## CONTENTS

1 Array objects  
1.1 The N-dimensional array *(ndarray)*  
1.2 Scalars  
1.3 Data type objects *(dtype)*  
1.4 Indexing  
1.5 Iterating Over Arrays  
1.6 Standard array subclasses  
1.7 Masked arrays  
1.8 The Array Interface  
1.9 Datetimes and Timedeltas  

2 Universal functions *(ufunc)*  
2.1 Broadcasting  
2.2 Output type determination  
2.3 Use of internal buffers  
2.4 Error handling  
2.5 Casting Rules  
2.6 *ufunc*  
2.7 Available ufuncs  

3 Routines  
3.1 Array creation routines  
3.2 Array manipulation routines  
3.3 Binary operations  
3.4 String operations  
3.5 C-Types Foreign Function Interface *(numpy.ctypeslib)*  
3.6 Datetime Support Functions  
3.7 Data type routines  
3.8 Optionally Scipy-accelerated routines *(numpy.dual)*  
3.9 Mathematical functions with automatic domain *(numpy.emath)*  
3.10 Floating point error handling  
3.11 Discrete Fourier Transform *(numpy.fft)*  
3.12 Financial functions  
3.13 Functional programming  
3.14 Numpy-specific help functions  
3.15 Indexing routines  
3.16 Input and output  
3.17 Linear algebra *(numpy.linalg)*  
3.18 Logic functions  

3 Routines 451

1 Array objects 3

2 Universal functions *(ufunc)* 431

3 Routines 451
<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.19</td>
<td>Masked array operations</td>
</tr>
<tr>
<td>3.20</td>
<td>Mathematical functions</td>
</tr>
<tr>
<td>3.21</td>
<td>Matrix library (numpy.matlib)</td>
</tr>
<tr>
<td>3.22</td>
<td>Numarray compatibility (numpy.numarray)</td>
</tr>
<tr>
<td>3.23</td>
<td>Old Numeric compatibility (numpy.oldnumeric)</td>
</tr>
<tr>
<td>3.24</td>
<td>Miscellaneous routines</td>
</tr>
<tr>
<td>3.25</td>
<td>Padding Arrays</td>
</tr>
<tr>
<td>3.26</td>
<td>Polynomials</td>
</tr>
<tr>
<td>3.27</td>
<td>Random sampling (numpy.random)</td>
</tr>
<tr>
<td>3.28</td>
<td>Set routines</td>
</tr>
<tr>
<td>3.29</td>
<td>Sorting, searching, and counting</td>
</tr>
<tr>
<td>3.30</td>
<td>Statistics</td>
</tr>
<tr>
<td>3.31</td>
<td>Test Support (numpy.testing)</td>
</tr>
<tr>
<td>3.32</td>
<td>Asserts</td>
</tr>
<tr>
<td>3.33</td>
<td>Window functions</td>
</tr>
<tr>
<td>4</td>
<td>Packaging (numpy.distutils)</td>
</tr>
<tr>
<td>4.1</td>
<td>Modules in numpy.distutils</td>
</tr>
<tr>
<td>4.2</td>
<td>Building Installable C libraries</td>
</tr>
<tr>
<td>4.3</td>
<td>Conversion of .src files</td>
</tr>
<tr>
<td>5</td>
<td>Numpy C-API</td>
</tr>
<tr>
<td>5.1</td>
<td>Python Types and C-Structures</td>
</tr>
<tr>
<td>5.2</td>
<td>System configuration</td>
</tr>
<tr>
<td>5.3</td>
<td>Data Type API</td>
</tr>
<tr>
<td>5.4</td>
<td>Array API</td>
</tr>
<tr>
<td>5.5</td>
<td>Array Iterator API</td>
</tr>
<tr>
<td>5.6</td>
<td>UFunc API</td>
</tr>
<tr>
<td>5.7</td>
<td>Generalized Universal Function API</td>
</tr>
<tr>
<td>5.8</td>
<td>Numpy core libraries</td>
</tr>
<tr>
<td>5.9</td>
<td>C API Deprecations</td>
</tr>
<tr>
<td>6</td>
<td>Numpy internals</td>
</tr>
<tr>
<td>6.1</td>
<td>Numpy C Code Explanations</td>
</tr>
<tr>
<td>6.2</td>
<td>Internal organization of numpy arrays</td>
</tr>
<tr>
<td>6.3</td>
<td>Multidimensional Array Indexing Order Issues</td>
</tr>
<tr>
<td>7</td>
<td>Numpy and SWIG</td>
</tr>
<tr>
<td>7.1</td>
<td>Numpy.i: a SWIG Interface File for NumPy</td>
</tr>
<tr>
<td>7.2</td>
<td>Testing the numpy.i Typemaps</td>
</tr>
<tr>
<td>8</td>
<td>Acknowledgements</td>
</tr>
</tbody>
</table>

Bibliography

Index
This reference manual details functions, modules, and objects included in Numpy, describing what they are and what they do. For learning how to use NumPy, see also user.
NumPy provides an N-dimensional array type, the `ndarray`, which describes a collection of “items” of the same type. The items can be indexed using for example N integers.

All ndarrays are homogenous: every item takes up the same size block of memory, and all blocks are interpreted in exactly the same way. How each item in the array is to be interpreted is specified by a separate data-type object, one of which is associated with every array. In addition to basic types (integers, floats, etc.), the data type objects can also represent data structures.

An item extracted from an array, e.g., by indexing, is represented by a Python object whose type is one of the array scalar types built in Numpy. The array scalars allow easy manipulation of also more complicated arrangements of data.

Figure 1.1: **Figure** Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned when a single element of the array is accessed.

1.1 The N-dimensional array (ndarray)

An `ndarray` is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its shape, which is a tuple of N positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype), one of which is associated with each ndarray.

As with other container objects in Python, the contents of an `ndarray` can be accessed and modified by indexing or slicing the array (using, for example, N integers), and via the methods and attributes of the `ndarray`. 
Different ndarrays can share the same data, so that changes made in one ndarray may be visible in another. That is, an ndarray can be a “view” to another ndarray, and the data it is referring to is taken care of by the “base” ndarray. ndarrays can also be views to memory owned by Python strings or objects implementing the buffer or array interfaces.

Example

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<type 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

The array can be indexed using Python container-like syntax:

```python
>>> x[1,2] # i.e., the element of x in the second row, third column, namely, 6.
```

For example slicing can produce views of the array:

```python
>>> y = x[:,1]
>>> y
array([2, 5])
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5])
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

1.1.1 Constructing arrays

New arrays can be constructed using the routines detailed in Array creation routines, and also by using the low-level ndarray constructor:

```python
ndarray
```

An array object represents a multidimensional, homogeneous array of fixed-size items.

class numpy.ndarray

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using array, zeros or empty (refer to the See Also section below). The parameters given here refer to a low-level method (ndarray(...)) for instantiating an array.

For more information, refer to the numpy module and examine the the methods and attributes of an array.

Parameters
(for the __new__ method; see Notes below)

- **shape**: tuple of ints
  Shape of created array.
**dtype**: data-type, optional

Any object that can be interpreted as a numpy data type.

**buffer**: object exposing buffer interface, optional

Used to fill the array with data.

**offset**: int, optional

Offset of array data in buffer.

**strides**: tuple of ints, optional

Strides of data in memory.

**order**: {'C', 'F'}, optional

Row-major or column-major order.

See Also:

- **array**: Construct an array.
- **zeros**: Create an array, each element of which is zero.
- **empty**: Create an array, but leave its allocated memory unchanged (i.e., it contains “garbage”).
- **dtype**: Create a data-type.

Notes

There are two modes of creating an array using **__new__**:

1. If **buffer** is None, then only **shape**, **dtype**, and **order** are used.
2. If **buffer** is an object exposing the buffer interface, then all keywords are interpreted.

No **__init__** method is needed because the array is fully initialized after the **__new__** method.

Examples

These examples illustrate the low-level **ndarray** constructor. Refer to the See Also section above for easier ways of constructing an ndarray.

First mode, **buffer** is None:

```python
>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[-1.13698227e+002, 4.25087011e-303],
       [ 2.88528414e-306, 3.27025015e-309]]) #random
```

Second mode:

```python
>>> np.ndarray((2,), buffer=np.array([1,2,3]),
... offset=np.int_().itemsize,
... dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

Attributes
**ndarray.T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

**Examples**

```python
>>> x = np.array([[1.,2.],[3.,4.]]
>>> x
array([[ 1., 2.],
       [ 3., 4.]])
>>> x.T
array([[ 1., 3.],
       [ 2., 4.]])
```

**ndarray.data**

Python buffer object pointing to the start of the array’s data.

**ndarray.dtype**

Data-type of the array’s elements.

**Parameters**

None

**Returns**

d : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```python
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
```
>>> type(x.dtype)
<type 'numpy.dtype'>

**ndarray.flags**

Information about the memory layout of the array.

### Notes

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercase attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- **UPDATEIFCOPY** can only be set `False`.
- **ALIGNED** can only be set `True` if the data is truly aligned.
- **WRITEABLE** can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be arbitrary if `arr.shape[dim] == 1` or the array has no elements. It does not generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CONTIGUOUS</td>
<td>The data is in a single, C-style contiguous segment.</td>
</tr>
<tr>
<td>F_CONTIGUOUS</td>
<td>The data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWN_DATA</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only.</td>
</tr>
<tr>
<td>ALIGNED</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATABLE</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will be</td>
</tr>
<tr>
<td></td>
<td>updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>

### ndarray.flat

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

### See Also:

- **flatten**
  - Return a copy of the array collapsed into one dimension.

### Examples

```python
gen
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x

array([[1, 2, 3],
       [4, 5, 6]])

>>> x.flat[3]
4

>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

**ndarray.imag**
The imaginary part of the array.

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([0.        , 0.70710678])
```

**ndarray.real**
The real part of the array.

**See Also:**

**numpy.real**
equivalent function

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([1.        , 0.70710678])
```

**ndarray.size**
Number of elements in the array.
Equivalent to `np.prod(a.shape)`, i.e., the product of the array’s dimensions.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

**ndarray.itemsize**
Length of one array element in bytes.
Examples

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**ndarray.nbytes**

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

Examples

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndarray.ndim**

Number of array dimensions.

Examples

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

**ndarray.shape**

Tuple of array dimensions.

Notes

May be used to “reshape” the array, as long as this would not require a change in the total number of elements.

Examples

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
```
ndarray.

**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element \((i[0], i[1], \ldots, i[n])\) in an array \(a\) is:

\[
\text{offset} = \sum(n\text{array}(i) \times a.\text{strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

**See Also:**

numpy.lib.stride_tricks.as_strided

**Notes**

Imagine an array of 32-bit integers (each 4 bytes):

\[
x = \text{np.array}([[0, 1, 2, 3, 4],
                    [5, 6, 7, 8, 9]], \text{dtype=\text{np.int32}})
\]

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array \(x\) will be \((20, 4)\).

**Examples**

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
```

```python
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
        [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

```python
>>> y.strides
(48, 16, 4)
```

```python
>>> y[1,1,1]
17
```

```python
>>> offset = sum(y.strides * np.array((1,1,1)))
```

```python
>>> offset / y.itemsize
17
```

```python
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
```

```python
array([[[[ 0,  1,  2,  3,  4,  5,  6,  7],
        [ 8,  9, 10, 11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20, 21, 22, 23],
        [24, 25, 26, 27, 28, 29, 30, 31],
        [32, 33, 34, 35, 36, 37, 38, 39]],
        [[40, 41, 42, 43, 44, 45, 46, 47],
        [48, 49, 50, 51, 52, 53, 54, 55],
        [56, 57, 58, 59, 60, 61, 62, 63],
        [64, 65, 66, 67, 68, 69, 70, 71],
        [72, 73, 74, 75, 76, 77, 78, 79]],
        [[80, 81, 82, 83, 84, 85, 86, 87],
        [88, 89, 90, 91, 92, 93, 94, 95],
        [96, 97, 98, 99, 100, 101, 102, 103],
        [104, 105, 106, 107, 108, 109, 110, 111],
        [112, 113, 114, 115, 116, 117, 118, 119]],
        [[120, 121, 122, 123, 124, 125, 126, 127],
        [128, 129, 130, 131, 132, 133, 134, 135],
        [136, 137, 138, 139, 140, 141, 142, 143],
        [144, 145, 146, 147, 148, 149, 150, 151],
        [152, 153, 154, 155, 156, 157, 158, 159]],
        [[160, 161, 162, 163, 164, 165, 166, 167],
        [168, 169, 170, 171, 172, 173, 174, 175],
        [176, 177, 178, 179, 180, 181, 182, 183],
        [184, 185, 186, 187, 188, 189, 190, 191],
        [192, 193, 194, 195, 196, 197, 198, 199]],
        [[200, 201, 202, 203, 204, 205, 206, 207],
        [208, 209, 210, 211, 212, 213, 214, 215],
        [216, 217, 218, 219, 220, 221, 222, 223],
        [224, 225, 226, 227, 228, 229, 230, 231],
        [232, 233, 234, 235, 236, 237, 238, 239]],
        [[240, 241, 242, 243, 244, 245, 246, 247],
        [248, 249, 250, 251, 252, 253, 254, 255],
        [256, 257, 258, 259, 260, 261, 262, 263],
        [264, 265, 266, 267, 268, 269, 270, 271],
        [272, 273, 274, 275, 276, 277, 278, 279]]])
```

```python
>>> x.strides
(32, 4, 224, 1344)
```

```python
>>> i = np.array([3,5,2,2])
```

```python
>>> offset = sum(i * x.strides)
```

```python
>>> x[3,5,2,2]
813
```

```python
>>> offset / x.itemsize
813
```

**ndarray.ctypes**

An object to simplify the interaction of the array with the ctypes module.
This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

None

**Returns**

c : Python object

Possessing attributes data, shape, strides, etc.

**See Also:**

numpy.ctypeslib

**Notes**

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

• data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

• shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

• strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

• data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

• shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

• strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

**Examples**

```python
>>> import ctypes
>>> x
```
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes._LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
_c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
_c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>

ndarray.base
Base object if memory is from some other object.

Examples
The base of an array that owns its memory is None:
>>> x = np.array([1,2,3,4])
>>> x.base is None
True

Slicing creates a view, whose memory is shared with x:
>>> y = x[2:]
>>> y.base is x
True

Methods

- `all([axis, out])`: Returns True if all elements evaluate to True.
- `any([axis, out])`: Returns True if any of the elements of a evaluate to True.
- `argmax([axis, out])`: Return indices of the maximum values along the given axis.
- `argmin([axis, out])`: Return indices of the minimum values along the given axis of a.
- `argpartition(kth, [axis, kind, order])`: Returns the indices that would partition this array.
- `argsort([axis, kind, order])`: Returns the indices that would sort this array.
- `astype(dtype, [order, casting, subok, copy])`: Copy of the array, cast to a specified type.
- `byteswap(inplace)`: Swap the bytes of the array elements
- `choose(choices[, out, mode])`: Use an index array to construct a new array from a set of choices.
- `clip(a_min, a_max[, out])`: Return an array whose values are limited to [a_min, a_max].
- `compress(condition[, axis, out])`: Return selected slices of this array along given axis.
- `conj()`: Complex-conjugate all elements.
- `conjugate()`: Return the complex conjugate, element-wise.
- `copy([order])`: Return a copy of the array.
- `cumprod([axis, dtype, out])`: Return the cumulative product of the elements along the given axis.
- `cumsum([axis, dtype, out])`: Return the cumulative sum of the elements along the given axis.
- `diagonal([offset, axis1, axis2])`: Return specified diagonals.
Table 1.3 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dot(b[, out])</code></td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td><code>dump(file)</code></td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td><code>dumps()</code></td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td><code>fill(value)</code></td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td><code>flatten([order])</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>item(*args)</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>itemset(*args)</code></td>
<td>Insert scalar into an array (scalar is cast to array’s dtype, if possible)</td>
</tr>
<tr>
<td><code>max([axis, out])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>mean([axis, dtype, out])</code></td>
<td>Returns the average of the array elements along given axis.</td>
</tr>
<tr>
<td><code>min([axis, out])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>newbyteorder([new_order])</code></td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>partition(kth[, axis, kind, order])</code></td>
<td>Rearranges the elements in the array in such a way that value of the element in kth position is in its sorted position.</td>
</tr>
<tr>
<td><code>prod([axis, dtype, out])</code></td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>ptp([axis, out])</code></td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td><code>put(indices, values[, mode])</code></td>
<td>Set a.flat[n] = values[n] for all n in indices.</td>
</tr>
<tr>
<td><code>ravel([order])</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>reshape(new_shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>round(decimals, out)</code></td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td><code>searchsorted(v[, side, sorter])</code></td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td><code>setfield(val, dtype[, offset])</code></td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td><code>setflags([write, align, uic])</code></td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td><code>sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td><code>std([axis, dtype, out, ddof])</code></td>
<td>Returns the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td><code>take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td><code>trace([offset, axis1, axis2, dtype, out])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>var([axis, dtype, out, ddof])</code></td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td><code>view([dtype])</code></td>
<td>New view of array with the same data.</td>
</tr>
</tbody>
</table>

**ndarray.all (axis=None, out=None)**

Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

See Also:

**numpy.all**

equivalent function

**ndarray.any (axis=None, out=None)**

Returns True if any of the elements of a evaluate to True.

Refer to `numpy.any` for full documentation.
See Also:

**ndarray.argmax** *(axis=None, out=None)*

Return indices of the maximum values along the given axis.

Refer to `ndarray.argmax` for full documentation.

See Also:

**ndarray.argmin** *(axis=None, out=None)*

Return indices of the minimum values along the given axis of `a`.

Refer to `ndarray.argmin` for detailed documentation.

See Also:

**ndarray.argpartition** *(kth, axis=-1, kind='introselect', order=None)*

Returns the indices that would partition this array.

Refer to `ndarray.argpartition` for full documentation. New in version 1.8.0.

See Also:

**ndarray.argsort** *(axis=-1, kind='quicksort', order=None)*

Returns the indices that would sort this array.

Refer to `ndarray.argsort` for full documentation.

See Also:

**ndarray.astype** *(dtype, order='K', casting='unsafe', subok=True, copy=True)*

Copy of the array, cast to a specified type.

Parameters:

- **dtype**: str or dtype
  
  Typecode or data-type to which the array is cast.

- **order**: {'C', 'F', 'A', 'K'}, optional
  
  Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

1.1. The N-dimensional array (`ndarray`)
casting: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok: bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy: bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the dtype, order, and subok requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t: ndarray

Unless copy is False and the other conditions for returning the input array are satisfied (see description for copy input parameter), arr_t is a new array of the same shape as the input array, with dtype, order given by dtype, order.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use a.real.astype(t).

Examples

>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1., 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])

ndarray.byteswap(inplace)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace: bool, optional

If True, swap bytes in-place, default is False.

Returns

out: ndarray

The byteswapped array. If inplace is True, this is a view to self.
Examples

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'], dtype='|S3')
```

`ndarray.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See Also:

- `numpy.choose`
equivalent function

`ndarray.clip(a_min, a_max, out=None)`

Return an array whose values are limited to \([a_{\text{min}}, a_{\text{max}}]\).

Refer to `numpy.clip` for full documentation.

See Also:

- `numpy.clip`
equivalent function

`ndarray.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also:

- `numpy.compress`
equivalent function

`ndarray.conj()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also:

- `numpy.conjugate`
equivalent function

`ndarray.conjugate()`

Return the complex conjugate, element-wise.
Refer to `numpy.conjugate` for full documentation.

See Also:

`numpy.conjugate`

equivalent function

`ndarray.copy(order='C')`

Return a copy of the array.

Parameters
order : {'C', 'F', 'A', 'K'}, optional
Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if `a` is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their order= arguments.)

See Also:

`numpy.copy, numpy.copyto`

Examples

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

`ndarray.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also:

`numpy.cumprod`

equivalent function

`ndarray.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also:

`numpy.cumsum`

equivalent function
ndarray.\texttt{diagonal}(\texttt{offset=0, axis1=0, axis2=1})

Return specified diagonals.

Refer to \texttt{numpy.diagonal} for full documentation.

\textbf{See Also:}

\texttt{numpy.diagonal}

equivalent function

\texttt{ndarray.dot}(\texttt{b, out=None})

Dot product of two arrays.

Refer to \texttt{numpy.dot} for full documentation.

\textbf{See Also:}

\texttt{numpy.dot}

equivalent function

\textbf{Examples}

\begin{verbatim}
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])

This array method can be conveniently chained:

>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
\end{verbatim}

\texttt{ndarray.dump}(\texttt{file})

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

\textbf{Parameters}

\texttt{file : str}

A string naming the dump file.

\texttt{ndarray.dumps()}

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

\textbf{Parameters}

\texttt{None}

\texttt{ndarray.fill}(\texttt{value})

Fill the array with a scalar value.

\textbf{Parameters}

\texttt{value : scalar}

All elements of \texttt{a} will be assigned this value.

\textbf{Examples}
```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

**ndarray.flatten(order='C')**

Return a copy of the array collapsed into one dimension.

**Parameters**

order : {'C', 'F', 'A'}, optional

Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from `a`. The default is ‘C’.

**Returns**

y : ndarray

A copy of the input array, flattened to one dimension.

**See Also:**

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

**Examples**

```python
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**ndarray.getfield(dtype, offset=0)**

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

**Examples**
```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```c
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```

ndarray.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

  • none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.
  • int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
  • tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

Examples

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```
**ndarray.itemset(*args)**

Insert scalar into an array (scalar is cast to array’s dtype, if possible)

There must be at least 1 argument, and define the last argument as `item`. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and `args` must select a single item in the array `a`.

**Parameters**

*args : Arguments

If one argument: a scalar, only used in case `a` is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

**Notes**

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

**Examples**

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

**ndarray.max(axis=None, out=None)**

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

**See Also:**

- `numpy.amax`
  equivalent function

**ndarray.mean(axis=None, dtype=None, out=None)**

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

**See Also:**

- `numpy.mean`
  equivalent function

**ndarray.min(axis=None, out=None)**

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

**See Also:**
**ndarray.newbyteorder** *(new_order='S')*

Return the array with the same data viewed with a different byte order.

Equivalent to:

```python
arr.view(arr.dtype.newbyteorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

- **new_order**: string, optional
  
  Byte order to force; a value from the byte order specifications above. *new_order* codes can be any of:

  - `'S'` - swap dtype from current to opposite endian
  - `'<'`, `'L'` - little endian
  - `'>'`, `'B'` - big endian
  - `'='`, `'N'` - native order
  - `'|'`, `'I'` - ignore (no change to byte order)

  The default value (`'S'`) results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of `B` or `b` or `biggish` are valid to specify big-endian.

**Returns**

- **new_arr**: array

  New array object with the dtype reflecting given change to the byte order.

**ndarray.nonzero()**

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

**See Also:**

- `numpy.nonzero`

**ndarray.partition** *(kth, axis=-1, kind='introselect', order=None)*

Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined. New in version 1.8.0.

**Parameters**

- **kth**: int or sequence of ints
  
  Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

- **axis**: int, optional
  
  Axis along which to sort. Default is -1, which means sort along the last axis.
kind : {'introselect'}, optional
    Selection algorithm. Default is ‘introselect’.

order : list, optional
    When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

numpy.partition
    Return a partitioned copy of an array.

argpartition
    Indirect partition.

sort
    Full sort.

Notes
    See np.partition for notes on the different algorithms.

Examples
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

ndarray.prod(axis=None, dtype=None, out=None)
    Return the product of the array elements over the given axis

    Refer to numpy.prod for full documentation.

    See Also:

    numpy.prod
        equivalent function

ndarray.ptp(axis=None, out=None)
    Peak to peak (maximum - minimum) value along a given axis.

    Refer to numpy.ptp for full documentation.

    See Also:

    numpy.ptp
        equivalent function

ndarray.put(indices, values, mode='raise')
    Set a.flat[n] = values[n] for all n in indices.

    Refer to numpy.put for full documentation.

    See Also:
**numpy.put**

equivalent function

**ndarray.ravel**([`order`])

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

See Also:

**numpy.ravel**

equivalent function

**ndarray.flat**

a flat iterator on the array.

**ndarray.repeat**(`repeats`, `axis=None`)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also:

**numpy.repeat**

equivalent function

**ndarray.reshape**(`shape`, `order='C'`)

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See Also:

**numpy.reshape**

equivalent function

**ndarray.resize**(`new_shape`, `refcheck=True`)

Change shape and size of array in-place.

**Parameters**

- `new_shape` : tuple of ints, or $n$ ints
  Shape of resized array.

- `refcheck` : bool, optional
  If False, reference count will not be checked. Default is True.

**Returns**

None

**Raises**

- `ValueError`
  If $a$ does not own its own data or references or views to it exist, and the data memory must be changed.

- `SystemError`
  If the `order` keyword argument is specified. This behaviour is a bug in NumPy.

See Also:
**resize**

Return a new array with the specified shape.

**Notes**

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

**Examples**

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
array([[0],
       [1]])
```

```python
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize((2, 3)) # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ...   ValueError: cannot resize an array that has been referenced ...
```

Unless `refcheck` is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

**ndarray.round**  

`ndarray.round(decimals=0, out=None)`  

Return `a` with each element rounded to the given number of decimals.  

Refer to `numpy.round` for full documentation.

**See Also:**
**numpy.around**

equivalent function

**ndarray.searchsorted** *(v, side='left', sorter=None)*

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

**See Also:**

**numpy.searchsorted**
equivalent function

**ndarray.setfield** *(val, dtype, offset=0)*

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*s field defined by *dtype* and beginning *offset* bytes into the field.

**Parameters**

*val*: object
  Value to be placed in field.

*dtype*: dtype object
  Data-type of the field in which to place *val*.

*offset*: int, optional
  The number of bytes into the field at which to place *val*.

**Returns**

None

**See Also:**

`getfield`

**Examples**

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

```python
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

```python
>>> x
array([[ 1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
       [ 1.48219694e-323, 1.00000000e+000, 1.48219694e-323],
       [ 1.48219694e-323, 1.48219694e-323, 1.00000000e+000]])
```

```python
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

**ndarray.setflags** *(write=None, align=None, uic=None)*

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
These Boolean-valued flags affect how numpy interprets the memory area used by \( a \) (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

- **write**: bool, optional
  
  Describes whether or not \( a \) can be written to.

- **align**: bool, optional
  
  Describes whether or not \( a \) is aligned properly for its type.

- **uic**: bool, optional
  
  Describes whether or not \( a \) is a copy of another “base” array.

**Notes**

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

- WRITEABLE (W) the data area can be written to;
- ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);
- UPDATEIFCOPY (U) this array is a copy of some other array (referenced by `.base`). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

**Examples**

```python
>>> y = np.array([[3, 1, 7],
                [2, 0, 0],
                [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

```python
>>> y.setflags(write=0, align=0)
```

```python
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
```
ndarray.sort (axis=-1, kind='quicksort', order=None)
Sort an array, in-place.

   Parameters
      axis : int, optional
         Axis along which to sort. Default is -1, which means sort along the last axis.
      kind : {'quicksort', 'mergesort', 'heapsort'}, optional
         Sorting algorithm. Default is 'quicksort'.
      order : list, optional
         When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

numpy.sort
   Return a sorted copy of an array.
argsort
   Indirect sort.
lexsort
   Indirect stable sort on multiple keys.
searchsorted
   Find elements in sorted array.
partition
   Partial sort.

Notes
See sort for notes on the different sorting algorithms.

Examples
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4], [1, 3]])

Use the order keyword to specify a field to use when sorting a structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])

ndarray.squeeze (axis=None)
Remove single-dimensional entries from the shape of a.

Refer to numpy.squeeze for full documentation.
See Also:

- `numpy.squeeze` equivalent function

`ndarray.std`\( (axis=\text{None}, \text{dtype}=\text{None}, \text{out}=\text{None}, ddof=0) \)
Returns the standard deviation of the array elements along given axis.
Refer to `numpy.std` for full documentation.

See Also:

- `numpy.std` equivalent function

`ndarray.sum`\( (axis=\text{None}, \text{dtype}=\text{None}, \text{out}=\text{None}) \)
Return the sum of the array elements over the given axis.
Refer to `numpy.sum` for full documentation.

See Also:

- `numpy.sum` equivalent function

`ndarray.swapaxes`\( (axis1, axis2) \)
Return a view of the array with \text{axis1} and \text{axis2} interchanged.
Refer to `numpy.swapaxes` for full documentation.

See Also:

- `numpy.swapaxes` equivalent function

`ndarray.take`\( (indices, axis=\text{None}, \text{out}=\text{None}, \text{mode}='\text{raise}') \)
Return an array formed from the elements of \text{a} at the given indices.
Refer to `numpy.take` for full documentation.

See Also:

- `numpy.take` equivalent function

`ndarray.tofile`\( (fid, sep='\text{""}', format='\text{%s}') \)
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of \text{a}. The data produced by this method can be recovered using the function fromfile().

Parameters

- \text{(fid)} : file or str
  An open file object, or a string containing a filename.

- \text{sep} : str
  Separator between array items for text output. If ‘‘’ (empty), a binary file is written, equivalent to \text{file.write(a.tostring())}.
**format**: str

Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

**Notes**

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

```python
ndarray.tolist()
```

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

**Parameters**

none

**Returns**

y : list

The possibly nested list of array elements.

**Notes**

The array may be recreated, `a = np.array(a.tolist())`.

**Examples**

```python
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

```python
ndarray.tostring(order='C')
```

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

**Parameters**

order : {‘C’, ‘F’, None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

s : str

A Python string exhibiting a copy of a’s raw data.
Examples

>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'  
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

darray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

See Also:

numpy.trace
equivalent function

darray.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape
   = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters

axes : None, tuple of ints, or n ints
   • None or no argument: reverses the order of the axes.
   • tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s
     j-th axis.
   • n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience”
     alternative to the tuple form)

Returns

out : ndarray
   View of a, with axes suitably permuted.

See Also:

ndarray.T
   Array property returning the array transposed.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
   [3, 4]])
>>> a.transpose()
array([[1, 3],
   [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
   [2, 4]])
ndarray.transpose(1, 0)
array([[1, 3],
[2, 4]])

ndarray.var (axis=None, dtype=None, out=None, ddof=0)
Returns the variance of the array elements, along given axis.
Refer to numpy.var for full documentation.
See Also:
numpy.var
equivalent function

ndarray.view(dtype=None, type=None)
New view of array with the same data.
Parameters
dtype : data-type or ndarray sub-class, optional
Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as a. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the type parameter).
type : Python type, optional
Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes
a.view() is used two different ways:
a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.
a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For a.view(some_dtype), if some_dtype has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of a (shown by print(a)). It also depends on exactly how a is stored in memory. Therefore if a is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
Viewing array data using a different type and dtype:

>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print type(y)
<class ‘numpy.matrixlib.defmatrix.matrix’>
Creating a view on a structured array so it can be used in calculations

```python
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2., 3.])
```

Making changes to the view changes the underlying array

```python
>>> xv[0,1] = 20
>>> print x
[(1, 20) (3, 4)]
```

Using a view to convert an array to a record array:

```python
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```python
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```python
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 2],
       [4, 5]], dtype=[('width', '<i2'), ('length', '<i2')])
```

### 1.1.2 Indexing arrays

Arrays can be indexed using an extended Python slicing syntax, `array[selection]`. Similar syntax is also used for accessing fields in a record array.

**See Also:**

_Array Indexing_.

### 1.1.3 Internal memory layout of an ndarray

An instance of class `ndarray` consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps \( N \) integers into the location of an item
in the block. The ranges in which the indices can vary is specified by the *shape* of the array. How many bytes each item takes and how the bytes are interpreted is defined by the *data-type object* associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an N-dimensional array in a 1-dimensional block. Numpy is flexible, and *ndarray* objects can accommodate any *strided indexing scheme*. In a strided scheme, the N-dimensional index \((n_0, n_1, ..., n_{N-1})\) corresponds to the offset (in bytes):

\[
n_{\text{offset}} = \sum_{k=0}^{N-1} s_k n_k
\]

from the beginning of the memory block associated with the array. Here, \(s_k\) are integers which specify the *strides* of the array. The *column-major* order (used, for example, in the Fortran language and in *Matlab*) and *row-major* order (used in C) schemes are just specific kinds of strided scheme, and correspond to memory that can be *addressed* by the strides:

\[
s_{\text{column}}^k = \prod_{j=0}^{k-1} d_j, \quad s_{\text{row}}^k = \prod_{j=k+1}^{N-1} d_j.
\]

where \(d_j = \text{self.itemsize} \times \text{self.shape}[j]\).

Both the C and Fortran orders are *contiguous*, i.e., single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

While a C-style and Fortran-style contiguous array, which has the corresponding flags set, can be addressed with the above strides, the actual strides may be different. This can happen in two cases:

1. If *self.shape*[k] == 1 then for any legal index *index*[k] == 0. This means that in the formula for the offset \(n_k = 0\) and thus \(s_k n_k = 0\) and the value of \(s_k = \text{self.strides}[k]\) is arbitrary.

2. If an array has no elements (*self.size* == 0) there is no legal index and the strides are never used. Any array with no elements may be considered C-style and Fortran-style contiguous.

Point 1. means that *self* and *self.squeeze()* always have the same contiguity and *aligned* flags value. This also means that even a high dimensional array could be C-style and Fortran-style contiguous at the same time.

An array is considered aligned if the memory offsets for all elements and the base offset itself is a multiple of *self.itemsize*.

---

**Note:** Points (1) and (2) are not yet applied by default. Beginning with Numpy 1.8.0, they are applied consistently only if the environment variable *NPY_RELAXED_STRIDES_CHECKING=1* was defined when NumPy was built. Eventually this will become the default.

You can check whether this option was enabled when your NumPy was built by looking at the value of `np.ones((10,1), order='C').flags.f_contiguous`. If this is *True*, then your NumPy has relaxed strides checking enabled.

---

**Warning:** It does not generally hold that *self.strides[−1] == self.itemsize* for C-style contiguous arrays or *self.strides[0] == self.itemsize* for Fortran-style contiguous arrays is true.

---

Data in new *ndarrays* is in the *row-major* (C) order, unless otherwise specified, but, for example, *basic array slicing* often produces *views* in a different scheme.

**Note:** Several algorithms in NumPy work on arbitrarily strided arrays. However, some algorithms require single-segment arrays. When an irregularly strided array is passed in to such algorithms, a copy is automatically made.
1.1.4 Array attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Information on each attribute is given below.

Memory layout

The following attributes contain information about the memory layout of the array:

- `ndarray.flags`: Information about the memory layout of the array.
- `ndarray.shape`: Tuple of array dimensions.
- `ndarray.strides`: Tuple of bytes to step in each dimension when traversing an array.
- `ndarray.ndim`: Number of array dimensions.
- `ndarray.data`: Python buffer object pointing to the start of the array’s data.
- `ndarray.size`: Number of elements in the array.
- `ndarray.itemsize`: Length of one array element in bytes.
- `ndarray.nbytes`: Total bytes consumed by the elements of the array.
- `ndarray.base`: Base object if memory is from some other object.

Notes

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `UPDATEIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `UPDATEIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be arbitrary if `arr.shape[dim] == 1` or the array has no elements. It does not generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CONTIGUOUS</td>
<td>The data is in a single, C-style contiguous segment.</td>
</tr>
<tr>
<td>F_CONTIGUOUS</td>
<td>The data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWNDATA</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only.</td>
</tr>
<tr>
<td>ALIGNED</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATEIFCOPY</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will</td>
</tr>
<tr>
<td></td>
<td>be updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>

`ndarray.shape`

Tuple of array dimensions.

Notes

May be used to “reshape” the array, as long as this would not require a change in the total number of elements.

Examples

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

`ndarray.strides`

Tuple of bytes to step in each dimension when traversing an array.
The byte offset of element \((i[0], i[1], \ldots, i[n])\) in an array \(a\) is:

\[
\text{offset} = \sum (\text{np.array}(i) \times a.\text{strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

**See Also:**

numpy.lib.stride_tricks.as_strided

**Notes**

Imagine an array of 32-bit integers (each 4 bytes):

\[
x = \text{np.array}([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9]], \text{dtype=\text{np.int32}})
\]

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array \(x\) will be \((20, 4)\).

**Examples**

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[0, 1, 2, 3],
        [4, 5, 6, 7],
        [8, 9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

**ndarray.ndim**

Number of array dimensions.

**Examples**

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```
**ndarray.data**
Python buffer object pointing to the start of the array’s data.

**ndarray.size**
Number of elements in the array.
Equivalent to `np.prod(a.shape)`, i.e., the product of the array’s dimensions.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

**ndarray.itemsize**
Length of one array element in bytes.

**Examples**

```python
>>> x = np.array([1, 2, 3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1, 2, 3], dtype=np.complex128)
>>> x.itemsize
16
```

**ndarray.nbytes**
Total bytes consumed by the elements of the array.

**Notes**
Does not include memory consumed by non-element attributes of the array object.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**ndarray.base**
Base object if memory is from some other object.

**Examples**
The base of an array that owns its memory is `None`:

```python
>>> x = np.array([[1, 2, 3, 4]])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```python
>>> y = x[2:]
>>> y.base is x
True
```
Data type

See Also:

Data type objects

The data type object associated with the array can be found in the `dtype` attribute:

```
ndarray.dtype  Data-type of the array's elements.
```

**ndarray.dtype**

Data-type of the array’s elements.

**Parameters**

None

**Returns**

d : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```python
def x
array([[0, 1],
       [2, 3]])
def x.dtype
dtype('int32')
def x
<type 'numpy.dtype'>
```

Other attributes

```
ndarray.T  Same as self.transpose(), except that self is returned if self.ndim < 2.
numpy.real  The real part of the array.
numpy.imag  The imaginary part of the array.
numpy.flat  A 1-D iterator over the array.
numpy.ctypes  An object to simplify the interaction of the array with the ctypes module.
```

**ndarray.T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

**Examples**

```python
def x = np.array([[1., 2.], [3., 4.]])
def x
array([[ 1.,  2.],
       [ 3.,  4.]])
def x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
def x = np.array([1., 2., 3., 4.])
def x
```
array([ 1., 2., 3., 4.])
>>> x.T
array([ 1., 2., 3., 4.])

**ndarray.real**
The real part of the array.

**See Also:**

numpy.real
equivalent function

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**ndarray.imag**
The imaginary part of the array.

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**ndarray.flat**
A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

**See Also:**

flatten
Return a copy of the array collapsed into one dimension.

**flatiter**

**Examples**

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
```
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
```
```python
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

ndarray.ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**
- None

**Returns**
- c : Python object
  Possessing attributes data, shape, strides, etc.

**See Also:**
- numpy.ctypeslib

**Notes**

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

- strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

- data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).
Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling \((a+b).\text{ctypes.data\_as}(\text{ctypes.c\_void\_p})\) returns a pointer to memory that is invalid because the array created as \((a+b)\) is deallocated before the next Python statement. You can avoid this problem using either \(c=a+b\) or \(ct=(a+b).\text{ctypes}\). In the latter case, \(ct\) will hold a reference to the array until \(ct\) is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

**Examples**

```python
g++ import ctypes
>>> x  
array([[0, 1],  
      [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data\_as(ctypes.POINTER(ctypes.c\_long))<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data\_as(ctypes.POINTER(ctypes.c\_long)).contents
\(c\_long(0)\)
>>> x.ctypes.data\_as(ctypes.POINTER(ctypes.c\_longlong)).contents
\(c\_longlong(4294967296L)\)
>>> x.ctypes.shape
<numpy.core\_internal.c\_long\_Array\_2 object at 0x01FFD580>
>>> x.ctypes.shape\_as(ctypes.c\_long)
<numpy.core\_internal.c\_long\_Array\_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core\_internal.c\_long\_Array\_2 object at 0x01FCE620>
>>> x.ctypes.strides\_as(ctypes.c\_longlong)
<numpy.core\_internal.c\_longlong\_Array\_2 object at 0x01F01300>
```

**Array interface**

See Also:

* The Array Interface.
  
<table>
<thead>
<tr>
<th><strong>array_interface</strong></th>
<th>Python-side of the array interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>array_struct</strong></td>
<td>C-side of the array interface</td>
</tr>
</tbody>
</table>

**ctypes foreign function interface**

```python
ndarray.ctypes An object to simplify the interaction of the array with the ctypes module.
```

**ndarray.ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

None
Returns

- `c`: Python object
  Possessing attributes data, shape, strides, etc.

See Also:

- `numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- `data`: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

- `shape (c_intp*self.ndim)`: A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `c_int`, `c_long`, or `c_longlong` depending on the platform. The `c_intp` type is defined accordingly in `numpy.ctypeslib`. The ctypes array contains the shape of the underlying array.

- `strides (c_intp*self.ndim)`: A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

- `data_as(obj)`: Return the data pointer cast to a particular C-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(cfloat.c_double))`.

- `shape_as(obj)`: Return the shape tuple as an array of some other C-types type. For example: `self.shape_as(cfloat.c_short)`.

- `strides_as(obj)`: Return the strides tuple as an array of some other C-types type. For example: `self.strides_as(cfloat.c_longlong)`.

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(cfloat.c_void_p)` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, `ct` will hold a reference to the array until `ct` is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(cfloat.POINTER(cfloat.c_long))
<ctypes.c_long object at 0x01F01300>
>>> x.ctypes.data_as(cfloat.POINTER(cfloat.c_long)).contents
```
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
  c_longlong(4294967296L)
>>> x.ctypes.shape
  <numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
  <numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
  <numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
  <numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>

1.1.5 Array methods

An ndarray object has many methods which operate on or with the array in some fashion, typically returning an array result. These methods are briefly explained below. (Each method’s docstring has a more complete description.)

For the following methods there are also corresponding functions in numpy: all, any, argmax, argmin, argpartition, argsort, choose, clip, compress, copy, cumprod, cumsum, diagonal, imag, max, mean, min, nonzero, partition, prod, ptp, put, ravel, real, repeat, reshape, round, searchsorted, sort, squeeze, std, sum, swapaxes, take, trace, transpose, var.

Array conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.item(*args)</td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td>ndarray.tolist()</td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td>ndarray.itemset(*args)</td>
<td>Insert scalar into an array (scalar is cast to array’s dtype, if possible)</td>
</tr>
<tr>
<td>ndarray.setasflat</td>
<td></td>
</tr>
<tr>
<td>ndarray.tostring(order)</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>ndarray.tofile(fid[, sep, format])</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>ndarray.dump(file)</td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td>ndarray.dumps()</td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td>ndarray.astype(dtype[, order, casting, ...])</td>
<td>Copy of the array, cast to a specified type.</td>
</tr>
<tr>
<td>ndarray.byteswap(inplace)</td>
<td>Swap the bytes of the array elements</td>
</tr>
<tr>
<td>ndarray.copy(order)</td>
<td>Return a copy of the array.</td>
</tr>
<tr>
<td>ndarray.view(dtype, type)</td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td>ndarray.getfield(dtype[, offset])</td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td>ndarray.setflags(write, align, uic)</td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td>ndarray.fill(value)</td>
<td>Fill the array with a scalar value.</td>
</tr>
</tbody>
</table>

**ndarray.item(*args)**

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

*args: Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is
interpreted as an nd-index into the array.

Returns

- z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

Examples

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

ndarray.tolist()  
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters

- none

Returns

- y : list

The possibly nested list of array elements.

Notes

The array may be recreated, a = np.array(a.tolist()).

Examples

```python
>>> a = np.array([[1, 2]])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```
ndarray.itemset(*args)
Insert scalar into an array (scalar is cast to array’s dtype, if possible)

There must be at least 1 argument, and define the last argument as item. Then, a.itemset(*args) is equivalent to but faster than a[args] = item. The item should be a scalar value and args must select a single item in the array a.

Parameters
*args : Arguments

If one argument: a scalar, only used in case a is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes
Compared to indexing syntax, itemset provides some speed increase for placing a scalar into a particular location in an ndarray, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using itemset (and item) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])

ndarray.tostring(order='C')
Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : str

A Python string exhibiting a copy of a’s raw data.

Examples
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'b\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x03\x00\x00\x00\x00'
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x00\x03\x00\x00\x00'

ndarray.tofile(fid, sep='', format='%s')

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can be recovered using the function fromfile().

Parameters

- **fid**: file or str
  An open file object, or a string containing a filename.
- **sep**: str
  Separator between array items for text output. If '' (empty), a binary file is written, equivalent to file.write(a.tostring()).
- **format**: str
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

ndarray.dump(file)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

- **file**: str
  A string naming the dump file.

ndarray.dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

ndarray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

Copy of the array, cast to a specified type.

Parameters

- **dtype**: str or dtype
  Typecode or data-type to which the array is cast.
  Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.
- **casting**: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
  Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.
• ‘no’ means the data types should not be cast at all.
• ‘equiv’ means only byte-order changes are allowed.
• ‘safe’ means only casts which can preserve values are allowed.
• ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
• ‘unsafe’ means any data conversions may be done.

subok : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the dtype, order, and subok requirements are satisfied, the input array is returned instead of a copy.

Returns
arr_t : ndarray

Unless copy is False and the other conditions for returning the input array are satisfied (see description for copy input parameter), arr_t is a new array of the same shape as the input array, with dtype, order given by dtype, order.

 Raises
ComplexWarning

When casting from complex to float or int. To avoid this, one should use a.real.astype(t).

Examples
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1., 2., 2.5])

>>> x.astype(int)
array([1, 2, 2])

daarray.byteswap (inplace)
Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters
inplace : bool, optional

If True, swap bytes in-place, default is False.

Returns
out : ndarray

The byteswapped array. If inplace is True, this is a view to self.

Examples
```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

`ndarray.copy(order='C')`

Return a copy of the array.

**Parameters**

- `order`: {'C', 'F', 'A', 'K'}, optional
  - Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their `order` arguments.)

**See Also:**

`numpy.copy`, `numpy.copyto`

**Examples**

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

`ndarray.view(dtype=None, type=None)`

New view of array with the same data.

**Parameters**

- `dtype`: data-type or ndarray sub-class, optional
  - Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as `a`. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

- `type`: Python type, optional
Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

a.view() is used two different ways:

a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For a.view(some_dtype), if some_dtype has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of a (shown by print(a)). It also depends on exactly how a is stored in memory. Therefore if a is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:

```python
>>> y = x.view(dtype=np.int16, type=np.matrix)
```  
```python
>>> y
matrix([[513]], dtype=int16)
```  
```python
>>> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
```  
Creating a view on a structured array so it can be used in calculations

```python
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
```  
```python
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2., 3.])
```  
Making changes to the view changes the underlying array

```python
>>> xv[0,1] = 20
```  
```python
>>> print x
[(1, 20) (3, 4)]
```  
Using a view to convert an array to a record array:

```python
>>> z = x.view(np.recarray)
```  
```python
>>> z.a
array([1], dtype=int8)
```  
Views share data:

```python
>>> x[0] = (9, 10)
```  
```python
>>> z[0]
(9, 10)
```
Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```python
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
ValueError: new type not compatible with array.
```

```python
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 2],
       [4, 5]], dtype=[('width', '<i2'), ('length', '<i2')])
```

```
ndarray.getfield(dtype, offset=0)
```

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

- `dtype` : str or dtype
  The data type of the view. The dtype size of the view can not be larger than that of the array itself.

- `offset` : int
  Number of bytes to skip before beginning the element view.

**Examples**

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```

```
ndarray.setflags(write=None, align=None, uic=None)
```

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)
Parameters

- **write**: bool, optional
  Describes whether or not `a` can be written to.

- **align**: bool, optional
  Describes whether or not `a` is aligned properly for its type.

- **uic**: bool, optional
  Describes whether or not `a` is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

- **WRITEABLE (W)** the data area can be written to;
- **ALIGNED (A)** the data and strides are aligned appropriately for the hardware (as determined by the compiler);
- **UPDATEIFCOPY (U)** this array is a copy of some other array (referenced by `.base`). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```python
>>> y = np.array([[3, 1, 7], [2, 0, 0], [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
```

```python
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

`ndarray.fill(value)`

Fill the array with a scalar value.

Parameters

- **value**: scalar
  
  All elements of `a` will be assigned this value.
Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with \( n \) integers which will be interpreted as an \( n \)-tuple.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ndarray.reshape</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>ndarray.resize</code></td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td><code>ndarray.transpose</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>ndarray.swapaxes</code></td>
<td>Return a view of the array with <code>axis1</code> and <code>axis2</code> interchanged.</td>
</tr>
<tr>
<td><code>ndarray.flatten</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>ndarray.ravel</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>ndarray.squeeze</code></td>
<td>Remove single-dimensional entries from the shape of ( a ).</td>
</tr>
</tbody>
</table>

`ndarray.reshape` *(shape, order='C')*

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

See Also:

`numpy.reshape`

equivalent function

`ndarray.resize` *(new_shape, refcheck=True)*

Change shape and size of array in-place.

Parameters

- `new_shape` : tuple of ints, or \( n \) ints
  
  Shape of resized array.

- `refcheck` : bool, optional
  
  If False, reference count will not be checked. Default is True.

Returns

None

Raises

- `ValueError`
  
  If \( a \) does not own its own data or references or views to it exist, and the data memory must be changed.

- `SystemError`
  
  If the `order` keyword argument is specified. This behaviour is a bug in NumPy.
See Also:

resize

Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python
object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure
that you have not shared the memory for this array with another Python object, then you may safely set refcheck
to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```python
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless refcheck is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

ndarray.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a ma-
trix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order in-
indicates how the axes are permuted (see Examples). If axes are not provided and \( a.\text{shape} = (i[0], i[1], \ldots i[n-2], i[n-1]) \), then \( a.\text{transpose}().\text{shape} = (i[n-1], i[n-2], \ldots i[1], i[0]) \).

Parameters

- **axes**: None, tuple of ints, or \( n \) ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: \( i \) in the \( j \)-th place in the tuple means \( a \)'s \( i \)-th axis becomes \( a.\text{transpose}() \)'s \( j \)-th axis.
  - \( n \) ints: same as an \( n \)-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns

- **out**: \( \text{ndarray} \)
  View of \( a \), with axes suitably permuted.

See Also:

- \( \text{ndarray.T} \)
  Array property returning the array transposed.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

**ndarray.swapaxes** (axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.

Refer to \( \text{numpy.swapaxes} \) for full documentation.

See Also:

- \( \text{numpy.swapaxes} \)
  equivalent function

**ndarray.flatten** (order='C')
Return a copy of the array collapsed into one dimension.

Parameters

- **order**: {'C', 'F', 'A'}, optional
  Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from \( a \). The default is ‘C’.

Returns

- **y**: \( \text{ndarray} \)
A copy of the input array, flattened to one dimension.

See Also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
aarray([1, 2, 3, 4])
>>> a.flatten('F')
aarray([1, 3, 2, 4])
```

```python
ndarray.ravel(order)
Return a flattened array.

Refer to numpy.ravel for full documentation.

See Also:

numpy.ravel
equivalent function
```

```python
ndarray.flat
a flat iterator on the array.
```

```python
ndarray.squeeze(axis=None)
Remove single-dimensional entries from the shape of a.

Refer to numpy.squeeze for full documentation.

See Also:

numpy.squeeze
equivalent function
```

Item selection and manipulation

For array methods that take an `axis` keyword, it defaults to `None`. If `axis` is `None`, then the array is treated as a 1-D array. Any other value for `axis` represents the dimension along which the operation should proceed.

```
ndarray.take(indices[, axis, out, mode])
Return an array formed from the elements of a at the given indices.

ndarray.put(indices, values[, mode])
Set a.flat[n] = values[n] for all n in indices.

ndarray.repeat(repeats[, axis])
Repeat elements of an array.

ndarray.choose(choices[, out, mode])
Use an index array to construct a new array from a set of choices.

ndarray.sort([axis, kind, order])
Sort an array, in-place.

ndarray.argpartition(kth[, axis, kind, order])
Returns the indices that would partition this array.

ndarray.partition(kth[, axis, kind, order])
Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array.

ndarray.searchsorted(v[, side, sorter])
Find indices where elements of v should be inserted in a to maintain order.

ndarray.nonzero()
Return the indices of the elements that are non-zero.
```
Table 1.10 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ndarray.compress()</code></td>
<td>Return selected slices of this array along given axis.</td>
</tr>
<tr>
<td><code>ndarray.diagonal()</code></td>
<td>Return specified diagonals.</td>
</tr>
</tbody>
</table>

**ndarray.take**

```python
indices, axis=None, out=None, mode='raise')
```

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.

**See Also:**

- `numpy.take`

  equivalent function

**ndarray.put**

```python
indices, values, mode='raise')
```

Set `a.flat[n] = values[n]` for all `n` in indices.

Refer to `numpy.put` for full documentation.

**See Also:**

- `numpy.put`

  equivalent function

**ndarray.repeat**

```python
repeats, axis=None)
```

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See Also:**

- `numpy.repeat`

  equivalent function

**ndarray.choose**

```python
choices, out=None, mode='raise')
```

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See Also:**

- `numpy.choose`

  equivalent function

**ndarray.sort**

```python
(axis=-1, kind='quicksort', order=None)
```

Sort an array, in-place.

**Parameters**

- `axis` : int, optional
  
  Axis along which to sort. Default is -1, which means sort along the last axis.

- `kind` : {'quicksort', 'mergesort', 'heapsort'}, optional
  
  Sorting algorithm. Default is 'quicksort'.

- `order` : list, optional
  
  When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.
See Also:

```
numpy.sort
  Return a sorted copy of an array.
argsort
  Indirect sort.
lexsort
  Indirect stable sort on multiple keys.
searchsorted
  Find elements in sorted array.
partition
  Partial sort.
```

Notes
See sort for notes on the different sorting algorithms.

Examples
```
>>> a = np.array([[1, 4], [3, 1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:
```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
       dtype=[('x', '|S1'), ('y', '<i4')])
```

\[\textit{ndarray}.\texttt{argsort}(axis=-1, \textit{kind}=\texttt{quicksort}, \textit{order}=None)\]
Returns the indices that would sort this array.

Refer to numpy.argsort for full documentation.

See Also:

```
numpy.argsort
  equivalent function
```

\[\textit{ndarray}.\texttt{partition}(kth, axis=-1, \textit{kind}=\texttt{introselect}, \textit{order}=None)\]
Rearranges the elements in the array in such a way that value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined. New in version 1.8.0.

**Parameters**

```
  \textit{kth} : \text{int or sequence of ints}
```

1.1. The N-dimensional array (ndarray)
Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

**axis**: int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**: {'introselect'}, optional

Selection algorithm. Default is ‘introselect’.

**order**: list, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

- `numpy.partition`
  Return a partitioned copy of an array.
- `argpartition`
  Indirect partition.
- `sort`
  Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```python
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
array([2, 1, 3, 4])
>>> a.partition((1, 3))
array([1, 2, 3, 4])
```

```python
ndarray.argpartition(kth, axis=-1, kind='introselect', order=None)
```

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation. New in version 1.8.0.

See Also:

- `numpy.argpartition`
  equivalent function

```python
ndarray.searchsorted(v, side='left', sorter=None)
```

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

See Also:

- `numpy.searchsorted`
  equivalent function
ndarray.nonzero()
    Return the indices of the elements that are non-zero.
    Refer to numpy.nonzero for full documentation.

See Also:

    numpy.nonzero
    equivalent function

ndarray.compress(condition, axis=None, out=None)
    Return selected slices of this array along given axis.
    Refer to numpy.compress for full documentation.

See Also:

    numpy.compress
    equivalent function

ndarray.diagonal(offset=0, axis1=0, axis2=1)
    Return specified diagonals.
    Refer to numpy.diagonal for full documentation.

See Also:

    numpy.diagonal
    equivalent function

Calculation

Many of these methods take an argument named axis. In such cases,

- If axis is None (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)

- If axis is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

Example of the axis argument

A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

>>> x
array([[[ 0,  1,  2],
    [ 3,  4,  5],
    [ 6,  7,  8]],
([[ 9, 10, 11],
   [12, 13, 14],
   [15, 16, 17]],
([[18, 19, 20],
  [21, 22, 23],
  [24, 25, 26]])])

>>> x.sum(axis=0)
array([[27, 30, 33],
    [36, 39, 42],
    [45, 48, 51]])
>>> # for sum, axis is the first keyword, so we may omit it,
>>> # specifying only its value
>>> x.sum(0), x.sum(1), x.sum(2)
(array([[27, 30, 33],
        [36, 39, 42],
        [45, 48, 51]]),
array([[ 9, 12, 15],
        [36, 39, 42],
        [63, 66, 69]]),
array([[ 3, 12, 21],
        [30, 39, 48],
        [57, 66, 75]]))

The parameter dtype specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of self. To avoid overflow, it can be useful to perform the reduction using a larger data type.

For several methods, an optional out argument can also be provided and the result will be placed into the output array given. The out argument must be an ndarray and have the same number of elements. It can have a different data type in which case casting will be performed.

ndarray.argmax(axis=None, out=None)  # Return indices of the maximum values along the given axis.
ndarray.min(axis=None, out=None)     # Return the minimum along a given axis.
ndarray.argmin(axis=None, out=None)  # Return indices of the minimum values along the given axis of a.
ndarray.ptp(axis=None, out=None)     # Peak to peak (maximum - minimum) value along a given axis.
ndarray.clip(a_min, a_max[, out])    # Return an array whose values are limited to [a_min, a_max].
ndarray.conj()                       # Complex-conjugate all elements.
ndarray.round([decimals, out])      # Return a with each element rounded to the given number of decimals.
ndarray.trace([offset, axis1, axis2, dtype, out])  # Return the sum along diagonals of the array.
ndarray.sum(axis=None, dtype=None[, out])  # Return the sum of the array elements over the given axis.
ndarray.cumsum(axis=None, dtype=None[, out])  # Return the cumulative sum of the elements along the given axis.
ndarray.mean(axis=None, dtype=None[, out])   # Returns the average of the array elements along given axis.
ndarray.var([axis, dtype, ddof])  # Returns the variance of the array elements, along given axis.
ndarray.std([axis, dtype, ddof])  # Returns the standard deviation of the array elements along given axis.
ndarray.prod(axis=None, dtype=None[, out])  # Return the product of the array elements over the given axis.
ndarray.cumprod(axis=None, dtype=None[, out])  # Return the cumulative product of the elements along the given axis.
ndarray.all(axis=None, out=None)       # Returns True if all elements evaluate to True.
ndarray.any(axis=None, out=None)       # Returns True if any of the elements of a evaluate to True.

ndarray.argmax (axis=None, out=None)
    Return indices of the maximum values along the given axis.
    Refer to numpy.argmax for full documentation.

See Also:

    numpy.argmax         equivalent function

ndarray.min (axis=None, out=None)
    Return the minimum along a given axis.
    Refer to numpy.amin for full documentation.
See Also:

```python
numpy.amin
```
equivalent function

```python
ndarray.argmin(axis=None, out=None)
```
Return indices of the minimum values along the given axis of `a`.
Refer to `numpy.argmin` for detailed documentation.

See Also:

```python
numpy.argmin
```
equivalent function

```python
ndarray.ptp(axis=None, out=None)
```
Peak to peak (maximum - minimum) value along a given axis.
Refer to `numpy.ptp` for full documentation.

See Also:

```python
numpy.ptp
```
equivalent function

```python
ndarray.clip(a_min, a_max, out=None)
```
Return an array whose values are limited to `[a_min, a_max]`.
Refer to `numpy.clip` for full documentation.

See Also:

```python
numpy.clip
```
equivalent function

```python
ndarray.conj()
```
Complex-conjugate all elements.
Refer to `numpy.conjugate` for full documentation.

See Also:

```python
numpy.conjugate
```
equivalent function

```python
ndarray.round(decimals=0, out=None)
```
Return `a` with each element rounded to the given number of decimals.
Refer to `numpy.around` for full documentation.

See Also:

```python
numpy.around
```
equivalent function

```python
ndarray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)
```
Return the sum along diagonals of the array.
Refer to `numpy.trace` for full documentation.
See Also:

```
numpy.trace
```
equivalent function

```
ndarray.sum (axis=None, dtype=None, out=None)
```
Return the sum of the array elements over the given axis.
Refer to `numpy.sum` for full documentation.

See Also:

```
numpy.sum
```
equivalent function

```
ndarray.cumsum (axis=None, dtype=None, out=None)
```
Return the cumulative sum of the elements along the given axis.
Refer to `numpy.cumsum` for full documentation.

See Also:

```
numpy.cumsum
```
equivalent function

```
ndarray.mean (axis=None, dtype=None, out=None)
```
Returns the average of the array elements along given axis.
Refer to `numpy.mean` for full documentation.

See Also:

```
numpy.mean
```
equivalent function

```
ndarray.var (axis=None, dtype=None, out=None, ddof=0)
```
Returns the variance of the array elements, along given axis.
Refer to `numpy.var` for full documentation.

See Also:

```
numpy.var
```
equivalent function

```
ndarray.std (axis=None, dtype=None, out=None, ddof=0)
```
Returns the standard deviation of the array elements along given axis.
Refer to `numpy.std` for full documentation.

See Also:

```
numpy.std
```
equivalent function

```
ndarray.prod (axis=None, dtype=None, out=None)
```
Return the product of the array elements over the given axis.
Refer to `numpy.prod` for full documentation.
See Also:

\texttt{numpy.prod}

equivalent function

\texttt{ndarray.cumprod} \texttt{(axis=\text{None}, dtype=\text{None}, out=\text{None})}

Return the cumulative product of the elements along the given axis.

Refer to \texttt{numpy.cumprod} for full documentation.

See Also:

\texttt{numpy.cumprod}
equivalent function

\texttt{ndarray.all} \texttt{(axis=\text{None}, out=\text{None})}

Returns True if all elements evaluate to True.

Refer to \texttt{numpy.all} for full documentation.

See Also:

\texttt{numpy.all}
equivalent function

\texttt{ndarray.any} \texttt{(axis=\text{None}, out=\text{None})}

Returns True if any of the elements of \texttt{a} evaluate to True.

Refer to \texttt{numpy.any} for full documentation.

See Also:

\texttt{numpy.any}
equivalent function

1.1.6 Arithmetic and comparison operations

Arithmetic and comparison operations on \texttt{ndarrays} are defined as element-wise operations, and generally yield \texttt{ndarray} objects as results.

Each of the arithmetic operations (+, −, *, /, //, %, divmod(), ** or \texttt{pow}(), \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{!}) and the comparisons (==, <, >, <=, >=, !=) is equivalent to the corresponding \texttt{universal function} (or \texttt{ufunc} for short) in Numpy. For more information, see the section on \textit{Universal Functions}.

Comparison operators:

\begin{verbatim}
ndarray.__lt__  x__lt__(y) <=> x<y
ndarray.__le__  x__le__(y) <=> x<=y
ndarray.__gt__  x__gt__(y) <=> x>y
ndarray.__ge__  x__ge__(y) <=> x>=y
ndarray.__eq__  x__eq__(y) <=> x==y
ndarray.__ne__  x__ne__(y) <=> x!=y
\end{verbatim}

\begin{verbatim}
ndarray.__lt__
 x__lt__(y) <=> x<y
ndarray.__le__
\end{verbatim}
x.__le__(y) <==> x<=y
ndarray.__gt__
x.__gt__(y) <==> x>y
ndarray.__ge__
x.__ge__(y) <==> x>=y
ndarray.__eq__
x.__eq__(y) <==> x==y
ndarray.__ne__
x.__ne__(y) <==> x!=y

Truth value of an array (bool):

```
ndarray.__nonzero__ x.__nonzero__() <==> x != 0
```

Note: Truth-value testing of an array invokes `ndarray.__nonzero__`, which raises an error if the number of elements in the array is larger than 1, because the truth value of such arrays is ambiguous. Use `.any()` and `.all()` instead to be clear about what is meant in such cases. (If the number of elements is 0, the array evaluates to False.)

Unary operations:

```
ndarray.__neg__ x.__neg__() <==> -x
ndarray.__pos__ x.__pos__() <==> +x
ndarray.__abs__ x.__abs__() <==> abs(x)
ndarray.__invert__ x.__invert__() <==> ~x
```

Arithmetic:

```
ndarray.__add__ x.__add__(y) <==> x+y
ndarray.__sub__ x.__sub__(y) <==> x-y
ndarray.__mul__ x.__mul__(y) <==> x*y
ndarray.__div__ x.__div__(y) <==> x/y
ndarray.__truediv__ x.__truediv__(y) <==> x/y
ndarray.__floordiv__ x.__floordiv__(y) <==> x/y
ndarray.__mod__ x.__mod__(y) <==> x%y
ndarray.__divmod__ x.__divmod__(y) <==> divmod(x, y)
```

Continued on next page
Table 1.15 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndarray.<strong>pow</strong>(y[, z])</td>
<td>equivalent to <code>pow(x, y[, z])</code></td>
</tr>
<tr>
<td>ndarray.<strong>lshift</strong></td>
<td>x.<strong>lshift</strong>(y) equivalent to <code>x&lt;&lt;y</code></td>
</tr>
<tr>
<td>ndarray.<strong>rshift</strong></td>
<td>x.<strong>rshift</strong>(y) equivalent to <code>x&gt;&gt;y</code></td>
</tr>
<tr>
<td>ndarray.<strong>and</strong></td>
<td>x.<strong>and</strong>(y) equivalent to <code>x&amp;y</code></td>
</tr>
<tr>
<td>ndarray.<strong>or</strong></td>
<td>x.<strong>or</strong>(y) equivalent to `x</td>
</tr>
<tr>
<td>ndarray.<strong>xor</strong></td>
<td>x.<strong>xor</strong>(y) equivalent to <code>x^y</code></td>
</tr>
<tr>
<td>ndarray.<strong>add</strong></td>
<td>x.<strong>add</strong>(y) equivalent to <code>x+y</code></td>
</tr>
<tr>
<td>ndarray.<strong>sub</strong></td>
<td>x.<strong>sub</strong>(y) equivalent to <code>x-y</code></td>
</tr>
<tr>
<td>ndarray.<strong>mul</strong></td>
<td>x.<strong>mul</strong>(y) equivalent to <code>x*y</code></td>
</tr>
<tr>
<td>ndarray.<strong>div</strong></td>
<td>x.<strong>div</strong>(y) equivalent to <code>x/y</code></td>
</tr>
<tr>
<td>ndarray.<strong>truediv</strong></td>
<td>x.<strong>truediv</strong>(y) equivalent to <code>x/y</code></td>
</tr>
<tr>
<td>ndarray.<strong>floordiv</strong></td>
<td>x.<strong>floordiv</strong>(y) equivalent to <code>x//y</code></td>
</tr>
<tr>
<td>ndarray.<strong>mod</strong></td>
<td>x.<strong>mod</strong>(y) equivalent to <code>x%y</code></td>
</tr>
<tr>
<td>ndarray.<strong>divmod</strong></td>
<td>divmod(x, y) equivalent to <code>divmod(x, y)</code></td>
</tr>
<tr>
<td>ndarray.<strong>pow</strong>(y[, z])</td>
<td>equivalent to <code>pow(x, y[, z])</code></td>
</tr>
</tbody>
</table>

Note:

- Any third argument to `pow` is silently ignored, as the underlying `ufunc` takes only two arguments.
- The three division operators are all defined; `div` is active by default, `truediv` is active when `__future__` division is in effect.
- Because `ndarray` is a built-in type (written in C), the `__r{op}` special methods are not directly defined.
- The functions called to implement many arithmetic special methods for arrays can be modified using `set_numeric_ops`.

1.1. The N-dimensional array (`ndarray`)
NumPy Reference, Release 1.8.1

Arithmetic, in-place:

```python
ndarray.__iadd__
x.__iadd__(y) ==> x+=y

ndarray.__isub__
x.__isub__(y) ==> x-=y

ndarray.__imul__
x.__imul__(y) ==> x*=y

ndarray.__idiv__
x.__idiv__(y) ==> x/=y

ndarray.__itruediv__
x.__itruediv__(y) ==> x/y

ndarray.__ifloordiv__
x.__ifloordiv__(y) ==> x//=y

ndarray.__imod__
x.__imod__(y) ==> x%=y

ndarray.__ipow__
x.__ipow__(y) ==> x**=y

ndarray.__ilshift__
x.__ilshift__(y) ==> x<<=y

ndarray.__irshift__
x.__irshift__(y) ==> x>>=y

ndarray.__iand__
x.__iand__(y) ==> x&=y

ndarray.__ior__
x.__ior__(y) ==> x|=y

ndarray.__ixor__
x.__ixor__(y) ==> x^=y
```

Chapter 1. Array objects
Warning: In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, \( A \ (\text{op}) = B \) can be different than \( A = A \ (\text{op}) \ B \). For example, suppose \( a = \text{ones}((3, 3)) \). Then, \( a += 3j \) is different than \( a = a + 3j \): while they both perform the same computation, \( a += 3 \) casts the result to fit back in \( a \), whereas \( a = a + 3j \) rebinds the name \( a \) to the result.

### 1.1.7 Special methods

For standard library functions:

```python
ndarray.__copy__([order])
Return a copy of the array.
```

```python
ndarray.__deepcopy__()
-> Deep copy of array.
Used if copy.deepcopy is called on an array.
```

```python
ndarray.__reduce__()
For pickling.
```

```python
ndarray.__setstate__(version, shape, dtype, ...)
For unpickling.
```

#### ndarray.__copy__([order])
Return a copy of the array.

**Parameters**

- **order**: {'C', 'F', 'A'}, optional
  - If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

#### ndarray.__deepcopy__()
-> Deep copy of array.

- Used if copy.deepcopy is called on an array.

#### ndarray.__reduce__()
For pickling.

#### ndarray.__setstate__(version, shape, dtype, isfortran, rawdata)
For unpickling.

**Parameters**

- **version**: int
  - optional pickle version. If omitted defaults to 0.
- **shape**: tuple
- **dtype**: data-type
- **isFortran**: bool
- **rawdata**: string or list
  - a binary string with the data (or a list if ‘a’ is an object array)

Basic customization:

```python
ndarray.__new__((S, ...))
ndarray.__array__(...)
Returns either a new reference to self if dtype is not given or a new array
```

```python
ndarray.__array_wrap__(...)
```

**static** `ndarray.__new__((S, ...))` → a new object with type S, a subtype of T
ndarray.__array__(dtype) → reference if type unchanged, copy otherwise.
Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

ndarray.__array_wrap__(obj) → Object of same type as ndarray object a.

Container customization: (see Indexing)

```
ndarray.__len__() <==> len(x)
ndarray.__getitem__ x.__getitem__(y) <==> x[y]
ndarray.__setitem__ x.__setitem__(i, y) <==> x[i]=y
ndarray.__getslice__ x.__getslice__(i, j) <==> x[i:j]
ndarray.__setslice__ x.__setslice__(i, j, y) <==> x[i:j]=y
ndarray.__contains__ x.__contains__(y) <==> y in x
```

Conversion; the operations `complex`, `int`, `long`, `float`, `oct`, and `hex`. They work only on arrays that have one element in them and return the appropriate scalar.

```
ndarray.__int__() <==> int(x)
ndarray.__long__() <==> long(x)
ndarray.__float__() <==> float(x)
ndarray.__oct__() <==> oct(x)
ndarray.__hex__() <==> hex(x)
```
ndarray.__hex__() <=> hex(x)

String representations:

\[
\begin{array}{c}
\text{ndarray.__str__()} \leftrightarrow \text{str(x)} \\
\text{ndarray.__repr__()} \leftrightarrow \text{repr(x)}
\end{array}
\]

ndarray.__str__() <=> str(x)

ndarray.__repr__() <=> repr(x)

## 1.2 Scalars

Python defines only one type of a particular data class (there is only one integer type, one floating-point type, etc.). This can be convenient in applications that don’t need to be concerned with all the ways data can be represented in a computer. For scientific computing, however, more control is often needed.

In NumPy, there are 24 new fundamental Python types to describe different types of scalars. These type descriptors are mostly based on the types available in the C language that CPython is written in, with several additional types compatible with Python’s types.

Array scalars have the same attributes and methods as `ndarrays`. This allows one to treat items of an array partly on the same footing as arrays, smoothing out rough edges that result when mixing scalar and array operations.

Array scalars live in a hierarchy (see the Figure below) of data types. They can be detected using the hierarchy: `isinstance(val, np.generic)` will return `True` if `val` is an array scalar object. Alternatively, what kind of array scalar is present can be determined using other members of the data type hierarchy. Thus, for example `isinstance(val, np.complexfloating)` will return `True` if `val` is a complex valued type, while `isinstance(val, np.flexible)` will return true if `val` is one of the flexible itemsize array types (string, unicode, void).

### 1.2.1 Built-in scalar types

The built-in scalar types are shown below. Along with their (mostly) C-derived names, the integer, float, and complex data-types are also available using a bit-width convention so that an array of the right size can always be ensured (e.g. `int8`, `float64`, `complex128`). Two aliases (`intp` and `uintp`) pointing to the integer type that is sufficiently large to hold a C pointer are also provided. The C-like names are associated with character codes, which are shown in the table. Use of the character codes, however, is discouraged.

Some of the scalar types are essentially equivalent to fundamental Python types and therefore inherit from them as well as from the generic array scalar type:

<table>
<thead>
<tr>
<th>Array scalar type</th>
<th>Related Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_</td>
<td>IntType (Python 2 only)</td>
</tr>
<tr>
<td>float_</td>
<td>FloatType</td>
</tr>
<tr>
<td>complex_</td>
<td>ComplexType</td>
</tr>
<tr>
<td>str_</td>
<td>StringType</td>
</tr>
<tr>
<td>unicode_</td>
<td>UnicodeType</td>
</tr>
</tbody>
</table>

1 However, array scalars are immutable, so none of the array scalar attributes are settable.
Figure 1.2: **Figure**: Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.
The boolean data type is very similar to the Python BooleanType but does not inherit from it because Python’s BooleanType does not allow itself to be inherited from, and on the C-level the size of the actual bool data is not the same as a Python Boolean scalar.

**Warning:** The bool type is not a subclass of the int type (the bool is not even a number type). This is different than Python’s default implementation of bool as a sub-class of int.

**Warning:** The int type does not inherit from the int built-in under Python 3, because type int is no longer a fixed-width integer type.

**Tip:** The default data type in Numpy is float_.

In the tables below, platform? means that the type may not be available on all platforms. Compatibility with different C or Python types is indicated: two types are compatible if their data is of the same size and interpreted in the same way.

### Booleans:

<table>
<thead>
<tr>
<th>Type</th>
<th>Remarks</th>
<th>Character code</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>compatible: Python bool</td>
<td>'?'</td>
</tr>
<tr>
<td>bool8</td>
<td>8 bits</td>
<td></td>
</tr>
</tbody>
</table>

### Integers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Remarks</th>
<th>Character code</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>compatible: C char</td>
<td>'b'</td>
</tr>
<tr>
<td>short</td>
<td>compatible: C short</td>
<td>'h'</td>
</tr>
<tr>
<td>intc</td>
<td>compatible: C int</td>
<td>'i'</td>
</tr>
<tr>
<td>int</td>
<td>compatible: Python int</td>
<td>'l'</td>
</tr>
<tr>
<td>longlong</td>
<td>compatible: C long long</td>
<td>'q'</td>
</tr>
<tr>
<td>intp</td>
<td>large enough to fit a pointer</td>
<td>'p'</td>
</tr>
<tr>
<td>int8</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>int16</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>int32</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>int64</td>
<td>64 bits</td>
<td></td>
</tr>
</tbody>
</table>

### Unsigned integers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Remarks</th>
<th>Character code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubyte</td>
<td>compatible: C unsigned char</td>
<td>'B'</td>
</tr>
<tr>
<td>ushort</td>
<td>compatible: C unsigned short</td>
<td>'H'</td>
</tr>
<tr>
<td>uintc</td>
<td>compatible: C unsigned int</td>
<td>'I'</td>
</tr>
<tr>
<td>uint</td>
<td>compatible: Python int</td>
<td>'L'</td>
</tr>
<tr>
<td>ulonglong</td>
<td>compatible: C long long</td>
<td>'Q'</td>
</tr>
<tr>
<td>uintp</td>
<td>large enough to fit a pointer</td>
<td>'P'</td>
</tr>
<tr>
<td>uint8</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>uint16</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>uint32</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>uint64</td>
<td>64 bits</td>
<td></td>
</tr>
</tbody>
</table>

### Floating-point numbers:
### Complex floating-point numbers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csingle</td>
<td>'F'</td>
<td>compatible: Python complex</td>
</tr>
<tr>
<td>complex_</td>
<td>'D'</td>
<td>compatible: Python complex</td>
</tr>
<tr>
<td>clongf仪表</td>
<td>'G'</td>
<td>compatible: Python long float</td>
</tr>
<tr>
<td>complex64</td>
<td></td>
<td>two 32-bit floats</td>
</tr>
<tr>
<td>complex128</td>
<td></td>
<td>two 64-bit floats</td>
</tr>
<tr>
<td>complex192</td>
<td></td>
<td>two 96-bit floats, platform?</td>
</tr>
<tr>
<td>complex256</td>
<td></td>
<td>two 128-bit floats, platform?</td>
</tr>
</tbody>
</table>

Any Python object:

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_</td>
<td>any Python object</td>
</tr>
</tbody>
</table>

#### Note:

The data actually stored in object arrays (i.e., arrays having dtype object_) are references to Python objects, not the objects themselves. Hence, object arrays behave more like usual Python lists, in the sense that their contents need not be of the same Python type.

The object type is also special because an array containing object_ items does not return an object_ object on item access, but instead returns the actual object that the array item refers to.

The following data types are flexible. They have no predefined size: the data they describe can be of different length in different arrays. (In the character codes # is an integer denoting how many elements the data type consists of.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str_</td>
<td>'S#'</td>
<td>compatible: Python str</td>
</tr>
<tr>
<td>unicode_</td>
<td>'U#'</td>
<td>compatible: Python unicode</td>
</tr>
<tr>
<td>void</td>
<td>'V#'</td>
<td></td>
</tr>
</tbody>
</table>

#### Warning:

Numeric Compatibility: If you used old typecode characters in your Numeric code (which was never recommended), you will need to change some of them to the new characters. In particular, the needed changes are c -> S1, b -> B, l -> b, s -> h, w -> H, and u -> I. These changes make the type character convention more consistent with other Python modules such as the struct module.

### 1.2.2 Attributes

The array scalar objects have an array priority of NPY_SCALAR_PRIORITY (-1,000,000.0). They also do not (yet) have a ctypes attribute. Otherwise, they share the same attributes as arrays:

- generic.flags: integer value of flags
- generic.shape: tuple of array dimensions
- generic.strides: tuple of bytes steps in each dimension
- generic.ndim: number of array dimensions
Table 1.22 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic.data</td>
<td>pointer to start of data</td>
</tr>
<tr>
<td>generic.size</td>
<td>number of elements in the gentype</td>
</tr>
<tr>
<td>generic.itemsize</td>
<td>length of one element in bytes</td>
</tr>
<tr>
<td>generic.base</td>
<td>base object</td>
</tr>
<tr>
<td>generic.dtype</td>
<td>get array data-descriptor</td>
</tr>
<tr>
<td>generic.real</td>
<td>real part of scalar</td>
</tr>
<tr>
<td>generic.imag</td>
<td>imaginary part of scalar</td>
</tr>
<tr>
<td>generic.flat</td>
<td>a 1-d view of scalar</td>
</tr>
<tr>
<td>generic.T</td>
<td>transpose</td>
</tr>
<tr>
<td>generic.<strong>array_interface</strong></td>
<td>Array protocol: Python side</td>
</tr>
<tr>
<td>generic.<strong>array_struct</strong></td>
<td>Array protocol: struct</td>
</tr>
<tr>
<td>generic.<strong>array_priority</strong></td>
<td>Array priority.</td>
</tr>
<tr>
<td>generic.<strong>array_wrap</strong></td>
<td>sc.<strong>array_wrap</strong>(obj) return scalar from array</td>
</tr>
</tbody>
</table>

generic.flags
integer value of flags

generic.shape
tuple of array dimensions

generic.strides
tuple of bytes steps in each dimension

generic.ndim
number of array dimensions

generic.data
pointer to start of data

generic.size
number of elements in the gentype

generic.itemsize
length of one element in bytes

generic.base
base object

generic.dtype
get array data-descriptor

generic.real
real part of scalar

generic.imag
imaginary part of scalar

generic.flat
a 1-d view of scalar

generic.T
transpose

generic.__array_interface__
Array protocol: Python side

generic.__array_struct__
Array protocol: struct

generic.__array_priority__
Array priority.

generic.__array_wrap__
sc.__array_wrap__(obj) return scalar from array
generic.__array_priority__
    Array priority.

generic.__array_wrap__(obj)
    sc.__array_wrap__(obj) return scalar from array

1.2.3 Indexing

See Also:

Indexing, Data type objects (dtype)

Array scalars can be indexed like 0-dimensional arrays: if \( x \) is an array scalar,
- \( x[()] \) returns a 0-dimensional ndarray
- \( x[\text{\textquoteleft\textfield-name\textquoteleft}] \) returns the array scalar in the field field-name. (\( x \) can have fields, for example, when it corresponds to a record data type.)

1.2.4 Methods

Array scalars have exactly the same methods as arrays. The default behavior of these methods is to internally convert the scalar to an equivalent 0-dimensional array and to call the corresponding array method. In addition, math operations on array scalars are defined so that the same hardware flags are set and used to interpret the results as for ufunc, so that the error state used for ufuncs also carries over to the math on array scalars.

The exceptions to the above rules are given below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic</td>
<td>Base class for numpy scalar types.</td>
</tr>
<tr>
<td>generic.<strong>array</strong></td>
<td>sc.<strong>array</strong>(type) return 0-dim array</td>
</tr>
<tr>
<td>generic.<strong>array_wrap</strong></td>
<td>sc.<strong>array_wrap</strong>(obj) return scalar from array</td>
</tr>
<tr>
<td>generic.squeeze</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>generic.byteswap</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>generic.<strong>reduce</strong></td>
<td></td>
</tr>
<tr>
<td>generic.setstate__</td>
<td></td>
</tr>
<tr>
<td>generic.setflags</td>
<td>Not implemented (virtual attribute)</td>
</tr>
</tbody>
</table>

class numpy.generic
    Base class for numpy scalar types.

Class from which most (all?) numpy scalar types are derived. For consistency, exposes the same API as ndarray, despite many consequent attributes being either “get-only,” or completely irrelevant. This is the class from which it is strongly suggested users should derive custom scalar types.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>transpose</td>
</tr>
<tr>
<td>base</td>
<td>base object</td>
</tr>
<tr>
<td>data</td>
<td>pointer to start of data</td>
</tr>
<tr>
<td>dtype</td>
<td>get array data-descriptor</td>
</tr>
<tr>
<td>flags</td>
<td>integer value of flags</td>
</tr>
<tr>
<td>flat</td>
<td>a 1-d view of scalar</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary part of scalar</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1.24 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemsize</td>
<td>length of one element in bytes</td>
</tr>
<tr>
<td>nbytes</td>
<td>length of item in bytes</td>
</tr>
<tr>
<td>ndim</td>
<td>number of array dimensions</td>
</tr>
<tr>
<td>real</td>
<td>real part of scalar</td>
</tr>
<tr>
<td>shape</td>
<td>tuple of array dimensions</td>
</tr>
<tr>
<td>size</td>
<td>number of elements in the gentype</td>
</tr>
<tr>
<td>strides</td>
<td>tuple of bytes steps in each dimension</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>any</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argmax</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argmin</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argsort</td>
<td>Not implemented (virtual attribute)</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1.25 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>astype</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>byteswap</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>choose</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>clip</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>compress</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>conj</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>conjugate</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>copy</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>cumprod</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>cumsum</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>diagonal</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>dump</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>dumps</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>fill</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>flatten</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>getfield</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>item</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>itemset</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>max</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>mean</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>min</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>newbyteorder</td>
<td>Return a new dtype with a different byte order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>prod</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>ptp</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>put</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>ravel</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>repeat</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>reshape</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>resize</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>round</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>searchsorted</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>setfield</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>setflags</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>sort</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>squeeze</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>std</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>sum</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>swapaxes</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>take</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tofile</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tolist</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tostring</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>trace</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>transpose</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>var</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>view</td>
<td>Not implemented (virtual attribute)</td>
</tr>
</tbody>
</table>

generic.all()   | Not implemented (virtual attribute)                                        |
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.any()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.argmax()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.argmin()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.argsort()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.astype()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.byteswap()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.choose()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.clip()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.compress()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.conj()

generic.conjugate()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.copy()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.cumprod()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the
   attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.cumsum()
   Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.diagonal()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.dump()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.dumps()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.fill()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.flatten()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

`generic.getfield()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The
generic.item()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.itemset()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.max()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.mean()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.min()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

generic.newbyteorder(new_order='S')
Return a new dtype with a different byte order.
Changes are also made in all fields and sub-arrays of the data type.
The new_order code can be any from the following:
*{'<', 'L'} - little endian
*{'>', 'B'} - big endian
*{'=', 'N'} - native order
*{'S'} - swap dtype from current to opposite endian
*{'I', 'I'} - ignore (no change to byte order)
Parameters

\texttt{new\_order} : str, optional

Byte order to force; a value from the byte order specifications above. The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of \texttt{new\_order} for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

Returns

\texttt{new\_dtype} : dtype

New dtype object with the given change to the byte order.

generic.\texttt{nonzero}()

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

generic.\texttt{prod}()

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

generic.\texttt{ptp}()

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

generic.\texttt{put}()

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

generic.\texttt{ravel}()

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

generic.\texttt{repeat}()

Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The

generic.**reshape**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.**resize**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.**round**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.**searchsorted**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.**setfield**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.**setflags**()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The
generic.sort()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.squeeze()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.std()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.sum()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.swapaxes()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.take()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
   The

generic.tofile()
   Not implemented (virtual attribute)
   Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
   See Also:
The

```
generic.tolist()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic.tostring()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic.trace()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic.transpose()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic.var()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic.view()
```
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

```
generic._array__()
```
sc._array__(type) return 0-dim array

```
generic._array_wrap__()
```
sc._array_wrap__(obj) return scalar from array
general.squeeze()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.byteswap()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
generic.__reduce__()

generic.__setstate__()

generic.setflags()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

1.2.5 Defining new types

There are two ways to effectively define a new array scalar type (apart from composing record dtypes from the built-in scalar types): One way is to simply subclass the ndarray and overwrite the methods of interest. This will work to a degree, but internally certain behaviors are fixed by the data type of the array. To fully customize the data type of an array you need to define a new data-type, and register it with NumPy. Such new types can only be defined in C, using the Numpy C-API.

1.3 Data type objects (dtype)

A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in e.g. the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is a record, an aggregate of other data types, (e.g., describing an array item consisting of an integer and a float),
   (a) what are the names of the “fields” of the record, by which they can be accessed,
   (b) what is the data-type of each field, and
To describe the type of scalar data, there are several built-in scalar types in Numpy for various precision of integers, floating-point numbers, etc. An item extracted from an array, e.g., by indexing, will be a Python object whose type is the scalar type associated with the data type of the array.

Note that the scalar types are not dtype objects, even though they can be used in place of one whenever a data type specification is needed in Numpy.

Struct data types are formed by creating a data type whose fields contain other data types. Each field has a name by which it can be accessed. The parent data type should be of sufficient size to contain all its fields; the parent is nearly always based on the void type which allows an arbitrary item size. Struct data types may also contain nested struct sub-array data types in their fields.

Finally, a data type can describe items that are themselves arrays of items of another data type. These sub-arrays must, however, be of a fixed size.

If an array is created using a data-type describing a sub-array, the dimensions of the sub-array are appended to the shape of the array when the array is created. Sub-arrays in a field of a record behave differently, see Record Access.

Sub-arrays always have a C-contiguous memory layout.

Example

A simple data type containing a 32-bit big-endian integer: (see Specifying and constructing data types for details on construction)

```python
>>> dt = np.dtype('>i4')
>>> dt.byteorder
'>'
>>> dt.itemsize
4
>>> dt.name
'int32'
>>> dt.type is np.int32
True
```

The corresponding array scalar type is int32.

Example

A record data type containing a 16-character string (in field ‘name’) and a sub-array of two 64-bit floating-point number (in field ‘grades’):

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt['name']
dtype('|S16')
>>> dt['grades']
dtype(('float64',(2,)))
```

Items of an array of this data type are wrapped in an array scalar type that also has two fields:

```python
>>> x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
>>> x[1]
('John', [6.0, 7.0])
>>> x[1][‘grades’]
array([ 6., 7.])
>>> type(x[1])
```

1.3.1 Specifying and constructing data types

Whenever a data-type is required in a NumPy function or method, either a dtype object or something that can be converted to one can be supplied. Such conversions are done by the dtype constructor:

```python
dtype  # Create a data type object.
```

**class numpy.dtype**

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

**Parameters**

- **obj**: bool, optional
  Object to be converted to a data type object.

- **align**: bool, optional
  Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be True only if obj is a dictionary or a comma-separated string. If a struct dtype is being created, this also sets a sticky alignment flag isalignedstruct.

- **copy**: bool, optional
  Make a new copy of the data-type object. If False, the result may just be a reference to a built-in data-type object.

**See Also:**

result_type

**Examples**

Using array-scalar type:

```python
>>> np.dtype(np.int16)
dtype('int16')
```

Record, one field name ‘f1’, containing int16:

```python
>>> np.dtype({'f1', np.int16})
dtype([('f1', '<i2')])
```

Record, one field named ‘f1’, in itself containing a record with one field:

```python
>>> np.dtype([('f1', [{'f1', np.int16}])])
dtype([('f1', [('f1', '<i2')])])
```

Record, two fields: the first field contains an unsigned int, the second an int32:

```python
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```
Using array-protocol type strings:

```python
>>> np.dtype([('a', 'f8'), ('b', 'S10')])
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```python
>>> np.dtype([('i4', (2, 3)), ('hello', (np.int, 3)), ('world', np.void, 10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide int16 into 2 int8's, called x and y. 0 and 1 are the offsets in bytes:

```python
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype(('i2', [('x', '|i1'), ('y', '|i1')])
```

Using dictionaries. Two fields named ‘gender’ and ‘age’:

```python
>>> np.dtype({'names':['gender','age'], 'formats':['S1',np.uint8]})
dtype([('gender', '|S1'), ('age', '|u1')])
```

Offsets in bytes, here 0 and 25:

```python
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Array-interface compliant full description of the data-type.</td>
</tr>
<tr>
<td>descr</td>
<td>Dictionary of named fields defined for this data type, or None.</td>
</tr>
<tr>
<td>fields</td>
<td>Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.</td>
</tr>
<tr>
<td>hasobject</td>
<td>Boolean indicating whether the dtype is a struct which maintains field alignment.</td>
</tr>
<tr>
<td>isalignedstruct</td>
<td>Integer indicating how this dtype relates to the built-in dtypes.</td>
</tr>
<tr>
<td>isbuiltin</td>
<td>Boolean indicating whether the byte order of this dtype is native</td>
</tr>
<tr>
<td>metadata</td>
<td>A bit-width name for this data-type.</td>
</tr>
<tr>
<td>name</td>
<td>Ordered list of field names, or None if there are no fields.</td>
</tr>
<tr>
<td>names</td>
<td>Shape tuple of the sub-array if this data type describes a sub-array,</td>
</tr>
<tr>
<td>str</td>
<td>The array-protocol typestring of this data-type object.</td>
</tr>
<tr>
<td>subdtype</td>
<td>Tuple (item_dtype, shape) if this dtype describes a sub-array, and</td>
</tr>
<tr>
<td></td>
<td>dtype.base</td>
</tr>
<tr>
<td>dtype.descr</td>
<td>Array-interface compliant full description of the data-type.</td>
</tr>
<tr>
<td></td>
<td>The format is that required by the ‘descr’ key in the <strong>array_interface</strong> attribute.</td>
</tr>
<tr>
<td>dtype.fields</td>
<td>Dictionary of named fields defined for this data type, or None.</td>
</tr>
<tr>
<td></td>
<td>The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple</td>
</tr>
</tbody>
</table>
fully describing the field:

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it’s meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the `ndarray.getfield` and `ndarray.setfield` methods.

**See Also:**

`ndarray.getfield`, `ndarray.setfield`

**Examples**

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print dt.fields
{'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

`dtype.hasobject`

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the `ndarray` memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won’t.

`dtype.isalignedstruct`

Boolean indicating whether the dtype is a struct which maintains field alignment. This flag is sticky, so when combining multiple structs together, it is preserved and produces new dtypes which are also aligned.

`dtype.isbuiltin`

Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if this is a structured array type, with fields</td>
</tr>
<tr>
<td>1</td>
<td>if this is a dtype compiled into numpy (such as ints, floats etc)</td>
</tr>
<tr>
<td>2</td>
<td>if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See <code>user.user-defined-data-types</code> in the Numpy manual</td>
</tr>
</tbody>
</table>

**Examples**

```python
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

`dtype.isnative`

Boolean indicating whether the byte order of this dtype is native to the platform.

`dtype.metadata`

`dtype.name`

A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.
**dtype.names**
Ordered list of field names, or None if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

**Examples**

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

**dtype.shape**
Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.

**dtype.str**
The array-protocol typestring of this data-type object.

**dtype.subdtype**
Tuple (item_dtype, shape) if this dtype describes a sub-array, and None otherwise.

The shape is the fixed shape of the sub-array described by this data type, and item_dtype the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by shape are tacked on to the end of the retrieved array.

**Methods**

```python
newbyteorder([new_order]) Return a new dtype with a different byte order.
```

**dtype.newbyteorder** (**new_order**='S')
Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

**Parameters**

- **new_order** : string, optional

  Byte order to force; a value from the byte order specifications below. The default value ('S') results in swapping the current byte order. new_order codes can be any of:

  - 'S' - swap dtype from current to opposite endian
  - {'<', 'L'} - little endian
  - {'>', 'B'} - big endian
  - {'=', 'N'} - native order
  - {'|', 'I'} - ignore (no change to byte order)

  The code does a case-insensitive check on the first letter of new_order for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

**Returns**

- **new_dtype** : dtype

  New dtype object with the given change to the byte order.

**Notes**

Changes are also made in all fields and sub-arrays of the data type.
Examples

```python
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>
>>> swapped_code = sys_is_le and '>' or '<
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

What can be converted to a data-type object is described below:

**dtype object**

Used as-is.

None

The default data type: `float_`.

Array-scalar types

The 24 built-in *array scalar type objects* all convert to an associated data-type object. This is true for their sub-classes as well.

Note that not all data-type information can be supplied with a type-object: for example, *flexible* data-types have a default *itemsize* of 0, and require an explicitly given size to be useful.

---

**Example**

```python
>>> dt = np.dtype(np.int32)  # 32-bit integer
>>> dt = np.dtype(np.complex128)  # 128-bit complex floating-point number
```

Generic types

The generic hierarchical type objects convert to corresponding type objects according to the associations:
Built-in Python types

Several python types are equivalent to a corresponding array scalar when used to generate a `dtype` object:

<table>
<thead>
<tr>
<th>Python Type</th>
<th>Corresponding Array Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>int_</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>bool_</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>float_</code></td>
</tr>
<tr>
<td><code>complex</code></td>
<td><code>cfloat</code></td>
</tr>
<tr>
<td><code>str</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td><code>unicode</code></td>
<td><code>unicode_</code></td>
</tr>
<tr>
<td><code>buffer</code></td>
<td><code>void</code></td>
</tr>
<tr>
<td>(all others)</td>
<td><code>object_</code></td>
</tr>
</tbody>
</table>

Example

```python
>>> dt = np.dtype(float)  # Python-compatible floating-point number
>>> dt = np.dtype(int)    # Python-compatible integer
>>> dt = np.dtype(object) # Python object
```

Types with `.dtype`

Any type object with a `dtype` attribute: The attribute will be accessed and used directly. The attribute must return something that is convertible into a dtype object.

Several kinds of strings can be converted. Recognized strings can be prepended with `'>'` (big-endian), `'<` (little-endian), or `='` (hardware-native, the default), to specify the byte order.

One-character strings

Each built-in data-type has a character code (the updated Numeric typecodes), that uniquely identifies it.

Example

```python
>>> dt = np.dtype('b')    # byte, native byte order
>>> dt = np.dtype('>H')   # big-endian unsigned short
>>> dt = np.dtype('<f')   # little-endian single-precision float
>>> dt = np.dtype('d')    # double-precision floating-point number
```

Array-protocol type strings (see *The Array Interface*)

The first character specifies the kind of data and the remaining characters specify how many bytes of data.

The supported kinds are

<table>
<thead>
<tr>
<th>Character</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>Boolean</td>
</tr>
<tr>
<td>'i'</td>
<td>(signed) integer</td>
</tr>
<tr>
<td>'u'</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>'f'</td>
<td>floating-point</td>
</tr>
<tr>
<td>'c'</td>
<td>complex-floating point</td>
</tr>
<tr>
<td>'S', 'a'</td>
<td>string</td>
</tr>
<tr>
<td>'U'</td>
<td>unicode</td>
</tr>
<tr>
<td>'V'</td>
<td>raw data (void)</td>
</tr>
</tbody>
</table>
String with comma-separated fields

Numarray introduced a short-hand notation for specifying the format of a record as a comma-separated string of basic formats.

A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it has more than one dimension. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. The generated data-type fields are named ‘f0’, ‘f1’, ..., ‘f<N-1>’ where N (>1) is the number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field describes a sub-array.

Example

- field named f0 containing a 32-bit integer
- field named f1 containing a 2 x 3 sub-array of 64-bit floating-point numbers
- field named f2 containing a 32-bit floating-point number

```python
>>> dt = np.dtype("i4, (2,3)f8, f4")
```

- field named f0 containing a 3-character string
- field named f1 containing a sub-array of shape (3,) containing 64-bit unsigned integers
- field named f2 containing a 3 x 4 sub-array containing 10-character strings

```python
>>> dt = np.dtype("a3, 3u8, (3,4)a10")
```

Type strings

Any string in `numpy.sctypeDict.keys()`:

Example

```python
>>> dt = np.dtype('uint32')  # 32-bit unsigned integer
>>> dt = np.dtype('Float64') # 64-bit floating-point number
```

(flexible_dtype, itemsize)

The first argument must be an object that is converted to a zero-sized flexible data-type object, the second argument is an integer providing the desired itemsize.

Example

```python
>>> dt = np.dtype((void, 10)) # 10-byte wide data block
>>> dt = np.dtype((str, 35))  # 35-character string
>>> dt = np.dtype(('U', 10))  # 10-character unicode string
```
(fixed_dtype, shape)

The first argument is any object that can be converted into a fixed-size data-type object. The second
argument is the desired shape of this type. If the shape parameter is 1, then the data-type object is
equivalent to fixed dtype. If shape is a tuple, then the new dtype defines a sub-array of the given shape.

Example

```python
>>> dt = np.dtype((np.int32, (2,2)))  # 2 x 2 integer sub-array
>>> dt = np.dtype(('S10', 1))         # 10-character string
>>> dt = np.dtype(('i4, (2,3)f8, f4', (2,3)))  # 2 x 3 record sub-array
```

([(field_name, field_dtype, field_shape), ...]

obj should be a list of fields where each field is described by a tuple of length 2 or 3. (Equivalent to the
descr item in the __array_interface__ attribute.)

The first element, field_name, is the field name (if this is " then a standard field name, 'f#', is assigned).
The field name may also be a 2-tuple of strings where the first string is either a “title” (which may be any
string or unicode string) or meta-data for the field which can be any object, and the second string is the
“name” which must be a valid Python identifier.

The second element, field_dtype, can be anything that can be interpreted as a data-type.

The optional third element field_shape contains the shape if this field represents an array of the data-type
in the second element. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.

This style does not accept align in the dtype constructor as it is assumed that all of the memory is
accounted for by the array interface description.

Example

Data-type with fields big (big-endian 32-bit integer) and little (little-endian 32-bit integer):

```python
>>> dt = np.dtype([('big', '>i4'), ('little', '<i4')])
```

Data-type with fields R, G, B, A, each being an unsigned 8-bit integer:

```python
>>> dt = np.dtype([('R','u1'), ('G','u1'), ('B','u1'), ('A','u1')])
```

({'names': ..., 'formats': ..., 'offsets': ..., 'titles': ..., 'itemsize': ...

This style has two required and three optional keys. The names and formats keys are required. Their
respective values are equal-length lists with the field names and the field formats. The field names must
be strings and the field formats can be any object accepted by dtype constructor.

When the optional keys offsets and titles are provided, their values must each be lists of the same length
as the names and formats lists. The offsets value is a list of byte offsets (integers) for each field, while the
titles value is a list of titles for each field (None can be used if no title is desired for that field). The titles
can be any string or unicode object and will add another entry to the fields dictionary keyed by the
title and referencing the same field tuple which will contain the title as an additional tuple member.

The itemsize key allows the total size of the dtype to be set, and must be an integer large enough so all the
fields are within the dtype. If the dtype being constructed is aligned, the itemsize must also be divisible
by the struct alignment.

Example
Data type with fields r, g, b, a, each being a 8-bit unsigned integer:

```python
>>> dt = np.dtype({'names': ['r', 'g', 'b', 'a'],
     ... 'formats': [uint8, uint8, uint8, uint8]})
```

Data type with fields r and b (with the given titles), both being 8-bit unsigned integers, the first at byte position 0 from the start of the field and the second at position 2:

```python
>>> dt = np.dtype({'names': ['r', 'b'], 'formats': ['u1', 'u1'],
     ... 'offsets': [0, 2],
     ... 'titles': ['Red pixel', 'Blue pixel']})
```

This usage is discouraged, because it is ambiguous with the other dict-based construction method. If you have a field called ‘names’ and a field called ‘formats’ there will be a conflict.

This style allows passing in the fields attribute of a data-type object. 

`obj` should contain string or unicode keys that refer to (data-type, offset) or (data-type, offset, title) tuples.

**Example**

Data type containing field col1 (10-character string at byte position 0), col2 (32-bit float at byte position 10), and col3 (integers at byte position 14):

```python
>>> dt = np.dtype({'col1': ('S10', 0), 'col2': (float32, 10),
     ... 'col3': (int, 14)})
```

1.3.2 `dtype`

Numpy data type descriptions are instances of the `dtype` class.
Attributes

The type of the data is described by the following `dtype` attributes:

| `dtype.type` | The type object used to instantiate a scalar of this data-type. |
| `dtype.kind` | A character code (one of ‘biufcSUV’) identifying the general kind of data. |
| `dtype.char` | A unique character code for each of the 21 different built-in types. |
| `dtype.num` | A unique number for each of the 21 different built-in types. |
| `dtype.str` | The array-protocol typestring of this data-type object. |

- `dtype.type`:
The type object used to instantiate a scalar of this data-type.

- `dtype.kind`:
A character code (one of ‘biufcSUV’) identifying the general kind of data.

- `dtype.char`:
A unique character code for each of the 21 different built-in types.

- `dtype.num`:
A unique number for each of the 21 different built-in types.

These are roughly ordered from least-to-most precision.

- `dtype.str`:
The array-protocol typestring of this data-type object.

Size of the data is in turn described by:

| `dtype.name` | A bit-width name for this data-type. |
| `dtype.itemsize` | The element size of this data-type object. |

- `dtype.name`:
A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

- `dtype.itemsize`:
The element size of this data-type object.

For 18 of the 21 types this number is fixed by the data-type. For the flexible data-types, this number can be anything.

Endianness of this data:

| `dtype.byteorder` | A character indicating the byte-order of this data-type object. |

- `dtype.byteorder`:
A character indicating the byte-order of this data-type object.

One of:

<table>
<thead>
<tr>
<th>Character</th>
<th>Endianness</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘=’</td>
<td>native</td>
</tr>
<tr>
<td>‘&lt;’</td>
<td>little-endian</td>
</tr>
<tr>
<td>‘&gt;’</td>
<td>big-endian</td>
</tr>
<tr>
<td>‘</td>
<td>’</td>
</tr>
</tbody>
</table>

All built-in data-type objects have byteorder either ‘=’ or ‘|’.
Examples

```
>>> dt = np.dtype('i2')
>>> dt.byteorder
'='
>>> # endian is not relevant for 8 bit numbers
>>> np.dtype('i1').byteorder
'|'
>>> # or ASCII strings
>>> np.dtype('S2').byteorder
'|'
>>> # Even if specific code is given, and it is native
>>> # '=' is the byteorder
>>> import sys

>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>
>>> swapped_code = sys_is_le and '>' or '<'
>>> dt = np.dtype(native_code + 'i2')
>>> dt.byteorder
'='
>>> # Swapped code shows up as itself
>>> dt = np.dtype(swapped_code + 'i2')
>>> dt.byteorder == swapped_code
True
```

Information about sub-data-types in a record:

- **dtype.fields**: Dictionary of named fields defined for this data type, or None.
- **dtype.names**: Ordered list of field names, or None if there are no fields.

**dtype.fields**

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

```
(dtype, offset[, title])
```

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the `ndarray.getfield` and `ndarray.setfield` methods.

**See Also:**

`ndarray.getfield`, `ndarray.setfield`

**Examples**

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print dt.fields
{'grades': (dtype([('float64', (2,))], 16), 'name': (dtype('|S16'), 0))}
```

**dtype.names**

Ordered list of field names, or None if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.
Examples

```python
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

For data types that describe sub-arrays:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype.subdtype</td>
<td>Tuple (item_dtype, shape) if this dtype describes a sub-array, and</td>
</tr>
<tr>
<td>dtype.shape</td>
<td>Shape tuple of the sub-array if this data type describes a sub-array,</td>
</tr>
</tbody>
</table>

**dtype.subdtype**

Tuple (item_dtype, shape) if this *dtype* describes a sub-array, and None otherwise.

The `shape` is the fixed shape of the sub-array described by this data type, and `item_dtype` the data type of the array.

If a field whose `dtype` object has this attribute is retrieved, then the extra dimensions implied by `shape` are tacked on to the end of the retrieved array.

**dtype.shape**

Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.

Attributes providing additional information:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype.hasobject</td>
<td>Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.</td>
</tr>
<tr>
<td>dtype.flags</td>
<td>Bit-flags describing how this data type is to be interpreted.</td>
</tr>
<tr>
<td>dtype.isbuiltin</td>
<td>Integer indicating how this dtype relates to the built-in dtypes.</td>
</tr>
<tr>
<td>dtype.isnative</td>
<td>Boolean indicating whether the byte order of this dtype is native</td>
</tr>
<tr>
<td>dtype.descr</td>
<td>Array-interface compliant full description of the data-type.</td>
</tr>
<tr>
<td>dtype.alignment</td>
<td>The required alignment (bytes) of this data-type according to the compiler.</td>
</tr>
</tbody>
</table>

**dtype.hasobject**

Boolean indicating whether this `dtype` contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won’t.

**dtype.flags**

Bit-flags describing how this data type is to be interpreted.

Bit-masks are in `numpy.core.multiarray` as the constants `ITEM_HASOBJECT`, `LIST_PICKLE`, `ITEM_IS_POINTER`, `NEEDS_INIT`, `NEEDS_PYAPI`, `USE_GETITEM`, `USE_SETITEM`. A full explanation of these flags is in C-API documentation; they are largely useful for user-defined data-types.

**dtype.isbuiltin**

Integer indicating how this `dtype` relates to the built-in dtypes.

Read-only.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if this is a structured array type, with fields</td>
</tr>
<tr>
<td>1</td>
<td>if this is a <code>dtype</code> compiled into numpy (such as ints, floats etc)</td>
</tr>
<tr>
<td>2</td>
<td>if the <code>dtype</code> is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See <code>user:user-defined-data-types</code> in the Numpy manual.</td>
</tr>
</tbody>
</table>
Examples

```python
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0
```

dtype.isnative
Boolean indicating whether the byte order of this dtype is native to the platform.

dtype.descr
Array-interface compliant full description of the data-type.

The format is that required by the ‘descr’ key in the `__array_interface__` attribute.

dtype.alignment
The required alignment (bytes) of this data-type according to the compiler.

More information is available in the C-API section of the manual.

Methods

Data types have the following method for changing the byte order:

```python
dtype.newbyteorder([new_order]) Return a new dtype with a different byte order.
```

dtype.newbyteorder(new_order='S')
Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

Parameters

new_order : string, optional
Byte order to force; a value from the byte order specifications below. The default value (`'S'`) results in swapping the current byte order. `new_order` codes can be any of:

- `'S'` - swap dtype from current to opposite endian
- `('<'`, `'L'`) - little endian
- `('>'`, `'B'`) - big endian
- `('=', `'N'`) - native order
- `('I'`, `'I'`) - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of `new_order` for these alternatives. For example, any of `>'` or `B` or `b` or `brian` are valid to specify big-endian.

Returns

new_dtype : dtype
New dtype object with the given change to the byte order.

Notes

Changes are also made in all fields and sub-arrays of the data type.
Examples

```python
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>
>>> swapped_code = sys_is_le and '>' or '<
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

The following methods implement the pickle protocol:

```
   _dtype.__reduce__
   _dtype.__setstate__
```

dtype._reduce__()

dtype._setstate__()

1.4 Indexing

ndarrays can be indexed using the standard Python `x[obj]` syntax, where `x` is the array and `obj` the selection. There are three kinds of indexing available: record access, basic slicing, advanced indexing. Which one occurs depends on `obj`.

Note: In Python, `x[(exp1, exp2, ..., expN)]` is equivalent to `x[exp1, exp2, ..., expN]`; the latter is just syntactic sugar for the former.

1.4.1 Basic Slicing

Basic slicing extends Python’s basic concept of slicing to N dimensions. Basic slicing occurs when `obj` is a slice object (constructed by `start:stop:step` notation inside of brackets), an integer, or a tuple of slice objects and
integers. **Ellipsis** and **newaxis** objects can be interspersed with these as well. In order to remain backward compatible with a common usage in Numeric, basic slicing is also initiated if the selection object is any sequence (such as a list) containing slice objects, the **Ellipsis** object, or the **newaxis** object, but no integer arrays or other embedded sequences.

The simplest case of indexing with \( N \) integers returns an **array scalar** representing the corresponding item. As in Python, all indices are zero-based: for the \( i \)-th index \( n_i \), the valid range is \( 0 \leq n_i < d_i \) where \( d_i \) is the \( i \)-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (i.e., if \( i < 0 \), it means \( n_i + i \)).

All arrays generated by basic slicing are always views of the original array.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is \( i:j:k \) where \( i \) is the starting index, \( j \) is the stopping index, and \( k \) is the step (\( k \neq 0 \)). This selects the \( m \) elements (in the corresponding dimension) with index values \( i, i + k, ..., i + (m - 1)k \) where \( m = q + (r \neq 0) \) and \( q \) and \( r \) are the quotient and remainder obtained by dividing \( j - i \) by \( k \): \( j - i = qk + r \), so that \( i + (m - 1)k < j \).

**Example**

```python
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]
array([1, 3, 5])
```

- Negative \( i \) and \( j \) are interpreted as \( n + i \) and \( n + j \) where \( n \) is the number of elements in the corresponding dimension. Negative \( k \) makes stepping go towards smaller indices.

**Example**

```python
>>> x[-2:10]
array([8, 9])
>>> x[-3:3:-1]
array([7, 6, 5, 4])
```

- Assume \( n \) is the number of elements in the dimension being sliced. Then, if \( i \) is not given it defaults to 0 for \( k > 0 \) and \( n \) for \( k < 0 \). If \( j \) is not given it defaults to \( n \) for \( k > 0 \) and -1 for \( k < 0 \). If \( k \) is not given it defaults to 1. Note that :: is the same as : and means select all indices along this axis.

**Example**

```python
>>> x[5:]
array([5, 6, 7, 8, 9])
```

- If the number of objects in the selection tuple is less than \( N \), then :: is assumed for any subsequent dimensions.

**Example**

```python
>>> x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
>>> x.shape
(2, 3, 1)
>>> x[1:2]
array([[[4]],
```

1.4. Indexing
• Ellipsis expand to the number of : objects needed to make a selection tuple of the same length as x.ndim. Only the first ellipsis is expanded, any others are interpreted as :.

Example

```python
>>> x[...,0]
array([[1, 2, 3],
       [4, 5, 6]])
```

• Each newaxis object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the newaxis object in the selection tuple.

Example

```python
>>> x[:,np.newaxis,:, :].shape
(2, 1, 3, 1)
```

• An integer, i, returns the same values as i:i+1 except the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the p-th element an integer (and all other entries :) returns the corresponding sub-array with dimension N - 1. If N = 1 then the returned object is an array scalar. These objects are explained in Scalars.

• If the selection tuple has all entries : except the p-th entry which is a slice object i:j:k, then the returned array has dimension N formed by concatenating the sub-arrays returned by integer indexing of elements i, i+k, ..., i+(m-1)*k < j.

• Basic slicing with more than one non-: entry in the slicing tuple, acts like repeated application of slicing using a single non-: entry, where the non-: entries are successively taken (with all other non-: entries replaced by :). Thus, x[ind1,...,ind2,: ] acts like x[ind1][...,ind2,: ] under basic slicing.

**Warning:** The above is not true for advanced slicing.

• You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in x[obj] = value must be (broadcastable) to the same shape as x[obj].

**Note:** Remember that a slicing tuple can always be constructed as obj and used in the x[obj] notation. Slice objects can be used in the construction in place of the [start:stop:step] notation. For example, x[1:10:5,: :-1] can also be implemented as obj = (slice(1,10,5), slice(None,None,-1)); x[obj]. This can be useful for constructing generic code that works on arrays of arbitrary dimension.

numpy.newaxis

The newaxis object can be used in all slicing operations as discussed above. None can also be used instead of newaxis.

### 1.4.2 Advanced indexing

Advanced indexing is triggered when the selection object, obj, is a non-tuple sequence object, an ndarray (of data type integer or bool), or a tuple with at least one sequence object or ndarray (of data type integer or bool). There are two types of advanced indexing: integer and Boolean.
Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a *view*).

**Integer**

Integer indexing allows selection of arbitrary items in the array based on their $N$-dimensional index. This kind of selection occurs when advanced indexing is triggered and the selection object is not an array of data type bool. For the discussion below, when the selection object is not a tuple, it will be referred to as if it had been promoted to a 1-tuple, which will be called the selection tuple. The rules of advanced integer-style indexing are:

- If the length of the selection tuple is larger than $N$ an error is raised.
- All sequences and scalars in the selection tuple are converted to `intp` indexing arrays.
- All selection tuple objects must be convertible to `intp` arrays, slice objects, or the Ellipsis object.
- The first Ellipsis object will be expanded, and any other Ellipsis objects will be treated as full slice (:) objects. The expanded Ellipsis object is replaced with as many full slice (:) objects as needed to make the length of the selection tuple $N$.
- If the selection tuple is smaller than $N$, then as many : objects as needed are added to the end of the selection tuple so that the modified selection tuple has length $N$.
- All the integer indexing arrays must be *broadcastable* to the same shape.
- The shape of the output (or the needed shape of the object to be used for setting) is the broadcasted shape.
- After expanding any ellipses and filling out any missing : objects in the selection tuple, then let $N_i$ be the number of indexing arrays, and let $N_s = N - N_i$ be the number of slice objects. Note that $N_i > 0$ (or we wouldn’t be doing advanced integer indexing).
- If $N_s = 0$ then the $M$-dimensional result is constructed by varying the index tuple $(i_1, \ldots, i_M)$ over the range of the result shape and for each value of the index tuple $(\text{ind}_1, \ldots, \text{ind}_M)$:

\[
\text{result}[i_1, \ldots, i_M] = x[\text{ind}_1[i_1, \ldots, i_M], \text{ind}_2[i_1, \ldots, i_M], \\
\ldots, \text{ind}_N[i_1, \ldots, i_M]]
\]

**Example**

Suppose the shape of the broadcasted indexing arrays is 3-dimensional and $N$ is 2. Then the result is found by letting $i, j, k$ run over the shape found by broadcasting $\text{ind}_1$ and $\text{ind}_2$, and each $i, j, k$ yields:

\[
\text{result}[i,j,k] = x[\text{ind}_1[i,j,k], \text{ind}_2[i,j,k]]
\]

- If $N_s > 0$, then partial indexing is done. This can be somewhat mind-boggling to understand, but if you think in terms of the shapes of the arrays involved, it can be easier to grasp what happens. In simple cases (*i.e.* one indexing array and $N - 1$ slice objects) it does exactly what you would expect (concatenation of repeated application of basic slicing). The rule for partial indexing is that the shape of the result (or the interpreted shape of the object to be used in setting) is the shape of $x$ with the indexed subspace replaced with the broadcasted indexing subspace. If the index subspaces are right next to each other, then the broadcasted indexing space directly replaces all of the indexed subspaces in $x$. If the indexing subspaces are separated (by slice objects), then the broadcasted indexing space is first, followed by the sliced subspace of $x$.

**Example**

Suppose $x$.shape is (10,20,30) and $\text{ind}$ is a (2,3,4)-shaped indexing `intp` array, then result = $x[\ldots,\text{ind},:]$ has shape (10,2,3,4,30) because the (20,)-shaped subspace has been replaced with a (2,3,4)-shaped broadcasted indexing subspace. If we let $i, j, k$ loop over the (2,3,4)-shaped subspace then
result[...,i,j,k,:] = x[... , ind[i,j,k,:]]. This example produces the same result as
x.take(ind, axis=-2).

Example

Now let x.shape be (10,20,30,40,50) and suppose ind_1 and ind_2 are broadcastable to the shape (2,3,4). Then x[:,ind_1,ind_2] has shape (10,2,3,4,40,50) because the (20,30)-shaped subspace from X has been replaced with the (2,3,4) subspace from the indices. However, x[:,ind_1,:,ind_2] has shape (2,3,4,10,30,50) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use .transpose() to move the subspace anywhere desired. (Note that this example cannot be replicated using take.)

Boolean

This advanced indexing occurs when obj is an array object of Boolean type (such as may be returned from comparison operators). It is always equivalent to (but faster than) x[obj.nonzero()] where, as described above, obj.nonzero() returns a tuple (of length obj.ndim) of integer index arrays showing the True elements of obj.

The special case when obj.ndim == x.ndim is worth mentioning. In this case x[obj] returns a 1-dimensional array filled with the elements of x corresponding to the True values of obj. The search order will be C-style (last index varies the fastest). If obj has True values at entries that are outside of the bounds of x, then an index error will be raised.

You can also use Boolean arrays as element of the selection tuple. In such instances, they will always be interpreted as nonzero(obj) and the equivalent integer indexing will be done.

Warning: The definition of advanced indexing means that x[(1,2,3),] is fundamentally different than x[(1,2,3)]. The latter is equivalent to x[1,2,3] which will trigger basic selection while the former will trigger advanced indexing. Be sure to understand why this is occurs.

Also recognize that x[[1,2,3]] will trigger advanced indexing, whereas x[[1,2,slice(None)]] will trigger basic slicing.

1.4.3 Record Access

See Also:

Data type objects (dtype), Scalars

If the ndarray object is a record array, i.e. its data type is a record data type, the fields of the array can be accessed by indexing the array with strings, dictionary-like.

Indexing x['field-name'] returns a new view to the array, which is of the same shape as x (except when the field is a sub-array) but of data type x.dtype['field-name'] and contains only the part of the data in the specified field. Also record array scalars can be “indexed” this way.

Indexing into a record array can also be done with a list of field names, e.g. x[['field-name1','field-name2']]. Currently this returns a new array containing a copy of the values in the fields specified in the list. As of NumPy 1.7, returning a copy is being deprecated in favor of returning a view. A copy will continue to be returned for now, but a FutureWarning will be issued when writing to the copy. If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e. use x[['field-name1','field-name2']].copy(). This will work with both past and future versions of NumPy.

If the accessed field is a sub-array, the dimensions of the sub-array are appended to the shape of the result.
Example

```python
>>> x = np.zeros((2,2), dtype=[('a', np.int32), ('b', np.float64, (3,3))])
>>> x['a'].shape
(2, 2)
>>> x['a'].dtype
dtype('int32')
>>> x['b'].shape
(2, 2, 3, 3)
>>> x['b'].dtype
dtype('float64')
```

1.4.4 Flat Iterator indexing

`x.flat` returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that `x.flat` is a 1-dimensional view. It can be used for integer indexing with 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

1.5 Iterating Over Arrays

The iterator object `nditer`, introduced in NumPy 1.6, provides many flexible ways to visit all the elements of one or more arrays in a systematic fashion. This page introduces some basic ways to use the object for computations on arrays in Python, then concludes with how one can accelerate the inner loop in Cython. Since the Python exposure of `nditer` is a relatively straightforward mapping of the C array iterator API, these ideas will also provide help working with array iteration from C or C++.

1.5.1 Single Array Iteration

The most basic task that can be done with the `nditer` is to visit every element of an array. Each element is provided one by one using the standard Python iterator interface.

Example

```python
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a):
...     print x,
... 0 1 2 3 4 5
```

An important thing to be aware of for this iteration is that the order is chosen to match the memory layout of the array instead of using a standard C or Fortran ordering. This is done for access efficiency, reflecting the idea that by default one simply wants to visit each element without concern for a particular ordering. We can see this by iterating over the transpose of our previous array, compared to taking a copy of that transpose in C order.

Example

```python
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a.T):
```
The elements of both $a$ and $a.T$ get traversed in the same order, namely the order they are stored in memory, whereas the elements of $a.T.copy(order='C')$ get visited in a different order because they have been put into a different memory layout.

**Controlling Iteration Order**

There are times when it is important to visit the elements of an array in a specific order, irrespective of the layout of the elements in memory. The `nditer` object provides an `order` parameter to control this aspect of iteration. The default, having the behavior described above, is order='K' to keep the existing order. This can be overridden with order='C' for C order and order='F' for Fortran order.

**Example**

```python
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, order='F'):
...     print x,
... 0 3 1 4 2 5
>>> for x in np.nditer(a.T, order='C'):
...     print x,
... 0 3 1 4 2 5
```

**Modifying Array Values**

By default, the `nditer` treats the input array as a read-only object. To modify the array elements, you must specify either read-write or write-only mode. This is controlled with per-operand flags.

Regular assignment in Python simply changes a reference in the local or global variable dictionary instead of modifying an existing variable in place. This means that simply assigning to $x$ will not place the value into the element of the array, but rather switch $x$ from being an array element reference to being a reference to the value you assigned. To actually modify the element of the array, $x$ should be indexed with the ellipsis.

**Example**

```python
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> for x in np.nditer(a, op_flags=['readwrite']):
...     x[...] = 2 * x
...>>> a
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```
Using an External Loop

In all the examples so far, the elements of a are provided by the iterator one at a time, because all the looping logic is internal to the iterator. While this is simple and convenient, it is not very efficient. A better approach is to move the one-dimensional innermost loop into your code, external to the iterator. This way, NumPy’s vectorized operations can be used on larger chunks of the elements being visited.

The `nditer` will try to provide chunks that are as large as possible to the inner loop. By forcing ‘C’ and ‘F’ order, we get different external loop sizes. This mode is enabled by specifying an iterator flag.

Observe that with the default of keeping native memory order, the iterator is able to provide a single one-dimensional chunk, whereas when forcing Fortran order, it has to provide three chunks of two elements each.

Example

```python
code
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop']):
...     print x,
...[0 1 2 3 4 5]
```

```python
code
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print x,
...[0 3] [1 4] [2 5]
```

Tracking an Index or Multi-Index

During iteration, you may want to use the index of the current element in a computation. For example, you may want to visit the elements of an array in memory order, but use a C-order, Fortran-order, or multidimensional index to look up values in a different array.

The Python iterator protocol doesn’t have a natural way to query these additional values from the iterator, so we introduce an alternate syntax for iterating with an `nditer`. This syntax explicitly works with the iterator object itself, so its properties are readily accessible during iteration. With this looping construct, the current value is accessible by indexing into the iterator, and the index being tracked is the property `index` or `multi_index` depending on what was requested.

The Python interactive interpreter unfortunately prints out the values of expressions inside the while loop during each iteration of the loop. We have modified the output in the examples using this looping construct in order to be more readable.

Example

```python
code
>>> a = np.arange(6).reshape(2,3)
>>> it = np.nditer(a, flags=['f_index'])
>>> while not it.finished:
...     print "%d <id>" % (it[0], it.index),
...     it.iternext()
```
...  
0 <0> 1 <2> 2 <4> 3 <1> 4 <3> 5 <5>  

>>> it = np.nditer(a, flags=['multi_index'])  
>>> while not it.finished:  
...    print "%d <" + str(it.multi_index) + ">",  
...    it.iternext()  
...  
0 <(0, 0)> 1 <(0, 1)> 2 <(0, 2)> 3 <(1, 0)> 4 <(1, 1)> 5 <(1, 2)>  

>>> it = np.nditer(a, flags=['multi_index'], op_flags=['writeonly'])  
>>> while not it.finished:  
...    it[0] = it.multi_index[1] - it.multi_index[0]  
...    it.iternext()  
...  
>>> a  
array([[ 0, 1, 2],  
[ 1, 2, 3]])  

Tracking an index or multi-index is incompatible with using an external loop, because it requires a different index  
value per element. If you try to combine these flags, the nditer object will raise an exception

Example

>>> a = np.zeros((2,3))  
>>> it = np.nditer(a, flags=['c_index', 'external_loop'])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: Iterator flag EXTERNAL_LOOP cannot be used if an index or multi-index is being tracked

Buffering the Array Elements

When forcing an iteration order, we observed that the external loop option may provide the elements in smaller chunks  
because the elements can’t be visited in the appropriate order with a constant stride. When writing C code, this is  
generally fine, however in pure Python code this can cause a significant reduction in performance.

By enabling buffering mode, the chunks provided by the iterator to the inner loop can be made larger, significantly  
reducing the overhead of the Python interpreter. In the example forcing Fortran iteration order, the inner loop gets to  
see all the elements in one go when buffering is enabled.

Example

>>> a = np.arange(6).reshape(2,3)  
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):  
...    print x,  
...  
[0 3] [1 4] [2 5]  

>>> for x in np.nditer(a, flags=['external_loop','buffered'], order='F'):  
...    print x,  
...  
[0 3 1 4 2 5]
Iterating as a Specific Data Type

There are times when it is necessary to treat an array as a different data type than it is stored as. For instance, one may want to do all computations on 64-bit floats, even if the arrays being manipulated are 32-bit floats. Except when writing low-level C code, it’s generally better to let the iterator handle the copying or buffering instead of casting the data type yourself in the inner loop.

There are two mechanisms which allow this to be done, temporary copies and buffering mode. With temporary copies, a copy of the entire array is made with the new data type, then iteration is done in the copy. Write access is permitted through a mode which updates the original array after all the iteration is complete. The major drawback of temporary copies is that the temporary copy may consume a large amount of memory, particularly if the iteration data type has a larger itemsize than the original one.

Buffering mode mitigates the memory usage issue and is more cache-friendly than making temporary copies. Except for special cases, where the whole array is needed at once outside the iterator, buffering is recommended over temporary copying. Within NumPy, buffering is used by the ufuncs and other functions to support flexible inputs with minimal memory overhead.

In our examples, we will treat the input array with a complex data type, so that we can take square roots of negative numbers. Without enabling copies or buffering mode, the iterator will raise an exception if the data type doesn’t match precisely.

Example

```python
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_dtypes=['complex128']):
...     print np.sqrt(x),
...     ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand required copying or buffering, but neither copying nor buffering was enabled
```

In copying mode, ‘copy’ is specified as a per-operand flag. This is done to provide control in a per-operand fashion. Buffering mode is specified as an iterator flag.

Example

```python
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_flags=['readonly','copy'],
...                    op_dtypes=['complex128']):
...     print np.sqrt(x),
... ...
1.73205080757j 1.41421356237j 1j 0j (1+0j) (1.41421356237+0j)
```

The iterator uses NumPy’s casting rules to determine whether a specific conversion is permitted. By default, it enforces ‘safe’ casting. This means, for example, that it will raise an exception if you try to treat a 64-bit float array as a 32-bit float array. In many cases, the rule ‘same_kind’ is the most reasonable rule to use, since it will allow conversion from 64 to 32-bit float, but not from float to int or from complex to float.

Example
NumPy Reference, Release 1.8.1

```python
>>> a = np.arange(6.)
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32']):
...    print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype('float32') according to the rule 'safe'

>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32'], casting='same_kind'):
...    print x,
...
0.0 1.0 2.0 3.0 4.0 5.0

>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['int32'], casting='same_kind'):
...    print x,
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype('int32') according to the rule 'same_kind'
```

One thing to watch out for is conversions back to the original data type when using a read-write or write-only operand. A common case is to implement the inner loop in terms of 64-bit floats, and use `same_kind` casting to allow the other floating-point types to be processed as well. While in read-only mode, an integer array could be provided, read-write mode will raise an exception because conversion back to the array would violate the casting rule.

---

### Example

```python
>>> a = np.arange(6)
>>> for x in np.nditer(a, flags=['buffered'], op_flags=['readwrite'],
... op_dtypes=['float64'], casting='same_kind'):
...    x[...] = x / 2.0
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Iterator requested dtype could not be cast from dtype('float64') to dtype('int64'), the operand 0 dtype, according to the rule 'same_kind'
```

---

### 1.5.2 Broadcasting Array Iteration

NumPy has a set of rules for dealing with arrays that have differing shapes which are applied whenever functions take multiple operands which combine element-wise. This is called broadcasting. The `nditer` object can apply these rules for you when you need to write such a function.

As an example, we print out the result of broadcasting a one and a two dimensional array together.

---

### Example

```python
>>> a = np.arange(3)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...    print "%d:%d" % (x,y),
...0:0 1:1 2:2 0:3 1:4 2:5
```
When a broadcasting error occurs, the iterator raises an exception which includes the input shapes to help diagnose the problem.

Example

```python
>>> a = np.arange(2)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...    print "%d:%d" % (x,y),
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2) (2,3)
```

Iterator-Allocated Output Arrays

A common case in NumPy functions is to have outputs allocated based on the broadcasting of the input, and additionally have an optional parameter called ‘out’ where the result will be placed when it is provided. The `nditer` object provides a convenient idiom that makes it very easy to support this mechanism.

We’ll show how this works by creating a function `square` which squares its input. Let’s start with a minimal function definition excluding ‘out’ parameter support.

Example

```python
>>> def square(a):
...    it = np.nditer([a, None])
...    for x, y in it:
...        y[...] = x*x
...    return it.operands[1]
...
>>> square([1,2,3])
array([1, 4, 9])
```

By default, the `nditer` uses the flags ‘allocate’ and ‘writeonly’ for operands that are passed in as None. This means we were able to provide just the two operands to the iterator, and it handled the rest.

When adding the ‘out’ parameter, we have to explicitly provide those flags, because if someone passes in an array as ‘out’, the iterator will default to ‘readonly’, and our inner loop would fail. The reason ‘readonly’ is the default for input arrays is to prevent confusion about unintentionally triggering a reduction operation. If the default were ‘readwrite’, any broadcasting operation would also trigger a reduction, a topic which is covered later in this document.

While we’re at it, let’s also introduce the ‘no_broadcast’ flag, which will prevent the output from being broadcast. This is important, because we only want one input value for each output. Aggregating more than one input value is a reduction operation which requires special handling. It would already raise an error because reductions must be explicitly enabled in an iterator flag, but the error message that results from disabling broadcasting is much more understandable for end-users. To see how to generalize the square function to a reduction, look at the sum of squares function in the section about Cython.

For completeness, we’ll also add the ‘external_loop’ and ‘buffered’ flags, as these are what you will typically want for performance reasons.

Example

```python
```

1.5. Iterating Over Arrays 113
Outer Product Iteration

Any binary operation can be extended to an array operation in an outer product fashion like in `outer`, and the `nditer` object provides a way to accomplish this by explicitly mapping the axes of the operands. It is also possible to do this with `newaxis` indexing, but we will show you how to directly use the `nditer` `op_axes` parameter to accomplish this with no intermediate views.

We’ll do a simple outer product, placing the dimensions of the first operand before the dimensions of the second operand. The `op_axes` parameter needs one list of axes for each operand, and provides a mapping from the iterator’s axes to the axes of the operand.

Suppose the first operand is one dimensional and the second operand is two dimensional. The iterator will have three dimensions, so `op_axes` will have two 3-element lists. The first list picks out the one axis of the first operand, and is -1 for the rest of the iterator axes, with a final result of `[0, -1, -1]`. The second list picks out the two axes of the second operand, but shouldn’t overlap with the axes picked out in the first operand. Its list is `[-1, 0, 1]`. The output operand maps onto the iterator axes in the standard manner, so we can provide None instead of constructing another list.

The operation in the inner loop is a straightforward multiplication. Everything to do with the outer product is handled by the iterator setup.

Example

```python
>>> a = np.arange(3)
>>> b = np.arange(8).reshape(2,4)
>>> it = np.nditer([a, b, None], flags=['external_loop'],
... op_axes=[[0, -1, -1], [-1, 0, 1], None])
>>> for x, y, z in it:
...    z[...] = x*y
... >>> it.operands[2]
array([[ 0,  0,  0,  0],
```
Reduction Iteration

Whenever a writeable operand has fewer elements than the full iteration space, that operand is undergoing a reduction. The `nditer` object requires that any reduction operand be flagged as read-write, and only allows reductions when ‘reduce_ok’ is provided as an iterator flag.

For a simple example, consider taking the sum of all elements in an array.

Example

```python
>>> a = np.arange(24).reshape(2,3,4)
>>> b = np.array(0)
>>> for x, y in np.nditer([a, b], flags=['reduce_ok', 'external_loop'],
... op_flags=[['readonly'], ['readwrite']]):
...     y[...] += x
...
>>> b
array(276)
>>> np.sum(a)
276
```

Things are a little bit more tricky when combining reduction and allocated operands. Before iteration is started, any reduction operand must be initialized to its starting values. Here’s how we can do this, taking sums along the last axis of `a`.

Example

```python
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop'],
... op_flags=[['readonly'], ['readwrite', 'allocate']],
... op_axes=[None, [0,1,-1]])
>>> it.operands[1][...] = 0
>>> for x, y in it:
...     y[...] += x
...
>>> it.operands[1]
array([[ 6, 22, 38],
       [54, 70, 86]])
>>> np.sum(a, axis=2)
array([[ 6, 22, 38],
       [54, 70, 86]])
```

To do buffered reduction requires yet another adjustment during the setup. Normally the iterator construction involves copying the first buffer of data from the readable arrays into the buffer. Any reduction operand is readable, so it may be read into a buffer. Unfortunately, initialization of the operand after this buffering operation is complete will not be reflected in the buffer that the iteration starts with, and garbage results will be produced.
The iterator flag “delay_bufalloc” is there to allow iterator-allocated reduction operands to exist together with buffering. When this flag is set, the iterator will leave its buffers uninitialized until it receives a reset, after which it will be ready for regular iteration. Here’s how the previous example looks if we also enable buffering.

Example

```python
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok', 'external_loop',
...       'buffered', 'delay_bufalloc'],
...       op_flags=[['readonly']], ['readwrite', 'allocate']],
...       op_axes=[None, [0,1,-1]])
>>> it.operands[1][...].fill(0)
>>> it.reset()
>>> for x, y in it:
...     y[...] += x
...     y[...]
array([[ 6, 22, 38],
       [54, 70, 86]])
```

1.5.3 Putting the Inner Loop in Cython

Those who want really good performance out of their low level operations should strongly consider directly using the iteration API provided in C, but for those who are not comfortable with C or C++, Cython is a good middle ground with reasonable performance tradeoffs. For the `nditer` object, this means letting the iterator take care of broadcasting, dtype conversion, and buffering, while giving the inner loop to Cython.

For our example, we’ll create a sum of squares function. To start, let’s implement this function in straightforward Python. We want to support an ‘axis’ parameter similar to the numpy `sum` function, so we will need to construct a list for the `op_axes` parameter. Here’s how this looks.

Example

```python
>>> def axis_to_axeslist(axis, ndim):
...     if axis is None:
...         return [-1] * ndim
...     else:
...         if type(axis) is not tuple:
...             axis = (axis,)
...         axeslist = [1] * ndim
...         for i in axis:
...             axeslist[i] = -1
...         ax = 0
...         for i in range(ndim):
...             if axeslist[i] != -1:
...                 axeslist[i] = ax
...                 ax += 1
...         return axeslist
...
>>> def sum_squares_py(arr, axis=None, out=None):
...     axeslist = axis_to_axeslist(axis, arr.ndim)
...     it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
...       'buffered', 'delay_bufalloc'],
...       op_flags=[['readonly']], ['readwrite', 'allocate']],
...       op_axes=[None, axeslist],
```
To Cython-ize this function, we replace the inner loop (y[... ] += x*x) with Cython code that’s specialized for the float64 dtype. With the ‘external_loop’ flag enabled, the arrays provided to the inner loop will always be one-dimensional, so very little checking needs to be done.

Here’s the listing of sum_squares.pyx:

```python
import numpy as np
cimport numpy as np
cimport cython
def axis_to_axeslist(axis, ndim):
    if axis is None:
        return [-1] * ndim
    else:
        if type(axis) is not tuple:
            axis = (axis,)
        axeslist = [1] * ndim
        for i in axis:
            axeslist[i] = -1
        ax = 0
        for i in range(ndim):
            if axeslist[i] != -1:
                axeslist[i] = ax
                ax += 1
        return axeslist
@cython.boundscheck(False)
def sum_squares_cy(arr, axis=None, out=None):
    cdef np.ndarray[double] x
    cdef np.ndarray[double] y
cdef int size
    cdef double value
    axeslist = axis_to_axeslist(axis, arr.ndim)
    it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
                                'buffered', 'delay_bufalloc'],
                    op_flags=[['readonly'], ['readwrite', 'allocate']],
                    op_axes=[None, axeslist],
                    op_dtypes=['float64', 'float64'])
    it.operands[1][...] = 0
    it.reset()
    for xarr, yarr in it:
        x = xarr
        y = yarr
        size = x.shape[0]
```

1.5. Iterating Over Arrays
for i in range(size):
    value = x[i]
    y[i] = y[i] + value * value
return it.operands[1]

On this machine, building the .pyx file into a module looked like the following, but you may have to find some Cython tutorials to tell you the specifics for your system configuration:

```bash
$ cython sum_squares.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -I/usr/include/python2.7 -fno-strict-aliasing -o sum_squares.so sum_squares.c
```

Running this from the Python interpreter produces the same answers as our native Python/NumPy code did.

Example

```bash
>>> from sum_squares import sum_squares_cy
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_cy(a)
array(55.0)
>>> sum_squares_cy(a, axis=-1)
array([ 5., 50.])
```

Doing a little timing in IPython shows that the reduced overhead and memory allocation of the Cython inner loop is providing a very nice speedup over both the straightforward Python code and an expression using NumPy’s built-in sum function:

```bash
>>> a = np.random.rand(1000,1000)
>>> timeit sum_squares_py(a, axis=-1)
10 loops, best of 3: 37.1 ms per loop
>>> timeit np.sum(a*a, axis=-1)
10 loops, best of 3: 20.9 ms per loop
>>> timeit sum_squares_cy(a, axis=-1)
100 loops, best of 3: 11.8 ms per loop
>>> np.all(sum_squares_cy(a, axis=-1) == np.sum(a*a, axis=-1))
True
>>> np.all(sum_squares_py(a, axis=-1) == np.sum(a*a, axis=-1))
True
```

1.6 Standard array subclasses

The `ndarray` in NumPy is a “new-style” Python built-in-type. Therefore, it can be inherited from (in Python or in C) if desired. Therefore, it can form a foundation for many useful classes. Often whether to sub-class the array object or to simply use the core array component as an internal part of a new class is a difficult decision, and can be simply a matter of choice. NumPy has several tools for simplifying how your new object interacts with other array objects, and so the choice may not be significant in the end. One way to simplify the question is by asking yourself if the object you are interested in can be replaced as a single array or does it really require two or more arrays at its core.

Note that `asarray` always returns the base-class `ndarray`. If you are confident that your use of the array object can handle any subclass of an `ndarray`, then `asanyarray` can be used to allow subclasses to propagate more cleanly through your subroutine. In principal a subclass could redefine any aspect of the array and therefore, under strict
guidelines, `asanyarray` would rarely be useful. However, most subclasses of the arrayobject will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One important example, however, of why your subroutine may not be able to handle an arbitrary subclass of an array is that matrices redefine the `*` operator to be matrix-multiplication, rather than element-by-element multiplication.

### 1.6.1 Special attributes and methods

**See Also:**

*Subclassing ndarray*

Numpy provides several hooks that subclasses of `ndarray` can customize:

- **`numpy.__array_finalize__`** (`self`)
  This method is called whenever the system internally allocates a new array from `obj`, where `obj` is a subclass (subtype) of the `ndarray`. It can be used to change attributes of `self` after construction (so as to ensure a 2-d matrix for example), or to update meta-information from the “parent.” Subclasses inherit a default implementation of this method that does nothing.

- **`numpy.__array_prepare__`** (`array`, `context=None`)
  At the beginning of every `ufunc`, this method is called on the input object with the highest array priority, or the output object if one was specified. The output array is passed in and whatever is returned is passed to the ufunc. Subclasses inherit a default implementation of this method which simply returns the output array unmodified. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the ufunc for computation.

- **`numpy.__array_wrap__`** (`array`, `context=None`)
  At the end of every `ufunc`, this method is called on the input object with the highest array priority, or the output object if one was specified. The ufunc-computed array is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method, which transforms the array into a new instance of the object’s class. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the user.

- **`numpy.__array_priority__`**
  The value of this attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. Subclasses inherit a default value of 1.0 for this attribute.

- **`numpy.__array__`** (`[dtype]`)
  If a class having the `__array__` method is used as the output object of an `ufunc`, results will be written to the object returned by `__array__`.

### 1.6.2 Matrix objects

`matrix` objects inherit from the `ndarray` and therefore, they have the same attributes and methods of ndarrays. There are six important differences of matrix objects, however, that may lead to unexpected results when you use matrices but expect them to act like arrays:

1. Matrix objects can be created using a string notation to allow Matlab-style syntax where spaces separate columns and semicolons (`';'`) separate rows.

2. Matrix objects are always two-dimensional. This has far-reaching implications, in that `m.ravel()` is still two-dimensional (with a 1 in the first dimension) and item selection returns two-dimensional objects so that sequence behavior is fundamentally different than arrays.

1.6. Standard array subclasses 119
3. Matrix objects over-ride multiplication to be matrix-multiplication. **Make sure you understand this for functions that you may want to receive matrices.** Especially in light of the fact that asanyarray(m) returns a matrix when m is a matrix.

4. Matrix objects over-ride power to be matrix raised to a power. The same warning about using power inside a function that uses asanyarray(...) to get an array object holds for this fact.

5. The default __array_priority__ of matrix objects is 10.0, and therefore mixed operations with ndarrays always produce matrices.

6. Matrices have special attributes which make calculations easier. These are

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matrix.T</code></td>
<td>transpose</td>
</tr>
<tr>
<td><code>matrix.H</code></td>
<td>hermitian (conjugate) transpose</td>
</tr>
<tr>
<td><code>matrix.I</code></td>
<td>inverse</td>
</tr>
<tr>
<td><code>matrix.A</code></td>
<td>base array</td>
</tr>
</tbody>
</table>

**Warning:** Matrix objects over-ride multiplication, ‘*’, and power, ‘**’, to be matrix-multiplication and matrix power, respectively. If your subroutine can accept sub-classes and you do not convert to base-class arrays, then you must use the ufuncs multiply and power to be sure that you are performing the correct operation for all inputs.

The matrix class is a Python subclass of the ndarray and can be used as a reference for how to construct your own subclass of the ndarray. Matrices can be created from other matrices, strings, and anything else that can be converted to an ndarray. The name “mat” is an alias for “matrix” in NumPy.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matrix</code></td>
<td>Returns a matrix from an array-like object, or from a string of data.</td>
</tr>
<tr>
<td><code>asmatrix(data[, dtype])</code></td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td><code>bmat(obj[, ldict, gdict])</code></td>
<td>Build a matrix object from a string, nested sequence, or array.</td>
</tr>
</tbody>
</table>

**class numpy.matrix**

Returns a matrix from an array-like object, or from a string of data. A matrix is a specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as * (matrix multiplication) and ** (matrix power).

**Parameters**

- `data`: array_like or string
  
  If `data` is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

- `dtype`: data-type
  
  Data-type of the output matrix.

- `copy`: bool
If `data` is already an `ndarray`, then this flag determines whether the data is copied (the default), or whether a view is constructed.

**See Also:**
array

**Examples**

```python
>>> a = np.matrix('1 2; 3 4')
>>> print a
[[1 2]
 [3 4]]
```

```python
>>> np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
 [3, 4]])
```

**Attributes**

- `A` base array
- `A1` 1-d base array
- `H` hermitian (conjugate) transpose
- `I` inverse
- `T` transpose

`base` Base object if memory is from some other object.
`ctypes` An object to simplify the interaction of the array with the ctypes module.
`data` Python buffer object pointing to the start of the array’s data.
`dtype` Data-type of the array’s elements.
`flags` Information about the memory layout of the array.
`flat` A 1-D iterator over the array.
`imag` The imaginary part of the array.
`itemsize` Length of one array element in bytes.
`nbytes` Total bytes consumed by the elements of the array.
`ndim` Number of array dimensions.
`real` The real part of the array.
`shape` Tuple of array dimensions.
`size` Number of elements in the array.
`strides` Tuple of bytes to step in each dimension when traversing an array.

```
matrix.A
    base array
matrix.A1
    1-d base array
matrix.H
    hermitian (conjugate) transpose
matrix.I
    inverse
matrix.T
    transpose
matrix.base
    Base object if memory is from some other object.
```

1.6. Standard array subclasses
Examples

The base of an array that owns its memory is None:

```python
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```python
>>> y = x[2:]
>>> y.base is x
True
```

**matrix.c**types

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

None

**Returns**

c : Python object

Possessing attributes `data`, `shape`, `strides`, etc.

**See Also:**

numpy.ctypeslib

**Notes**

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- **data**: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_`[`data`] [0].

- **shape** (`c_intp*`self.ndim): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `c_int`, `c_long`, or `c_longlong` depending on the platform. The `c_intp` type is defined accordingly in `numpy.ctypeslib`. The ctypes array contains the shape of the underlying array.

- **strides** (`c_intp*`self.ndim): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

- **data_as**(obj): Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

- **shape_as**(obj): Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

- **strides_as**(obj): Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.
Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling \((a+b).\text{ctypes.data\_as}(\text{ctypes.c\_void\_p})\) returns a pointer to memory that is invalid because the array created as \((a+b)\) is deallocated before the next Python statement. You can avoid this problem using either \(c=a+b\) or \(ct=(a+b).\text{ctypes}\). In the latter case, \(ct\) will hold a reference to the array until \(ct\) is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

**Examples**

```python
>>> import ctypes
>>> x
array([[ 0, 1],
       [ 2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data\_as(\text{ctypes.POINTER(\text{ctypes.c\_long})})
<ctypes.LP\_c\_long object at 0x01F01300>
>>> x.ctypes.data\_as(\text{ctypes.POINTER(\text{ctypes.c\_long})}).\text{contents}
c\_long(0)
>>> x.ctypes.data\_as(\text{ctypes.POINTER(\text{ctypes.c\_longlong})}).\text{contents}
c\_longlong(4294967296L)
>>> x.ctypes.shape
<\text{numpy.core\_internal.c\_long\_Array\_2 object at 0x01FFD580}>
>>> x.ctypes.shape\_as(\text{ctypes.c\_long})
<\text{numpy.core\_internal.c\_long\_Array\_2 object at 0x01FCE620}>
>>> x.ctypes.strides
<\text{numpy.core\_internal.c\_long\_Array\_2 object at 0x01FCE620}>
>>> x.ctypes.strides\_as(\text{ctypes.c\_longlong})
<\text{numpy.core\_internal.c\_longlong\_Array\_2 object at 0x01F01300}>
```

**matrix.data**

Python buffer object pointing to the start of the array’s data.

**matrix.dtype**

Data-type of the array’s elements.

- **Parameters**
  - None

- **Returns**
  - \(d\): \text{numpy dtype object}

**See Also:**

numpy.dtype

**Examples**

```python
>>> x
array([[ 0, 1],
       [ 2, 3]])
>>> x.dtype
dtype(‘int32’)
>>> type(x.dtype)
<\text{type ‘\text{numpy.dtype}’}>
```

**matrix.flags**

Information about the memory layout of the array.
Notes

The `flags` object can be accessed dictionary-like (as in `a.flags[‘WRITEABLE’]`), or by using lowercase attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `UPDATEIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `UPDATEIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be arbitrary if `arr.shape[dim] == 1` or the array has no elements. It does not generally hold that `self.strides[-1] == self.itemsize for C-style contiguous arrays or self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

| C_CONTIGUOUS (C) | This data is in a single, C-style contiguous segment. |
| F_CONTIGUOUS (F) | This data is in a single, Fortran-style contiguous segment. |
| OWN_DATA (O) | The array owns the memory it uses or borrows it from another object. |
| WRITEABLE (W) | The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a `RuntimeError` exception. |
| ALIGNED (A) | The data and all elements are aligned appropriately for the hardware. |
| UPDATEIFCOPY (U) | This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array. |
| FNC | F_CONTIGUOUS and not C_CONTIGUOUS. |
| FORC | F_CONTIGUOUS or C_CONTIGUOUS (one-segment test). |
| BEHAVED (B) | ALIGNED and WRITEABLE. |
| CARRAY (CA) | BEHAVED and C_CONTIGUOUS. |
| FARRAY (FA) | BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS. |
`matrix.flat`
A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

**See Also:**

`flatten`
Return a copy of the array collapsed into one dimension.

`flatiter`

**Examples**

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

`matrix.imag`
The imaginary part of the array.

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

`matrix.itemsize`
Length of one array element in bytes.

**Examples**

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
```

1.6. Standard array subclasses
>>> x.itemsize
16

**matrix.nbytes**
Total bytes consumed by the elements of the array.

**Notes**
Does not include memory consumed by non-element attributes of the array object.

**Examples**

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**matrix.ndim**
Number of array dimensions.

**Examples**

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

**matrix.real**
The real part of the array.

**See Also:**

`numpy.real` equivalent function

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

**matrix.shape**
Tuple of array dimensions.

**Notes**
May be used to “reshape” the array, as long as this would not require a change in the total number of elements.

**Examples**

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
```
Matrix.size

Number of elements in the array.

Equivalent to np.prod(a.shape), i.e., the product of the array’s dimensions.

Examples

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

Matrix.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array `a` is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See Also:

numpy.lib.stride_tricks.as_strided

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```python
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be `(20, 4)`.

Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```
```python
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Methods

- `all([axis, out])` Test whether all matrix elements along a given axis evaluate to True.
- `any([axis, out])` Test whether any array element along a given axis evaluates to True.
- `argmax([axis, out])` Indices of the maximum values along an axis.
- `argmin([axis, out])` Return the indices of the minimum values along an axis.
- `argpartition(kth[, axis, kind, order])` Returns the indices that would partition this array.
- `argsort([axis, kind, order])` Returns the indices that would sort this array.
- `astype(dtype[, order, casting, subok, copy])` Copy of the array, cast to a specified type.
- `byteswap(inplace)` Swap the bytes of the array elements.
- `choose(choices[, out, mode])` Use an index array to construct a new array from a set of choices.
- `clip(a_min, a_max[, out])` Return an array whose values are limited to [a_min, a_max].
- `compress(condition[, axis, out])` Return selected slices of this array along given axis.
- `conj()` Complex-conjugate all elements.
- `conjugate()` Return the complex conjugate, element-wise.
- `copy([order])` Return a copy of the array.
- `cumprod([axis, dtype, out])` Return the cumulative product of the elements along the given axis.
- `cumsum([axis, dtype, out])` Return the cumulative sum of the elements along the given axis.
- `diagonal([offset, axis1, axis2])` Return specified diagonals.
- `dot(b[, out])` Dot product of two arrays.
- `dump(file)` Dump a pickle of the array to the specified file.
- `dumps()` Returns the pickle of the array as a string.
- `fill(value)` Fill the array with a scalar value.
- `flatten([order])` Return a copy of the array collapsed into one dimension.
- `getA()` Return `self` as an `ndarray` object.
- `getA1()` Return `self` as a flattened `ndarray`.
- `getH()` Returns the (complex) conjugate transpose of `self`.
- `getI()` Returns the (multiplicative) inverse of invertible `self`.
- `getT()` Returns the transpose of the matrix.
- `getfield(dtype[, offset])` Returns a field of the given array as a certain type.
- `item(*args)` Copy an element of an array to a standard Python scalar and return it.
- `itemset(*args)` Insert scalar into an array (scalar is cast to array’s dtype, if possible)
- `max([axis, out])` Return the maximum value along an axis.
- `mean([axis, dtype, out])` Returns the average of the matrix elements along the given axis.
- `min([axis, out])` Return the minimum value along an axis.
Table 1.40 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newbyteorder</td>
<td>Return the array with the same data viewed with a different byte order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td>partition</td>
<td>Rearranges the elements in the array in such a way that value of the element in kth position is in its sorted position.</td>
</tr>
<tr>
<td>prod</td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td>ptp</td>
<td>Peak-to-peak (maximum - minimum) value along the given axis.</td>
</tr>
<tr>
<td>put</td>
<td>Set a.flat[n] = values[n] for all n in indices.</td>
</tr>
<tr>
<td>ravel</td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td>repeat</td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td>reshape</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>resize</td>
<td>Return an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>round</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>searchsorted</td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td>setfield</td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td>setflags</td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td>sort</td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td>squeeze</td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td>std</td>
<td>Return the standard deviation of the array elements along the given axis.</td>
</tr>
<tr>
<td>sum</td>
<td>Returns the sum of the matrix elements, along the given axis.</td>
</tr>
<tr>
<td>swapaxes</td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td>take</td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td>tofile</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>tostring</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>transpose</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td>var</td>
<td>Returns the variance of the matrix elements, along the given axis.</td>
</tr>
<tr>
<td>view</td>
<td>New view of array with the same data.</td>
</tr>
</tbody>
</table>

matrix.all (axis=None, out=None)
Test whether all matrix elements along a given axis evaluate to True.

Parameters
See ‘numpy.all’ for complete descriptions

See Also:
numpy.all

Notes
This is the same as ndarray.all, but it returns a matrix object.

Examples
```python
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]])
>>> y = x[0]; y
matrix([[0, 1, 2, 3]])
>>> (x == y)
matrix([[ True, True, True, True],
        [False, False, False, False],
        [False, False, False, False]], dtype=bool)
>>> (x == y).all()
```

1.6. Standard array subclasses
False
>>> (x == y).all(0)
matrix([[False, False, False, False]], dtype=bool)
>>> (x == y).all(1)
matrix([[ True],
        [False],
        [False]], dtype=bool)

matrix.any (axis=None, out=None)
Test whether any array element along a given axis evaluates to True.

Refer to numpy.any for full documentation.

Parameters
axis : int, optional
    Axis along which logical OR is performed
out : ndarray, optional
    Output to existing array instead of creating new one, must have same shape as expected output

Returns
any : bool, ndarray
    Returns a single bool if axis is None; otherwise, returns ndarray

matrix.argmax (axis=None, out=None)
Indices of the maximum values along an axis.

Parameters
See ‘numpy.argmax’ for complete descriptions

See Also:
numpy.argmax

Notes
This is the same as ndarray.argmax, but returns a matrix object where ndarray.argmax would return an ndarray.

Examples
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]])
>>> x.argmax()
11
>>> x.argmax(0)
matrix([[2, 2, 2, 2]])
>>> x.argmax(1)
matrix([[3],
        [3],
        [3]])

matrix.argmin (axis=None, out=None)
Return the indices of the minimum values along an axis.

Parameters
See ‘numpy.argmin’ for complete descriptions.
See Also:

numpy.argmin

Notes
This is the same as ndarray.argmin, but returns a matrix object where ndarray.argmin would return an ndarray.

Examples

```python
>>> x = -np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0, -1, -2, -3],
        [ -4, -5, -6, -7],
        [ -8, -9, -10, -11]])
>>> x.argmin()
11
>>> x.argmin(0)
matrix([[2, 2, 2, 2]])
>>> x.argmin(1)
matrix([[3],
        [3],
        [3]])
```

matrix.argpartition (kth, axis=-1, kind='introselect', order=None)

Returns the indices that would partition this array.

Refer to numpy.argpartition for full documentation. New in version 1.8.0.

See Also:

numpy.argpartition
equivalent function

matrix.argsort (axis=-1, kind='quicksort', order=None)

Returns the indices that would sort this array.

Refer to numpy.argsort for full documentation.

See Also:

numpy.argsort
equivalent function

matrix.astype (dtype, order='K', casting='unsafe', subok=True, copy=True)

Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype
    Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.

- ‘no’ means the data types should not be cast at all.
- ‘equiv’ means only byte-order changes are allowed.
- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

**subok** : bool, optional

If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy** : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the dtype, order, and subok requirements are satisfied, the input array is returned instead of a copy.

**Returns**

- **arr_t** : ndarray

  Unless copy is False and the other conditions for returning the input array are satisfied (see description for copy input parameter), arr_t is a new array of the same shape as the input array, with dtype, order given by dtype, order.

**Raises**

- **ComplexWarning**

  When casting from complex to float or int. To avoid this, one should use a.real.astype(t).

**Examples**

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

**matrix.byteswap(inplace)**

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

**Parameters**

- **inplace** : bool, optional

  If True, swap bytes in-place, default is False.

**Returns**

- **out** : ndarray

  The byteswapped array. If inplace is True, this is a view to self.
Examples

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

matrix.choose (choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to numpy.choose for full documentation.

See Also:

numpy.choose
equivalent function

matrix.clip (a_min, a_max, out=None)

Return an array whose values are limited to [a_min, a_max].

Refer to numpy.clip for full documentation.

See Also:

numpy.clip
equivalent function

matrix.compress (condition, axis=None, out=None)

Return selected slices of this array along given axis.

Refer to numpy.compress for full documentation.

See Also:

numpy.compress
equivalent function

matrix.conj()

Complex-conjugate all elements.

Refer to numpy.conjugate for full documentation.

See Also:

numpy.conjugate
equivalent function

matrix.conjugate()

Return the complex conjugate, element-wise.
Refer to `numpy.conjugate` for full documentation.

See Also:

`numpy.conjugate`

equivalent function

`matrix.copy(order='C')`

Return a copy of the array.

Parameters

- `order`: {'C', 'F', 'A', 'K'}, optional
  Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if `a` is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their `order=` arguments.)

See Also:

`numpy.copy`, `numpy.copyto`

Examples

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

`matrix.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also:

`numpy.cumprod`

equivalent function

`matrix.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also:

`numpy.cumsum`

equivalent function
matrix\texttt{.diagonal}(offset=0, axis1=0, axis2=1)

Return specified diagonals.

Refer to \texttt{numpy.diagonal} for full documentation.

\textbf{See Also:}

\texttt{numpy.diagonal}

equivalent function

matrix\texttt{.dot}(b, out=None)

Dot product of two arrays.

Refer to \texttt{numpy.dot} for full documentation.

\textbf{See Also:}

\texttt{numpy.dot}

equivalent function

\textbf{Examples}

\begin{verbatim}
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2., 2.],
       [2., 2.]])

This array method can be conveniently chained:

>>> a.dot(b).dot(b)
array([[8., 8.],
       [8., 8.]])
\end{verbatim}

matrix\texttt{.dump}(file)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

\textbf{Parameters}

\begin{itemize}
\item \texttt{file} : str
\end{itemize}

A string naming the dump file.

matrix\texttt{.dumps}()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

\textbf{Parameters}

None

matrix\texttt{.fill}(value)

Fill the array with a scalar value.

\textbf{Parameters}

\begin{itemize}
\item \texttt{value} : scalar
\end{itemize}

All elements of \texttt{a} will be assigned this value.

\textbf{Examples}
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([[ 1.,  1.]])

matrix.flatten(order='C')
Return a copy of the array collapsed into one dimension.

Parameters

order: {'C', 'F', 'A'}, optional
Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from a. The default is ‘C’.

Returns

y : ndarray
A copy of the input array, flattened to one dimension.

See Also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])

matrix.getA()
Return self as an ndarray object.
Equivalent to np.asarray(self).

Parameters

None

Returns

ret : ndarray
self as an ndarray

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.getA()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
       dtype=int32)
matrix.getA1()  
Return *self* as a flattened *ndarray*.

Equivalent to *np.asarray(x).ravel()*

**Parameters**

None

**Returns**

*ret*: *ndarray*

*self*, 1-D, as an *ndarray*

**Examples**

```python
g = np.matrix(np.arange(12).reshape((3,4))); g
matrix([[ 0, 1, 2, 3],  
        [ 4, 5, 6, 7],  
        [ 8, 9, 10, 11]])
```

```python
g.getA1()
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

matrix.getH()  
Returns the (complex) conjugate transpose of *self*.

Equivalent to *np.transpose(self)* if *self* is real-valued.

**Parameters**

None

**Returns**

*ret*: *matrix object*

complex conjugate transpose of *self*

**Examples**

```python
g = np.matrix(np.arange(12).reshape((3,4)))
g
matrix([[ 0, 1, 2, 3],  
        [ 4, 5, 6, 7],  
        [ 8, 9, 10, 11]])
```

```python
g = g - 1j*g; g
matrix([[ 0. +0.j, 1. -1.j, 2. -2.j, 3. -3.j],  
```

```python
g.getH()
matrix([[ 0. +0.j,  4. +4.j,  8. +8.j],  
        [ 1. +1.j,  5. +5.j,  9. +9.j],  
        [ 2. +2.j,  6. +6.j, 10.+10.j],  
        [ 3. +3.j,  7. +7.j, 11.+11.j]])
```

matrix.getI()  
Returns the (multiplicative) inverse of invertible *self*.

**Parameters**

None

**Returns**

*ret*: *matrix object*

If *self* is non-singular, *ret* is such that *ret * self == self * ret == np.matrix(np.eye(self[0,:].size)) all return True.
Raises

`numpy.linalg.LinAlgError`: Singular matrix

If `self` is singular.

See Also:

`linalg.inv`

Examples

```python
>>> m = np.matrix('[[1, 2; 3, 4]]'); m
matrix([[1, 2],
        [3, 4]])

>>> m.getI()
matrix([[-2., 1.],
        [1.5, -0.5]])

>>> m.getI() * m
matrix([[ 1., 0.],
        [ 0., 1.]]))
```

`matrix.getT()`

Returns the transpose of the matrix.

Does not conjugate! For the complex conjugate transpose, use `getH`.

Parameters

None

Returns

`ret`: matrix object

The (non-conjugated) transpose of the matrix.

See Also:

transpose, getH

Examples

```python
>>> m = np.matrix('[[1, 2; 3, 4]]')
>>> m
matrix([[1, 2],
        [3, 4]])

>>> m.getT()
matrix([[1, 3],
        [2, 4]])
```

`matrix.getfield` *(dtype, offset=0)*

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

`dtype`: str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

`offset`: int
Number of bytes to skip before beginning the element view.

**Examples**

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j,  0.+0.j],
       [ 0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1.,  0.],
       [ 0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
       [ 0.,  4.]])
```

**matrix.item(*args)**

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

**Notes**

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

**item** is very similar to [args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

**Examples**

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
```

1.6. Standard array subclasses
1
>>> x.item((2, 2))
3

matrix.itemset(*args)
Insert scalar into an array (scalar is cast to array’s dtype, if possible)

There must be at least 1 argument, and define the last argument as item. Then, a.itemset(*args) is equivalent to but faster than a[args] = item. The item should be a scalar value and args must select a single item in the array a.

Parameters
*args : Arguments
   If one argument: a scalar, only used in case a is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes
Compared to indexing syntax, itemset provides some speed increase for placing a scalar into a particular location in an ndarray, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using itemset (and item) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])

matrix.max (axis=None, out=None)
Return the maximum value along an axis.

Parameters
See ‘amax‘ for complete descriptions

See Also:
amax, ndarray.max

Notes
This is the same as ndarray.max, but returns a matrix object where ndarray.max would return an ndarray.

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.max()
matrix.max

Returns the average of the matrix elements along the given axis.

See Also:
	numpy.mean

Notes

Same as ndarray.mean except that, where that returns an ndarray, this returns a matrix object.

Examples

```python
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.mean()
5.5
>>> x.mean(0)
matrix([[ 4.,  5.,  6.,  7.]])
>>> x.mean(1)
matrix([[ 1.5],
        [ 5.5],
        [ 9.5]])
```

matrix.min

Returns the minimum value along an axis.

Parameters

See ‘amin’ for complete descriptions.

See Also:

amin, ndarray.min

Notes

This is the same as ndarray.min, but returns a matrix object where ndarray.min would return an ndarray.

Examples

```python
>>> x = -np.matrix(np.arange(12).reshape((3, 4))); x
matrix([[  0,  -1,  -2,  -3],
        [ -4,  -5,  -6,  -7],
        [ -8,  -9, -10, -11]])
>>> x.min()
-11
>>> x.min(0)
```

1.6. Standard array subclasses
matrix([[ -8,  -9,  -10,  -11]])

>>> x.min(1)
matrix([[ -3],
        [ -7],
        [-11]])

matrix.newbyteorder(new_order='S')
Return the array with the same data viewed with a different byte order.
Equivalent to:
arr.view(arr.dtype.newbyteorder(new_order))
Changes are also made in all fields and sub-arrays of the array data type.

Parameters
new_order : string, optional
    Byte order to force; a value from the byte order specifications above. new_order
can be any of:

    * ‘S’ - swap dtype from current to opposite endian
    * {'<', 'L'} - little endian
    * {'>', 'B'} - big endian
    * {'=', 'N'} - native order
    * {'|', 'I'} - ignore (no change to byte order)

The default value (‘S’) results in swapping the current byte order. The code does a
case-insensitive check on the first letter of new_order for the alternatives above. For
example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

Returns
new_arr : array
    New array object with the dtype reflecting given change to the byte order.

matrix.nonzero()
Return the indices of the elements that are non-zero.
Refer to numpy.nonzero for full documentation.

See Also:

numpy.nonzero
equivalent function

matrix.partition(kth, axis=-1, kind='introselect', order=None)
Rearranges the elements in the array in such a way that value of the element in kth position
is in the position it would be in a sorted array. All elements smaller than the kth element
are moved before this element and all equal or greater are moved behind it. The ordering
of the elements in the two partitions is undefined. New in version 1.8.0.

Parameters
kth : int or sequence of ints
    Element index to partition by. The kth element value will be in its final sorted position
    and all smaller elements will be moved before it and all equal or greater elements
    behind it. The order all elements in the partitions is undefined. If provided with a
    sequence of kth it will partition all elements indexed by kth of them into their sorted
    position at once.

axis : int, optional
Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'introselect'}, optional
         Selection algorithm. Default is ‘introselect’.

order : list, optional
         When a is an array with fields defined, this argument specifies which fields to compare
         first, second, etc. Not all fields need be specified.

See Also:

numpy.partition
         Return a partitioned copy of an array.

argpartition
         Indirect partition.

sort
         Full sort.

Notes
See np.partition for notes on the different algorithms.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

matrix.prod(axis=None, dtype=None, out=None)
Return the product of the array elements over the given axis.

Refer to prod for full documentation.

See Also:

prod, ndarray.prod

Notes
Same as ndarray.prod, except, where that returns an ndarray, this returns a matrix object instead.

Examples

>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]])
>>> x.prod()
0
>>> x.prod(0)
matrix([[  0,  45, 120, 231]])
>>> x.prod(1)
matrix([[  0],
        [ 840],
        [7920]])

1.6. Standard array subclasses
matrix.ptp

```python
matrix.ptp(axis=None, out=None)
```

Peak-to-peak (maximum - minimum) value along the given axis.

Refer to numpy.ptp for full documentation.

See Also:

numpy.ptp

Notes

Same as ndarray.ptp, except, where that would return an ndarray object, this returns a matrix object.

Examples

```python
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.ptp()
11
>>> x.ptp(0)
matrix([[8, 8, 8, 8]])
>>> x.ptp(1)
matrix([[3],
        [3],
        [3]])
```

matrix.put

```python
matrix.put(indices, values, mode='raise')
```

Set a.flat[n] = values[n] for all n in indices.

Refer to numpy.put for full documentation.

See Also:

numpy.put

equivalent function

matrix.ravel

```python
matrix.ravel(order)
```

Return a flattened array.

Refer to numpy.ravel for full documentation.

See Also:

numpy.ravel

equivalent function

ndarray.flat

a flat iterator on the array.

matrix.repeat

```python
matrix.repeat(repeats, axis=None)
```

Repeat elements of an array.

Refer to numpy.repeat for full documentation.

See Also:
**numpy.repeat**

Equivalent function

**matrix.reshape** *(shape, order='C')*

Returns an array containing the same data with a new shape.

Refer to **numpy.reshape** for full documentation.

See Also:

**numpy.reshape**

Equivalent function

**matrix.resize** *(new_shape, refcheck=True)*

Change shape and size of array in-place.

**Parameters**

- **new_shape** : tuple of ints, or n ints
  Shape of resized array.
- **refcheck** : bool, optional
  If False, reference count will not be checked. Default is True.

**Returns**

None

**Raises**

- **ValueError**
  If `a` does not own its own data or references or views to it exist, and the data memory must be changed.

- **SystemError**
  If the `order` keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

**resize**

Return a new array with the specified shape.

**Notes**

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

**Examples**

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
```

---

1.6. Standard array subclasses
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
        [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
        [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ... ValueError: cannot resize an array that has been referenced ...

Unless refcheck is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

matrix.round(decimals=0, out=None)
Return a with each element rounded to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

    numpy.around
    equivalent function

matrix.searchsorted(v, side='left', sorter=None)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see numpy.searchsorted

See Also:

    numpy.searchsorted
    equivalent function

matrix.setfield(val, dtype, offset=0)
Put a value into a specified place in a field defined by a data-type.

Place val into a’s field defined by dtype and beginning offset bytes into the field.

Parameters
    val : object
Value to be placed in field.

dtype : dtype object

Data-type of the field in which to place val.

offset : int, optional

The number of bytes into the field at which to place val.

Returns

None

See Also:

ggetField

Examples

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[ 1.00000000e+000,  1.48219694e-323,  1.48219694e-323],
       [ 1.48219694e-323,  1.00000000e+000,  1.48219694e-323],
       [ 1.48219694e-323,  1.48219694e-323,  1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

matrix.setflags (write=None, align=None, uic=None)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional

Describes whether or not a can be written to.

align : bool, optional

Describes whether or not a is aligned properly for its type.

uic : bool, optional

Describes whether or not a is a copy of another “base” array.
Notes
Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;
ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);
UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```python
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])

>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

>>> y.setflags(write=0, align=0)

>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False

>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

matrix.sort (axis=-1, kind='quicksort', order=None)
Sort an array, in-place.

Parameters
axis : int, optional
  Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm. Default is 'quicksort'.

order : list, optional
  When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

numpy.sort
  Return a sorted copy of an array.
argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes
See sort for notes on the different sorting algorithms.

Examples
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [3, 1]])

Use the order keyword to specify a field to use when sorting a structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', 'S1'), ('y', '<i4')])

matrix.squeeze (axis=None)
Remove single-dimensional entries from the shape of a.

Refer to numpy.squeeze for full documentation.

See Also:

numpy.squeeze
equivalent function

matrix.std (axis=None, dtype=None, out=None, ddof=0)
Return the standard deviation of the array elements along the given axis.

Refer to numpy.std for full documentation.

See Also:

numpy.std

Notes
This is the same as ndarray.std, except that where an ndarray would be returned, a matrix object is returned instead.
Examples

```python
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
>>> x.std()
3.4520525295346629
>>> x.std(0)
matrix([[ 3.26598632,  3.26598632,  3.26598632,  3.26598632]])
>>> x.std(1)
matrix([[ 1.11803399],
         [ 1.11803399],
         [ 1.11803399]])
```

`matrix.sum` *(axis=None, dtype=None, out=None)*

Returns the sum of the matrix elements, along the given axis.

Refer to `numpy.sum` for full documentation.

See Also:

`numpy.sum`

Notes

This is the same as `ndarray.sum`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

Examples

```python
>>> x = np.matrix([[1, 2], [4, 3]])
>>> x.sum()
10
>>> x.sum(axis=0)
matrix([[3],
        [7]])
>>> x.sum(axis=1, dtype='float')
matrix([[ 3.],
        [ 7.]])
>>> out = np.zeros((1, 2), dtype='float')
>>> x.sum(axis=1, dtype='float', out=out)
matrix([[ 3.],
        [ 7.]])
```

`matrix.swapaxes` *(axis1, axis2)*

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also:

`numpy.swapaxes`

equivalent function

`matrix.take` *(indices, axis=None, out=None, mode='raise')*

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.

See Also:
**numpy.take**
equivalent function

**matrix.tofile** *(fid, sep='', format='s')*
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can be recovered using the function fromfile().

**Parameters**
- **fid** : file or str
  An open file object, or a string containing a filename.
- **sep** : str
  Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tostring())`.
- **format** : str
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

**Notes**
This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

**matrix.tolist()**
Return the matrix as a (possibly nested) list.
See `ndarray.tolist` for full documentation.

**See Also:**
`ndarray.tolist`

**Examples**
```python
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```
```python
>>> x.tolist()
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

**matrix.tostring** *(order='C')*
Construct a Python string containing the raw data bytes in the array.
Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

**Parameters**
- **order** : {‘C’, ‘F’, None}, optional
  Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.
Returns

\[ s : \text{str} \]

A Python string exhibiting a copy of \( a \)'s raw data.

Examples

```python
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tobytes()
'b\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00\x00\x00''
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
'b\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00\x00\x00''
```

`matrix.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

`numpy.trace`

equivalent function

`matrix.transpose(*axes)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and \( a \).shape = \((i[0], i[1], \ldots, i[n-2], i[n-1])\), then \( a.transpose() \).shape = \((i[n-1], i[n-2], \ldots, i[1], i[0])\).

Parameters

\[ \text{axes} : \text{None}, \text{tuple of ints, or } n \text{ ints} \]

- None or no argument: reverses the order of the axes.
- tuple of ints: \( i \) in the \( j \)-th place in the tuple means \( a \)'s \( i \)-th axis becomes \( a.transpose() \)'s \( j \)-th axis.
- \( n \) ints: same as an \( n \)-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns

\[ \text{out} : \text{ndarray} \]

View of \( a \), with axes suitably permuted.

See Also:

`ndarray.T`

Array property returning the array transposed.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
```
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])

matrix.var(axis=None, dtype=None, out=None, ddof=0)
Returns the variance of the matrix elements, along the given axis.

Refer to numpy.var for full documentation.

See Also:
numpy.var

Notes
This is the same as ndarray.var, except that where an ndarray would be returned, a matrix object is returned instead.

Examples
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[0, 1, 2, 3],
        [4, 5, 6, 7],
        [8, 9, 10, 11]])
>>> x.var()
11.916666666666666
>>> x.var(0)
matrix([[10.66666667, 10.66666667, 10.66666667, 10.66666667]])
>>> x.var(1)
matrix([[1.25],
        [1.25],
        [1.25]])

matrix.view(dtype=None, type=None)
New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional
Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as a. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the type parameter).

type : Python type, optional
Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes
a.view() is used two different ways:
a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For a.view(some_dtype), if some_dtype has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of a (shown by print(a)). It also depends on exactly how a is stored in memory. Therefore if a is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

Examples

>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:

>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)

Creating a view on a structured array so it can be used in calculations

>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)

Making changes to the view changes the underlying array

>>> xv[0,1] = 20

Using a view to convert an array to a record array:

>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)

Views share data:

>>> x[0] = (9, 10)
>>> z[0]
(9, 10)

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.

>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 2],
       [4, 5]], dtype=[('width', '<i2'), ('length', '<i2')])

numpy.asmatrix(data, dtype=None)
Interpret the input as a matrix.

Unlike matrix, asmatrix does not make a copy if the input is already a matrix or an ndarray. Equivalent to matrix(data, copy=False).

Parameters

data : array_like
   Input data.

Returns

mat : matrix
   data interpreted as a matrix.

Examples

>>> x = np.array([[1, 2], [3, 4]])

>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
       [3, 4]])

numpy.bmat(obj, ldict=None, gdict=None)
Build a matrix object from a string, nested sequence, or array.

Parameters

obj : str or array_like
   Input data. Names of variables in the current scope may be referenced, even if obj is a string.

Returns

out : matrix
   Returns a matrix object, which is a specialized 2-D array.

See Also:

matrix

Examples

>>> A = np.mat("1 1; 1 1")
>>> B = np.mat("2 2; 2 2")
>>> C = np.mat("3 4; 5 6")
>>> D = np.mat("7 8; 9 0")

1.6. Standard array subclasses
All the following expressions construct the same block matrix:

```python
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

```python
>>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

```python
>>> np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

Example 1: Matrix creation from a string

```python
>>> a=mat('1 2 3; 4 5 3')
>>> print (a*a.T).I
[[ 0.2924 -0.1345]
 [-0.1345 0.0819]]
```

Example 2: Matrix creation from nested sequence

```python
>>> mat([[1,5,10],[1.0,3,4j]])
matrix([[ 1.+0.j, 5.+0.j, 10.+0.j],
        [ 1.+0.j, 3.+0.j, 0.+4.j]])
```

Example 3: Matrix creation from an array

```python
>>> mat(random.rand(3,3)).T
matrix([[ 0.7699, 0.7922, 0.3294],
        [ 0.2792, 0.0101, 0.9219],
        [ 0.3398, 0.7571, 0.8197]])
```

### 1.6.3 Memory-mapped file arrays

Memory-mapped files are useful for reading and/or modifying small segments of a large file with regular layout, without reading the entire file into memory. A simple subclass of the ndarray uses a memory-mapped file for the data buffer of the array. For small files, the over-head of reading the entire file into memory is typically not significant, however for large files using memory mapping can save considerable resources.

Memory-mapped-file arrays have one additional method (besides those they inherit from the ndarray): `.flush()` which must be called manually by the user to ensure that any changes to the array actually get written to disk.

**Note:** Memory-mapped arrays use the the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

<table>
<thead>
<tr>
<th>memmap</th>
<th>Create a memory-map to an array stored in a <em>binary</em> file on disk.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memmap.flush()</code></td>
<td>Write any changes in the array to the file on disk.</td>
</tr>
</tbody>
</table>

```python
class numpy.memmap
Create a memory-map to an array stored in a *binary* file on disk.
```
Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. NumPy’s memmap’s are array-like objects. This differs from Python’s `mmap` module, which uses file-like objects.

This subclass of `ndarray` has some unpleasant interactions with some operations, because it doesn’t quite fit properly as a subclass. An alternative to using this subclass is to create the `mmap` object yourself, then create an `ndarray` with `ndarray.__new__` directly, passing the object created in its ‘buffer=’ parameter.

This class may at some point be turned into a factory function which returns a view into an mmap buffer.

**Parameters**

- **filename**: str or file-like object
  
The file name or file object to be used as the array data buffer.

- **dtype**: data-type, optional
  
The data-type used to interpret the file contents. Default is `uint8`.

- **mode**: {'r+', 'r', 'w+', 'c'}, optional
  
The file is opened in this mode:
  
<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td>Open existing file for reading only.</td>
</tr>
<tr>
<td><code>r+</code></td>
<td>Open existing file for reading and writing.</td>
</tr>
<tr>
<td><code>w+</code></td>
<td>Create or overwrite existing file for reading and writing.</td>
</tr>
<tr>
<td><code>c</code></td>
<td>Copy-on-write: assignments affect data in memory, but changes are not saved to disk. The file on disk is read-only.</td>
</tr>
</tbody>
</table>

Default is ‘`r+`’.

- **offset**: int, optional
  
In the file, array data starts at this offset. Since `offset` is measured in bytes, it should normally be a multiple of the byte-size of `dtype`. When `mode` ! = ‘`r`’, even positive offsets beyond end of file are valid; The file will be extended to accommodate the additional data. The default offset is 0.

- **shape**: tuple, optional
  
The desired shape of the array. If `mode` == ‘`r`’ and the number of remaining bytes after `offset` is not a multiple of the byte-size of `dtype`, you must specify `shape`. By default, the returned array will be 1-D with the number of elements determined by file size and data-type.

- **order**: {'C', 'F'}, optional
  
Specify the order of the ndarray memory layout: C (row-major) or Fortran (column-major). This only has an effect if the shape is greater than 1-D. The default order is ‘C’.

**Notes**

The memmap object can be used anywhere an `ndarray` is accepted. Given a memmap `fp`, `isinstance(fp, numpy.ndarray)` returns True.

Memory-mapped arrays use the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

**Examples**

```python
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))
```
This example uses a temporary file so that doctest doesn’t write files to your directory. You would use a ‘normal’
filename.

>>> from tempfile import mkdtemp
>>> import os.path as path

Create a memmap with dtype and shape that matches our data:

>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))

Write data to memmap array:

>>> fp[:] = data[:]

Deletion flushes memory changes to disk before removing the object:

>>> del fp

Load the memmap and verify data was stored:

>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))

Read-only memmap:

>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))

Copy-on-write memmap:

>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))

It’s possible to assign to copy-on-write array, but values are only written into the memory copy of the array, and
not written to disk:

>>> fpc[0,:] = 0
File on disk is unchanged:

```python
>>> fpr
memmap([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]], dtype=float32)
```

Offset into a memmap:

```python
>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([ 4.,  5.,  6.,  7.,  8.,  9., 10., 11.], dtype=float32)
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>(str) Path to the mapped file.</td>
</tr>
<tr>
<td>offset</td>
<td>(int) Offset position in the file.</td>
</tr>
<tr>
<td>mode</td>
<td>(str) File mode.</td>
</tr>
</tbody>
</table>

Methods

- **flush()** Write any changes in the array to the file on disk.

  ```python
  memmap.flush()
  ```

  Write any changes in the array to the file on disk.

  For further information, see `memmap`.

  **Parameters**

  None

  **See Also:**

  `memmap`

- **close** Close the memmap file.

  ```python
  memmap.close()
  ```

  Write any changes in the array to the file on disk.

  For further information, see `memmap`.

  **Parameters**

  None

  **See Also:**

  `memmap`

Example:

```python
>>> a = memmap('newfile.dat', dtype=float, mode='w+', shape=1000)
>>> a[10] = 10.0
>>> a[30] = 30.0
>>> del a
>>> b = fromfile('newfile.dat', dtype=float)
>>> print b[10], b[30]
10.0 30.0
```
```python
>>> a = memmap('newfile.dat', dtype=float)
>>> print a[10], a[30]
10.0 30.0
```

### 1.6.4 Character arrays (numpy.char)

See Also:

*Creating character arrays (numpy.char)*

**Note:** The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of `dtype object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

These are enhanced arrays of either `string_` or `unicode_` type. These arrays inherit from the `ndarray`, but specially-define the operations `+`, `*`, and `%` on a (broadcasting) element-by-element basis. These operations are not available on the standard `ndarray` of character type. In addition, the `chararray` has all of the standard `string` (and `unicode`) methods, executing them on an element-by-element basis. Perhaps the easiest way to create a chararray is to use `self.view(chararray)` where `self` is an `ndarray` of `str` or `unicode` data-type. However, a chararray can also be created using the `numpy.chararray` constructor, or via the `numpy.char.array` function:

<table>
<thead>
<tr>
<th>chararray</th>
<th>Provides a convenient view on arrays of string and unicode values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>core.defchararray.array(obj[, itemsize, ...])</td>
<td>Create a chararray.</td>
</tr>
</tbody>
</table>

**class** `numpy.chararray`

Provides a convenient view on arrays of string and unicode values.

**Note:** The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of `dtype object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

Versus a regular Numpy array of type `str` or `unicode`, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. `endswith`) and infix operators (e.g. `"+", "*", "%"`)

Chararrays should be created using `numpy.char.array` or `numpy.char.asarray`, rather than this constructor directly.

This constructor creates the array, using `buffer` (with `offset` and `strides`) if it is not None. If `buffer` is None, then constructs a new array with `strides` in “C order”, unless both `len(shape) >= 2 and order='Fortran'`, in which case `strides` is in “Fortran order”.

**Parameters**

- `shape`: tuple
  - Shape of the array.
- `itemsize`: int, optional

---

160 Chapter 1. Array objects
Length of each array element, in number of characters. Default is 1.

**unicode** : bool, optional

Are the array elements of type unicode (True) or string (False). Default is False.

**buffer** : int, optional

Memory address of the start of the array data. Default is None, in which case a new array is created.

**offset** : int, optional

Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be $\geq 0$.

**strides** : array_like of ints, optional

Strides for the array (see `ndarray.strides` for full description). Default is None.

**order** : {'C', 'F'}, optional

The order in which the array data is stored in memory: ‘C’ -> “row major” order (the default), ‘F’ -> “column major” (Fortran) order.

**Examples**

```python
>>> charar = np.chararray((3, 3))
>>> charar[:] = 'a'
>>> charar
chararray([[‘a’, ‘a’, ‘a’],
          [‘a’, ‘a’, ‘a’],
          [‘a’, ‘a’, ‘a’]],
dtype=|S1)
```

```python
>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:] = 'abc'
>>> charar
chararray([[‘abc’, ‘abc’, ‘abc’],
          [‘abc’, ‘abc’, ‘abc’],
          [‘abc’, ‘abc’, ‘abc’]],
dtype=|S5’)
```

**Attributes**

- **T** Same as self.transpose(), except that self is returned if self.ndim < 2.
- **base** Base object if memory is from some other object.
- **ctypes** An object to simplify the interaction of the array with the ctypes module.
- **data** Python buffer object pointing to the start of the array’s data.
- **dtype** Data-type of the array’s elements.
- **flags** Information about the memory layout of the array.
- **flat** A 1-D iterator over the array.
- **imag** The imaginary part of the array.
- **itemsize** Length of one array element in bytes.
- **nbytes** Total bytes consumed by the elements of the array.
- **ndim** Number of array dimensions.
- **real** The real part of the array.
- **shape** Tuple of array dimensions.
- **size** Number of elements in the array.
- **strides** Tuple of bytes to step in each dimension when traversing an array.
chararray.\textbf{T}
Same as self.transpose(), except that self is returned if self.ndim < 2.

\textbf{Examples}

\begin{verbatim}
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
\end{verbatim}

chararray.\textbf{base}
Base object if memory is from some other object.

\textbf{Examples}

The base of an array that owns its memory is None:

\begin{verbatim}
>>> x = np.array([1,2,3,4])
>>> x.base is None
True

Slicing creates a view, whose memory is shared with x:

>>> y = x[2:]
>>> y.base is x
True
\end{verbatim}

chararray.\textbf{ctypes}
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

\textbf{Parameters}

None

\textbf{Returns}

c : Python object

Possessing attributes data, shape, strides, etc.

\textbf{See Also:}

typewriter.numpy.ctypeslib

\textbf{Notes}

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- \textit{data}: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary
C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

•shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

•strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

•data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self.as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

•shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

•strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents

  c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents

  c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

chararray.data
Python buffer object pointing to the start of the array’s data.
chararray.

**dtype**

Data-type of the array's elements.

**Parameters**

None

**Returns**

d : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```python
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

chararray.

**flags**

Information about the memory layout of the array.

**Notes**

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercase attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- **UPDATEIFCOPY** can only be set `False`.
- **ALIGNED** can only be set `True` if the data is truly aligned.
- **WRITEABLE** can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be arbitrary if `arr.shape[dim] == 1` or the array has no elements. It does not generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.
Attributes

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CONTIGUOUS (C)</td>
<td>This data is in a single, C-style contiguous segment.</td>
</tr>
<tr>
<td>F_CONTIGUOUS (F)</td>
<td>This data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWN_DATA (O)</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE (W)</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only.</td>
</tr>
<tr>
<td>ALIGNED (A)</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATEIFCOPY (U)</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED (B)</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY (CA)</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY (FA)</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>

`chararray.flat`  
A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

`flatten`  
Return a copy of the array collapsed into one dimension.

`flatiter`  
Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
```

```python
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
```

```python
>>> x.flat[3]
4
```

```python
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

**chararray.imag**
The imaginary part of the array.

**Examples**

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**chararray.itemsize**
Length of one array element in bytes.

**Examples**

```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**chararray.nbytes**
Total bytes consumed by the elements of the array.

**Notes**
Does not include memory consumed by non-element attributes of the array object.

**Examples**

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**chararray.ndim**
Number of array dimensions.

**Examples**

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
```
```python
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

chararray.real
The real part of the array.

See Also:
	numpy.real
equivalent function

Examples
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')

chararray.shape
Tuple of array dimensions.

Notes
May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

chararray.size
Number of elements in the array.
Equivalent to `np.prod(a.shape)`, i.e., the product of the array’s dimensions.

Examples
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```
chararray.strides

TUPLE OF BYTES TO STEP IN EACH DIMENSION WHEN TRAVERSING AN ARRAY.

The byte offset of element (i[0], i[1], ..., i[n]) in an array a is:

\[
\text{offset} = \sum (\text{np.array}(i) \times a.\text{strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See Also:

numpy.lib.stride_tricks.as_strided

Notes

Imagine an array of 32-bit integers (each 4 bytes):

\[
x = \text{np.array([[0, 1, 2, 3, 4],}
\quad [5, 6, 7, 8, 9]], \text{dtype=\text{np.int32}})
\]

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Methods

- `astype(dtype[, order, casting, subok, copy])` Copy of the array, cast to a specified type.
- `copy([order])` Return a copy of the array.
<table>
<thead>
<tr>
<th>Method (Parameters)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count(sub[, start, end])</code></td>
<td>Returns an array with the number of non-overlapping occurrences of substring <code>sub</code> in <code>self</code>.</td>
</tr>
<tr>
<td><code>decode((encoding, errors))</code></td>
<td>Calls <code>str.decode</code> element-wise.</td>
</tr>
<tr>
<td><code>dump(file)</code></td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td><code>dumps()</code></td>
<td>Returns the pickle of the array as a string.</td>
</tr>
<tr>
<td><code>encode((encoding, errors))</code></td>
<td>Calls <code>str.encode</code> element-wise.</td>
</tr>
<tr>
<td><code>endswith((suffix[, start, end])</code></td>
<td>Returns a boolean array which is <code>True</code> where the string element is found.</td>
</tr>
<tr>
<td><code>expandtabs((tabsize))</code></td>
<td>Return a copy of each string element where all tab characters are replaced by one or more spaces.</td>
</tr>
<tr>
<td><code>fill(value)</code></td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td><code>find(sub[, start, end])</code></td>
<td>For each element, return the lowest index in the string where substring <code>sub</code> is found.</td>
</tr>
<tr>
<td><code>flatten((order))</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>index(sub[, start, end])</code></td>
<td>Like <code>find</code>, but raises <code>ValueError</code> when the substring is not found.</td>
</tr>
<tr>
<td><code>isalnum()</code></td>
<td>Returns true for each element if all characters in the string are alphanumeric and there is at least one character.</td>
</tr>
<tr>
<td><code>isalpha()</code></td>
<td>Returns true for each element if all characters in the string are alphabetic and there is at least one character.</td>
</tr>
<tr>
<td><code>isdecimal()</code></td>
<td>Returns true for each element in <code>self</code>, return True if there are digits and there is at least one digit.</td>
</tr>
<tr>
<td><code>isdigit()</code></td>
<td>Returns true for each element if all characters in the string are digits and there is at least one digit.</td>
</tr>
<tr>
<td><code>islower()</code></td>
<td>Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character.</td>
</tr>
</tbody>
</table>
| `isnumeric()`                                                                     | Returns true for each element in `self`, return True if there are only numeric characters.
| `isspace()`                                                                       | Returns true for each element if there are only whitespace characters in the string and there is at least one whitespace character. |
| `isupper()`                                                                       | Returns true for each element if all cased characters in the string is titlecased and there is at least one cased character. |
| `item(*args)`                                                                     | Copy an element of an array to a standard Python scalar and return it.      |
| `join(seq)`                                                                       | Return a string which is the concatenation of the strings in the sequence `seq`. |
| `ljust(width[, fillchar])`                                                         | Return an array with the elements of `self` left-justified in a string of length `width`. |
| `lower()`                                                                         | Return an array with the elements of `self` converted to lowercase.          |
| `lstrip([chars])`                                                                  | For each element in `self`, return a copy with the leading characters removed. |
| `nonzero()`                                                                       | Return the indices of the elements that are non-zero.                       |
| `put(indices, values[, mode])`                                                     | Set `a.flat[n] = values[n]` for all `n` in `indices`.                       |
| `ravel(order)`                                                                    | Return a flattened array.                                                   |
| `repeat(repeats[, axis])`                                                          | Repeat elements of an array.                                                |
| `replace(old, new[, count])`                                                       | For each element in `self`, return a copy of the string with all occurrences of substring `old` replaced by `new`. |
| `reshape(shape[, order])`                                                          | Returns an array containing the same data with a new shape.                 |
| `resize(new_shape[, refcheck])`                                                     | Change shape and size of array in-place.                                    |
| `rfind(sub[, start, end])`                                                         | For each element in `self`, return the highest index in the string where substring `sub` is found. |
| `rindex(sub[, start, end])`                                                         | Like `rfind`, but raises `ValueError` when the substring `sub` is not found. |
| `rjust(width[, fillchar])`                                                         | Return an array with the elements of `self` right-justified in a string of length `width`. |
| `rsplit((sep, maxsplit))`                                                           | For each element in `self`, return a list of the words in the string, using `sep` as the delimiter. |
| `rstrip([chars])`                                                                  | For each element in `self`, return a copy with the trailing characters removed. |
| `searchsorted(v[, side, sorter])`                                                   | Find indices where elements of `v` should be inserted in `a` to maintain order. |
| `setfield(val, dtype[, offset])`                                                   | Put a value into a specified place in a field defined by a data-type.      |
| `setflags([write, align, uic])`                                                    | Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.          |
| `sort([axis, kind, order])`                                                        | Sort an array, in-place.                                                   |
| `split(sep, maxsplit)`                                                             | For each element in `self`, return a list of the words in the string, using `sep` as the delimiter. |
| `splitlines([keepends])`                                                           | For each element in `self`, return a list of the lines in the element, breaking at line boundaries. |
| `squeeze(axis)`                                                                   | Remove single-dimensional entries from the shape of `a`.                   |
| `startswith(prefix[, start, end])`                                                 | Returns a boolean array which is `True` where the string element is contained within `prefix`. |
| `strip([chars])`                                                                   | For each element in `self`, return a copy with the leading and trailing characters removed. |
| `swapaxes(axis1, axis2)`                                                           | Return a view of the array with `axis1` and `axis2` interchanged.            |
| `swapcase()`                                                                      | For each element in `self`, return a copy of the string with uppercase characters converted to lowercase. |
| `take(indices[, axis, out, mode])`                                                 | Return an array formed from the elements of `a` at the given indices.       |
### Table 1.45 – continued

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>title()</code></td>
<td>For each element in <code>self</code>, return a titlecased version of the string: words start with uppercase.</td>
</tr>
<tr>
<td><code>tofile(fid[, sep, format])</code></td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td><code>tolist()</code></td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td><code>tostring([order])</code></td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td><code>translate(table[, deletechars])</code></td>
<td>For each element in <code>self</code>, return a copy of the string where all characters occurring in <code>deletechars</code> are removed, and the remaining characters have been mapped through the given translation table.</td>
</tr>
<tr>
<td><code>transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td><code>upper()</code></td>
<td>Return an array with the elements of <code>self</code> converted to uppercase.</td>
</tr>
<tr>
<td><code>view([dtype, type])</code></td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td><code>zfill(width)</code></td>
<td>Return the numeric string left-filled with zeros in a string of length <code>width</code>.</td>
</tr>
</tbody>
</table>

**chararray.** **astype** *(dtype, order='K', casting='unsafe', subok=True, copy=True)*

Copy of the array, cast to a specified type.

**Parameters**

- **dtype** : str or dtype
  - Typecode or data-type to which the array is cast.

- **order** : {'C', 'F', 'A', 'K'}, optional
  - Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

- **casting** : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
  - Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.
    - ‘no’ means the data types should not be cast at all.
    - ‘equiv’ means only byte-order changes are allowed.
    - ‘safe’ means only casts which can preserve values are allowed.
    - ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
    - ‘unsafe’ means any data conversions may be done.

- **subok** : bool, optional
  - If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

- **copy** : bool, optional
  - By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

**Returns**

- **arr_t** : ndarray
  - Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), `arr_t` is a new array of the same shape as the input array, with `dtype`, `order` given by `dtype`, `order`.

**Raises**

- **ComplexWarning**
When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

**Examples**

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([1., 2., 2.5])
>>> x.astype(int)
array([1, 2, 2])
```

**chararray.copy(order='C')**

Return a copy of the array.

**Parameters**

- **order**: {'C', 'F', 'A', 'K'}, optional
  Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if `a` is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their order= arguments.)

**See Also:**

numpy.copy, numpy.copyto

**Examples**

```python
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy()
>>> x.fill(0)
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
>>> y.flags['C_CONTIGUOUS']
True
```

**chararray.count(sub, start=0, end=None)**

Returns an array with the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`.

**See Also:**

char.count

**chararray.decode(encoding=None, errors=None)**

Calls `str.decode` element-wise.

**See Also:**

char.decode
chararray.dump(file)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters
file : str
A string naming the dump file.

chararray.dumps()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None

chararray.encode(encoding=None, errors=None)
Calls str.encode element-wise.

See Also:
char.encode

chararray.endswith(suffix, start=0, end=None)
Returns a boolean array which is True where the string element in self ends with suffix, otherwise False.

See Also:
char.endswith

chararray.expandtabs(tabsize=8)
Return a copy of each string element where all tab characters are replaced by one or more spaces.

See Also:
char.expandtabs

chararray.fill(value)
Fill the array with a scalar value.

Parameters
value : scalar
All elements of a will be assigned this value.

Examples
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])

chararray.find(sub, start=0, end=None)
For each element, return the lowest index in the string where substring sub is found.

See Also:
char.find

chararray.flatten(order='C')
Return a copy of the array collapsed into one dimension.
Parameters

order : {'C', 'F', 'A'}, optional

Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the
C/Fortran ordering from a. The default is 'C'.

Returns

y : ndarray

A copy of the input array, flattened to one dimension.

See Also:

ravel

Return a flattened array.

flat

A 1-D flat iterator over the array.

Examples

```python
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

chararray.getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the
given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits
in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with
a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the
array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```
chararray.index(sub, start=0, end=None)
   Like find, but raises ValueError when the substring is not found.

   See Also:
   char.index

chararray.isalnum()
   Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

   See Also:
   char.isalnum

chararray.isalpha()
   Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

   See Also:
   char.isalpha

chararray.isdecimal()
   For each element in self, return True if there are only decimal characters in the element.

   See Also:
   char.isdecimal

chararray.isdigit()
   Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

   See Also:
   char.isdigit

chararray.islower()
   Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

   See Also:
   char.islower

chararray.isnumeric()
   For each element in self, return True if there are only numeric characters in the element.

   See Also:
   char.isnumeric

chararray.isspace()
   Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

   See Also:
   char.isspace

chararray.istitle()
   Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

   See Also:
char.istitle

chararray.isupper()

Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

See Also:

char.isupper

chararray.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (\(a.size == 1\)), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

\(z\): Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of \(a\) is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to \(a[args]\), except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

Examples

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

chararray.join(seq)

Return a string which is the concatenation of the strings in the sequence seq.

See Also:

char.join
**chararray.ljust**(width, fillchar=’ ‘)
Return an array with the elements of self left-justified in a string of length width.

**See Also:**
char.ljust

**chararray.lower()**
Return an array with the elements of self converted to lowercase.

**See Also:**
char.lower

**chararray.lstrip**(chars=None)
For each element in self, return a copy with the leading characters removed.

**See Also:**
char.lstrip

**chararray.nonzero()**
Return the indices of the elements that are non-zero.
Refer to numpy.nonzero for full documentation.

**See Also:**
numpy.nonzero
equivalent function

**chararray.put**(indices, values, mode=’raise’)
Set a.flat[n] = values[n] for all n in indices.
Refer to numpy.put for full documentation.

**See Also:**
numpy.put
equivalent function

**chararray.ravel**(order)
Return a flattened array.
Refer to numpy.ravel for full documentation.

**See Also:**
numpy.ravel
equivalent function

**ndarray.flat**
a flat iterator on the array.

**chararray.repeat**(repeats, axis=None)
Repeat elements of an array.
Refer to numpy.repeat for full documentation.

**See Also:**
numpy.repeat
equivalent function
chararray.replace(old, new, count=None)

For each element in self, return a copy of the string with all occurrences of substring old replaced by new.

See Also:

char.replace

cchararray.reshape(shape, order='C')

Returns an array containing the same data with a new shape.

Refer to numpy.reshape for full documentation.

See Also:

numpy.reshape
equivalent function

cchararray.resize(new_shape, refcheck=True)

Change shape and size of array in-place.

Parameters

new_shape : tuple of ints, or n ints
    Shape of resized array.

refcheck : bool, optional
    If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError
    If a does not own its own data or references or views to it exist, and the data memory
    must be changed.

SystemError
    If the order keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

resize
    Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another
Python object and then reallocate the memory. However, reference counts can increase in other ways so if
you are sure that you have not shared the memory for this array with another Python object, then you may
safely set refcheck to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-
shaped:
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
        [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
        [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
        [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ...
ValueError: cannot resize an array that has been referenced ...

Unless refcheck is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

cchararray.rfind(sub, start=0, end=None)
For each element in self, return the highest index in the string where substring sub is found, such that sub is contained within [start, end].

See Also:

c.char.rfind
cchararray.rindex(sub, start=0, end=None)
Like rfind, but raises ValueError when the substring sub is not found.

See Also:

c.char.rindex
cchararray.rjust(width, fillchar=' ')
Return an array with the elements of self right-justified in a string of length width.

See Also:

c.char.rjust
cchararray.rsplit(sep=None, maxsplit=None)
For each element in self, return a list of the words in the string, using sep as the delimiter string.

See Also:
char.rsplit

chararray.rstrip( chars=None)
For each element in self, return a copy with the trailing characters removed.

See Also:
char.rsplit

chararray.searchsorted (v, side='left', sorter=None)
Find indices where elements of v should be inserted in a to maintain order.
For full documentation, see numpy.searchsorted

See Also:

numpy.searchsorted
equivalent function

chararray.setfield( val, dtype, offset=0)
Put a value into a specified place in a field defined by a data-type.
Place val into a’s field defined by dtype and beginning offset bytes into the field.

Parameters
val : object
    Value to be placed in field.
dtype : dtype object
    Data-type of the field in which to place val.
offset : int, optional
    The number of bytes into the field at which to place val.

Returns
None

See Also:
getfield

Examples
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[ 1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
       [ 1.48219694e-323, 1.00000000e+000, 1.48219694e-323],
       [ 1.48219694e-323, 1.48219694e-323, 1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1., 0., 0.],
       [ 1., 0., 0.],
       [ 1., 0., 0.]])
chararray\[0., 1., 0.],
[0., 0., 1.]]\]

```
chararray.setflags(write=None, align=None, uic=None)
```

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below).

The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

- **write**: bool, optional
  Describes whether or not a can be written to.

- **align**: bool, optional
  Describes whether or not a is aligned properly for its type.

- **uic**: bool, optional
  Describes whether or not a is a copy of another “base” array.

**Notes**

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

- **WRITEABLE** (W) the data area can be written to;
- **ALIGNED** (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);
- **UPDATEIFCOPY** (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

**Examples**

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
```

```
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

```
>>> y.setflags(write=0, align=0)
```

```
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
```
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True

cchararray.sort (axis=-1, kind='quicksort', order=None)
Sort an array, in-place.

Parameters
axis : int, optional
    Axis along which to sort. Default is -1, which means sort along the last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm. Default is 'quicksort'.
order : list, optional
    When a is an array with fields defined, this argument specifies which fields to compare
    first, second, etc. Not all fields need be specified.

See Also:

numpy.sort
    Return a sorted copy of an array.
argsort
    Indirect sort.
lexsort
    Indirect stable sort on multiple keys.
searchsorted
    Find elements in sorted array.
partition
    Partial sort.

Notes
See sort for notes on the different sorting algorithms.

Examples
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])

Use the order keyword to specify a field to use when sorting a structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', 'S1'), ('y', '<i4')])

1.6. Standard array subclasses
chararray.split(sep=None, maxsplit=None)
For each element in self, return a list of the words in the string, using sep as the delimiter string.

See Also:
char.split

chararray.splitlines(keepends=None)
For each element in self, return a list of the lines in the element, breaking at line boundaries.

See Also:
char.splitlines

chararray.squeeze(axis=None)
Remove single-dimensional entries from the shape of a.
Refer to numpy.squeeze for full documentation.

See Also:
numpy.squeeze
equivalent function

chararray.startswith(prefix, start=0, end=None)
Returns a boolean array which is True where the string element in self starts with prefix, otherwise False.

See Also:
char.startswith

chararray.strip(chars=None)
For each element in self, return a copy with the leading and trailing characters removed.

See Also:
char.strip

chararray.swapaxes(axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.
Refer to numpy.swapaxes for full documentation.

See Also:
numpy.swapaxes
equivalent function

chararray.swapcase()
For each element in self, return a copy of the string with uppercase characters converted to lowercase and vice versa.

See Also:
char.swapcase

chararray.take(indices, axis=None, out=None, mode='raise')
Return an array formed from the elements of a at the given indices.

See Also:
numpy.take for full documentation.
**numpy.take**
equivalent function

chararray.title()
For each element in `self`, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

See Also:
char.title

cchararray.tofile(fid, sep='', format='%s')
Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of `a`. The data produced by this method can be recovered using the function fromfile().

Parameters
- **fid**: file or str
  An open file object, or a string containing a filename.
- **sep**: str
  Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tostring())`.
- **format**: str
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes
This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

cchararray.tolist()
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
- none

Returns
- **y**: list
  The possibly nested list of array elements.

Notes
The array may be recreated, `a = np.array(a.tolist())`.

Examples
```python
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
```
>>> list(a)
[array([1, 2]), array([3, 4])]

```python
d >> a.tolist()
d [[1, 2], [3, 4]]
```

chararray.tostring(order='C')

Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

**Parameters**

- **order**: {'C', 'F', None}, optional
  - Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

- **s**: str
  - A Python string exhibiting a copy of a’s raw data.

**Examples**

```python
d >>> x = np.array([[0, 1], [2, 3]])
d >>> x.tostring()
d '\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x03\x00\x00\x00'
d >>> x.tostring('C') == x.tostring()
d True
d >>> x.tostring('F')
d '\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00'
```

chararray.translate(table, deletechars=None)

For each element in self, return a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table.

**See Also:**

char.translate

chararray.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

**Parameters**

- **axes**: None, tuple of ints, or n ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
  - n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)
Returns

```python
out : ndarray
```
View of a, with axes suitably permuted.

See Also:

```python
ndarray.T
```
Array property returning the array transposed.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```python
chararray.upper()
```
Return an array with the elements of self converted to uppercase.

See Also:

```python
char.upper
```

```python
chararray.view(dtype=None, type=None)
```
New view of array with the same data.

Parameters

- **dtype**: data-type or ndarray sub-class, optional
  Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as a. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the type parameter).

- **type**: Python type, optional
  Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

- a.view() is used two different ways:
  - a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.
  - a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarraysubclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

  For a.view(some_dtype), if some_dtype has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

**Examples**

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```python
>>> y = x.view(dtype=np.int16, type=np.matrix)
```

```python
>>> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```python
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
```

Making changes to the view changes the underlying array

```python
>>> xv[0,1] = 20
>>> print x
[(1, 20) (3, 4)]
```

Using a view to convert an array to a record array:

```python
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```python
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```python
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
```
chararray.zfill(width)
Return the numeric string left-filled with zeros in a string of length width.

See Also:
char.zfill

numpy.core.defchararray.array(obj, itemsize=None, copy=True, unicode=None, order=None)
Create a chararray.

Note: This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type string_ or unicode_ and use the free functions in numpy.char for fast vectorized string operations instead.

Versus a regular Numpy array of type str or unicode, this class adds the following functionality:
1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. str.endswith) and infix operators (e.g. +, *, %)

Parameters
obj : array of str or unicode-like
itemsize : int, optional
    itemsize is the number of characters per scalar in the resulting array. If itemsize is None, and obj is an object array or a Python list, the itemsize will be automatically determined. If itemsize is provided and obj is of type str or unicode, then the obj string will be chunked into itemsize pieces.
copy : bool, optional
    If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (itemsize, unicode, order, etc.).
unicode : bool, optional
    When true, the resulting chararray can contain Unicode characters, when false only 8-bit characters. If unicode is None and obj is one of the following:
    • a chararray,
    • an ndarray of type str or unicode
    • a Python str or unicode object,
    then the unicode setting of the output array will be automatically determined.
order : {'C', 'F', 'A'}, optional
    Specify the order of the array. If order is ‘C’ (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is ‘F’, then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is ‘A’, then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).
Another difference with the standard ndarray of str data-type is that the chararray inherits the feature introduced by Numarray that white-space at the end of any element in the array will be ignored on item retrieval and comparison operations.

### 1.6.5 Record arrays (`numpy.rec`)

See Also:

*Creating record arrays (`numpy.rec`), Data type routines, Data type objects (`dtype`).*

NumPy provides the `recarray` class which allows accessing the fields of a record/structured array as attributes, and a corresponding scalar data type object `record`.

| recarray | Construct an ndarray that allows field access using attributes. |
| record | A data-type scalar that allows field access as attribute lookup. |

#### class `numpy.recarray`

Construct an ndarray that allows field access using attributes.

Arrays may have a data-types containing fields, analogous to columns in a spread sheet. An example is `[(x, int), (y, float)]`, where each entry in the array is a pair of `(int, float)`. Normally, these attributes are accessed using dictionary lookups such as `arr['x']` and `arr['y']`. Record arrays allow the fields to be accessed as members of the array, using `arr.x` and `arr.y`.

**Parameters**

- **shape**: tuple
  
  Shape of output array.

- **dtype**: data-type, optional
  
  The desired data-type. By default, the data-type is determined from `formats`, `names`, `titles`, `aligned` and `byteorder`.

- **formats**: list of data-types, optional
  
  A list containing the data-types for the different columns, e.g. `['i4', 'f8', 'i4']`. `formats` does not support the new convention of using types directly, i.e. `(int, float, int)`. Note that `formats` must be a list, not a tuple. Given that `formats` is somewhat limited, we recommend specifying `dtype` instead.

- **names**: tuple of str, optional
  
  The name of each column, e.g. `('x', 'y', 'z')`.

- **buf**: buffer, optional
  
  By default, a new array is created of the given shape and data-type. If `buf` is specified and is an object exposing the buffer interface, the array will use the memory from the existing buffer. In this case, the `offset` and `strides` keywords are available.

**Returns**

- **rec**: recarray
  
  Empty array of the given shape and type.

**Other Parameters**

- **titles**: tuple of str, optional
Aliases for column names. For example, if `names` were (`'x'`, `'y'`, `'z'`) and `titles` is (`'x_coordinate'`, `'y_coordinate'`, `'z_coordinate'`), then `arr['x']` is equivalent to both `arr.x` and `arr.x_coordinate`.

`byteorder` : {'<', '>', '='}, optional

Byte-order for all fields.

`aligned` : bool, optional

Align the fields in memory as the C-compiler would.

`strides` : tuple of ints, optional

Buffer (`buf`) is interpreted according to these strides (strides define how many bytes each array element, row, column, etc. occupy in memory).

`offset` : int, optional

Start reading buffer (`buf`) from this offset onwards.

`order` : {'C', 'F'}, optional

Row-major or column-major order.

See Also:

rec.fromrecords

Construct a record array from data.

record

Fundamental data-type for recarray.

format_parser

determine a data-type from formats, names, titles.

Notes

This constructor can be compared to empty: it creates a new record array but does not fill it with data. To create a record array from data, use one of the following methods:

1. Create a standard ndarray and convert it to a record array, using `arr.view(np.recarray)`

2. Use the `buf` keyword.

3. Use `np.rec.fromrecords`.

Examples

Create an array with two fields, `x` and `y`:

```python
>>> x = np.array([[1.0, 2], [3.0, 4]], dtype=[('x', float), ('y', int)])
>>> x
array([[ 1.,  2],
       [ 3.,  4]],
      dtype=[('x', '<f8'), ('y', '<i4')])
>>> x['x']
array([ 1.,  3.])
```

View the array as a record array:

```python
>>> x = x.view(np.recarray)
```
Create a new, empty record array:

```python
>>> np.recarray((2,),
            ... dtype=[('x', int), ('y', float), ('z', int)])
rec.array([(-1073741821, 1.2249118382103472e-301, 24547520),
           (3471280, 1.2134086255804012e-316, 0)],
           dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
```

**Attributes**

- **T**
  Same as self.transpose(), except that self is returned if self.ndim < 2.

- **base**
  Base object if memory is from some other object.

- **ctypes**
  An object to simplify the interaction of the array with the ctypes module.

- **data**
  Python buffer object pointing to the start of the array’s data.

- **dtype**
  Data-type of the array’s elements.

- **flags**
  Information about the memory layout of the array.

- **flat**
  A 1-D iterator over the array.

- **imag**
  The imaginary part of the array.

- **itemsize**
  Length of one array element in bytes.

- **nbytes**
  Total bytes consumed by the elements of the array.

- **ndim**
  Number of array dimensions.

- **real**
  The real part of the array.

- **shape**
  Tuple of array dimensions.

- **size**
  Number of elements in the array.

- **strides**
  Tuple of bytes to step in each dimension when traversing an array.

**Examples**

```python
>>> x = np.array([[1., 2.], [3., 4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1., 2., 3., 4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

**recarray.base**
Base object if memory is from some other object.

**Examples**

The base of an array that owns its memory is None:
Recastarray ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
c : Python object
Possessing attributes data, shape, strides, etc.

See Also:
numpy.ctypeslib

Notes
Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

•data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

•shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

•strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

•data_as(obj): Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

•shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

•strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that
is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

**Examples**

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]]))
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
<ctypes.LP_c_longlong object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
<ctypes.LP_c_longlong object at 0x01F01300>
```

recarray.**data**
Python buffer object pointing to the start of the array’s data.

recarray.**dtype**
Data-type of the array’s elements.

**Parameters**
None

**Returns**

d : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```python
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
```

recarray.**flags**
Information about the memory layout of the array.
Notes

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercase attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be arbitrary if `arr.shape[dim] == 1` or the array has no elements. It does not generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Attributes

<table>
<thead>
<tr>
<th>C_CONTIGUOUS (C)</th>
<th>This data is in a single, C-style contiguous segment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_CONTIGUOUS (F)</td>
<td>This data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWN-DATA (O)</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE (W)</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.</td>
</tr>
<tr>
<td>ALIGNED (A)</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATEIFCOPY (U)</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED (B)</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY (CA)</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY (FA)</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>
recarray.flat
A 1-D iterator over the array.
This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:
flatten
Return a copy of the array collapsed into one dimension.

Examples
```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:
```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
```

```python
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

recarray.imag
The imaginary part of the array.

Examples
```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

recarray.itemsize
Length of one array element in bytes.

Examples
```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
```

```python
>>> x = np.array([1,2,3], dtype=np.complex128)
```
>>> x.itemsize
16

recarray.nbytes
Total bytes consumed by the elements of the array.

Notes
Does not include memory consumed by non-element attributes of the array object.

Examples

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

recarray.ndim
Number of array dimensions.

Examples

```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

recarray.real
The real part of the array.

See Also:

numpy.real
equivalent function

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
```

recarray.shape
Tuple of array dimensions.

Notes
May be used to “reshape” the array, as long as this would not require a change in the total number of elements.

Examples

```python
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
```
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged

recarray.size
Number of elements in the array.
Equivalent to np.prod(a.shape), i.e., the product of the array’s dimensions.

Examples

>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30

recarray.strides
Tuple of bytes to step in each dimension when traversing an array.
The byte offset of element (i[0], i[1], ..., i[n]) in an array a is:
offset = sum(np.array(i) * a.strides)
A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See Also:
numpy.lib.stride_tricks.as_strided

Notes
Imagine an array of 32-bit integers (each 4 bytes):

x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

Examples

>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```python
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=np.sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = np.sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

**Methods**

- `all([axis, out])`  
  Returns True if all elements evaluate to True.

- `any([axis, out])`  
  Returns True if any of the elements of `a` evaluate to True.

- `argmax([axis, out])`  
  Return indices of the maximum values along the given axis.

- `argmin([axis, out])`  
  Return indices of the minimum values along the given axis of `a`.

- `argpartition(kth[, axis, kind, order])`  
  Returns the indices that would partition this array.

- `argsort([axis, kind, order])`  
  Returns the indices that would sort this array.

- `astype(dtype[, order, casting, subok, copy])`  
  Copy of the array, cast to a specified type.

- `byteswap(inplace)`  
  Swap the bytes of the array elements

- `choose(choices[, out, mode])`  
  Use an index array to construct a new array from a set of choices.

- `clip(a_min, a_max[, out])`  
  Return an array whose values are limited to `[a_min, a_max]`.

- `compress(condition[, axis, out])`  
  Return selected slices of this array along given axis.

- `conj()`  
  Complex-conjugate all elements.

- `conjugate()`  
  Return the complex conjugate, element-wise.

- `copy([order])`  
  Return a copy of the array.

- `cumprod([axis, dtype, out])`  
  Return the cumulative product of the elements along the given axis.

- `cumsum([axis, dtype, out])`  
  Return the cumulative sum of the elements along the given axis.

- `diagonal([offset, axis1, axis2])`  
  Return specified diagonals.

- `dot(b[, out])`  
  Dot product of two arrays.

- `dump(file)`  
  Dump a pickle of the array to the specified file.

- `dumps()`  
  Returns the pickle of the array as a string.

- `field(attr[, val])`  
  Fill the array with a scalar value.

- `flatten([order])`  
  Return a copy of the array collapsed into one dimension.

- `getfield(dtype[, offset])`  
  Returns a field of the given array as a certain type.

- `item(*args)`  
  Copy an element of an array to a standard Python scalar and return it.

- `itemset(*args)`  
  Insert scalar into an array (scalar is cast to array’s dtype, if possible)

- `max([axis, out])`  
  Return the maximum along a given axis.

- `mean([axis, dtype, out])`  
  Returns the average of the array elements along given axis.

- `min([axis, dtype, out])`  
  Return the minimum along a given axis.

- `newbyteorder([new_order])`  
  Return the array with the same data viewed with a different byte order.

- `nonzero()`  
  Return the indices of the elements that are non-zero.

- `partition(kth[, axis, kind, order])`  
  Rearranges the elements in the array in such a way that value of the element in kth position

- `prod([axis, dtype, out])`  
  Return the product of the array elements over the given axis

---

**1.6. Standard array subclasses**
Table 1.48 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptp([axis, out])</td>
<td>Peak to peak (maximum - minimum) value along a given axis.</td>
</tr>
<tr>
<td>put(indices, values[, mode])</td>
<td>Set a.flat[n] = values[n] for all n in indices.</td>
</tr>
<tr>
<td>ravel([order])</td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td>repeat(repeats[, axis])</td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td>reshape(shape[, order])</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>resize(new_shape[, refcheck])</td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td>round([decimals, out])</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td>searchsorted(v[, side, sorter])</td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td>setfield(val, dtype[, offset])</td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td>setflags([write, align, uic])</td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td>sort([axis, kind, order])</td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td>squeeze(axis)</td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td>std([axis, dtype, out, ddof])</td>
<td>Return the standard deviation of the array elements along given axis.</td>
</tr>
<tr>
<td>sum([axis, dtype, out])</td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td>swapaxes(axis1, axis2)</td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td>take(indices[, axis, out, mode])</td>
<td>Return an array formed from the elements of a at the given indices.</td>
</tr>
<tr>
<td>tofile(fid[, sep, format])</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>tolist()</td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
<tr>
<td>tostring([order])</td>
<td>Construct a Python string containing the raw data bytes in the array.</td>
</tr>
<tr>
<td>trace([offset, axis1, axis2, dtype, out])</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td>transpose(*axes)</td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
<tr>
<td>var([axis, dtye, out, ddof])</td>
<td>Returns the variance of the array elements, along given axis.</td>
</tr>
<tr>
<td>view([dtype, type])</td>
<td></td>
</tr>
</tbody>
</table>

```
recarray.all (axis=None, out=None)
   Returns True if all elements evaluate to True.
   Refer to numpy.all for full documentation.
   See Also:
   numpy.all
      equivalent function
```

```
recarray.any (axis=None, out=None)
   Returns True if any of the elements of a evaluate to True.
   Refer to numpy.any for full documentation.
   See Also:
   numpy.any
      equivalent function
```

```
recarray.argmax (axis=None, out=None)
   Return indices of the maximum values along the given axis.
   Refer to numpy.argmax for full documentation.
   See Also:
   numpy.argmax
      equivalent function
```
recarray.argmin(axis=None, out=None)

Return indices of the minimum values along the given axis of a.

Refer to numpy.argmin for detailed documentation.

See Also:

numpy.argmin
equivalent function

recarray.argpartition(kth, axis=-1, kind='introselect', order=None)

Returns the indices that would partition this array.

Refer to numpy.argpartition for full documentation. New in version 1.8.0.

See Also:

numpy.argpartition
equivalent function

recarray.argsort(axis=-1, kind='quicksort', order=None)

Returns the indices that would sort this array.

Refer to numpy.argsort for full documentation.

See Also:

numpy.argsort
equivalent function

recarray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)

Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype
    Typecode or data-type to which the array is cast.

order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
    Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.

    • ‘no’ means the data types should not be cast at all.

    • ‘equiv’ means only byte-order changes are allowed.

    • ‘safe’ means only casts which can preserve values are allowed.

    • ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.

    • ‘unsafe’ means any data conversions may be done.

subok : bool, optional
If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

`copy` : bool, optional

By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

`arr_t` : ndarray

Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), `arr_t` is a new array of the same shape as the input array, with dtype, order given by `dtype`, `order`.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1.,  2.,  2.5])

>>> x.astype(int)
array([1, 2, 2])
```

```
recarray.byteswap(inplace)
```

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

`inplace` : bool, optional

If True, swap bytes in-place, default is False.

Returns

`out` : ndarray

The byteswapped array. If `inplace` is True, this is a view to self.

Examples

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']

>>> A.byteswap(True)
array([ 256, 1, 13090], dtype=int16)

>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```python
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
```

array(['ceg', 'fac'],
      dtype='|S3')

recarray.choose(choices, out=None, mode='raise')
Use an index array to construct a new array from a set of choices.
Refer to numpy.choose for full documentation.

See Also:

numpy.choose
  equivalent function

recarray.clip(a_min, a_max, out=None)
Return an array whose values are limited to \([a_{\min}, a_{\max}]\).
Refer to numpy.clip for full documentation.

See Also:

numpy.clip
  equivalent function

recarray.compress(condition, axis=None, out=None)
Return selected slices of this array along given axis.
Refer to numpy.compress for full documentation.

See Also:

numpy.compress
  equivalent function

recarray.conj()
Complex-conjugate all elements.
Refer to numpy.conjugate for full documentation.

See Also:

numpy.conjugate
  equivalent function

recarray.conjugate()
Return the complex conjugate, element-wise.
Refer to numpy.conjugate for full documentation.

See Also:

numpy.conjugate
  equivalent function

recarray.copy(order='C')
Return a copy of the array.

Parameters
  order : \{'C', 'F', 'A', 'K'\}, optional
Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

See Also:

numpy.copy, numpy.copyto

Examples

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

recarray.cumprod(axis=None, dtype=None, out=None)
Return the cumulative product of the elements along the given axis.

Refer to numpy.cumprod for full documentation.

See Also:

numpy.cumprod
equivalent function

recarray.cumsum(axis=None, dtype=None, out=None)
Return the cumulative sum of the elements along the given axis.

Refer to numpy.cumsum for full documentation.

See Also:

numpy.cumsum
equivalent function

recarray.diagonal(offset=0, axis1=0, axis2=1)
Return specified diagonals.

Refer to numpy.diagonal for full documentation.

See Also:

numpy.diagonal
equivalent function
recarray.dot (b, out=None)

Dot product of two arrays.

Refer to numpy.dot for full documentation.

See Also:

numpy.dot

equivalent function

Examples

```python
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```python
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

recarray.dump (file)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file : str

A string naming the dump file.

recarray.dumps ()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters

None

recarray.field(attr, val=None)

recarray.fill (value)

Fill the array with a scalar value.

Parameters

value : scalar

All elements of a will be assigned this value.

Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```
**recarray.flaten**(order='C')

Return a copy of the array collapsed into one dimension.

Parameters

- **order**: {'C', 'F', 'A'}, optional
  
  Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the
  C/Fortran ordering from a. The default is ‘C’.

Returns

- **y**: ndarray
  
  A copy of the input array, flattened to one dimension.

See Also:

- **ravel**: Return a flattened array.
- **flat**: A 1-D flat iterator over the array.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**recarray.getfield**(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the
given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits
in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with
a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

- **dtype**: str or dtype
  
  The data type of the view. The dtype size of the view can not be larger than that of the
  array itself.

- **offset**: int
  
  Number of bytes to skip before beginning the element view.

**Examples**

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
```

```python
>>> xgetField(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:
```python
>>> x.getfield(np.float64, offset=8)
array([[ 1.,  0.],
       [ 0.,  4.]])
```

```python
recarray.item(*args)
```

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

z : Standard Python scalar object

A copy of the specified element of the array as a suitable Python scalar

**Notes**

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

**Examples**

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

```python
recarray.itemset(*args)
```

Insert scalar into an array (scalar is cast to array’s dtype, if possible)

There must be at least 1 argument, and define the last argument as item. Then, a.itemset(*args) is equivalent to but faster than a[args] = item. The item should be a scalar value and args must select a single item in the array a.

**Parameters**

*args : Arguments
If one argument: a scalar, only used in case \( a \) is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

**Notes**

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

**Examples**

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

`recarray.max` *(axis=None, out=None)*

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

**See Also:**

`numpy.amax`

equivalent function

`recarray.mean` *(axis=None, dtype=None, out=None)*

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

**See Also:**

`numpy.mean`

equivalent function

`recarray.min` *(axis=None, out=None)*

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

**See Also:**

`numpy.amin`

equivalent function

`recarray.newbyteorder` *(new_order='S')*

Return the array with the same data viewed with a different byte order.

Equivalent to:
arr.view(arr.dtype.newbytorder(new_order))

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

*new_order*: string, optional

Byte order to force; a value from the byte order specifications above. `new_order` codes can be any of:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of `new_order` for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

*new_arr*: array

New array object with the dtype reflecting given change to the byte order.

def recarray.nonzero()  

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

See Also:

* `numpy.nonzero` equivalent function

**Parameters**

kth : int or sequence of ints

Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order all elements in the partitions is undefined. New in version 1.8.0.

*axis*: int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.

*kind*: {'introselect'}, optional

Selection algorithm. Default is ‘introselect’.

*order*: list, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.
See Also:

numpy.partition
   Return a partitioned copy of an array.
argpartition
   Indirect partition.
sort
   Full sort.

Notes
See numpy.partition for notes on the different algorithms.

Examples

>>> a = np.array([3, 4, 2, 1])
>>> a.partition(a, 3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
array([1, 2, 3, 4])

recarray.prod(axis=None, dtype=None, out=None)
   Return the product of the array elements over the given axis
   Refer to numpy.prod for full documentation.
   See Also:

   numpy.prod
      equivalent function

recarray.ptp(axis=None, out=None)
   Peak to peak (maximum - minimum) value along a given axis.
   Refer to numpy.ptp for full documentation.
   See Also:

   numpy.ptp
      equivalent function

recarray.put(indices, values, mode='raise')
   Set a.flat[n] = values[n] for all n in indices.
   Refer to numpy.put for full documentation.
   See Also:

   numpy.put
      equivalent function

recarray.ravel(order=None)
   Return a flattened array.
   Refer to numpy.ravel for full documentation.
   See Also:
**numpy.ravel**
equivalent function

**ndarray.flat**
a flat iterator on the array.

**recarray.repeat (repeats, axis=None)**
Repeat elements of an array.
Refer to numpy.repeat for full documentation.

See Also:

**numpy.repeat**
equivalent function

**recarray.reshape (shape, order='C')**
Returns an array containing the same data with a new shape.
Refer to numpy.reshape for full documentation.

See Also:

**numpy.reshape**
equivalent function

**recarray.resize (new_shape, refcheck=True)**
Change shape and size of array in-place.

Parameters

- **new_shape**
  : tuple of ints, or n ints
  Shape of resized array.

- **refcheck**
  : bool, optional
  If False, reference count will not be checked. Default is True.

Returns

None

Raises

**ValueError**
If a does not own its own data or references or views to it exist, and the data memory must be changed.

**SystemError**
If the order keyword argument is specified. This behaviour is a bug in NumPy.

See Also:

**resize**
Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.
The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

**Examples**

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

Unless `refcheck` is False:

```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

**recarray.round**(decimals=0, out=None)
Return `a` with each element rounded to the given number of decimals.
Refer to `numpy.around` for full documentation.

**See Also:**

- `numpy.around` equivalent function

- `recarray.searchsorted`(v, side='left', sorter=None)
Find indices where elements of `v` should be inserted in `a` to maintain order.
For full documentation, see `numpy.searchsorted`
See Also:

```python
numpy.searchsorted
```
equivalent function

```python
recarray.setfield(val, dtype, offset=0)
```
Put a value into a specified place in a field defined by a data-type.

Place `val` into `a`’s field defined by `dtype` and beginning `offset` bytes into the field.

**Parameters**

- `val`: object
  Value to be placed in field.
- `dtype`: dtype object
  Data-type of the field in which to place `val`.
- `offset`: int, optional
  The number of bytes into the field at which to place `val`.

**Returns**

None

See Also:

`getfield`

**Examples**

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[ 1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
       [ 1.48219694e-323, 1.00000000e+000, 1.48219694e-323],
       [ 1.48219694e-323, 1.48219694e-323, 1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

```python
recarray.setflags(write=None, align=None, uic=None)
```
Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by `a` (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)
Parameters

write : bool, optional

Describes whether or not a can be written to.

align : bool, optional

Describes whether or not a is aligned properly for its type.

uic : bool, optional

Describes whether or not a is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```python
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])

>>> y.flags
C_CONTINUOUS : True
F_CONTINUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

>>> y.setflags(write=0, align=0)

>>> y.flags
C_CONTINUOUS : True
F_CONTINUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False

>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

recarray.sort (axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

axis : int, optional

Axis along which to sort. Default is -1, which means sort along the last axis.
kind : {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm. Default is ‘quicksort’.

order : list, optional

When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:

numpy.sort
Return a sorted copy of an array.

argsort
Indirect sort.

lexsort
Indirect stable sort on multiple keys.

searchsorted
Find elements in sorted array.

partition
Partial sort.

Notes

See sort for notes on the different sorting algorithms.

Examples

```python
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:

```python
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '<S1'), ('y', '<i4')])
```

recarray.squeeze (axis=None)
Remove single-dimensional entries from the shape of a.

Refer to numpy.squeeze for full documentation.

See Also:

numpy.squeeze
equivalent function
NumPy Reference, Release 1.8.1

recarray.std(axis=None, dtype=None, out=None, ddof=0)
Returns the standard deviation of the array elements along given axis.
Refer to numpy.std for full documentation.
See Also:
numpy.std
equivalent function
recarray.sum(axis=None, dtype=None, out=None)
Return the sum of the array elements over the given axis.
Refer to numpy.sum for full documentation.
See Also:
numpy.sum
equivalent function
recarray.swapaxes(axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.
Refer to numpy.swapaxes for full documentation.
See Also:
numpy.swapaxes
equivalent function
recarray.take(indices, axis=None, out=None, mode=’raise’)
Return an array formed from the elements of a at the given indices.
Refer to numpy.take for full documentation.
See Also:
numpy.take
equivalent function
recarray.tofile(fid, sep=”“, format=”%s”)
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can
be recovered using the function fromfile().
Parameters
fid : file or str
An open file object, or a string containing a filename.
sep : str
Separator between array items for text output. If “” (empty), a binary file is written,
equivalent to file.write(a.tostring()).
format : str
Format string for text file output. Each entry in the array is formatted to text by first
converting it to the closest Python type, and then using “format” % item.

214

Chapter 1. Array objects


Notes
This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

recarray.tolist()
Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Parameters
none

Returns
y : list

The possibly nested list of array elements.

Notes
The array may be recreated, a = np.array(a.tolist()).

Examples
>>> a = np.array([[1, 2]])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]

recarray.tostring(order='C')
Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

Parameters
order : {'C', 'F', None}, optional

Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : str

A Python string exhibiting a copy of a’s raw data.

Examples
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tolist()
'\x00\x00\x00\x00\x00\x01\x00\x00\x00\x02\x00\x00\x00\x00\x03\x00\x00\x00\x00'
>>> x.tolist('C') == x.tolist()
True
`x.tostring('F')`
'\x00\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00\x03\x00\x00\x00

```
recarray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)
```
Return the sum along diagonals of the array.
Refer to `numpy.trace` for full documentation.

**See Also:**

`numpy.trace`
equivalent function

```
recarray.transpose(*axes)
```
Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).`

**Parameters**

axes : None, tuple of ints, or n ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes `a.transpose()`’s j-th axis.
  - n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

**Returns**

out : ndarray
  View of a, with axes suitably permuted.

**See Also:**

`ndarray.T`
Array property returning the array transposed.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 0],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 0],
       [2, 4]])
```
recarray.var (axis=None, dtype=None, out=None, ddof=0)
Returns the variance of the array elements, along given axis.
Refer to numpy.var for full documentation.

See Also:

numpy.var
equivalent function

recarray.view (dtype=None, type=None)

class numpy.record
A data-type scalar that allows field access as attribute lookup.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>transpose</td>
</tr>
<tr>
<td>base</td>
<td>base object</td>
</tr>
<tr>
<td>data</td>
<td>pointer to start of data</td>
</tr>
<tr>
<td>dtype</td>
<td>dtype object</td>
</tr>
<tr>
<td>flags</td>
<td>integer value of flags</td>
</tr>
<tr>
<td>flat</td>
<td>a 1-d view of scalar</td>
</tr>
<tr>
<td>imag</td>
<td>imaginary part of scalar</td>
</tr>
<tr>
<td>itemsize</td>
<td>length of one element in bytes</td>
</tr>
<tr>
<td>nbytes</td>
<td>length of item in bytes</td>
</tr>
<tr>
<td>ndim</td>
<td>number of array dimensions</td>
</tr>
<tr>
<td>real</td>
<td>real part of scalar</td>
</tr>
<tr>
<td>shape</td>
<td>tuple of array dimensions</td>
</tr>
<tr>
<td>size</td>
<td>number of elements in the genotype</td>
</tr>
<tr>
<td>strides</td>
<td>tuple of bytes steps in each dimension</td>
</tr>
</tbody>
</table>

record.T
transpose

record.base
base object

record.data
pointer to start of data

record.dtype
dtype object

record.flags
integer value of flags

record.flat
a 1-d view of scalar

record.imag
imaginary part of scalar

record.itemsize
length of one element in bytes
**record.nbytes**
length of item in bytes

**record.ndim**
number of array dimensions

**record.real**
real part of scalar

**record.shape**
tuple of array dimensions

**record.size**
number of elements in the gentype

**record.strides**
tuple of bytes steps in each dimension

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>any</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argmax</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argmin</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>argsort</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>astype</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>byteswap</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>choose</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>clip</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>compress</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>conj</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>conjugate</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>copy</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>cumprod</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>cumsum</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>diagonal</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>dump</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>dumps</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>fill</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>flatten</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>getfield</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>item</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>itemset</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>max</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>mean</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>min</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>newbyteorder</td>
<td>Return a new dtype with a different byte order.</td>
</tr>
<tr>
<td>nonzero</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>pprint()</td>
<td>Pretty-print all fields.</td>
</tr>
<tr>
<td>prod</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>ptp</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>put</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>ravel</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>repeat</td>
<td>Not implemented (virtual attribute)</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 1.50 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reshape</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>resize</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>round</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>searchsorted</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>setfield</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>setflags</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>sort</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>squeeze</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>std</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>sum</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>swapaxes</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>take</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tofile</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tolist</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>tostring</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>trace</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>transpose</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>var</td>
<td>Not implemented (virtual attribute)</td>
</tr>
<tr>
<td>view</td>
<td>Not implemented (virtual attribute)</td>
</tr>
</tbody>
</table>

**record.all()**

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

**record.any()**

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

**record.argmax()**

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The

**record.argmin()**

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**

The
record.argsort()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.astype()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.byteswap()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.choose()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.clip()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.compress()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.conj()

record.conjugate()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.copy()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.cumprod()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.cumsum()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.diagonal()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.dump()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.dumps()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.fill()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.flatten()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.getfield()

record.item()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.itemset()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.max()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.mean()  
Not implemented (virtual attribute)  
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

The

record.min()  
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The

```python
record.newbyteorder(new_order='S')
```

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The `new_order` code can be any from the following:

- `{'<', 'L'}` - little endian
- `{'>', 'B'}` - big endian
- `{'=', 'N'}` - native order
- `{'S'` - swap dtype from current to opposite endian
- `{'I', 'T'}` - ignore (no change to byte order)

**Parameters**

- `new_order`: str, optional
  Byte order to force; a value from the byte order specifications above. The default value (`'S'`) results in swapping the current byte order. The code does a case-insensitive check on the first letter of `new_order` for the alternatives above. For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

**Returns**

- `new_dtype`: dtype
  New `dtype` object with the given change to the byte order.

```python
record.nonzero()
```

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The

```python
record.pprint()
```

Pretty-print all fields.

```python
record.prod()
```

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

**See Also:**
The

```python
record.ptp()
```

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
See Also:
The
record.put()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.ravel()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.repeat()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.reshape()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.resize()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.round()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The
record.searchsorted()
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.setfield()

record.setflags()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.sort()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.squeeze()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.std()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.sum()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:

record.swapaxes()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.take()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.tofile()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.tolist()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.tostring()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.trace()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.transpose()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See Also:
The

record.var()
Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.
See Also:
The `record.view()`
Not implemented (virtual attribute)
Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

1.6.6 Masked arrays (numpy.ma)

See Also:
Masked arrays

1.6.7 Standard container class

For backward compatibility and as a standard “container” class, the UserArray from Numeric has been brought over to NumPy and named `numpy.lib.user_array.container`. The container class is a Python class whose self.array attribute is an ndarray. Multiple inheritance is probably easier with `numpy.lib.user_array.container` than with the ndarray itself and so it is included by default. It is not documented here beyond mentioning its existence because you are encouraged to use the ndarray class directly if you can.

```python
numpy.lib.user_array.container(data[,...])
```

```python
class numpy.lib.user_array.container(data, dtype=None, copy=True)
```

Methods

```python
container.astype(typecode)
container.byteswap()
container.copy()
container.tostring()
```

1.6. Standard array subclasses
1.6.8 Array Iterators

Iterators are a powerful concept for array processing. Essentially, iterators implement a generalized for-loop. If myiter is an iterator object, then the Python code:

```python
for val in myiter:
    ...
    some code involving val
    ...
```

calls `val = myiter.next()` repeatedly until `StopIteration` is raised by the iterator. There are several ways to iterate over an array that may be useful: default iteration, flat iteration, and $N$-dimensional enumeration.

Default iteration

The default iterator of an ndarray object is the default Python iterator of a sequence type. Thus, when the array object itself is used as an iterator. The default behavior is equivalent to:

```python
for i in range(arr.shape[0]):
    val = arr[i]
```

This default iterator selects a sub-array of dimension $N - 1$ from the array. This can be a useful construct for defining recursive algorithms. To loop over the entire array requires $N$ for-loops.

```python
>>> a = arange(24).reshape(3,2,4)+10
>>> for val in a:
...    print 'item:', val
item: [[10 11 12 13]
[14 15 16 17]]
item: [[18 19 20 21]
[22 23 24 25]]
item: [[26 27 28 29]
[30 31 32 33]]
```

Flat iteration

```python
ndarray.flat  A 1-D iterator over the array.
```

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

- flatten
  Return a copy of the array collapsed into one dimension.

Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
```
Array:

```python
array([[1, 2, 3],
       [4, 5, 6]])
```

```python
>>> x.flat[3]
4
```

```python
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```python
>>> x.T.flat[3]
5
```

```python
>>> type(x.flat)
<type 'numpy.flatiter'>
```

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
```

```python
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

As mentioned previously, the flat attribute of ndarray objects returns an iterator that will cycle over the entire array in C-style contiguous order.

```python
>>> for i, val in enumerate(a.flat):
...     if i%5 == 0:
...         print i, val
0 10
5 15
10 20
15 25
20 30
```

Here, I’ve used the built-in enumerate iterator to return the iterator index as well as the value.

**N-dimensional enumeration**

```python
ndenumerate(arr)  # Multidimensional index iterator.
```

```python
class numpy.ndenumerate(arr)
Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

Parameters
  a : ndarray
     Input array.

See Also:
  ndindex, flatiter
```

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(a):
...     print index, x
```

---

1.6. Standard array subclasses

229
Methods

next()  Standard iterator method, returns the index tuple and array value.

ndenumerate.next()  
Standard iterator method, returns the index tuple and array value.

Returns

coords : tuple of ints
   The indices of the current iteration.
val : scalar
   The array element of the current iteration.

Sometimes it may be useful to get the N-dimensional index while iterating. The ndenumerate iterator can achieve this.

```python
>>> for i, val in ndenumerate(a):
...     if sum(i)%5 == 0:
...         print i, val
(0, 0, 0) 10
(1, 1, 3) 25
(2, 0, 3) 29
(2, 1, 2) 32
```

Iterator for broadcasting

broadcast()  Produce an object that mimics broadcasting.

class numpy.broadcast
   Produce an object that mimics broadcasting.

Parameters

in1, in2, ... : array_like
   Input parameters.

Returns

b : broadcast object
   Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has shape and nd properties, and may be used as an iterator.

Examples

Manually adding two vectors, using broadcasting:

```python
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
```
>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[ 5.,  6.,  7.],
        [ 6.,  7.,  8.],
        [ 7.,  8.,  9.]])

Compare against built-in broadcasting:

>>> x + y
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])

Attributes

- **index**: current index in broadcasted result
- **iters**: tuple of iterators along self’s “components.”
- **shape**: Shape of broadcasted result.
- **size**: Total size of broadcasted result.

**broadcast.index**

Current index in broadcasted result

**Examples**

```python
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()  
((1, 4), (1, 5), (1, 6))
>>> b.index
3
```

**broadcast.iters**

tuple of iterators along self's “components.”

Returns a tuple of `numpy.flatiter` objects, one for each “component” of self.

**See Also:**

`numpy.flatiter`

**Examples**

```python
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> row, col = b.iters
>>> row.next(), col.next()  
(1, 4)
```

**broadcast.shape**

Shape of broadcasted result.
Examples

```python
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.shape
(3, 3)
```

`broadcast.size`
Total size of broadcasted result.

Examples

```python
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.size
9
```

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next</code></td>
<td><code>x.next()</code> -&gt; the next value, or raise StopIteration</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>Reset the broadcasted result’s iterator(s).</td>
</tr>
</tbody>
</table>

The general concept of broadcasting is also available from Python using the `broadcast` iterator. This object takes \( N \) objects as inputs and returns an iterator that returns tuples providing each of the input sequence elements in the broadcasted result.

```python
>>> for val in broadcast([[1,0],[2,3]],[0,1]):
...     print val
(1, 0)
(0, 1)
```
1.7 Masked arrays

Masked arrays are arrays that may have missing or invalid entries. The numpy.ma module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

1.7.1 The numpy.ma module

Rationale

Masked arrays are arrays that may have missing or invalid entries. The numpy.ma module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

What is a masked array?

In many circumstances, datasets can be incomplete or tainted by the presence of invalid data. For example, a sensor may have failed to record a data, or recorded an invalid value. The numpy.ma module provides a convenient way to address this issue, by introducing masked arrays.

A masked array is the combination of a standard numpy.ndarray and a mask. A mask is either nomask, indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not. When an element of the mask is False, the corresponding element of the associated array is valid and is said to be unmasked. When an element of the mask is True, the corresponding element of the associated array is said to be masked (invalid).

The package ensures that masked entries are not used in computations.

As an illustration, let’s consider the following dataset:

```python
>>> import numpy as np
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
```

We wish to mark the fourth entry as invalid. The easiest is to create a masked array:

```python
>>> mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
```

We can now compute the mean of the dataset, without taking the invalid data into account:

```python
>>> mx.mean()
2.75
```

The numpy.ma module

The main feature of the numpy.ma module is the MaskedArray class, which is a subclass of numpy.ndarray. The class, its attributes and methods are described in more details in the MaskedArray class section.

The numpy.ma module can be used as an addition to numpy:

```python
>>> import numpy as np
>>> import numpy.ma as ma
```
To create an array with the second element invalid, we would do:

```python
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values close to 1.e20 are invalid, we would do:

```python
>>> z = masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section *Constructing masked arrays*.

### 1.7.2 Using numpy.ma

#### Constructing masked arrays

There are several ways to construct a masked array.

- A first possibility is to directly invoke the `MaskedArray` class.
- A second possibility is to use the two masked array constructors, `array` and `masked_array`.

```python
array(data[, dtype, copy, order, mask, ...]) An array class with possibly masked values.
masked_array An array class with possibly masked values.
```

**numpy.ma.array**

```python
data, dtype=None, copy=False, order=False, mask=False, fill_value=None, keep_mask=True, hard_mask=False, shrink=True, subok=True, ndmin=0
```

An array class with possibly masked values.

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None,
                 copy=False, subok=True, ndmin=0, fill_value=None,
                 keep_mask=True, hard_mask=False, shrink=True)
```

**Parameters**

- `data`: array_like
  - Input data.
- `mask`: sequence, optional
  - Mask. Must be convertible to an array of booleans with the same shape as `data`. True indicates a masked (i.e. invalid) data.
- `dtype`: dtype, optional
  - Data type of the output. If `dtype` is None, the type of the data argument (`data.dtype`) is used. If `dtype` is not None and different from `data.dtype`, a copy is performed.
- `copy`: bool, optional
  - Whether to copy the input data (True), or to use a reference instead. Default is False.
- `subok`: bool, optional
  - Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.
- `ndmin`: int, optional
Minimum number of dimensions. Default is 0.

**fill_value**: scalar, optional

Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

**keep_mask**: bool, optional

Whether to combine `mask` with the mask of the input data, if any (True), or to use only `mask` for the output (False). Default is True.

**hard_mask**: bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked. Default is False.

**shrink**: bool, optional

Whether to force compression of an empty mask. Default is True.

```python
numpy.ma.masked_array
alias of MaskedArray
```

- A third option is to take the view of an existing array. In that case, the mask of the view is set to `nomask` if the array has no named fields, or an array of boolean with the same structure as the array otherwise.

```python
>>> x = np.array([1, 2, 3])
>>> x.view(ma.MaskedArray)
masked_array(data=[1 2 3],
            mask=False,
            fill_value=999999)
>>> x = np.array([(1, 1.), (2, 2.)], dtype=[('a', int), ('b', float)])
>>> x.view(ma.MaskedArray)
masked_array(data=[(1, 1.0) (2, 2.0)],
            mask=((False, False) (False, False)),
            fill_value=(999999, 1e+20),
            dtype=[('a', '<i4'), ('b', '<f8')])
```

- Yet another possibility is to use any of the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asarray(a, dtype, order)</code></td>
<td>Convert the input to a masked array of the given data-type.</td>
</tr>
<tr>
<td><code>asanyarray(a, dtype)</code></td>
<td>Convert the input to a masked array, conserving subclasses.</td>
</tr>
<tr>
<td><code>fix_invalid(a, mask, copy, fill_value)</code></td>
<td>Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>masked_equal(x, value, copy)</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>masked_greater(x, value, copy)</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>masked_greater_equal(x, value, copy)</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>masked_inside(x, v1, v2, copy)</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>masked_invalid(a, copy)</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>masked_less(x, value, copy)</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>masked_less_equal(x, value, copy)</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>masked_not_equal(x, value, copy)</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>masked_object(x, value, copy, shrink)</code></td>
<td>Mask the array where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>masked_outside(x, v1, v2, copy)</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>masked_values(x, value, rtol, atol, copy, ...)</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>masked_where(condition, a, copy)</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

```python
numpy.ma.asarray(a, dtype=None, order=None)
```

Convert the input to a masked array of the given data-type.

1.7. Masked arrays
No copy is performed if the input is already an ndarray. If \( a \) is a subclass of \texttt{MaskedArray}, a base class \texttt{MaskedArray} is returned.

**Parameters**

\( a \): array_like

Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists, ndarrays and masked arrays.

\texttt{dtype}: dtype, optional

By default, the data-type is inferred from the input data.

\texttt{order}: {'C', 'F'}, optional

Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**

\texttt{out}: MaskedArray

Masked array interpretation of \( a \).

**See Also:**

\texttt{asanyarray}

Similar to \texttt{asarray}, but conserves subclasses.

**Examples**

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asarray(x)
masked_array(data =
     [[ 0.  1.  2.  3.  4.]
      [ 5.  6.  7.  8.  9.]],
    mask = False,
   fill_value = 1e+20)
```

**numpy.ma.asanyarray** (\( a, \texttt{dtype} = \text{None} \))

Convert the input to a masked array, conserving subclasses.

If \( a \) is a subclass of \texttt{MaskedArray}, its class is conserved. No copy is performed if the input is already an ndarray.

**Parameters**

\( a \): array_like

Input data, in any form that can be converted to an array.

\texttt{dtype}: dtype, optional

By default, the data-type is inferred from the input data.

\texttt{order}: {'C', 'F'}, optional

Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.
Returns

\textbf{out} : MaskedArray

MaskedArray interpretation of \(a\).

See Also:

\texttt{asarray}

Similar to \texttt{asanyarray}, but does not conserve subclass.

Examples

\begin{verbatim}
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0., 1., 2., 3., 4.],
       [ 5., 6., 7., 8., 9.]])
>>> np.ma.asanyarray(x)
masked_array(data =
             [[ 0.  1.  2.  3.  4.]
             [ 5.  6.  7.  8.  9.]],
            mask = False,
            fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
\end{verbatim}

\texttt{numpy.ma.fix_invalid(a, mask=False, copy=True, fill_value=None)}

Return input with invalid data masked and replaced by a fill value.

Invalid data means values of \texttt{nan}, \texttt{inf}, etc.

Parameters

- \(a\) : array_like

  Input array, a (subclass of) \texttt{ndarray}.

- \(\text{copy}\) : bool, optional

  Whether to use a copy of \(a\) (True) or to fix \(a\) in place (False). Default is True.

- \(\text{fill\_value}\) : scalar, optional

  Value used for fixing invalid data. Default is None, in which case the \(a.fill\_value\) is used.

Returns

- \(b\) : MaskedArray

  The input array with invalid entries fixed.

Notes

A copy is performed by default.

Examples

\begin{verbatim}
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
             mask = [ True False False False],
             fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
\end{verbatim}
masked_array(data = [-- -1.0 -- --],
         mask = [ True False True True],
       fill_value = 1e+20)

>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00, 1.00000000e+20,
       1.00000000e+20])

>>> x.data
array([ 1., -1., NaN, Inf])

numpy.ma.masked_equal(x, value, copy=True)

Mask an array where equal to a given value.

This function is a shortcut to masked_where, with condition = (x == value). For floating point arrays, consider using masked_values(x, value).

See Also:

masked_where
    Mask where a condition is met.

masked_values
    Mask using floating point equality.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
         mask = [False False True False],
       fill_value=999999)

numpy.ma.masked_greater(x, value, copy=True)

Mask an array where greater than a given value.

This function is a shortcut to masked_where, with condition = (x > value).

See Also:

masked_where
    Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
         mask = [False False False True],
       fill_value=999999)

numpy.ma.masked_greater_equal(x, value, copy=True)

Mask an array where greater than or equal to a given value.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 2 --],
         mask = [False False False True],
       fill_value=999999)
This function is a shortcut to `masked_where`, with `condition = (x >= value)`.

See Also:

`masked_where`
Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
g feather a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
             mask = [False False True True],
             fill_value=999999)
```

`numpy.ma.masked_inside(x, v1, v2, copy=True)`
Mask an array inside a given interval.

Shortcut to `masked_where`, where `condition` is True for `x` inside the interval [v1,v2] (v1 <= x <= v2). The boundaries v1 and v2 can be given in either order.

See Also:

`masked_where`
Mask where a condition is met.

Notes
The array x is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma

>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

The order of v1 and v2 doesn’t matter.

```python
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
             fill_value=1e+20)
```

`numpy.ma.masked_invalid(a, copy=True)`
Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to `masked_where`, with `condition = ~(np.isfinite(a))`. Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

See Also:

`masked_where`
Mask where a condition is met.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0., 1., NaN, Inf, 4.])
```
```python
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
mask = [False False True True False],
fill_value=1e+20)
```

**numpy.ma.masked_less** *(x, value, copy=True)*

Mask an array where less than a given value.

This function is a shortcut to `masked_where`, with `condition = (x < value)`.

**See Also:**

`masked_where`  
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
```
```python
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
mask = [ True True False False],
fill_value=999999)
```

**numpy.ma.masked_less_equal** *(x, value, copy=True)*

Mask an array where less than or equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x <= value)`.

**See Also:**

`masked_where`  
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
```
```python
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
mask = [ True True True False],
fill_value=999999)
```

**numpy.ma.masked_not_equal** *(x, value, copy=True)*

Mask an array where not equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x != value)`.

**See Also:**
**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
              mask = [ True True False True],
              fill_value=999999)
```

```
**numpy.ma.masked_object**(x, value, copy=True, shrink=True)
Mask the array x where the data are exactly equal to value.

This function is similar to **masked_values**, but only suitable for object arrays: for floating point, use **masked_values** instead.

**Parameters**
- x : array_like
  Array to mask
- value : object
  Comparison value
- copy : {True, False}, optional
  Whether to return a copy of x.
- shrink : {True, False}, optional
  Whether to collapse a mask full of False to nomask

**Returns**
- result : MaskedArray
  The result of masking x where equal to value.

**See Also:**

**masked_where**
Mask where a condition is met.

**masked_equal**
Mask where equal to a given value (integers).

**masked_values**
Mask using floating point equality.

**Examples**

```python
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol' ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
```
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]

Note that mask is set to nomask if possible.

>>> eat
masked_array(data = [cheese ham pineapple],
       mask = False,
      fill_value=?)

numpy.ma.masked_outside(x, v1, v2, copy=True)
Mask an array outside a given interval.

Shortcut to masked_where, where condition is True for x outside the interval [v1,v2] (x < v1)|(x > v2). The boundaries v1 and v2 can be given in either order.

See Also:

masked_where
Mask where a condition is met.

Notes
The array x is prefilled with its filling value.

Examples

>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
mixed_array(data = [-- -- 0.01 0.2 -- --],
       mask = [ True True False False True True],
      fill_value=1e+20)

The order of v1 and v2 doesn’t matter.

>>> ma.masked_outside(x, 0.3, -0.3)
mixed_array(data = [-- -- 0.01 0.2 -- --],
       mask = [ True True False False True True],
      fill_value=1e+20)

numpy.ma.masked_values(x, value, rtol=1e-05, atol=1e-08, copy=True, shrink=True)
Mask using floating point equality.

Return a MaskedArray, masked where the data in array x are approximately equal to value, i.e. where the following condition is True
(abs(x - value) <= atol+rtol*abs(value))

The fill_value is set to value and the mask is set to nomask if possible. For integers, consider using masked_equal.

Parameters

x : array_like
Array to mask.

value : float
Masking value.
rtol : float, optional
    Tolerance parameter.

atol : float, optional
    Tolerance parameter (1e-8).

copy : bool, optional
    Whether to return a copy of x.

shrink : bool, optional
    Whether to collapse a mask full of False to nomask.

Returns

result : MaskedArray
    The result of masking x where approximately equal to value.

See Also:

masked_where
    Mask where a condition is met.

masked_equal
    Mask where equal to a given value (integers).

Examples

>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
             mask = [False  True False  True False],
             fill_value=1.1)

Note that mask is set to nomask if possible.

>>> ma.masked_values(x, 1.5)
masked_array(data = [ 1. 1.1 2. 1.1 3. ],
             mask = False,
             fill_value=1.5)

For integers, the fill value will be different in general to the result of masked_equal.

>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=2)
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
             mask = [False False True False False],
             fill_value=999999)

numpy.ma.masked_where(condition, a, copy=True)
    Mask an array where a condition is met.
Return $a$ as an array masked where $condition$ is True. Any masked values of $a$ or $condition$ are also masked in the output.

**Parameters**

- **condition**: array_like
  - Masking condition. When $condition$ tests floating point values for equality, consider using `masked_values` instead.

- **a**: array_like
  - Array to mask.

- **copy**: bool
  - If True (default) make a copy of $a$ in the result. If False modify $a$ in place and return a view.

**Returns**

- **result**: MaskedArray
  - The result of masking $a$ where $condition$ is True.

**See Also:**

- `masked_values`
  - Mask using floating point equality.

- `masked_equal`
  - Mask where equal to a given value.

- `masked_not_equal`
  - Mask where not equal to a given value.

- `masked_less_equal`
  - Mask where less than or equal to a given value.

- `masked_greater_equal`
  - Mask where greater than or equal to a given value.

- `masked_less`
  - Mask where less than a given value.

- `masked_greater`
  - Mask where greater than a given value.

- `masked_inside`
  - Mask inside a given interval.

- `masked_outside`
  - Mask outside a given interval.

- `masked_invalid`
  - Mask invalid values (NaNs or infs).

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
```
mask = [ True  True  True False],
      fill_value=999999)

Mask array b conditional on a.

>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data = ['a' 'b' -- 'd'],
      mask = [False False True False],
      fill_value=N/A)

Effect of the copy argument.

>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data = [-- -- -- 3],
      mask = [False True True False],
      fill_value=999999)

>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
      mask = [False True True False],
      fill_value=999999)

>> a
array([0, 1, 2, 3])

>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c
masked_array(data = [99 -- -- 3],
      mask = [False True True False],
      fill_value=999999)

>> a
array([99, 1, 2, 3])

When condition or a contain masked values.

>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)

>> a
masked_array(data = [0 1 -- 3],
      mask = [False False True False],
      fill_value=999999)

>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)

>> b
masked_array(data = [-- 1 2 3],
      mask = [ True False False False],
      fill_value=999999)

>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
      mask = [ True False True True],
      fill_value=999999)

Accessing the data

The underlying data of a masked array can be accessed in several ways:
• through the `data` attribute. The output is a view of the array as a `numpy.ndarray` or one of its subclasses, depending on the type of the underlying data at the masked array creation.
• through the `__array__` method. The output is then a `numpy.ndarray`.
• by directly taking a view of the masked array as a `numpy.ndarray` or one of its subclass (which is actually what using the `data` attribute does).
• by using the `getdata` function.

None of these methods is completely satisfactory if some entries have been marked as invalid. As a general rule, where a representation of the array is required without any masked entries, it is recommended to fill the array with the `filled` method.

Accessing the mask

The mask of a masked array is accessible through its `mask` attribute. We must keep in mind that a `True` entry in the mask indicates an invalid data.

Another possibility is to use the `getmask` and `getmaskarray` functions. `getmask(x)` outputs the mask of `x` if `x` is a masked array, and the special value `nomask` otherwise. `getmaskarray(x)` outputs the mask of `x` if `x` is a masked array. If `x` has no invalid entry or is not a masked array, the function outputs a boolean array of `False` with as many elements as `x`.

Accessing only the valid entries

To retrieve only the valid entries, we can use the inverse of the mask as an index. The inverse of the mask can be calculated with the `numpy.logical_not` function or simply with the `~` operator:

```python
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> x[~x.mask]
masked_array(data = [1 4],
             mask = [False False],
             fill_value = 999999)
```

Another way to retrieve the valid data is to use the `compressed` method, which returns a one-dimensional `ndarray` (or one of its subclasses, depending on the value of the `baseclass` attribute):

```python
>>> x.compressed()
array([1, 4])
```

Note that the output of `compressed` is always 1D.

Modifying the mask

Masking an entry

The recommended way to mark one or several specific entries of a masked array as invalid is to assign the special value `masked` to them:

```python
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data = [-- 2 3],
             mask = [ True False False],
             fill_value = 999999)
>>> y = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```
```python
>>> y[(0, 1, 2), (1, 2, 0)] = ma.masked
>>> y
masked_array(data =
[[1 -- 3]
 [4 5 --]
[-- 8 9]],
mask =
[[False  True False]
 [False False True]
 [ True False False]],
fill_value = 999999)
```

```python
>>> z = ma.array([1, 2, 3, 4])
>>> z[:-2] = ma.masked
>>> z
masked_array(data = [-- -- 3 4],
mask = [ True True False False],
fill_value = 999999)
```

A second possibility is to modify the `mask` directly, but this usage is discouraged.

**Note:** When creating a new masked array with a simple, non-structured datatype, the mask is initially set to the special value `nomask`, that corresponds roughly to the boolean `False`. Trying to set an element of `nomask` will fail with a `TypeError` exception, as a boolean does not support item assignment.

All the entries of an array can be masked at once by assigning `True` to the mask:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x.mask = True
>>> x
masked_array(data = [-- -- --],
mask = [ True True True],
fill_value = 999999)
```

Finally, specific entries can be masked and/or unmasked by assigning to the mask a sequence of booleans:

```python
>>> x = ma.array([1, 2, 3])
>>> x.mask = [0, 1, 0]
>>> x
masked_array(data = [1 -- 3],
mask = [False 3 False],
fill_value = 999999)
```

**Unmasking an entry**

To unmask one or several specific entries, we can just assign one or several new valid values to them:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
mask = [False False True],
fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
mask = [False False False],
fill_value = 999999)
```

---

1.7. Masked arrays
Note: Unmasking an entry by direct assignment will silently fail if the masked array has a hard mask, as shown by the hardmask attribute. This feature was introduced to prevent overwriting the mask. To force the unmasking of an entry where the array has a hard mask, the mask must first to be softened using the soften_mask method before the allocation. It can be re-hardened with harden_mask:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1], hard_mask=True)
>>> x
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x.soften_mask()
>>> x[-1] = 5
>>> x
masked_array(data = [1 2 5],
    mask = [False False False],
    fill_value = 999999)
>>> x.harden_mask()
```

To unmask all masked entries of a masked array (provided the mask isn’t a hard mask), the simplest solution is to assign the constant nomask to the mask:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data = [1 2 --],
    mask = [False False True],
    fill_value = 999999)
>>> x.mask = ma.nomask
>>> x
masked_array(data = [1 2 3],
    mask = [False False False],
    fill_value = 999999)
```

## Indexing and slicing

As a `MaskedArray` is a subclass of `numpy.ndarray`, it inherits its mechanisms for indexing and slicing.

When accessing a single entry of a masked array with no named fields, the output is either a scalar (if the corresponding entry of the mask is `False`) or the special value `masked` (if the corresponding entry of the mask is `True`):

```python
>>> x = ma.array([[1, 2, 3], mask=[0, 0, 1]])
>>> x[0]
1
>>> x[-1]
masked_array(data = --,
    mask = True,
    fill_value = 1e+20)
>>> x[-1] is ma.masked
True
```

If the masked array has named fields, accessing a single entry returns a `numpy.void` object if none of the fields are masked, or a 0d masked array with the same dtype as the initial array if at least one of the fields is masked.
```python
>>> y = ma.masked_array([[1, 2], (3, 4)],
...                      mask=[[0, 0], (0, 1)],
...                      dtype=[('a', int), ('b', int)])
>>> y[0]
(1, 2)
>>> y[-1]
masked_array(data = (3, --),
              mask = (False, True),
              fill_value = (999999, 999999),
              dtype = [('a', '<i4'), ('b', '<i4')])
```

When accessing a slice, the output is a masked array whose `data` attribute is a view of the original data, and whose mask is either `nomask` (if there was no invalid entries in the original array) or a copy of the corresponding slice of the original mask. The copy is required to avoid propagation of any modification of the mask to the original.

```python
>>> x = ma.array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
>>> mx = x[:3]
>>> mx
masked_array(data = [1 -- 3],
              mask = [False True False],
              fill_value = 999999)
>>> mx[1] = -1
>>> mx
masked_array(data = [1 -1 3],
              mask = [False True False],
              fill_value = 999999)
```

Accessing a field of a masked array with structured datatype returns a `MaskedArray`.

## Operations on masked arrays

Arithmetic and comparison operations are supported by masked arrays. As much as possible, invalid entries of a masked array are not processed, meaning that the corresponding `data` entries *should* be the same before and after the operation.

**Warning:** We need to stress that this behavior may not be systematic, that masked data may be affected by the operation in some cases and therefore users should not rely on this data remaining unchanged.

The `numpy.ma` module comes with a specific implementation of most ufuncs. Unary and binary functions that have a validity domain (such as `log` or `divide`) return the `masked` constant whenever the input is masked or falls outside the validity domain:

```python
>>> ma.log([-1, 0, 1, 2])
masked_array(data = [-- -- 0.0 0.69314718056],
             mask = [ True True False False],
             fill_value = 1e+20)
```

Masked arrays also support standard `numpy` ufuncs. The output is then a masked array. The result of a unary ufunc is masked wherever the input is masked. The result of a binary ufunc is masked wherever any of the input is masked. If the ufunc also returns the optional context output (a 3-element tuple containing the name of the ufunc, its arguments and its domain), the context is processed and entries of the output masked array are masked wherever the corresponding input fall outside the validity domain:

## 1.7. Masked arrays
1.7.3 Examples

Data with a given value representing missing data

Let’s consider a list of elements, x, where values of -9999. represent missing data. We wish to compute the average value of the data and the vector of anomalies (deviations from the average):

```python
>>> import numpy.ma as ma
>>> x = [0.,1.,-9999.,3.,4.]
>>> mx = ma.masked_values(x, -9999.)
>>> print mx.mean()
2.0
>>> print mx - mx.mean()
[-2.0 -1.0 -- 1.0 2.0]
>>> print mx.anom()
[-2.0 -1.0 -- 1.0 2.0]
```

Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```python
>>> print mx.filled(mx.mean())
[ 0. 1. 2. 3. 4.]
```

Numerical operations

Numerical operations can be easily performed without worrying about missing values, dividing by zero, square roots of negative numbers, etc.:

```python
>>> import numpy as np, numpy.ma as ma
>>> x = ma.array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,1,0,0])
>>> y = ma.array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,1,1])
>>> print np.sqrt(x/y)
[1.0 -- 1.0 -- --]
```

Four values of the output are invalid: the first one comes from taking the square root of a negative number, the second from the division by zero, and the last two where the inputs were masked.

Ignoring extreme values

Let’s consider an array d of random floats between 0 and 1. We wish to compute the average of the values of d while ignoring any data outside the range [0.1, 0.9]:

```python
>>> print ma.masked_outside(d, 0.1, 0.9).mean()
```
1.7.4 Constants of the `numpy.ma` module

In addition to the `MaskedArray` class, the `numpy.ma` module defines several constants.

`numpy.ma.masked`

The `masked` constant is a special case of `MaskedArray`, with a float datatype and a null shape. It is used to test whether a specific entry of a masked array is masked, or to mask one or several entries of a masked array:

```python
>>> x = ma.array([1, 2, 3], mask=[0, 1, 0])
>>> x[1] is ma.masked
True
>>> x[-1] = ma.masked
```

`numpy.ma.nomask`

Value indicating that a masked array has no invalid entry. `nomask` is used internally to speed up computations when the mask is not needed.

`numpy.ma.masked_print_options`

String used in lieu of missing data when a masked array is printed. By default, this string is ‘--’.

1.7.5 The `MaskedArray` class

`class numpy.ma.MaskedArray`

A subclass of `ndarray` designed to manipulate numerical arrays with missing data.

An instance of `MaskedArray` can be thought as the combination of several elements:

- The `data`, as a regular `numpy.ndarray` of any shape or datatype (the data).
- A boolean `mask` with the same shape as the data, where a `True` value indicates that the corresponding element of the data is invalid. The special value `nomask` is also acceptable for arrays without named fields, and indicates that no data is invalid.
- A `fill_value`, a value that may be used to replace the invalid entries in order to return a standard `numpy.ndarray`.

Attributes and properties of masked arrays

See Also:

`Array Attributes`

`MaskedArray.data`

Returns the underlying data, as a view of the masked array. If the underlying data is a subclass of `numpy.ndarray`, it is returned as such.

```python
>>> x = ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.data
matrix([[1, 2],
        [3, 4]])
```

The type of the data can be accessed through the `baseclass` attribute.
MaskedArray.\texttt{mask}

Returns the underlying mask, as an array with the same shape and structure as the data, but where all fields are atomically booleans. A value of \texttt{True} indicates an invalid entry.

MaskedArray.\texttt{recordmask}

Returns the mask of the array if it has no named fields. For structured arrays, returns a \texttt{ndarray} of booleans where entries are \texttt{True} if all the fields are masked, \texttt{False} otherwise:

\begin{verbatim}
>>> x = ma.array([(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)],
...               mask=[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)],
...               dtype=[('a', int), ('b', int)])
>>> x.recordmask
array([False, False, True, False, False], dtype=bool)
\end{verbatim}

MaskedArray.\texttt{fill\_value}

Returns the value used to fill the invalid entries of a masked array. The value is either a scalar (if the masked array has no named fields), or a 0-D \texttt{ndarray} with the same \texttt{dtype} as the masked array if it has named fields.

The default filling value depends on the datatype of the array:

<table>
<thead>
<tr>
<th>datatype</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>int</td>
<td>999999</td>
</tr>
<tr>
<td>float</td>
<td>1.e20</td>
</tr>
<tr>
<td>complex</td>
<td>1.e20+0j</td>
</tr>
<tr>
<td>object</td>
<td>‘?’</td>
</tr>
<tr>
<td>string</td>
<td>‘N/A’</td>
</tr>
</tbody>
</table>

MaskedArray.\texttt{baseclass}

Returns the class of the underlying data.

\begin{verbatim}
>>> x = ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 0], [1, 0]])
>>> x.baseclass
<class ‘numpy.matrixlib.defmatrix.matrix’>
\end{verbatim}

MaskedArray.\texttt{sharedmask}

Returns whether the mask of the array is shared between several masked arrays. If this is the case, any modification to the mask of one array will be propagated to the others.

MaskedArray.\texttt{hardmask}

Returns whether the mask is hard (\texttt{True}) or soft (\texttt{False}). When the mask is hard, masked entries cannot be unmasked.

As \texttt{MaskedArray} is a subclass of \texttt{ndarray}, a masked array also inherits all the attributes and properties of a \texttt{ndarray} instance.
Table 1.61 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.real</td>
<td>Real part</td>
</tr>
<tr>
<td>MaskedArray.flat</td>
<td>Flat version of the array.</td>
</tr>
<tr>
<td>MaskedArray.<strong>array_priority</strong></td>
<td>int(x=0) -&gt; int or long</td>
</tr>
</tbody>
</table>

MaskedArray.base
Base object if memory is from some other object.

Examples
The base of an array that owns its memory is None:

```python
g >> x = np.array([1,2,3,4])
>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
g >> y = x[2:]
>> y.base is x
True
```

MaskedArray.ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters
None

Returns
- c : Python object
  Possessing attributes data, shape, strides, etc.

See Also:
- numpy.ctypeslib

Notes
Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

- • data: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

- • shape (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

- • strides (c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.
•`data_as(obj)`: Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

•`shape_as(obj)`: Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

•`strides_as(obj)`: Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as `(a+b)` is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, `ct` will hold a reference to the array until `ct` is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as` parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
array([[ 0,  1],
       [ 2,  3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
  c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
  c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

MaskedArray.**dtype**

Data-type of the array’s elements.

**Parameters**

None

**Returns**

d : numpy dtype object

**See Also:**

numpy.dtype

**Examples**

```python
>>> x
array([[ 0,  1],
       [ 2,  3]])
```
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

MaskedArray.flags
Information about the memory layout of the array.

Notes
The flags object can be accessed dictionary-like (as in a.flags['WRITEABLE']), or by using lowercased attribute names (as in a.flags.writeable). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling ndarray.setflags.

The array flags cannot be set arbitrarily:
  • UPDATEIFCOPY can only be set False.
  • ALIGNED can only be set True if the data is truly aligned.
  • WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension arr.strides[dim] may be arbitrary if arr.shape[dim] == 1 or the array has no elements. It does not generally hold that self.strides[-1] == self.itemsize for C-style contiguous arrays or self.strides[0] == self.itemsize for Fortran-style contiguous arrays is true.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CONTIGUOUS (C)</td>
<td>The data is in a single, C-style contiguous segment.</td>
</tr>
<tr>
<td>F_CONTIGUOUS (F)</td>
<td>The data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWNDATA (O)</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE (W)</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.</td>
</tr>
<tr>
<td>ALIGNED (A)</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATEIFCOPY (U)</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED (B)</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY (CA)</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY (FA)</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>

**MaskedArray.itemsize**

Length of one array element in bytes.

**Examples**

```python
>>> x = np.array([1, 2, 3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1, 2, 3], dtype=np.complex128)
>>> x.itemsize
16
```

**MaskedArray.nbytes**

Total bytes consumed by the elements of the array.

**Notes**

Does not include memory consumed by non-element attributes of the array object.

**Examples**

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```
**MaskedArray.ndim**

Number of array dimensions.

**Examples**

```python
g >> x = np.array([1, 2, 3])
g >>> x.ndim
1
g >>> y = np.zeros((2, 3, 4))
g >>> y.ndim
3
```

**MaskedArray.shape**

Tuple of array dimensions.

**Notes**

May be used to “reshape” the array, as long as this would not require a change in the total number of elements.

**Examples**

```python
g >>> x = np.array([1, 2, 3, 4])
g >>> x.shape
(4,)
g >>> y = np.zeros((2, 3, 4))
g >>> y.shape
(2, 3, 4)
g >>> y.shape = (3, 8)
g >>> y
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
g >>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
ValueError: total size of new array must be unchanged
```

**MaskedArray.size**

Number of elements in the array.

Equivalent to `np.prod(a.shape)`, i.e., the product of the array’s dimensions.

**Examples**

```python
g >>> x = np.zeros((3, 5, 2), dtype=np.complex128)
g >>> x.size
30
g >>> np.prod(x.shape)
30
```

**MaskedArray.strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element \((i[0], i[1], \ldots, i[n])\) in an array \(a\) is:

\[
\text{offset} = \text{sum}(\text{np.array}(i) \times a\text{.strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

**See Also:**

`numpy.lib.stride_tricks.as_strided`
Notes

Imagine an array of 32-bit integers (each 4 bytes):

```python
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be `(20, 4)`.

Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
>>> y.strides
(48, 16, 4)
>>> y[i,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17

>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[i,5,2,2]
813
>>> offset / x.itemsize
813
```

MaskedArray.

- **imag**
  - Imaginary part.

- **real**
  - Real part

- **flat**
  - Flat version of the array.

MaskedArray.

- **__array_priority__** = 15

1.7.6 MaskedArray methods

See Also:

- Array methods
Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy Reference, Release 1.8.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>float</strong>()</td>
<td>Convert to float.</td>
</tr>
<tr>
<td>MaskedArray.<strong>hex</strong>()</td>
<td>&lt;==&gt; hex(x)</td>
</tr>
<tr>
<td>MaskedArray.<strong>int</strong>()</td>
<td>Convert to int.</td>
</tr>
<tr>
<td>MaskedArray.<strong>long</strong>()</td>
<td>&lt;==&gt; long(x)</td>
</tr>
<tr>
<td>MaskedArray.<strong>oct</strong>()</td>
<td>&lt;==&gt; oct(x)</td>
</tr>
<tr>
<td>MaskedArray.view([dtype, type])</td>
<td>New view of array with the same data.</td>
</tr>
<tr>
<td>MaskedArray.astype(newtype)</td>
<td>Returns a copy of the MaskedArray cast to given newtype.</td>
</tr>
<tr>
<td>MaskedArray.byteswap(inplace)</td>
<td>Swap the bytes of the array elements.</td>
</tr>
<tr>
<td>MaskedArray.compressed()</td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td>MaskedArray.filled([fill_value])</td>
<td>Return a copy of self, with masked values filled with a given value.</td>
</tr>
<tr>
<td>MaskedArray.tofile(fid[, sep, format])</td>
<td>Save a masked array to a file in binary format.</td>
</tr>
<tr>
<td>MaskedArray.toflex()</td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td>MaskedArray.tolist([fill_value])</td>
<td>Return the data portion of the masked array as a hierarchical Python list.</td>
</tr>
<tr>
<td>MaskedArray.torecords()</td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td>MaskedArray.tostring([fill_value, order])</td>
<td>Return the array data as a string containing the raw bytes in the array.</td>
</tr>
</tbody>
</table>

MaskedArray.__float__()  
Convert to float.

MaskedArray.__hex__()    
<==> hex(x)

MaskedArray.__int__()    
Convert to int.

MaskedArray.__long__()   
<==> long(x)

MaskedArray.__oct__()    
<==> oct(x)

MaskedArray.view([dtype, type])  
New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as a. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the type parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

a.view() is used two different ways:
a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.
a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation.
of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

**Examples**

```python
g >> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```python
g >> y = x.view(dtype=np.int16, type=np.matrix)
g >> y
matrix([[513]], dtype=int16)
g >> print type(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```python
g >> x = np.array([[(1, 2),(3,4)]], dtype=[('a', np.int8), ('b', np.int8)])
g >> xv = x.view(np.int8).reshape(-1,2)
g >> xv
array([[1, 2],
       [3, 4]], dtype=int8)
g >> xv.mean(0)
array([ 2., 3.])
```

Making changes to the view changes the underlying array

```python
g >> xv[0,1] = 20
g >> print x
[(1, 20) (3, 4)]
```

Using a view to convert an array to a record array:

```python
g >> z = x.view(np.recarray)
g >> z.a
array([1], dtype=int8)
```

Views share data:

```python
g >> z[0] = (9, 10)
g >> print z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```python
g >> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
g >> y = x[:, 0:2]
g >> y
array([[1, 2],
       [4, 5]], dtype=int16)
g >> y.view(dtype=[('width', np.int16), ('length', np.int16)])
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
```

```python
g >> z = y.copy()
```
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 2]],
[(4, 5)], dtype=[('width', '<i2'), ('length', '<i2')])

MaskedArray.astype(newtype)

Returns a copy of the MaskedArray cast to given newtype.

Parameters
output : MaskedArray

Returns
A copy of self cast to input newtype. The returned record shape matches self.shape.

Examples

>>> x = np.ma.array([[1, 2, 3.1], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)

>>> print x
[[1.0 -- 3.1]
 [-- 5.0 --]
 [7.0 -- 9.0]]

>>> print x.astype(int32)
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]

MaskedArray.byteswap(inplace)

Swap the bytes of the array elements.

Parameters
inplace : bool, optional
If True, swap bytes in-place, default is False.

Returns
out : ndarray
The byteswapped array. If inplace is True, this is a view to self.

Examples

>>> A = np.array([1, 256, 8755], dtype=np.int16)

>>> map(hex, A)
['0x1', '0x100', '0x2233']

>>> A.byteswap(True)
array([256, 1, 13090], dtype=int16)

>>> map(hex, A)
['0x100', '0x1', '0x3322']

Arrays of strings are not swapped

>>> A = np.array(['ceg', 'fac'])

>>> A.byteswap()
array(['ceg', 'fac'],
     dtype='|S3')

MaskedArray.compressed()

Return all the non-masked data as a 1-D array.

Returns
data : ndarray

1.7. Masked arrays

261
A new *ndarray* holding the non-masked data is returned.

**Notes**

The result is **not** a MaskedArray!

**Examples**

```python
g >> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
g >>> x.compressed()
g array([0, 1])
g >>> type(x.compressed())
g <type 'numpy.ndarray'>
```

MaskedArray.*filled*(fill_value=None)

Return a copy of self, with masked values filled with a given value.

**Parameters**

- fill_value : scalar, optional
  The value to use for invalid entries (None by default). If None, the *fill_value*
  attribute of the array is used instead.

**Returns**

- filled_array : ndarray
  A copy of *self* with invalid entries replaced by *fill_value* (be it the function argument
  or the attribute of *self*).

**Notes**

The result is **not** a MaskedArray!

**Examples**

```python
g >>> x = np.ma.array([1,2,3,4,5], mask=[0,0,1,0,1], fill_value=-999)
g >>> x.filled()
g array([1, 2, -999, 4, -999])
g >>> type(x.filled())
g <type 'numpy.ndarray'>
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, *filled* returns a matrix:

```python
g >>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
g >>> x.filled()
g matrix([[1, 999999],
         [999999, 4]])
```

MaskedArray.*tofile*(fid, sep='', format='%s')

Save a masked array to a file in binary format.

**Warning:** This function is not implemented yet.

**Raises**

- **NotImplementedError**
  When *tofile* is called.

MaskedArray.*toflex*()

Transforms a masked array into a flexible-type array.
The flexible type array that is returned will have two fields:

- the \_data field stores the \_data part of the array.
- the \_mask field stores the \_mask part of the array.

**Parameters**
None

**Returns**

record : ndarray
A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

**Notes**
A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value,...) will be lost.

**Examples**

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.toflex()
[[1, False) (2, True) (3, False)]
[(4, True) (5, False) (6, True)]
[(7, False) (8, True) (9, False)]
```

MaskedArray.tolist(fill_value=None)
Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to fill_value. If fill_value is None, the corresponding entries in the output list will be None.

**Parameters**

- fill_value : scalar, optional
  The value to use for invalid entries. Default is None.

**Returns**

result : list
  The Python list representation of the masked array.

**Examples**

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]
>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]
```

MaskedArray.torecords
Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:
•the _data field stores the _data part of the array.
•the _mask field stores the _mask part of the array.

Parameters
None

Returns
record : ndarray
A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

Notes
A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value, ...) will be lost.

Examples
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
[7 -- 9]]

>>> print x.toflex()
[[[1, False) (2, True) (3, False)]
 [[4, True) (5, False) (6, True)]
 [[7, False) (8, True) (9, False)]]

MaskedArray.tostream (fill_value=None, order='C')
Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

Parameters
fill_value : scalar, optional
Value used to fill in the masked values. Default is None, in which case MaskedArray.fill_value is used.

order : {'C', 'F', 'A'}, optional
Order of the data item in the copy. Default is ‘C’.
• ‘C’ – C order (row major).
• ‘F’ – Fortran order (column major).
• ‘A’ – Any, current order of array.
• None – Same as ‘A’.

See Also:
ndarray.tostream, tolist, tofile

Notes
As for ndarray.tostream, information about the shape, dtype, etc., but also about fill_value, will be lost.
Examples

```python
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00?B\x0f\x00?B\x0f\x00\x04\x00\x00\x00'
```

Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with n integers which will be interpreted as an n-tuple.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaskedArray.flatten(order)</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>MaskedArray.ravel()</code></td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td><code>MaskedArray.reshape(*s, **kwargs)</code></td>
<td>Give a new shape to the array without changing its data.</td>
</tr>
<tr>
<td><code>MaskedArray.resize(newshape[, refcheck, order])</code></td>
<td></td>
</tr>
<tr>
<td><code>MaskedArray.squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of <code>a</code>.</td>
</tr>
<tr>
<td><code>MaskedArray.swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with <code>axis1</code> and <code>axis2</code> interchanged.</td>
</tr>
<tr>
<td><code>MaskedArray.transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

**MaskedArray.flatten(order='C')**

Return a copy of the array collapsed into one dimension.

**Parameters**

- `order` : {'C', 'F', 'A'}, optional
  - Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from `a`. The default is 'C'.

**Returns**

- `y` : ndarray
  - A copy of the input array, flattened to one dimension.

**See Also**

- `ravel`
  - Return a flattened array.
- `flat`
  - A 1-D flat iterator over the array.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

**MaskedArray.ravel()**

Returns a 1D version of self, as a view.

**Returns**

- MaskedArray
  - Output view is of shape (self.size,) (or (np.ma.product(self.shape),)).

1.7. Masked arrays 265
Examples

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]

>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

**MaskedArray.reshape(*s, **kwargs)**

Give a new shape to the array without changing its data.

Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

- **shape** : int or tuple of ints
  
The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order** : {'C', 'F'}, optional
  
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

- **reshaped_array** : array
  
  A new view on the array.

**See Also:**

- **reshape**
  
  Equivalent function in the masked array module.

- **numpy.ndarray.reshape**
  
  Equivalent method on ndarray object.

- **numpy.reshape**
  
  Equivalent function in the NumPy module.

**Notes**

The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`.

**Examples**

```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])

>>> print x
[|-- 2] [3 --]

>>> x = x.reshape((4,1))

>>> print x
[|--] [2] [3] [|--]
```
MaskedArray.**resize**(*newshape*, *refcheck=True*, *order=False*)

**Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the **numpy.ma.resize** function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

MaskedArray.**squeeze**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to **numpy.squeeze** for full documentation.

See Also:

**numpy.squeeze**

equivalent function

MaskedArray.**swapaxes**(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to **numpy.swapaxes** for full documentation.

See Also:

**numpy.swapaxes**

equivalent function

MaskedArray.**transpose**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and *a*.shape = (i[0], i[1], ...

... i[n-2], i[n-1]), then *a*.transpose().shape = (i[n-1], i[n-2], ...

... i[1], i[0]).

Parameters

- **axes** : None, tuple of ints, or *n* ints
  - None or no argument: reverses the order of the axes.
  - tuple of ints: *i* in the *j*-th place in the tuple means *a*’s *i*-th axis becomes *a*.transpose()’s *j*-th axis.
  - *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns

- **out** : ndarray
  View of *a*, with axes suitably permuted.

See Also:

**ndarray.T**

Array property returning the array transposed.

Examples
Item selection and manipulation

For array methods that take an *axis* keyword, it defaults to *None*. If axis is *None*, then the array is treated as a 1-D array. Any other value for *axis* represents the dimension along which the operation should proceed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argmax</code></td>
<td>Returns array of indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td><code>argmin</code></td>
<td>Returns array of indices to the minimum values along the given axis.</td>
</tr>
<tr>
<td><code>argsort</code></td>
<td>Return an ndarray of indices that sort the array along the specified axis.</td>
</tr>
<tr>
<td><code>choose</code></td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td><code>compress</code></td>
<td>Return <em>a</em> where condition is <em>True</em>.</td>
</tr>
<tr>
<td><code>diagonal</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>fill</code></td>
<td>Fill the array with a scalar value.</td>
</tr>
<tr>
<td><code>item</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>nonzero</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>put</code></td>
<td>Set storage-indexed locations to corresponding values.</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>searchsorted</code></td>
<td>Find indices where elements of v should be inserted in a to maintain order.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort the array, in-place</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.</td>
</tr>
</tbody>
</table>

**Parameters**

- **axis** : {None, integer}
  
  If None, the index is into the flattened array, otherwise along the specified axis

- **fill_value** : {var}, optional
  
  Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

- **out** : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.
Returns

\[\text{index\_array : \{integer\_array\}}\]

Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

MaskedArray.\texttt{argmin}(axis=None, fill\_value=None, out=None)
Return array of indices to the minimum values along the given axis.

Parameters

- **axis : \{None, integer\}**
  - If None, the index is into the flattened array, otherwise along the specified axis
- **fill\_value : \{var\}, optional**
  - Value used to fill in the masked values. If None, the output of minimum\_fill\_value(self._data) is used instead.
- **out : \{None, array\}, optional**
  - Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

\{ndarray, scalar\}
If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

Examples

```python
>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[- --]
 [2 3]]
>>> print x.argmin(axis=0, fill\_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill\_value=9)
[1 1]
```

MaskedArray.\texttt{argsort}(axis=None, kind='quicksort', order=None, fill\_value=None)
Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill\_value.

Parameters

- **axis : \text{int, optional}**
  - Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.
- **fill\_value : \text{var, optional}**
Value used to fill the array before sorting. The default is the `fill_value` attribute of the input array.

**kind**: {'quicksort', 'mergesort', 'heapsort'}, optional

Sorting algorithm.

**order**: list, optional

When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

**index_array**: ndarray, int

Array of indices that sort `a` along the specified axis. In other words, `a[index_array]` yields a sorted `a`.

**See Also:**

**sort**

Describes sorting algorithms used.

**lexsort**

Indirect stable sort with multiple keys.

**ndarray.sort**

Inplace sort.

**Notes**

See `sort` for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.ma.array([3,2,1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --]
mask = [False False True],
fill_value = 999999)
```

```python
>>> a.argsort()
array([1, 0, 2])
```

**MaskedArray.choose**(choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See Also:**

`numpy.choose`
equivalent function

**MaskedArray.compress**(condition, axis=None, out=None)

Return `a` where `condition` is True.

If condition is a MaskedArray, missing values are considered as False.

**Parameters**

**condition**: var

Boolean 1-d array selecting which entries to return. If `len(condition)` is less than the size of `a` along the axis, then output is truncated to length of condition array.
**axis** : {None, int}, optional

Axis along which the operation must be performed.

**out** : {None, ndarray}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

**result** : MaskedArray

A MaskedArray object.

**Notes**

Please note the difference with `compressed`! The output of `compress` has a mask, the output of `compressed` does not.

**Examples**

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]

>>> x.compress([1, 0, 1])
mixed_array(data = [1 3],
            mask = [False False],
            fill_value=999999)

>>> x.compress([1, 0, 1], axis=1)
mixed_array(data =
            [[1 3]
             [-- --]
             [7 9]],
            mask =
            [[False False]
             [ True  True]
             [False False]],
            fill_value=999999)
```

MaskedArray.``diagonal`` *(offset=0, axis1=0, axis2=1)*

Return specified diagonals.

Refer to numpy.``diagonal`` for full documentation.

**See Also:**

numpy.``diagonal``

equivalent function

MaskedArray.``fill`` *(value)*

Fill the array with a scalar value.

**Parameters**

**value** : scalar

All elements of `a` will be assigned this value.
Examples

```python
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

MaskedArray. `item(*args)`
Copy an element of an array to a standard Python scalar and return it.

**Parameters**

* *args: Arguments (variable number and type)*

  - none: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar object and returned.
  - `int_type`: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
  - tuple of `int_types`: functions as does a single `int_type` argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

* `z`: Standard Python scalar object

  A copy of the specified element of the array as a suitable Python scalar

**Notes**

When the data type of `a` is longdouble or clongdouble, `item()` returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for `item()`, unless fields are defined, in which case a tuple is returned.

`item` is very similar to `a[args]`, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

**Examples**

```python
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

MaskedArray. `nonzero()`
Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:
a[a.nonzero()]

To group the indices by element, rather than dimension, use instead:

np.transpose(a.nonzero())

The result of this is always a 2d array, with a row for each non-zero element.

Parameters
None

Returns

tuple_of_arrays : tuple

Indices of elements that are non-zero.

See Also:

numpy.nonzero
Function operating on ndarrays.

flatnonzero
Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero
Equivalent ndarray method.

count_nonzero
Counts the number of non-zero elements in the input array.

Examples

>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]],
   fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[ 1.0  0.0  0.0]
 [ 0.0 --  0.0]
 [ 0.0  0.0  1.0]],
   fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
Indices can also be grouped by element.

```python
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `ma.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```python
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
              [[False False False]
               [ True  True  True]
               [ True  True  True]],
              mask = False,
              fill_value=999999)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the condition array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`MaskedArray.put(indices, values, mode='raise')`

Set storage-indexed locations to corresponding values.

Sets self._data.flat[n] = values[n] for each n in indices. If `values` is shorter than `indices` then it will repeat. If `values` has some masked values, the initial mask is updated in consequence, else the corresponding values are unmasked.

**Parameters**
- `indices` : 1-D array_like
  Target indices, interpreted as integers.
- `values` : array_like
  Values to place in self._data copy at target indices.
- `mode` : {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices will behave. ‘raise’: raise an error. ‘wrap’: wrap around. ‘clip’: clip to the range.

**Notes**
`values` can be a scalar or length 1 array.

**Examples**

```python
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-5 --]
 [7 -- 9]]
>>> x.put([0,4,8],[10,20,30])
>>> print x
[[10 -- 3]
 [... 20]
 [7 -- 9]]
```

274 Chapter 1. Array objects
>>> x.put(4, 999)
>>> print x
[[10 -- 3]
[ -- 999 --]
[7 -- 30]]

MaskedArray method `repeat`:

- `repeat(repeats, axis=None)`
- Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

See Also:

- `numpy.repeat` equivalent function

MaskedArray method `searchsorted`:

- `searchsorted(v, side='left', sorter=None)`
  - Find indices where elements of `v` should be inserted in `a` to maintain order.
  - For full documentation, see `numpy.searchsorted`.

See Also:

- `numpy.searchsorted` equivalent function

MaskedArray method `sort`:

- `sort(axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)`
  - Sort the array, in-place.

**Parameters**

- `a`: array_like
  - Array to be sorted.

- `axis`: int, optional
  - Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

- `kind`: {'quicksort', 'mergesort', 'heapsort'}, optional
  - Sorting algorithm. Default is 'quicksort'.

- `order`: list, optional
  - When `a` is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

- `endwith`: {True, False}, optional
  - Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

- `fill_value`: {var}, optional
  - Value used internally for the masked values. If `fill_value` is not None, it supersedes `endwith`.

**Returns**

- `sorted_array`: ndarray

---

1.7. Masked arrays
Array of the same type and shape as $a$.

See Also:

**ndarray.sort**
  Method to sort an array in-place.

**argsort**
  Indirect sort.

**lexsort**
  Indirect stable sort on multiple keys.

**searchsorted**
  Find elements in a sorted array.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

```python
>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Default
>>> a.sort()
>>> print(a)
[1 3 5 -- --]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Put missing values in the front
>>> a.sort(endwith=False)
>>> print(a)
[-- -- 1 3 5]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # fill_value takes over endwith
>>> a.sort(endwith=False, fill_value=3)
>>> print(a)
[1 -- -- 3 5]
```

MaskedArray . take (indices, axis=None, out=None, mode='raise')

Pickling and copy

```python
MaskedArray.copy(order='C')
Return a copy of the array.

MaskedArray.dump(file)
Dump a pickle of the array to the specified file.

MaskedArray.dumps()
Returns the pickle of the array as a string.
```

MaskedArray . copy (order='C')
Return a copy of the array.

**Parameters**

  order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’
    means ‘F’ if $a$ is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of $a$ as
closely as possible. (Note that this function and \texttt{numpy.copy} are very similar, but have different default values for their \texttt{order=} arguments.)

**See Also:**

\texttt{numpy.copy}, \texttt{numpy.copyto}

**Examples**

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')
```

```python
>>> y = x.copy()
```

```python
>>> x.fill(0)
```

```python
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```python
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```python
>>> y.flags['C_CONTIGUOUS']
True
```

\texttt{MaskedArray.dump(file)}

Dump a pickle of the array to the specified file. The array can be read back with \texttt{pickle.load} or \texttt{numpy.load}.

**Parameters**

- **file**: str
  - A string naming the dump file.

\texttt{MaskedArray.dumps()}

Returns the pickle of the array as a string. \texttt{pickle.loads} or \texttt{numpy.loads} will convert the string back to an array.

**Parameters**

- **None**

**Calculations**

<table>
<thead>
<tr>
<th>MaskedArray.all([axis, out])</th>
<th>Check if all of the elements of (a) are true.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.anom([axis, dtype])</td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.any([axis, out])</td>
<td>Check if any of the elements of (a) are true.</td>
</tr>
<tr>
<td>MaskedArray.clip(a_min, a_max[, out])</td>
<td>Return an array whose values are limited to ([a_{\text{min}}, a_{\text{max}}]).</td>
</tr>
<tr>
<td>MaskedArray.conj()</td>
<td>Complex-conjugate all elements.</td>
</tr>
<tr>
<td>MaskedArray.conjugate()</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td>MaskedArray.cumprod([axis, dtype, out])</td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.cumsum([axis, dtype, out])</td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.max([axis, out, fill_value])</td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td>MaskedArray.mean([axis, dtype, out])</td>
<td>Returns the average of the array elements.</td>
</tr>
<tr>
<td>MaskedArray.min([axis, out, fill_value])</td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td>MaskedArray.prod([axis, dtype, out])</td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td>MaskedArray.product([axis, dtype, out])</td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td>MaskedArray.ptp([axis, out, fill_value])</td>
<td>Return ((\text{maximum} - \text{minimum})) along the the given dimension (i.e.</td>
</tr>
</tbody>
</table>

Continued on next page
**Table 1.66 – continued from previous page**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaskedArray.round()</code></td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
<tr>
<td><code>MaskedArray.std()</code></td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td><code>MaskedArray.sum()</code></td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td><code>MaskedArray.trace()</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>MaskedArray.var()</code></td>
<td>Compute the variance along the specified axis.</td>
</tr>
</tbody>
</table>

**MaskedArray.all**

Check if all of the elements of a are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**
- **axis**: {None, integer}
  - Axis to perform the operation over. If None, perform over flattened array.
- **out**: {None, array}, optional
  - Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

- `all`
  - equivalent function

**Examples**

```python
>>> np.ma.array([1, 2, 3]).all()
True
>>> a = np.ma.array([1, 2, 3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

**MaskedArray.anom**

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**
- **axis**: int, optional
  - Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.
- **dtype**: dtype, optional
  - *Type to use in computing the variance. For arrays of integer type*
    - the default is float32; for arrays of float types it is the same as the array type.

**See Also:**

- `mean`
  - Compute the mean of the array.
Examples

```python
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1.  0.  1.],
             mask = False,
             fill_value = 1e+20)
```

MaskedArray.**any** *(axis=None, out=None)*

Check if any of the elements of `a` are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

**Parameters**

- **axis** : {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array and return a scalar.

- **out** : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

- **any** equivalent function

MaskedArray.**clip**(a_min, a_max, out=None)

Return an array whose values are limited to `[a_min, a_max]`.

Refer to `numpy.clip` for full documentation.

**See Also:**

- `numpy.clip` equivalent function

MaskedArray.**conj**()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

**See Also:**

- `numpy.conjugate` equivalent function

MaskedArray.**conjugate**()

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

**See Also:**

- `numpy.conjugate` equivalent function
MaskedArray.cumprod(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

axis : {None, -1, int}, optional

Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

cumprod : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

**Notes**

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

MaskedArray.cumsum(axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array. axis may be negative, in which case it counts from the last to the first axis.

dtype : {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.
Returns

cumsum : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

MaskedArray.max(\(axis=\text{None}, out=\text{None}, fill\_value=\text{None})

Return the maximum along a given axis.

Parameters

axis : {None, int}, optional

Axis along which to operate. By default, axis is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

fill_value : {var}, optional

Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns

amax : array_like

New array holding the result. If out was specified, out is returned.

See Also:

maximum_fill_value

Returns the maximum filling value for a given datatype.

MaskedArray.mean(\(axis=\text{None}, dtype=\text{None}, out=\text{None})

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to numpy.mean for the full documentation.

Parameters

a : array_like

Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : dtype, optional
Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

**mean**: ndarray, see dtype parameter above

If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also:**

* numpy.ma.mean
  Equivalent function.

* numpy.mean
  Equivalent function on non-masked arrays.

* numpy.ma.average
  Weighted average.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> a.mean()
1.5
```

MaskedArray.min (axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

**Parameters**

**axis**: {None, int}, optional

Axis along which to operate. By default, `axis` is None and the flattened input is used.

**out**: array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value**: {var}, optional

Value used to fill in the masked values. If None, use the output of `minimum_fill_value`.

**Returns**

**amin**: array_like

New array holding the result. If `out` was specified, `out` is returned.

**See Also:**

* minimum_fill_value
  Returns the minimum filling value for a given datatype.
MaskedArray.prod(axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

axis : {None, int}, optional

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

See Also:

prod

equivalent function

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([  2.,  12.])

MaskedArray.product(axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

axis : {None, int}, optional

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

**out**: `{None, array}`, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

`product_along_axis` : `{array, scalar}`, see `dtype` parameter above.

Returns an array whose shape is the same as `a` with the specified axis removed. Returns a 0d array when `a` is 1d or `axis=None`. Returns a reference to the specified output array if specified.

**See Also:**

`prod`  
equivalent function

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
given examples
```

MaskedArray.ptp(`axis=None, out=None, fill_value=None`)  
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

**Parameters**

`axis` : `{None, int}`, optional  
Axis along which to find the peaks. If `None` (default) the flattened array is used.

`out` : `{None, array_like}`, optional  
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

`fill_value` : `{var}`, optional  
Value used to fill in the masked values.

**Returns**

`ptp` : ndarray.

A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

MaskedArray.round(`decimals=0, out=None`)  
Return `a` with each element rounded to the given number of decimals.
Refer to `numpy.around` for full documentation.

See Also:

```
numpy.around
```
equivalent function

```
MaskedArray.std(axis=None, dtype=None, out=None, ddof=0)
```
Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

```
a : array_like
```
Calculate the standard deviation of these values.

```
axis : int, optional
```
Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

```
dtype : dtype, optional
```
Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

```
out : ndarray, optional
```
Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

```
ddof : int, optional
```
Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof \) is zero.

```
keepdims : bool, optional
```
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \( arr \).

Returns

```
standard_deviation : ndarray, see dtype parameter above.
```
If \( out \) is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

```
var, mean, nanmean, nanstd, nanvar
```

```
numpy.doc.ufuncs
```
Section "Output arguments"

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{mean}(x))^2)} \).

The average squared deviation is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate
of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with ddof=1, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the std is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1.,  1.])
>>> np.std(a, axis=1)
array([ 0.5,  0.5])
```

In single precision, std() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

_maskedarray_.sum(axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

**Parameters**

- **axis**: {None, -1, int}, optional
  Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

- **sum_along_axis**: MaskedArray or scalar
  An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.
Examples

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.sum()
25
>>> print x.sum(axis=1)
[4 5 16]
>>> print x.sum(axis=0)
[8 5 12]
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>
```

MaskedArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

See Also:

numpy.trace
equivalent function

MaskedArray.var(axis=None, dtype=None, out=None, ddof=0)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- `a` : array_like
  Array containing numbers whose variance is desired. If `a` is not an array, a conversion is attempted.

- `axis` : int, optional
  Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- `dtype` : data-type, optional
  Type to use in computing the variance. For arrays of integer type the default is `float32`; for arrays of float types it is the same as the array type.

- `out` : ndarray, optional
  Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

- `ddof` : int, optional
  “Delta Degrees of Freedom”: the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero.

- `keepdims` : bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.  

1.7. Masked arrays 287
Returns

variance : ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise, a reference to
the output array is returned.

See Also:

std, mean, nanmean, nanstd, nanvar

numpy.doc.ufuncs

Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., 
var = mean(abs(x - x.mean())**2).

The mean is normally calculated as x.sum() / N, where N = len(x). If, however, ddof is specified, the
divisor N - ddof is used instead. In standard statistical practice, ddof=1 provides an unbiased estimator
of the variance of a hypothetical infinite population. ddof=0 provides a maximum likelihood estimate of the
variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and
nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the
input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a
higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1., 1.])
>>> np.var(a, axis=1)
array([0.25, 0.25])

In single precision, var() can be inaccurate:

>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:]=1.0
>>> a[1,:]=0.1
>>> np.var(a)
0.20405951142311096

Computing the variance in float64 is more accurate:

>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.5)**2 + (0.1-0.5)**2)/2
0.20250000000000001

Arithmetic and comparison operations

Comparison operators:
MaskedArray.__lt__
   x.__lt__(y) <==> x<y

MaskedArray.__le__
   x.__le__(y) <==> x<=y

MaskedArray.__gt__
   x.__gt__(y) <==> x>y

MaskedArray.__ge__
   x.__ge__(y) <==> x>=y

MaskedArray.__eq__ (other)
   Check whether other equals self elementwise

MaskedArray.__ne__ (other)
   Check whether other doesn’t equal self elementwise

**Truth value of an array (bool):**

MaskedArray.__nonzero__
   x.__nonzero__() <==> x != 0

**Arithmetic:**

MaskedArray.__abs__ () <==> abs(x)

MaskedArray.__add__ (other)
   Add other to self, and return a new masked array.

MaskedArray.__radd__ (other)
   Add other to self, and return a new masked array.

MaskedArray.__sub__ (other)
   Subtract other to self, and return a new masked array.

MaskedArray.__rsub__ (other)
   Subtract other to self, and return a new masked array.

MaskedArray.__mul__ (other)
   Multiply other by self, and return a new masked array.

MaskedArray.__rmul__ (other)
   Multiply other by self, and return a new masked array.

MaskedArray.__div__ (other)
   Divide other into self, and return a new masked array.

MaskedArray.__rdiv__ (y) <==> y/x

MaskedArray.__truediv__ (other)
   Divide other into self, and return a new masked array.

MaskedArray.__rtruediv__ (other)
   Divide other into self, and return a new masked array.

MaskedArray.__floordiv__ (other)
   Divide other into self, and return a new masked array.

MaskedArray.__rfloordiv__ (other)
   Divide other into self, and return a new masked array.

MaskedArray.__mod__
   x.__mod__(y) <==> x%y

MaskedArray.__rmod__
   x.__rmod__(y) <==> y%x

MaskedArray.__divmod__ (y) <==> divmod(x, y)

MaskedArray.__rdivmod__ (y) <==> divmod(y, x)

MaskedArray.__pow__ (other)
   Raise self to the power other, masking the potential NaNs/Inf
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaskedArray.__rpow__(other)</code></td>
<td>Raise self to the power other, masking the potential NaNs/Infs</td>
</tr>
<tr>
<td><code>MaskedArray.__lshift__</code></td>
<td>x.<strong>lshift</strong>(y) (\iff) (x\ll y)</td>
</tr>
<tr>
<td><code>MaskedArray.__rlshift__</code></td>
<td>x.<strong>rlshift</strong>(y) (\iff) (y\ll x)</td>
</tr>
<tr>
<td><code>MaskedArray.__rshift__</code></td>
<td>x.<strong>rshift</strong>(y) (\iff) (x\gg y)</td>
</tr>
<tr>
<td><code>MaskedArray.__rrshift__</code></td>
<td>x.<strong>rrshift</strong>(y) (\iff) (y\gg x)</td>
</tr>
<tr>
<td><code>MaskedArray.__and__</code></td>
<td>x.<strong>and</strong>(y) (\iff) (x&amp; y)</td>
</tr>
<tr>
<td><code>MaskedArray.__rand__</code></td>
<td>x.<strong>rand</strong>(y) (\iff) (y&amp; x)</td>
</tr>
<tr>
<td><code>MaskedArray.__or__</code></td>
<td>x.<strong>or</strong>(y) (\iff) (x| y)</td>
</tr>
<tr>
<td><code>MaskedArray.__ror__</code></td>
<td>x.<strong>ror</strong>(y) (\iff) (y| x)</td>
</tr>
<tr>
<td><code>MaskedArray.__xor__</code></td>
<td>x.<strong>xor</strong>(y) (\iff) (x\xor y)</td>
</tr>
<tr>
<td><code>MaskedArray.__rxor__</code></td>
<td>x.<strong>rxor</strong>(y) (\iff) (y\xor x)</td>
</tr>
</tbody>
</table>

MaskedArray.__abs__() \(\iff\) \(\text{abs}(x)\)

MaskedArray.__add__(other)
Add other to self, and return a new masked array.

MaskedArray.__radd__(other)
Add other to self, and return a new masked array.

MaskedArray.__sub__(other)
Subtract other to self, and return a new masked array.

MaskedArray.__rsub__(other)
Subtract other to self, and return a new masked array.

MaskedArray.__mul__(other)
Multiply other by self, and return a new masked array.

MaskedArray.__rmul__(other)
Multiply other by self, and return a new masked array.

MaskedArray.__div__(other)
Divide other into self, and return a new masked array.

MaskedArray.__rdiv__(other)
Divide other into self, and return a new masked array.

MaskedArray.__mod__(other)
Divide other into self, and return a new masked array.

MaskedArray.__rmod__(other)
Divide other into self, and return a new masked array.

MaskedArray.__divmod__(other)
\(\text{divmod}(x, y)\)

290 Chapter 1. Array objects
MaskedArray.__rdivmod__(y) <=> divmod(y, x)

MaskedArray.__pow__(other)
    Raise self to the power other, masking the potential NaNs/Infs

MaskedArray.__rpow__(other)
    Raise self to the power other, masking the potential NaNs/Infs

MaskedArray.__lshift__
    x.__lshift__(y) <=> x<<y

MaskedArray.__rlshift__
    x.__rlshift__(y) <=> y<<x

MaskedArray.__rshift__
    x.__rshift__(y) <=> x>>y

MaskedArray.__rrshift__
    x.__rrshift__(y) <=> y>>x

MaskedArray.__and__
    x.__and__(y) <=> x&y

MaskedArray.__rand__
    x.__rand__(y) <=> y&x

MaskedArray.__or__
    x.__or__(y) <=> x|y

MaskedArray.__ror__
    x.__ror__(y) <=> y|x

MaskedArray.__xor__
    x.__xor__(y) <=> x^y

MaskedArray.__rxor__
    x.__rxor__(y) <=> y^x

Arithmetic, in-place:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>iadd</strong></td>
<td>Add other to self in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>isub</strong></td>
<td>Subtract other from self in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>imul</strong></td>
<td>Multiply self by other in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>idiv</strong></td>
<td>Divide self by other in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>itruediv</strong></td>
<td>True divide self by other in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>ifloordiv</strong></td>
<td>Floor divide self by other in-place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>imod</strong></td>
<td>x.<strong>imod</strong>(y) &lt;=&gt; x %= y</td>
</tr>
<tr>
<td>MaskedArray.<strong>ipow</strong></td>
<td>Raise self to the power other, in place.</td>
</tr>
<tr>
<td>MaskedArray.<strong>ilshift</strong></td>
<td>x.<strong>ilshift</strong>(y) &lt;=&gt; x&lt;&lt;=y</td>
</tr>
<tr>
<td>MaskedArray.<strong>irshift</strong></td>
<td>x.<strong>irshift</strong>(y) &lt;=&gt; x&gt;&gt;=y</td>
</tr>
<tr>
<td>MaskedArray.<strong>iand</strong></td>
<td>x.<strong>iand</strong>(y) &lt;=&gt; x&amp;=y</td>
</tr>
<tr>
<td>MaskedArray.<strong>ior</strong></td>
<td>x.<strong>ior</strong>(y) &lt;=&gt; xl=y</td>
</tr>
<tr>
<td>MaskedArray.<strong>ixor</strong></td>
<td>x.<strong>ixor</strong>(y) &lt;=&gt; x^=y</td>
</tr>
</tbody>
</table>

MaskedArray.__iadd__(other)
    Add other to self in-place.

MaskedArray.__isub__(other)
Subtract other from self in-place.

```
MaskedArray.__imul__(other)
Multiply self by other in-place.
```

```
MaskedArray.__idiv__(other)
Divide self by other in-place.
```

```
MaskedArray.__itruediv__(other)
True divide self by other in-place.
```

```
MaskedArray.__ifloordiv__(other)
Floor divide self by other in-place.
```

```
MaskedArray.__imod__
x.__imod__(y) <=> x%=y
```

```
MaskedArray.__ipow__(other)
Raise self to the power other, in place.
```

```
MaskedArray.__ilshift__
x.__ilshift__(y) <=> x<<=y
```

```
MaskedArray.__irshift__
x.__irshift__(y) <=> x>>=y
```

```
MaskedArray.__iand__
x.__iand__(y) <=> x&=y
```

```
MaskedArray.__ior__
x.__ior__(y) <=> x|=y
```

```
MaskedArray.__ixor__
x.__ixor__(y) <=> x^=y
```

**Representation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>repr</strong>()</td>
<td>Literal string representation.</td>
</tr>
<tr>
<td>MaskedArray.<strong>str</strong>()</td>
<td>String representation.</td>
</tr>
<tr>
<td>MaskedArray.ids()</td>
<td>Return the addresses of the data and mask areas.</td>
</tr>
<tr>
<td>MaskedArray.iscontiguous()</td>
<td>Return a boolean indicating whether the data is contiguous.</td>
</tr>
</tbody>
</table>

```
MaskedArray.__repr__()
Literal string representation.
```

```
MaskedArray.__str__()
String representation.
```

```
MaskedArray.ids()
Return the addresses of the data and mask areas.
```

```
MaskedArray.iscontiguous()
Return a boolean indicating whether the data is contiguous.
```

**Examples**

```
>>> x = np.ma.array([1, 2, 3], mask=[0, 1, 1])
>>> x.ids()
(166670640, 166659832)
```
If the array has no mask, the address of \texttt{nomask} is returned. This address is typically not close to the data in memory:

\begin{verbatim}
>>> x = np.ma.array([1, 2, 3])
>>> x.ids()
(166691080, 3083169284L)
\end{verbatim}

\texttt{MaskedArray.iscontiguous()}  
Return a boolean indicating whether the data is contiguous.

\textbf{Parameters}

None

\textbf{Examples}

\begin{verbatim}
>>> x = np.ma.array([1, 2, 3])
>>> x.iscontiguous()
True
\end{verbatim}

\texttt{iscontiguous} returns one of the flags of the masked array:

\begin{verbatim}
>>> x.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
\end{verbatim}

\textbf{Special methods}

For standard library functions:

\begin{verbatim}
MaskedArray.__copy__((order))
Return a copy of the array.

MaskedArray.__deepcopy__((memo))

MaskedArray.__getstate__()
Return the internal state of the masked array, for pickling.

MaskedArray.__reduce__()
Return a 3-tuple for pickling a MaskedArray.

MaskedArray.__setstate__(state)
Restore the internal state of the masked array, for pickling purposes.
\end{verbatim}

\texttt{MaskedArray.__copy__((order))}

Return a copy of the array.

\textbf{Parameters}

\texttt{order} : \{‘C’, ‘F’, ‘A’\}, optional

If order is ‘C’ (False) then the result is contiguous (default). If order is ‘Fortran’ (True) then the result has fortran order. If order is ‘Any’ (None) then the result has fortran order only if the array already is in fortran order.

\texttt{MaskedArray.__deepcopy__(memo=)}

\texttt{MaskedArray.__getstate__()}

Return the internal state of the masked array, for pickling purposes.

\texttt{MaskedArray.__reduce__()}

Return a 3-tuple for pickling a MaskedArray.
**MaskedArray.__setstate__(state)**

Restore the internal state of the masked array, for pickling purposes. `state` is typically the output of the `__getstate__` output, and is a 5-tuple:

- class name
- a tuple giving the shape of the data
- a typecode for the data
- a binary string for the data
- a binary string for the mask.

**Basic customization:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaskedArray.__new__(...)</code></td>
<td>Create a new masked array from scratch.</td>
</tr>
<tr>
<td><code>MaskedArray.__array__(...)</code></td>
<td>Returns either a new reference to self if dtype is not given or a new array</td>
</tr>
<tr>
<td><code>MaskedArray.__array_wrap__(obj[, context])</code></td>
<td>Special hook for ufuncs.</td>
</tr>
</tbody>
</table>

**static MaskedArray.__new__(data=None, mask=False, dtype=None, copy=False, subok=True, ndmin=0, fill_value=None, keep_mask=True, hard_mask=None, shrink=True, **options)**

Create a new masked array from scratch.

**Notes**

A masked array can also be created by taking a `.view(MaskedArray)`.

**MaskedArray.__array__(ldtype) → reference if type unchanged, copy otherwise.**

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

**MaskedArray.__array_wrap__(obj, context=None)**

Special hook for ufuncs. Wraps the numpy array and sets the mask according to context.

**Container customization:** (see Indexing)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MaskedArray.__len__() &lt;==&gt; len(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>MaskedArray.__getitem__(indx)</code></td>
<td>x.<strong>getitem</strong>(y) &lt;==&gt; x[y]</td>
</tr>
<tr>
<td><code>MaskedArray.__setitem__(indx, value)</code></td>
<td>x.<strong>setitem</strong>(y) &lt;==&gt; x[y] = value</td>
</tr>
<tr>
<td><code>MaskedArray.__delitem__(i)</code></td>
<td>x.<strong>delitem</strong>(y) &lt;==&gt; del x[y]</td>
</tr>
<tr>
<td><code>MaskedArray.__getslice__(i, j)</code></td>
<td>x.<strong>getslice</strong>(y) &lt;==&gt; x[i:j]</td>
</tr>
<tr>
<td><code>MaskedArray.__setslice__(i, j, value)</code></td>
<td>x.<strong>setslice</strong>(y) &lt;==&gt; x[i:j] = value</td>
</tr>
<tr>
<td><code>MaskedArray.__contains__(i, j, value)</code></td>
<td>x.<strong>contains</strong>(y) &lt;==&gt; y in x</td>
</tr>
</tbody>
</table>

**MaskedArray.__len__(...) <==> len(x)**

**Return the item described by i, as a masked array.**

**MaskedArray.__getitem__(indx)**

x.__getitem__(y) <==> x[y]

**Return the item described by i, as a masked array.**

**MaskedArray.__setitem__(indx)**

x.__setitem__(y) <==> x[y]

**Set item described by index. If value is masked, masks those locations.**

**MaskedArray.__delitem__**
x.__delitem__(y) ﬁ___ del x[y]

MaskedArray.__getslice__(i, j)
x.__getslice__(i, j) ﬁ___ x[i:j]

Return the slice described by (i, j). The use of negative indices is not supported.

MaskedArray.__setslice__(i, j, value)
x.__setslice__(i, j, value) ﬁ___ x[i:j]=value

Set the slice (i,j) of a to value. If value is masked, mask those locations.

MaskedArray.__contains__
x.__contains__(y) ﬁ___ y in x

**Specific methods**

**Handling the mask**

The following methods can be used to access information about the mask or to manipulate the mask.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaskedArray.<strong>setmask</strong>(mask[, copy])</td>
<td>Set the mask.</td>
</tr>
<tr>
<td>MaskedArray.harden_mask()</td>
<td>Force the mask to hard.</td>
</tr>
<tr>
<td>MaskedArray.soften_mask()</td>
<td>Force the mask to soft.</td>
</tr>
<tr>
<td>MaskedArray.unshare_mask()</td>
<td>Copy the mask and set the sharedmask flag to False.</td>
</tr>
<tr>
<td>MaskedArray.shrink_mask()</td>
<td>Reduce a mask to nomask when possible.</td>
</tr>
</tbody>
</table>

MaskedArray.__setmask__(mask, copy=False)
Set the mask.

MaskedArray.harden_mask()
Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. harden_mask sets hardmask to True.

**See Also:**

hardmask

MaskedArray.soften_mask()
Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. soften_mask sets hardmask to False.

**See Also:**

hardmask

MaskedArray.unshare_mask()
Copy the mask and set the sharedmask flag to False.

Whether the mask is shared between masked arrays can be seen from the sharedmask property. unshare_mask ensures the mask is not shared. A copy of the mask is only made if it was shared.

**See Also:**

sharedmask

MaskedArray.shrink_mask()
Reduce a mask to nomask when possible.

1.7. Masked arrays 295
Handling the fill_value

**MaskedArray.get_fill_value()**  Return the filling value of the masked array.

**MaskedArray.set_fill_value(value)**  Set the filling value of the masked array.

MaskedArray.get_fill_value()

Return the filling value of the masked array.

**Returns**

fill_value : scalar

The filling value.

**Examples**

```python
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
    ...    np.ma.array([0, 1], dtype=dt).get_fill_value()
    ...
999999
999999
1e+20
(1e+20+0j)
```

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf
```

MaskedArray.set_fill_value(value=None)

Set the filling value of the masked array.

**Parameters**

value : scalar, optional

The new filling value. Default is None, in which case a default based on the data type is used.

**See Also:**

ma.set_fill_value

Equivalent function.
Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931

Reset to default:

```python
>>> x.set_fill_value()
```x.fill_value
1e+20

Counting the missing elements

MaskedArray.count([axis])  Count the non-masked elements of the array along the given axis.

MaskedArray.count(axis=None)
Count the non-masked elements of the array along the given axis.

Parameters

axis : int, optional
Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

Returns

result : int or ndarray
If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

count_masked
Count masked elements in array or along a given axis.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :) = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- -- --]],
mask =
[[False False False]
[ True  True  True]],
fill_value = 999999)
>>> a.count()
3
```

When the axis keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([[1, 1, 1]])
```
1.7.7 Masked array operations

Constants

```
ma.MaskType  Numpy's Boolean type. Character code: ?. Alias: bool8
```

```
numpy.ma.MaskType
  alias of bool8
```

Creation

From existing data

```
ma.masked_array  An array class with possibly masked values.
ma.array(data[, dtype, copy, order, mask, ...])  An array class with possibly masked values.
ma.copy          copy
ma.frombuffer(buffer[, dtype, count, offset])  Interpret a buffer as a 1-dimensional array.
ma.fromfunction(function, shape, **kwargs)  Construct an array by executing a function over each coordinate.
ma.MaskedArray.copy([order])  Return a copy of the array.
```

```
numpy.ma.masked_array  alias of MaskedArray
```

```
numpy.ma.array(data, dtype=None, copy=False, order=\_\_\_\_\_\_, mask=\_\_\_\_\_\_, fill_value=None, keep_mask=True, hard_mask=False, shrink=True, subok=True, ndmin=0)  An array class with possibly masked values.
```

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None,
                copy=False, subok=True, ndmin=0, fill_value=None,
                keep_mask=True, hard_mask=None, shrink=True)
```

Parameters

- **data**: array_like
  - Input data.
- **mask**: sequence, optional
  - Mask. Must be convertible to an array of booleans with the same shape as data. True indicates a masked (i.e. invalid) data.
- **dtype**: dtype, optional
  - Data type of the output. If dtype is None, the type of the data argument (data.dtype) is used. If dtype is not None and different from data.dtype, a copy is performed.
- **copy**: bool, optional
  -
Whether to copy the input data (True), or to use a reference instead. Default is False.

subok : bool, optional

Whether to return a subclass of MaskedArray if possible (True) or a plain MaskedArray. Default is True.

ndmin : int, optional

Minimum number of dimensions. Default is 0.

fill_value : scalar, optional

Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

keep_mask : bool, optional

Whether to combine mask with the mask of the input data, if any (True), or to use only mask for the output (False). Default is True.

hard_mask : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be un-masked. Default is False.

shrink : bool, optional

Whether to force compression of an empty mask. Default is True.

numpy.ma.copy = <numpy.ma.core._frommethod instance at 0x2820290>

copy a.copy(order='C')

Return a copy of the array.

Parameters


Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a as closely as possible. (Note that this function and :func:numpy.copy are very similar, but have different default values for their order= arguments.)

Examples

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```
**numpy.ma.frombuffer** (buffer, dtype=float, count=-1, offset=0) = <numpy.ma.core._convert2ma instance at 0x2820dd0>

Interpret a buffer as a 1-dimensional array.

**Parameters**

- **buffer**: buffer_like
  An object that exposes the buffer interface.
- **dtype**: data-type, optional
  Data-type of the returned array; default: float.
- **count**: int, optional
  Number of items to read. -1 means all data in the buffer.
- **offset**: int, optional
  Start reading the buffer from this offset; default: 0.

**Notes**

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

**Examples**

>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
dtype='|S1')

**numpy.ma.fromfunction** (function, shape, **kwargs) = <numpy.ma.core._convert2ma instance at 0x2820e18>

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value \(fn(x, y, z)\) at coordinate \((x, y, z)\).

**Parameters**

- **function**: callable
  The function is called with N parameters, where N is the rank of `shape`. Each parameter represents the coordinates of the array varying along a specific axis. For example, if `shape` were (2, 2), then the parameters in turn be (0, 0), (0, 1), (1, 0), (1, 1).
- **shape**: (N,) tuple of ints
  Shape of the output array, which also determines the shape of the coordinate arrays passed to `function`.
- **dtype**: data-type, optional
  Data-type of the coordinate arrays passed to `function`. By default, `dtype` is float.

**Returns**

- **fromfunction**: any
The result of the call to function is passed back directly. Therefore the shape of fromfunction is completely determined by function. If function returns a scalar value, the shape of fromfunction would match the shape parameter.

See Also:
indices, meshgrid

Notes
Keywords other than dtype are passed to function.

Examples
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)

>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

MaskedArray.copy(order='C')
Return a copy of the array.

Parameters
order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’
    means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a as
    closely as possible. (Note that this function and :func:numpy.copy are very similar, but
    have different default values for their order= arguments.)

See Also:
numpy.copy, numpy.copyto

Examples
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True

Ones and zeros
NumPy Reference, Release 1.8.1

numpy.ma.empty(shape, dtype=\texttt{float}, order='C') = <numpy.ma.core._convert2ma instance at 0x2820cf8>

Return a new array of given shape and type, without initializing entries.

Parameters

- \texttt{shape} : int or tuple of int
  Shape of the empty array

- \texttt{dtype} : data-type, optional
  Desired output data-type.

- \texttt{order} : {\texttt{C}, \texttt{F}}, optional
  Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See Also:

- empty_like, zeros, ones

Notes

\texttt{empty}, unlike \texttt{zeros}, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

```python
code readable
```

```python
>>> np.empty([2, 2])
array([[ 4.01540289e-408,  4.01540289e-408],
       [ 1.07496955e-398,  1.07496955e-398]]) #random
```

```python
>>> np.empty([2, 2], dtype=int)
array([[1, 2],
       [3, 4]]) #random
```

numpy.ma.empty_like(a, dtype=None, order='K', subok=True) = <numpy.ma.core._convert2ma instance at 0x2820d88>

Return a new array with the same shape and type as a given array.

Parameters

- \texttt{a} : array_like
  The shape and data-type of \texttt{a} define these same attributes of the returned array.

- \texttt{dtype} : data-type, optional
  New in version 1.6.0. Overrides the data type of the result.

- \texttt{order} : {\texttt{C}, \texttt{F}, \texttt{A}, or \texttt{K}}, optional
  New in version 1.6.0. Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if \texttt{a} is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of \texttt{a} as closely as possible.
subok : bool, optional.

If True, then the newly created array will use the sub-class type of ‘a’, otherwise it will
be a base-class array. Defaults to True.

Returns
out : ndarray

Array of uninitialized (arbitrary) data with the same shape and type as a.

See Also:
ones_like
Return an array of ones with shape and type of input.
zeros_like
Return an array of zeros with shape and type of input.
empty
Return a new uninitialized array.
ones
Return a new array setting values to one.
zeros
Return a new array setting values to zero.

Notes
This function does not initialize the returned array; to do that use zeros_like or ones_like instead. It may be
marginally faster than the functions that do set the array values.

Examples
>>> a = ([1,2,3], [4,5,6])                    # a is array-like
>>> np.empty_like(a)
array([[-1073741821, -1073741821, 3],   #random
       [ 0, 0, -1073741821]])
>>> a = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> np.empty_like(a)
array([[-2.00000715e+000, 1.48219694e-323, -2.00000572e+000],  #random
       [ 4.38791518e-305, -2.00000715e+000, 4.17269252e-309]])

numpy.ma.masked_all(shape, dtype=<type 'float'>)

Empty masked array with all elements masked.

Return an empty masked array of the given shape and dtype, where all the data are masked.

Parameters
shape : tuple
   Shape of the required MaskedArray.

dtype : dtype, optional
   Data type of the output.

Returns
a : MaskedArray
   A masked array with all data masked.

See Also:
**masked_all_like**

Empty masked array modelled on an existing array.

**Examples**

```python
>>> import numpy.ma as ma
>>> ma.masked_all((3, 3))
masked_array(data =
[-- -- --]
[-- -- --]
[-- -- --],
mask =
[ True True True]
[ True True True]
[ True True True],
fill_value=1e+20)
```

The `dtype` parameter defines the underlying data type.

```python
>>> a = ma.masked_all((3, 3))
>>> a.dtype
dtype('float64')
>>> a = ma.masked_all((3, 3), dtype=np.int32)
>>> a.dtype
dtype('int32')
```

```
import numpy.ma as ma
arr = np.zeros((2, 3), dtype=np.float32)
arr
array([[ 0., 0., 0.],
[ 0., 0., 0.]], dtype=float32)
ma.masked_all_like(arr)
mixed_array(data =
[-- -- --])
```

numPy.ma.masked_all_like(arr)

Empty masked array with the properties of an existing array.

Return an empty masked array of the same shape and dtype as the array `arr`, where all the data are masked.

**Parameters**

- `arr` : ndarray
  
  An array describing the shape and dtype of the required `MaskedArray`.

**Returns**

- `a` : MaskedArray
  
  A masked array with all data masked.

**Raises**

AttributeError

If `arr` doesn’t have a shape attribute (i.e. not an ndarray)

**See Also:**

**masked_all**

Empty masked array with all elements masked.

**Examples**

```python
>>> import numpy.ma as ma
>>> arr = np.zeros((2, 3), dtype=np.float32)
>>> arr
array([[ 0., 0., 0.],
[ 0., 0., 0.]], dtype=float32)
>>> ma.masked_all_like(arr)
mixed_array(data =
[-- -- --])
```
The dtype of the masked array matches the dtype of `arr`.

```python
>>> arr.dtype
dtype('float32')
```

```python
>>> ma.masked_all_like(arr).dtype
dtype('float32')
```

**numpy.ma.ones** *(shape*, *dtype=None*, *order='C'*)  =  `<numpy.ma.core._convert2ma instance at 0x2820ef0>`

Return a new array of given shape and type, filled with ones.

**Parameters**

- **shape**: int or sequence of ints
  Shape of the new array, e.g., (2, 3) or 2.

- **dtype**: data-type, optional
  The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

- **order**: {'C', 'F'}, optional
  Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

- **out**: ndarray
  Array of ones with the given shape, dtype, and order.

**See Also:**

`zeros`, `ones_like`

**Examples**

```python
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```

```python
>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])
```

```python
>>> np.ones((2, 1))
array([[1.],
       [1.]])
```

```python
>>> s = (2, 2)
>>> np.ones(s)
array([[1., 1.],
       [1., 1.]])
```

**numpy.ma.zeros** *(shape*, *dtype=float*, *order='C'*)  =  `<numpy.ma.core._convert2ma instance at 0x2820f80>`

Return a new array of given shape and type, filled with zeros.

**Parameters**

- **shape**: int or sequence of ints
Shape of the new array, e.g., (2, 3) or 2.

**dtype**: data-type, optional

The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order**: {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

**out**: ndarray

Array of zeros with the given shape, dtype, and order.

**See Also:**

zeros_like

Return an array of zeros with shape and type of input.

ones_like

Return an array of ones with shape and type of input.

empty_like

Return an empty array with shape and type of input.

ones

Return a new array setting values to one.

empty

Return a new uninitialized array.

**Examples**

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])

>>> s = (2,2)
>>> np.zeros(s)
array([[ 0., 0.],
       [ 0., 0.]])

>>> np.zeros((2,2), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([[0, 0],
       [0, 0]],
       dtype=[('x', '<i4'), ('y', '<i4')])
```

---

**Inspecting the array**

```python
ma.all(self[, axis, out])
```

Check if all of the elements of a are true.
Table 1.81 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.any(self[, axis, out])</code></td>
<td>Check if any of the elements of a are true.</td>
</tr>
<tr>
<td><code>ma.count(a[, axis])</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.count_masked(arr[, axis])</code></td>
<td>Count the number of masked elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.getmask(a)</code></td>
<td>Return the mask of a masked array, or nomask.</td>
</tr>
<tr>
<td><code>ma.getmaskarray(arr)</code></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><code>ma.getdata(a[, subok])</code></td>
<td>Return the data of a masked array as an ndarray.</td>
</tr>
<tr>
<td><code>ma.nonzero[](self)</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.shape(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.data</code></td>
<td>Return the current data, as a view of the original</td>
</tr>
<tr>
<td><code>ma.MaskedArray.mask</code></td>
<td>Mask</td>
</tr>
<tr>
<td><code>ma.MaskedArray.recordmask</code></td>
<td>Return the mask of the records.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.all([][, axis, out])</code></td>
<td>Check if all of the elements of a are true.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.any([][, axis, out])</code></td>
<td>Check if any of the elements of a are true.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.count([axis])</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.nonzero()</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.shape(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.size(obj[, axis])</code></td>
<td>Return the number of elements along a given axis.</td>
</tr>
</tbody>
</table>

**numpy.ma.all(self, axis=None, out=None)** = `<numpy.ma.core._frommethod instance at 0x2819e18>

Check if all of the elements of a are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

- `axis` : {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array.

- `out` : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

- `all`
  
  equivalent function

**Examples**

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

**numpy.ma.any(self, axis=None, out=None)** = `<numpy.ma.core._frommethod instance at 0x2819fc8>

Check if any of the elements of a are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

**Parameters**

- `axis` : {None, integer}

  Axis to perform the operation over. If None, perform over flattened array.
Axis to perform the operation over. If None, perform over flattened array and return a scalar.

**out**: {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

*any*  
equivalent function

**numpy.ma.count** *(a, axis=None)*  
Count the non-masked elements of the array along the given axis.

**Parameters**

*axis*: int, optional

Axis along which to count the non-masked elements. If *axis* is None, all non-masked elements are counted.

**Returns**

*result*: int or ndarray

If *axis* is None, an integer count is returned. When *axis* is not None, an array with shape determined by the lengths of the remaining axes, is returned.

**See Also:**

*count_masked*  
Count masked elements in array or along a given axis.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
[[ 0  1  2]
 [-- -- --]],
mask =
[[False False False]
 [ True  True  True]],
fill_value = 999999)
>>> a.count()
3

When the *axis* keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

**numpy.ma.count_masked** *(arr, axis=None)*  
Count the number of masked elements along the given axis.

**Parameters**

*arr*: array_like
An array with (possibly) masked elements.

**axis** : int, optional

Axis along which to count. If None (default), a flattened version of the array is used.

**Returns**

**count** : int, ndarray

The total number of masked elements (axis=None) or the number of masked elements
along each slice of the given axis.

**See Also:**

**MaskedArray.count**
Count non-masked elements.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(9).reshape((3,3))
>>> a = ma.array(a)
>>> a[1, 0] = ma.masked
>>> a[1, 2] = ma.masked
>>> a[2, 1] = ma.masked
>>> a
masked_array(data =
 [[0 1 2]
  [-- 4 --]
  [6 -- 8]],
mask =
[[False False False]
 [ True False True]
 [False True False]],
fill_value=999999)
>>> ma.count_masked(a)
3

When the *axis* keyword is used an array is returned.

```python
>>> ma.count_masked(a, axis=0)
array([1, 1, 1])
```
getmaskarray

Return the mask of a masked array, or full array of False.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =

   [[1 --]
    [3 4]],

   mask =

   [[False  True]
    [False False]],

   fill_value=999999)
```

```python
>>> ma.getmask(a)
array([[False, True],
   [False, False]], dtype=bool)
```

Equivalently use the MaskedArray mask attribute.

```python
>>> a.mask
array([[False, True],
   [False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =

   [[1 2]
    [3 4]],

   mask = False,

   fill_value=999999)
```

```python
>>> ma.nomask
False
```

```python
>>> ma.getmask(b) == ma.nomask
True
```

```python
>>> b.mask == ma.nomask
True
```

numpy.ma.getmaskarray(arr)

Return the mask of a masked array, or full boolean array of False.

Return the mask of arr as an ndarray if arr is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as arr.

Parameters

    arr : array_like

    Input MaskedArray for which the mask is required.

See Also:

    getmask

    Return the mask of a masked array, or nomask.

    getdata

    Return the data of a masked array as an ndarray.
Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],
 mask =
[[False True]
 [False False]],
 fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
 [False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
 [3 4]],
 mask =
False,
 fill_value=999999)
>>> >ma.getmaskarray(b)
array([[False, False],
 [False, False]], dtype=bool)
```

```python
numpy.ma.getdata(a, subok=True)
Return the data of a masked array as an ndarray.
Return the data of a (if any) as an ndarray if a is a MaskedArray, else return a as a ndarray or subclass (depending on subok) if not.

Parameters
a : array_like
    Input MaskedArray, alternatively a ndarray or a subclass thereof.
subok : bool
    Whether to force the output to be a pure ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

See Also:

getmask
Return the mask of a masked array, or nomask.
getmaskarray
Return the mask of a masked array, or full array of False.
```

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],
 mask =
[[False True]
 [False False]],
 fill_value=999999)
```
>>> [3 4],
    mask =
    [[False True]
    [False False]],
    fill_value=999999)

>>> ma.getdata(a)
array([[1, 2],
    [3, 4]])

Equivalently use the MaskedArray data attribute.

>>> a.data
array([[1, 2],
    [3, 4]])

numpy.ma.nonzero(self) = <numpy.ma.core._frommethod instance at 0x28204d0>

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

a[a.nonzero()]

To group the indices by element, rather than dimension, use instead:

np.transpose(a.nonzero())

The result of this is always a 2d array, with a row for each non-zero element.

Parameters

None

Returns

tuple_of_arrays : tuple

Indices of elements that are non-zero.

See Also:

numpy.nonzero

Function operating on ndarrays.

flatnonzero

Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero

Equivalent ndarray method.

count_nonzero

Counts the number of non-zero elements in the input array.

Examples

>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
    [[ 1.  0.  0.]
    [ 0.  1.  0.]
    [ 0.  0.  1.]],
    mask =
False,
      fill_value=1e+20)

>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
 [0.0 -- 0.0]
 [0.0 0.0 1.0]],
 mask =
[[False False False]
 [False True False]
 [False False False]],
 fill_value=1e+20)

>>> x.nonzero()
(array([0, 2]), array([0, 2]))

Indices can also be grouped by element.

>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array a, the condition a > 3 is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the a where the condition is true.

>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])

>>> a > 3
masked_array(data =
[[False False False]
 [ True True True]
 [ True True True]],
 mask =
False,
      fill_value=999999)

>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

The nonzero method of the condition array can also be called.

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

numpy.ma.shape(obj)

Return the shape of an array.

Parameters

a : array_like

Input array.

Returns

shape : tuple of ints

The elements of the shape tuple give the lengths of the corresponding array dimensions.
See Also:

alen
ndarray.shape
   Equivalent array method.

Examples

```python
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
```

```
numpy.ma.size(obj, axis=None)
   Return the number of elements along a given axis.

Parameters

   a : array_like
      Input data.
   axis : int, optional
      Axis along which the elements are counted. By default, give the total number of ele-
      ments.

Returns

   element_count : int
      Number of elements along the specified axis.

See Also:

shape
dimensions of array
ndarray.shape
dimensions of array
ndarray.size
   number of elements in array

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
>>> np.size(a, 0)
2
```
MaskedArray.\texttt{data}  
Return the current data, as a view of the original underlying data.

MaskedArray.\texttt{mask}  
Mask

MaskedArray.\texttt{recordmask}  
Return the mask of the records. A record is masked when all the fields are masked.

MaskedArray.\texttt{all} (\texttt{axis=\text{n}one, out=\text{n}one})  
Check if all of the elements of a are true.

   Performs a \texttt{logical_and} over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is \texttt{masked}.

   \begin{description}
   \item[Parameters]\begin{itemize}
   \item \texttt{axis} : {\text{n}one, integer}
   \end{itemize}
   \end{description}

   \begin{description}
   \item[See Also:]
   \begin{itemize}
   \item \texttt{all}
   \end{itemize}
   \end{description}

   \begin{description}
   \item[Examples]
   \begin{itemize}
   \item \texttt{np.ma.array([1,2,3]).all()}
   \item \texttt{True}
   \item \texttt{a = np.ma.array([1,2,3], mask=True)}
   \item \texttt{(a.all() is np.ma.masked)}
   \item \texttt{True}
   \end{itemize}
   \end{description}

MaskedArray.\texttt{any} (\texttt{axis=\text{n}one, out=\text{n}one})  
Check if any of the elements of a are true.

   Performs a \texttt{logical_or} over the given axis and returns the result. Masked values are considered as False during computation.

   \begin{description}
   \item[Parameters]\begin{itemize}
   \item \texttt{axis} : {\text{n}one, integer}
   \end{itemize}
   \end{description}

   \begin{description}
   \item[See Also:]
   \begin{itemize}
   \item \texttt{any}
   \end{itemize}
   \end{description}

   \begin{description}
   \item[Examples]
   \begin{itemize}
   \item \texttt{np.ma.array([1,2,3]).all()}
   \item \texttt{True}
   \item \texttt{a = np.ma.array([1,2,3], mask=True)}
   \item \texttt{(a.all() is np.ma.masked)}
   \item \texttt{True}
   \end{itemize}
   \end{description}
MaskedArray.count(axis=None)
Count the non-masked elements of the array along the given axis.

Parameters
axis : int, optional
Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

Returns
result : int or ndarray
If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:
count_masked
Count masked elements in array or along a given axis.

Examples
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :]= ma.masked
>>> a
masked_array(data =
[[0 1 2]],
mask =
[False False False]
[ True True True],
fill_value = 999999)
>>> a.count()
3
When the axis keyword is specified an array of appropriate size is returned.

>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])

MaskedArray.nonzero()
Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

a[a.nonzero()]

To group the indices by element, rather than dimension, use instead:

np.transpose(a.nonzero())

The result of this is always a 2d array, with a row for each non-zero element.

Parameters
None

Returns
tuple_of_arrays : tuple
Indices of elements that are non-zero.

See Also:

numpy.nonzero
Function operating on ndarrays.

flatnonzero
Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero
Equivalent ndarray method.

count_nonzero
Counts the number of non-zero elements in the input array.

Examples

>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]],
mask =
False, fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
[0.0 -- 0.0]
[0.0 0.0 1.0]],
mask =
[[False False False]
[False  True False]
[False False False]],
fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))

Indices can also be grouped by element.

>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array \( a \), the condition \( a > 3 \) is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the \( a \) where the condition is true.

>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
The \texttt{nonzero} method of the condition array can also be called.

\begin{verbatim}
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
\end{verbatim}

\texttt{numpy.ma.shape} \texttt{(obj)}

Return the shape of an array.

\begin{itemize}
  \item \textbf{Parameters}
    \begin{itemize}
      \item \texttt{a} : array_like
    \end{itemize}
  \item \textbf{Input array.}
  \item \textbf{Returns}
    \begin{itemize}
      \item \texttt{shape} : tuple of ints
    \end{itemize}
  \item The elements of the shape tuple give the lengths of the corresponding array dimensions.
\end{itemize}

\textbf{See Also:}

\texttt{alen}

\texttt{ndarray.shape}

Equivalent array method.

\textbf{Examples}

\begin{verbatim}
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
>>> a = np.array([[1, 2], (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
\end{verbatim}

\texttt{numpy.ma.size} \texttt{(obj, axis=None)}

Return the number of elements along a given axis.

\begin{itemize}
  \item \textbf{Parameters}
    \begin{itemize}
      \item \texttt{a} : array_like
    \end{itemize}
  \item \texttt{Input data.}
  \item \texttt{axis} : int, optional
\end{itemize}
Axis along which the elements are counted. By default, give the total number of elements.

Returns
  element_count : int

Number of elements along the specified axis.

See Also:

shape
dimensions of array

ndarray.shape
dimensions of array

ndarray.size
number of elements in array

Examples

>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
>>> np.size(a, 0)
2

Manipulating a MaskedArray

Changing the shape

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.ravel(self)</td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td>ma.reshape(a, new_shape[, order])</td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td>ma.resize(x, new_shape)</td>
<td>Return a new masked array with the specified size and shape.</td>
</tr>
<tr>
<td>ma.MaskedArray.flatten([order])</td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td>ma.MaskedArray.reshape(*s, **kwargs)</td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td>ma.MaskedArray.resize(newshape[, refcheck, ...])</td>
<td>Give a new shape to the array without changing its data.</td>
</tr>
</tbody>
</table>

numpy.ma.ravel(self) = <numpy.ma.core._frommethod instance at 0x2820680>

Returns a 1D version of self, as a view.

Returns
  MaskedArray

Output view is of shape (self.size,) (or (np.ma.product(self.shape),)).

Examples

>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
>>> print x
[[1 -- 3]

1.7. Masked arrays
```python
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]

numpy.ma.reshape(a, new_shape, order='C')
Returns an array containing the same data with a new shape.

Refer to MaskedArray.reshape for full documentation.

See Also:

MaskedArray.reshape
equivalent function

numpy.ma.resize(x, new_shape)
Return a new masked array with the specified size and shape.

This is the masked equivalent of the numpy.resize function. The new array is filled with repeated copies of x (in the order that the data are stored in memory). If x is masked, the new array will be masked, and the new mask will be a repetition of the old one.

See Also:

numpy.resize
Equivalent function in the top level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.array([[1, 2], [3, 4]])
>>> a[0, 1] = ma.masked
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False True]
[False False]],
fill_value = 999999)
>>> np.resize(a, (3, 3))
array([[1, 2, 3],
[4, 1, 2],
[3, 4, 1]])
>>> ma.resize(a, (3, 3))
masked_array(data =
[[1 -- 3]
[4 1 --]
[3 4 1]],
mask =
[[False True False]
[False False True]
[False False False]],
fill_value = 999999)
```

A MaskedArray is always returned, regardless of the input type.

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> ma.resize(a, (3, 3))
```
masked_array(data =
    [[1 2 3]
     [4 1 2]
     [3 4 1]],
    mask =
    False,
    fill_value = 999999)

MaskedArray.flatten(order='C')
Return a copy of the array collapsed into one dimension.

Parameters
    order : {'C', 'F', 'A'}, optional
        Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the
        C/Fortran ordering from a. The default is 'C'.

Returns
    y : ndarray
        A copy of the input array, flattened to one dimension.

See Also:
    ravel
        Return a flattened array.
    flat
        A 1-D flat iterator over the array.

Examples
    >>> a = np.array([[1,2], [3,4]])
    >>> a.flatten()
    array([1, 2, 3, 4])
    >>> a.flatten('F')
    array([1, 3, 2, 4])

MaskedArray.ravel()
Returns a 1D version of self, as a view.

Returns
    MaskedArray

    Output view is of shape (self.size,) (or
    (np.ma.product(self.shape),)).

Examples
    >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
    >>> print x
    [[1 -- 3]
     [-- 5 --]
     [7 -- 9]]
    >>> print x.ravel()
    [1 -- 3 -- 5 -- 7 -- 9]

MaskedArray.reshape(*s, **kwargs)
Give a new shape to the array without changing its data.
Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

- **shape**: int or tuple of ints
  The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order**: {'C', 'F'}, optional
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

- **reshaped_array**: array
  A new view on the array.

**See Also:**

- `reshape` - Equivalent function in the masked array module.
- `numpy.ma.reshape` - Equivalent method on ndarray object.
- `numpy.reshape` - Equivalent function in the NumPy module.

**Notes**

The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`

**Examples**

```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> print x
[-- 2]
[3 --]

>>> x = x.reshape((4,1))
>>> print x
[|--
 [2]
 [3]
[|--]

MaskedArray.resize(newshape, refcheck=True, order=False)
```

**Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the `numpy.ma.resize` function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

**Modifying axes**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.swapaxes</code></td>
<td>swapaxes</td>
</tr>
<tr>
<td><code>ma.transpose(a[, axes])</code></td>
<td>Permute the dimensions of an array.</td>
</tr>
</tbody>
</table>
Table 1.83 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.MaskedArray.swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with (axis1) and (axis2) interchanged.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

**numpy.ma.swapaxes**

Return a view of the array with \(axis1\) and \(axis2\) interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See Also:**

- `numpy.swapaxes`
  - equivalent function

**numpy.ma.transpose**

Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

**See Also:**

- `numpy.transpose`
  - Equivalent function in top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked

>>> x
masked_array(data =
[[ 0  1]
 [ 2 --]],
         mask =
[[False False]
 [False  True]],
    fill_value = 999999)
>>> ma.transpose(x)
mapped_array(data =
[[ 0  2]
 [ 1 --]],
         mask =
[[False False]
 [False  True]],
    fill_value = 999999)
```

**MaskedArray.swapaxes** (\(axis1\), \(axis2\))

Return a view of the array with \(axis1\) and \(axis2\) interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See Also:**

- `numpy.swapaxes`
  - equivalent function
MaskedArray.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters:
axes : None, tuple of ints, or n ints

• None or no argument: reverses the order of the axes.
• tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
• n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns:
out : ndarray

View of a, with axes suitably permuted.

See Also:

ndarray.T

Array property returning the array transposed.

Examples:

>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])

Changing the number of dimensions

ma.atleast_1d(*arys) Convert inputs to arrays with at least one dimension.
ma.atleast_2d(*arys) View inputs as arrays with at least two dimensions.
ma.atleast_3d(*arys) View inputs as arrays with at least three dimensions.
ma.expand_dims(x, axis) Expand the shape of an array.
ma.squeeze(a[, axis]) Remove single-dimensional entries from the shape of an array.
ma.MaskedArray.squeeze([axis]) Remove single-dimensional entries from the shape of a.
ma.column_stack(tup) Stack 1-D arrays as columns into a 2-D array.
ma.concatenate(arrays[, axis]) Concatenate a sequence of arrays along the given axis.
ma.dstack(tup) Stack arrays in sequence depth wise (along third axis).
ma.hstack(tup) Stack arrays in sequence horizontally (column wise).
Table 1.84 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.hsplit(ary, indices_or_sections)</td>
<td>Split an array into multiple sub-arrays horizontally (column-wise).</td>
</tr>
<tr>
<td>ma.mr_</td>
<td>Translate slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td>ma.row_stack(tup)</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
<tr>
<td>ma.vstack(tup)</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

```
numpy.ma.atleast_1d(*arys) = <numpy.ma.extras._fromnxfunction instance at 0x28216c8>
```

Convert inputs to arrays with at least one dimension.
Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

arys1, arys2, ... : array_like

One or more input arrays.

**Returns**

ret : ndarray

An array, or sequence of arrays, each with a.ndim >= 1. Copies are made only if necessary.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> np.atleast_1d(1.0)
array([ 1.])

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> np.atleast_1d(x) is x
True
```

```
numpy.ma.atleast_2d(*arys) = <numpy.ma.extras._fromnxfunction instance at 0x28217e8>
```

View inputs as arrays with at least two dimensions.

**Parameters**

arys1, arys2, ... : array_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**

res, res2, ... : ndarray

An array, or tuple of arrays, each with a.ndim >= 2. Copies are avoided where possible, and views with two or more dimensions are returned.

1.7. Masked arrays
Notes

The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> np.atleast_2d(3.0)
array([[ 3.]])

>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[ 0.,  1.,  2.]])
>>> np.atleast_2d(x).base is x
True

>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

numpy.ma.atleast_3d(*arys) = <numpy.ma.extras._fromnxfunction instance at 0x2821878>

View inputs as arrays with at least three dimensions.

Parameters

arys1, arys2, ... : array_like

One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

Returns

res1, res2, ... : ndarray

An array, or tuple of arrays, each with a.ndim >= 3. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape (N,) becomes a view of shape (1, N, 1), and a 2-D array of shape (M, N) becomes a view of shape (M, N, 1).

Notes

The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> np.atleast_3d(3.0)
array([[3.]])

>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)

>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True

>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...    print(arr, arr.shape
...[[[1]]]
```
Expand the shape of an array.

Expands the shape of the array by including a new axis before the one specified by the axis parameter. This function behaves the same as `numpy.expand_dims` but preserves masked elements.

See Also:

`numpy.expand_dims`

Equivalent function in top-level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> x = ma.array([1, 2, 4])
>>> x[1] = ma.masked
>>> x
masked_array(data = [1 -- 4],
             mask = [False True False],
             fill_value = 999999)
>>> np.expand_dims(x, axis=0)
array([[1, 2, 4]])
>>> ma.expand_dims(x, axis=0)
masked_array(data =
             [[1 -- 4]],
             mask =
             [[False True False]],
             fill_value = 999999)
```

The same result can be achieved using slicing syntax with `np.newaxis`.

```python
>>> x[np.newaxis, :]
masked_array(data =
             [[1 -- 4]],
             mask =
             [[False True False]],
             fill_value = 999999)
```

Remove single-dimensional entries from the shape of an array.

Parameters

- `a` : array_like
  Input data.

- `axis` : None or int or tuple of ints, optional
  New in version 1.7.0. Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

Returns

- `squeezed` : ndarray
  The input array, but with with all or a subset of the dimensions of length 1 removed. This is always `a` itself or a view into `a`.
Examples

```python
>>> x = np.array([[0], [1], [2]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=(2,)).shape
(1, 3)
```

**MaskedArray.squeeze** *(axis=None)*

Remove single-dimensional entries from the shape of `a`. Refer to `numpy.squeeze` for full documentation.

**See Also:**

`numpy.squeeze`

equivalent function

```python
numpy.ma.column_stack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821a28>
```

Stack 1-D arrays as columns into a 2-D array. Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**

- **tup**: sequence of 1-D or 2-D arrays.
  
  Arrays to stack. All of them must have the same first dimension.

**Returns**

- **stacked**: 2-D array
  
  The array formed by stacking the given arrays.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

**numpy.ma.concatenate** *(arrays, axis=0)*

Concateenate a sequence of arrays along the given axis.

**Parameters**

- **arrays**: sequence of array_like

  The arrays must have the same shape, except in the dimension corresponding to `axis` (the first, by default).

- **axis**: int, optional
The axis along which the arrays will be joined. Default is 0.

Returns
result : MaskedArray

The concatenated array with any masked entries preserved.

See Also:

numpy.concatenate
Equivalent function in the top-level NumPy module.

Examples
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
             fill_value = 999999)
>>> b
masked_array(data = [2 3 4],
             mask = False,
             fill_value = 999999)
>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
             fill_value = 999999)

numpy.ma.dstack(tup) = <numpy.maextras_fromnxfunction instance at 0x2821ab8>

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds
arrays divided by dsplit. This is a simple way to stack 2D arrays (images) into a single 3D array for
processing.

Parameters
tup : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

Returns
stacked : ndarray

The array formed by stacking the given arrays.

See Also:

vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.
dsplit
Split array along third axis.

Notes
The function is applied to both the _data and the _mask, if any.

Examples
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

numpy.ma.hstack (tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821998>

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided
by hsplit.

Parameters
tup : sequence of ndarrays
    All arrays must have the same shape along all but the second axis.

Returns
stacked : ndarray
    The array formed by stacking the given arrays.

See Also:
vstack
Stack arrays in sequence vertically (row wise).
dstack
Stack arrays in sequence depth wise (along third axis).
concatenate
Join a sequence of arrays together.
hsplit
Split array along second axis.

Notes
The function is applied to both the _data and the _mask, if any.
Examples

```python
>>> a = np.array((1, 2, 3))
>>> b = np.array((2, 3, 4))
>>> np.hstack((a, b))
array([1, 2, 3, 2, 3, 4])
```  
```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```  
```python
def numpy.ma.hsplit(ary, indices_or_sections) = <numpy.ma.extras._fromnxfunction instance at 0x2821b00>
```

Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the `split` documentation. `hsplit` is equivalent to `split` with `axis=1`, the array is always split along the second axis regardless of the array dimension.

See Also:

- `split`
  Split an array into multiple sub-arrays of equal size.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]]])
```  
```python
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]]),
 array([], dtype=float64)]
```  

With a higher dimensional array the split is still along the second axis.
>>> x = np.arange(8.0).reshape(2, 2, 2)

>>> x

array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])

>>> np.hsplit(x, 2)

[array([[[ 0.,  1.],
        [ 4.,  5.]]]),
 array([[[ 2.,  3.],
        [ 6.,  7.]]])]

numpy.ma.mr_ = <numpy.ma.extras.mr_class object at 0x281edd0>

Translate slice objects to concatenation along the first axis.

This is the masked array version of lib.index_tricks.RClass.

See Also:

lib.index_tricks.RClass

Examples

>>> np.ma.mr_[np.ma.array([1, 2, 3]), 0, 0, np.ma.array([4, 5, 6])]

array([1, 2, 3, 0, 0, 4, 5, 6])

numpy.ma.row_stack(tup) = <numpy.ma.extras_fromnxfnction instance at 0x2821908>

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

Parameters

- **tup**: sequence of ndarrays

  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

- **stacked**: ndarray

  The array formed by stacking the given arrays.

See Also:

- **hstack**
  Stack arrays in sequence horizontally (column wise).

- **dstack**
  Stack arrays in sequence depth wise (along third dimension).

- **concatenate**
  Join a sequence of arrays together.

- **vsplit**
  Split array into a list of multiple sub-arrays vertically.
Notes
The function is applied to both the _data and the _mask, if any.

Examples
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])

numpy.ma.vstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821908>

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided
by vsplit.

Parameters
  tup : sequence of ndarrays
    Tuple containing arrays to be stacked. The arrays must have the same shape along all
    but the first axis.

Returns
  stacked : ndarray
    The array formed by stacking the given arrays.

See Also:

hstack
    Stack arrays in sequence horizontally (column wise).

dstack
    Stack arrays in sequence depth wise (along third dimension).

concatenate
    Join a sequence of arrays together.

vsplit
    Split array into a list of multiple sub-arrays vertically.

Notes
The function is applied to both the _data and the _mask, if any.
Examples

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>ma.concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td><code>ma.dstack(tup)</code></td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td><code>ma.hstack(tup)</code></td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td><code>ma.vstack(tup)</code></td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

```python
numpy.ma.column_stack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821a28>
```

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**
- **tup**: sequence of 1-D or 2-D arrays.

  Arrays to stack. All of them must have the same first dimension.

**Returns**
- **stacked**: 2-D array

  The array formed by stacking the given arrays.

**Notes**

The function is applied to both the `_data` and the `_mask`, if any.

**Examples**

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

```python
numpy.ma.concatenate(arrays, axis=0)
```

Concatenate a sequence of arrays along the given axis.
Parameters
arrays : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to axis 
(the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns
result : MaskedArray

The concatenated array with any masked entries preserved.

See Also:

numpy.concatenate
Equivalent function in the top-level NumPy module.

Examples
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)

 masked_array(data = [0 -- 2],
              mask = [False True False],
           fill_value = 999999)

 masked_array(data = [2 3 4],
              mask = False,
           fill_value = 999999)

 >>> ma.concatenate([a, b])
 masked_array(data = [0 -- 2 2 3 4],
              mask = [False True False False False False],
           fill_value = 999999)

numpy.ma.dstack (tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821ab8>

Stack arrays in sequence depth wise (along third axis).
Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds 
arrays divided by dsplit. This is a simple way to stack 2D arrays (images) into a single 3D array for 
processing.

Parameters
tup : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

Returns
stacked : ndarray

The array formed by stacking the given arrays.

See Also:
vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.

dsplit
Split array along third axis.

Notes
The function is applied to both the _data and the _mask, if any.

Examples
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.dstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])

numpy.ma.hstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821998>

Stack arrays in sequence horizontally (column wise).
Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided
by hsplit.

Parameters
tup : sequence of ndarrays
All arrays must have the same shape along all but the second axis.

Returns
stacked : ndarray
The array formed by stacking the given arrays.

See Also:

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third axis).

concatenate
Join a sequence of arrays together.
**hsplit**

Split array along second axis.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
generate_example_code
```

```python
numpy.ma.vstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821908>
```

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by `vsplit`.

**Parameters**

- `tup`: sequence of ndarrays

  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

**Returns**

- `stacked`: ndarray

  The array formed by stacking the given arrays.

**See Also:**

**hstack**

Stack arrays in sequence horizontally (column wise).

**dstack**

Stack arrays in sequence depth wise (along third dimension).

**concatenate**

Join a sequence of arrays together.

**vsplit**

Split array into a list of multiple sub-arrays vertically.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**
Operations on masks

Creating a mask

numpy.ma.make_mask(m, copy=False, shrink=True, dtype=<type 'numpy.bool_'>)
Create a boolean mask from an array.

Parameters

- m : array_like
  Potential mask.
- copy : bool, optional
  Whether to return a copy of m (True) or m itself (False).
- shrink : bool, optional
  Whether to shrink m to nomask if all its values are False.
- dtype : dttype, optional
  Data-type of the output mask. By default, the output mask has a dtype of MaskType (bool). If the dtype is flexible, each field has a boolean dtype.

Returns

- result : ndarray
  A boolean mask derived from m.
Examples

```python
>>> import numpy.ma as ma
>>> m = [True, False, True, True]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 1, 1]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
>>> m = [1, 0, 2, -3]
>>> ma.make_mask(m)
array([ True, False, True, True], dtype=bool)
```

Effect of the `shrink` parameter.

```python
>>> m = np.zeros(4)
>>> m
array([ 0., 0., 0., 0.])
>>> ma.make_mask(m)
False
>>> ma.make_mask(m, shrink=False)
array([False, False, False, False], dtype=bool)
```

Using a flexible `dtype`.

```python
>>> m = [1, 0, 1, 1]
>>> n = [0, 1, 0, 0]
>>> arr = []
... for man, mouse in zip(m, n):
...    arr.append((man, mouse))
>>> arr
[(1, 0), (0, 1), (1, 0), (1, 0)]
>>> dtype = np.dtype({'names': ['man', 'mouse'],
                   'formats': [np.int, np.int]})
>>> arr = np.array(arr, dtype=dtype)
>>> arr
array([(1, 0), (0, 1), (1, 0), (1, 0)],
      dtype=[('man', '<i4'), ('mouse', '<i4')])
```

```
numpy.ma.make_mask_none(newshape, dtype=None)

Return a boolean mask of the given shape, filled with False.

This function returns a boolean ndarray with all entries False, that can be used in common mask manipulations. If a complex dtype is specified, the type of each field is converted to a boolean type.

Parameters

- `newshape`: tuple
  A tuple indicating the shape of the mask.

- `dtype`: {None, dtype}, optional
  If None, use a MaskType instance. Otherwise, use a new datatype with the same fields as `dtype`, converted to boolean types.

Returns

- `result`: ndarray
  An ndarray of appropriate shape and dtype, filled with False.

1.7. Masked arrays
See Also:

**make_mask**
Create a boolean mask from an array.

**make_mask_descr**
Construct a dtype description list from a given dtype.

**Examples**

```python
defining a more complex dtype.
>>> dtype = np.dtype({'names': ['foo', 'bar'],
                 'formats': [np.float32, np.int]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4')])
>>> ma.make_mask_none((3,), dtype=dtype)
array([(False, False), (False, False), (False, False)],
dtype=[('foo', '|b1'), ('bar', '|b1')])
```

**numpy.ma.mask_or**(m1, m2, copy=False, shrink=True)

Combine two masks with the logical_or operator.

The result may be a view on m1 or m2 if the other is nomask (i.e. False).

**Parameters**

- **m1, m2**: array_like
  Input masks.

- **copy**: bool, optional
  If copy is False and one of the inputs is nomask, return a view of the other input mask. Defaults to False.

- **shrink**: bool, optional
  Whether to shrink the output to nomask if all its values are False. Defaults to True.

**Returns**

- **mask**: output mask
  The result masks values that are masked in either m1 or m2.

**Raises**

- **ValueError**
  If m1 and m2 have different flexible dtypes.

**Examples**

```python
>>> m1 = np.ma.make_mask([[0, 1, 1, 0]])
>>> m2 = np.ma.make_mask([[1, 0, 0, 0]])
>>> np.ma.mask_or(m1, m2)
array([[ True,  True,  True,  False]], dtype=bool)
```

**numpy.ma.make_mask_descr**(ndtype)

Construct a dtype description list from a given dtype.
Returns a new dtype object, with the type of all fields in ndtype to a boolean type. Field names are not altered.

**Parameters**

- **ndtype**: dtype
  
The dtype to convert.

**Returns**

- **result**: dtype
  
  A dtype that looks like ndtype, the type of all fields is boolean.

**Examples**

```python
>>> import numpy.ma as ma
>>> dtype = np.dtype([('names': ['foo', 'bar'],
                     'formats': [np.float32, np.int])])
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4')])
>>> ma.make_mask_descr(dtype)
dtype([('foo', '|b1'), ('bar', '|b1')])
>>> ma.make_mask_descr(np.float32)
<type 'numpy.bool_'>
```

### Accessing a mask

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.getmask(a)</code></td>
<td>Return the mask of a masked array, or nomask.</td>
</tr>
<tr>
<td><code>ma.getmaskarray(arr)</code></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><code>ma.masked_array.mask</code></td>
<td>Mask</td>
</tr>
</tbody>
</table>

```
numpy.ma.getmask(a)
Return the mask of a masked array, or nomask.
```

Return the mask of a as an ndarray if a is a MaskedArray and the mask is not nomask, else return nomask. To guarantee a full array of booleans of the same shape as a, use getmaskarray.

**Parameters**

- **a**: array_like

  Input MaskedArray for which the mask is required.

**See Also:**

- `getdata`
  
  Return the data of a masked array as an ndarray.

- `getmaskarray`
  
  Return the mask of a masked array, or full array of False.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --],
[3 4]],
mask =
[[False  True]])
```

1.7. Masked arrays
Equivalently use the MaskedArray mask attribute.

```python
>>> a.mask
array([[False,  True],
       [False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
 [3 4]],
mask =
False,
fill_value=999999)
```

```python
>>> ma.nomask
False
```

```python
>>> ma.getmask(b) == ma.nomask
True
```

```python
>>> b.mask == ma.nomask
True
```

numpy.ma.getmaskarray(arr)

Return the mask of a masked array, or full boolean array of False.

Return the mask of arr as an ndarray if arr is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as arr.

Parameters

arr : array_like

Input MaskedArray for which the mask is required.

See Also:

getmask

Return the mask of a masked array, or nomask.

getdata

Return the data of a masked array as an ndarray.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
 [3 4]],
mask =
[[False  True]
 [False False]],
fill_value=999999)
```
>>> ma.getmaskarray(a)
array([[False,  True],
       [False, False]], dtype=bool)

Result when mask == nomask

>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
    [[1 2]
     [3 4]],
    mask =
    False,
    fill_value=999999)

>>> ma.getmaskarray(b)
array([[False, False],
       [False, False]], dtype=bool)

masked_array

Finding masked data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.flatnotmasked_contiguous(a)</code></td>
<td>Find contiguous unmasked data in a masked array along the given axis.</td>
</tr>
<tr>
<td><code>ma.flatnotmasked_edges(a)</code></td>
<td>Find the indices of the first and last unmasked values.</td>
</tr>
<tr>
<td><code>ma.notmasked_contiguous(a[, axis])</code></td>
<td>Find contiguous unmasked data in a masked array along the given axis.</td>
</tr>
<tr>
<td><code>ma.notmasked_edges(a[, axis])</code></td>
<td>Find the indices of the first and last unmasked values along an axis.</td>
</tr>
</tbody>
</table>

numpy.ma.flatnotmasked_contiguous(a)

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

- `a` : array

  The input array.

**Returns**

- `slice_list` : list

  A sorted sequence of slices (start index, end index).

**See Also:**

flatnotmasked_edges, notmasked_contiguous, notmasked_edges, clump_masked, clump_unmasked

**Notes**

Only accepts 2-D arrays at most.

**Examples**

```python
>>> a = np.ma.arange(10)
>>> np.ma.extras.flatnotmasked_contiguous(a)
slice(0, 10, None)
```

```python
>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> a[mask] = np.ma.masked
```
```python
gnp.array(a[~a.mask])
array([3, 4, 6, 7, 8])

gnp.ma.extras.flatnotmasked_contiguous(a)
[slice(3, 5, None), slice(6, 9, None)]

g[::] = np.ma.masked
gnp.ma.extras.flatnotmasked_edges(a)
None
```

```
numpy.ma.flatnotmasked_edges(a)

Find the indices of the first and last unmasked values.

Expects a 1-D MaskedArray, returns None if all values are masked.

**Parameters**

- **arr**: array_like
  Input 1-D MaskedArray

**Returns**

- **edges**: ndarray or None
  The indices of first and last non-masked value in the array. Returns None if all values are masked.

**See Also:**

flatnotmasked_contiguous, notmasked_contiguous, notmasked_edges, clump_masked, clump_unmasked

**Notes**

Only accepts 1-D arrays.

**Examples**

```python
>>> a = np.ma.arange(10)
>>> flatnotmasked_edges(a)
[0, -1]

>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> a[mask] = np.ma.masked
>>> np.array(a[~a.mask])
array([3, 4, 6, 7, 8])

>>> flatnotmasked_edges(a)
array([3, 8])

>>> a[::] = np.ma.masked
>>> print flatnotmasked_edges(ma)
None
```

```
numpy.ma.notmasked_contiguous(a, axis=None)

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

- **a**: array_like
  The input array.

- **axis**: int, optional
  The axis along which to find contiguous unmasked data.
```

---

344 Chapter 1. Array objects
Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

**Returns**

**endpoints** : list

A list of slices (start and end indexes) of unmasked indexes in the array.

**See Also:**

flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges, clump_masked, clump_unmasked

**Notes**

Only accepts 2-D arrays at most.

**Examples**

```python
g = np.arange(9).reshape((3, 3)).mask = np.zeros_like(a) mask[1:, 1:] = 1
>>> ma = np.ma.array(a, mask=mask) np.array(ma[~ma.mask])
array([0, 1, 2, 3, 6])
```

numpy.ma.notmasked_edges(a, axis=None)

Find the indices of the first and last unmasked values along an axis.

If all values are masked, return None. Otherwise, return a list of two tuples, corresponding to the indices of the first and last unmasked values respectively.

**Parameters**

a : array_like

The input array.

axis : int, optional

Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

**Returns**

edges : ndarray or list

An array of start and end indexes if there are any masked data in the array. If there are no masked data in the array, edges is a list of the first and last index.

**See Also:**

flatnotmasked_contiguous, flatnotmasked_edges, notmasked_contiguous, clump_masked, clump_unmasked

**Examples**

```python
g = np.arange(9).reshape((3, 3)).m = np.zeros_like(a) m[1:, 1:] = 1
```
NumPy Reference, Release 1.8.1

```python
>>> am = np.ma.array(a, mask=m)
>>> np.array(am[~am.mask])
array([0, 1, 2, 3, 6])

>>> np.ma.extras.notmasked_edges(ma)
array([0, 6])
```

### Modifying a mask

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.mask_cols(a[, axis])</code></td>
<td>Mask columns of a 2D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.mask_or(m1, m2[, copy, shrink])</code></td>
<td>Combine two masks with the <code>logical_or</code> operator.</td>
</tr>
<tr>
<td><code>ma.mask_rowcols(a[, axis])</code></td>
<td>Mask rows and/or columns of a 2D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.mask_rows(a[, axis])</code></td>
<td>Mask rows of a 2D array that contain masked values.</td>
</tr>
<tr>
<td><code>ma.harden_mask(self)</code></td>
<td>Force the mask to hard.</td>
</tr>
<tr>
<td><code>ma.soften_mask(self)</code></td>
<td>Force the mask to soft.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.harden_mask()</code></td>
<td>Force the mask to hard.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.soften_mask()</code></td>
<td>Force the mask to soft.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.shrink_mask()</code></td>
<td>Reduce a mask to nomask when possible.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.unshare_mask()</code></td>
<td>Copy the mask and set the sharedmask flag to False.</td>
</tr>
</tbody>
</table>

```python
numpy.ma.mask_cols(a, axis=None)
Mask columns of a 2D array that contain masked values.

This function is a shortcut to `mask_rowcols` with `axis` equal to 1.

See Also:

- `mask_rowcols`  
  Mask rows and/or columns of a 2D array.

- `masked_where`  
  Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
             [[0 0 0],
              [0 -- 0],
              [0 0 0]],
            mask =
             [[False False False],
              [False  True False],
              [False False False]],
            fill_value=999999)
```

```python
>>> ma.mask_cols(a)
masked_array(data =
             [[0 0 .],
              [0 - 0],
              [0 0 .]],
```
Mask two masks with the `logical_or` operator.

The result may be a view on *m1* or *m2* if the other is **nomask** (i.e. False).

**Parameters**

*m1, m2* : array_like
Input masks.

*copy* : bool, optional
If *copy* is False and one of the inputs is **nomask**, return a view of the other input mask. Defaults to False.

*shrink* : bool, optional
Whether to shrink the output to **nomask** if all its values are False. Defaults to True.

**Returns**

*mask* : output mask
The result masks values that are masked in either *m1* or *m2*.

**Raises**

*ValueError*
If *m1* and *m2* have different flexible dtypes.

**Examples**

```python
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([True, True, True, False], dtype=bool)
```

Mask rows and/or columns of a 2D array that contain masked values.

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected using the *axis* parameter.

- If *axis* is None, rows and columns are masked.
- If *axis* is 0, only rows are masked.
- If *axis* is 1 or -1, only columns are masked.

**Parameters**

*a* : array_like, MaskedArray
The array to mask. If not a MaskedArray instance (or if no array elements are masked). The result is a MaskedArray with *mask* set to **nomask** (False). Must be a 2D array.

*axis* : int, optional
Axis along which to perform the operation. If None, applies to a flattened version of the array.

**Returns**

a : MaskedArray

A modified version of the input array, masked depending on the value of the axis parameter.

**Raises**

NotImplementedError

If input array a is not 2D.

**See Also:**

`mask_rows`  
Mask rows of a 2D array that contain masked values.

`mask_cols`  
Mask cols of a 2D array that contain masked values.

`masked_where`  
Mask where a condition is met.

**Notes**

The input array's mask is modified by this function.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
             [[0 0 0],
              [0 -- 0],
              [0 0 0]],
            mask =
             [[False False False],
              [False  True False],
              [False False False]],
          fill_value=999999)
>>> ma.mask_rowcols(a)
masked_array(data =
             [[0 -- 0],
              [-- -- --],
              [0 -- 0]],
            mask =
             [[False  True False],
              [ True  True True],
              [False  True False]],
          fill_value=999999)
```
numpy.ma.mask_rows(a, axis=None)

Mask rows of a 2D array that contain masked values.

This function is a shortcut to mask_rowcols with axis equal to 0.

See Also:

mask_rowcols
Mask rows and/or columns of a 2D array.

masked_where
Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(data =
  [[0 0 0],
   [-- -- --],
   [0 0 0]],
  mask =
  [[False False False],
   [ True False False],
   [False False False]],
  fill_value=999999)
>>> ma.mask_rows(a)
masked_array(data =
  [[0 0 0],
   [-- -- --],
   [0 0 0]],
  mask =
  [[False False False],
   [ True True True],
   [False False False]],
  fill_value=999999)
```

numpy.ma.harden_mask(self) = <numpy.ma.core._frommethod instance at 0x2820368>

Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. harden_mask sets hardmask to True.

See Also:

hardmask

numpy.ma.soften_mask(self) = <numpy.ma.core._frommethod instance at 0x28207a0>

Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. soften_mask sets hardmask to False.

See Also:
hardmask

MaskedArray.harden_mask()
Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. 
harden_mask sets hardmask to True.

See Also:
  hardmask

MaskedArray.soften_mask()
Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its hardmask property. 
soften_mask sets hardmask to False.

See Also:
  hardmask

MaskedArray.shrink_mask()
Reduce a mask to nomask when possible.

Parameters
  None

Returns
  None

Examples
>>> x = np.ma.array([[1, 2], [3, 4]], mask=[0]*4)
>>> x.mask
array([[False, False],
       [False, False]], dtype=bool)
>>> x.shrink_mask()
>>> x.mask
False

MaskedArray.unshare_mask()
Copy the mask and set the sharedmask flag to False.

Whether the mask is shared between masked arrays can be seen from the sharedmask property. 
unshare_mask ensures the mask is not shared. A copy of the mask is only made if it was shared.

See Also:
  sharedmask

Conversion operations
> to a masked array

ma.asarray(a[, dtype, order])  Convert the input to a masked array of the given data-type.
ma.asanyarray(a[, dtype])  Convert the input to a masked array, conserving subclasses.
ma.fix_invalid(a[, mask, copy, fill_value])  Return input with invalid data masked and replaced by a fill value.

Continued on next page
### Table 1.90 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.masked_equal(x, value[, copy])</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater(x, value[, copy])</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater_equal(x, value[, copy])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_inside(x, v1, v2[, copy])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>ma.masked_less(x, value[, copy])</code></td>
<td>Mask an array where less than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_less_equal(x, value[, copy])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_not_equal(x, value[, copy])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_object(x, value[, copy, shrink])</code></td>
<td>Mask the array x where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>ma.masked_outside(x, v1, v2[, copy])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_values(x, value[, rtol, atol, ...])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>ma.masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

#### numpy.ma.asarray

Convert the input to a masked array of the given data-type.

No copy is performed if the input is already an `ndarray`. If `a` is a subclass of `MaskedArray`, a base class `MaskedArray` is returned.

**Parameters**
- `a`: array_like
  - Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples of tuples, tuples of lists, ndarrays and masked arrays.
- `dtype`: dtype, optional
  - By default, the data-type is inferred from the input data.
- `order`: {'C', 'F'}, optional
  - Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**
- `out`: MaskedArray
  - Masked array interpretation of `a`.

**See Also:**
- `asanyarray`
  - Similar to `asarray`, but conserves subclasses.

#### Examples

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])

>>> np.ma.asarray(x)
masked_array(data =
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]],
mask =
False,
fill_value = 1e+20)
```
numpy.ma.asanyarray(a, dtype=None)

Convert the input to a masked array, conserving subclasses.

If a is a subclass of MaskedArray, its class is conserved. No copy is performed if the input is already an ndarray.

Parameters

a : array_like
    Input data, in any form that can be converted to an array.

dtype : dtype, optional
    By default, the data-type is inferred from the input data.

order : {'C', 'F'}, optional
    Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

Returns

out : MaskedArray
    MaskedArray interpretation of a.

See Also:

asarray
    Similar to asanyarray, but does not conserve subclass.

Examples

>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asanyarray(x)
masked_array(data =
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]],
mask =
False,
fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
fill_value : scalar, optional

Value used for fixing invalid data. Default is None, in which case the a.fill_value is used.

Returns

b : MaskedArray

The input array with invalid entries fixed.

Notes

A copy is performed by default.

Examples

```python
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
               mask = [ True False False False],
               fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [-- -1.0 -- --],
               mask = [ True False True True],
               fill_value = 1e+20)
>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00, 1.00000000e+20,
       1.00000000e+20])
```

numpy.ma.masked_equal (x, value, copy=True)

Mask an array where equal to a given value.

This function is a shortcut to masked_where, with condition = (x == value). For floating point arrays, consider using masked_values(x, value).

See Also:

masked_where
Mask where a condition is met.

masked_values
Mask using floating point equality.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 2 --],
               mask = [False False False True],
               fill_value=999999)
```

numpy.ma.masked_greater (x, value, copy=True)

Mask an array where greater than a given value.
This function is a shortcut to masked_where, with condition = (x > value).

See Also:

masked_where
    Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
mapped_array(data = [0 1 2 --],
    mask = [False False False True],
    fill_value=999999)

numpy.ma.masked_greater_equal (x, value, copy=True)
    Mask an array where greater than or equal to a given value.

    This function is a shortcut to masked_where, with condition = (x >= value).

    See Also:

    masked_where
        Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
mapped_array(data = [0 1 -- --],
    mask = [False False True True],
    fill_value=999999)

numpy.ma.masked_inside (x, v1, v2, copy=True)
    Mask an array inside a given interval.

    Shortcut to masked_where, where condition is True for x inside the interval [v1,v2] (v1 <= x <= v2). The boundaries v1 and v2 can be given in either order.

    See Also:

    masked_where
        Mask where a condition is met.

Notes

The array x is prefilled with its filling value.

Examples

>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
mapped_array(data = [0.31 1.2 -- -- -0.4 -1.1],
mask = [False False  True  True False False],
fill_value=1e+20)

The order of v1 and v2 doesn’t matter.

>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
            mask = [False False  True  True False False],
            fill_value=1e+20)

numpy.ma.masked_invalid(a, copy=True)
Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to masked_where, with condition = ~(np.isfinite(a)). Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0., 1., NaN, Inf, 4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
            mask = [False False  True  True False],
            fill_value=1e+20)

numpy.ma.masked_less(x, value, copy=True)
Mask an array where less than a given value.

This function is a shortcut to masked_where, with condition = (x < value).

See Also:

masked_where
Mask where a condition is met.

Examples

>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
            mask = [ True  True False False],
            fill_value=999999)

numpy.ma.masked_less_equal(x, value, copy=True)
Mask an array where less than or equal to a given value.

This function is a shortcut to masked_where, with condition = (x <= value).
See Also:

**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
mask = [ True True True False],
fill_value=999999)
```

NumPy's `.masked_not_equal(x, value, copy=True)`

Mask an array where not equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x != value)`.

See Also:

**masked_where**
Mask where a condition is met.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
mask = [ True True False True],
fill_value=999999)
```

NumPy's `.masked_object(x, value, copy=True, shrink=True)`

Mask the array `x` where the data are exactly equal to `value`.

This function is similar to `masked_values`, but only suitable for object arrays: for floating point, use `masked_values` instead.

**Parameters**

- `x`: array_like
  Array to mask

- `value`: object
  Comparison value

- `copy`: {True, False}, optional
  Whether to return a copy of `x`.

- `shrink`: {True, False}, optional
  Whether to collapse a mask full of False to nomask

**Returns**

- `result`: MaskedArray
The result of masking \(x\) where equal to \(value\).

See Also:

masked_where

Mask where a condition is met.

masked_equal

Mask where equal to a given value (integers).

masked_values

Mask using floating point equality.

Examples

```python
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_equal(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol' ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]
```

Note that `mask` is set to `nomask` if possible.

```python
>>> eat
masked_array(data = [cheese ham pineapple],
         mask = False,
        fill_value=?)
```

numpy.ma.masked_outside\((x, v1, v2, copy=True)\)

Mask an array outside a given interval.

Shortcut to masked_where, where `condition` is True for \(x\) outside the interval \([v1,v2]\) \((x < v1)\|(x > v2)\). The boundaries \(v1\) and \(v2\) can be given in either order.

See Also:

masked_where

Mask where a condition is met.

Notes

The array \(x\) is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
         mask = [ True True False False True True],
        fill_value=1e+20)
```

The order of \(v1\) and \(v2\) doesn’t matter.
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data = [-- -- 0.01 0.2 -- --],
              mask = [ True  True False False  True  True],
              fill_value=1e+20)

numpy.ma.masked_values(x, value, rtol=1e-05, atol=1e-08, copy=True, shrink=True)

Mask using floating point equality.

Return a MaskedArray, masked where the data in array x are approximately equal to value, i.e. where the following condition is True
(abs(x - value) <= atol+rtol*abs(value))

The fill value is set to value and the mask is set to nomask if possible. For integers, consider using masked_equal.

Parameters
- x: array_like
  Array to mask.
- value: float
  Masking value.
- rtol: float, optional
  Tolerance parameter.
- atol: float, optional
  Tolerance parameter (1e-8).
- copy: bool, optional
  Whether to return a copy of x.
- shrink: bool, optional
  Whether to collapse a mask full of False to nomask.

Returns
- result: MaskedArray
  The result of masking x where approximately equal to value.

See Also:

masked_where
Mask where a condition is met.

masked_equal
Mask where equal to a given value (integers).

Examples

>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
              mask = [False  True False  True False],
              fill_value=1.1)

Note that mask is set to nomask if possible.
>>> ma.masked_values(x, 1.5)
mixed_array(data = [ 1. 1.1 2. 1.1 3. ],
    mask = False,
    fill_value=1.5)

For integers, the fill value will be different in general to the result of `masked_equal`.

```python
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> ma.masked_values(x, 2)
mixed_array(data = [0 1 -- 3 4],
    mask = [False False True False False],
    fill_value=2)

>>> ma.masked_equal(x, 2)
mixed_array(data = [0 1 -- 3 4],
    mask = [False False True False False],
    fill_value=999999)
```

```python
numpy.ma.
masked_where(condition, a, copy=True)

Mask an array where a condition is met.

Return `a` as an array masked where `condition` is True. Any masked values of `a` or `condition` are also masked in the output.

Parameters

  condition : array_like
    Masking condition. When `condition` tests floating point values for equality, consider using `masked_values` instead.

  a : array_like
    Array to mask.

  copy : bool
    If True (default) make a copy of `a` in the result. If False modify `a` in place and return a view.

Returns

  result : MaskedArray
    The result of masking `a` where `condition` is True.

See Also:

  `masked_values`
  Mask using floating point equality.

  `masked_equal`
  Mask where equal to a given value.

  `masked_not_equal`
  Mask where not equal to a given value.

  `masked_less_equal`
  Mask where less than or equal to a given value.

  `masked_greater_equal`
  Mask where greater than or equal to a given value.
```
**masked_less**  
Mask where less than a given value.

**masked_greater**  
Mask where greater than a given value.

**masked_inside**  
Mask inside a given interval.

**masked_outside**  
Mask outside a given interval.

**masked_invalid**  
Mask invalid values (NaNs or infs).

### Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
             mask = [ True True True False],
             fill_value=999999)

Mask array \( b \) conditional on \( a \).

```python
>>> b = [‘a’, ‘b’, ‘c’, ‘d’]
>>> ma.masked_where(a == 2, b)
masked_array(data = [a b -- d],
             mask = [False False True False],
             fill_value=N/A)
```  
Effect of the `copy` argument.

```python
>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
             mask = [False True True False],
             fill_value=999999)
>>> a
array([0, 1, 2, 3])
```  
When `condition` or `a` contain masked values.
>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
    mask = [False False True False],
    fill_value=999999)

>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data = [-- 1 2 3],
    mask = [ True False False False],
    fill_value=999999)

>>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
    mask = [ True False True True],
    fill_value=999999)

> to a ndarray

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.compress_cols(a)</td>
<td>Suppress whole columns of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td>ma.compress_rowcols(x[, axis])</td>
<td>Suppress the rows and/or columns of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td>ma.compress_rows(a)</td>
<td>Suppress whole rows of a 2-D array that contain masked values.</td>
</tr>
<tr>
<td>ma.compressed(x)</td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td>ma.filled(a[, fill_value])</td>
<td>Return input as an array with masked data replaced by a fill value.</td>
</tr>
<tr>
<td>ma.MaskedArray.compressed()</td>
<td>Return all the non-masked data as a 1-D array.</td>
</tr>
<tr>
<td>ma.MaskedArray.filled([fill_value])</td>
<td>Return a copy of self, with masked values filled with a given value.</td>
</tr>
</tbody>
</table>

**numpy.ma.compress_cols(a)**

Suppress whole columns of a 2-D array that contain masked values.

This is equivalent to `np.ma.extras.compress_rowcols(a, 1)`, see `extras.compress_rowcols` for details.

See Also:

`extras.compress_rowcols`

**numpy.ma.compress_rowcols(x, axis=None)**

Suppress the rows and/or columns of a 2-D array that contain masked values.

The suppression behavior is selected with the `axis` parameter.

- If `axis` is None, both rows and columns are suppressed.
- If `axis` is 0, only rows are suppressed.
- If `axis` is 1 or -1, only columns are suppressed.

Parameters

- `axis` : int, optional
  
  Axis along which to perform the operation. Default is None.

Returns

- `compressed_array` : ndarray
  
  The compressed array.
Examples

```python
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
...       [1, 0, 0],
...       [0, 0, 0]])
>>> x
masked_array(data =
[[- 1 2]
[ -- 4 5]
[6 7 8]],
mask =
[[ True False False]
[ True False False]
[False False False]],
fill_value = 999999)

>>> np.ma.extras.compress_rowcols(x)
array([[7, 8]])
>>> np.ma.extras.compress_rowcols(x, 0)
array([[6, 7, 8]])
>>> np.ma.extras.compress_rowcols(x, 1)
array([[1, 2],
[4, 5],
[7, 8]])
```

**numpy.ma.compress_rowcols(a)**

Suppress whole rows of a 2-D array that contain masked values.

This is equivalent to `np.ma.extras.compress_rowcols(a, 0)`, see `extras.compress_rowcols` for details.

**See Also:**

`extras.compress_rowcols`

**numpy.ma.compressed(x)**

Return all the non-masked data as a 1-D array.

This function is equivalent to calling the “compressed” method of a `MaskedArray`, see `MaskedArray.compressed` for details.

**See Also:**

`MaskedArray.compressed`

**Equivalent method.**

**numpy.ma.filled(a, fill_value=None)**

Return input as an array with masked data replaced by a fill value.

If `a` is not a `MaskedArray`, `a` itself is returned. If `a` is a `MaskedArray` and `fill_value` is None, `fill_value` is set to `a.fill_value`.

**Parameters**

- `a`: MaskedArray or array_like
  An input object.

- `fill_value`: scalar, optional
  Filling value. Default is None.
Returns
   a : ndarray
      The filled array.

See Also:
   compressed

Examples

>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
         ...       [1, 0, 0],
         ...       [0, 0, 0]])
>>> x.filled()
array([[999999, 1, 2],
       [999999, 4, 5],
       [ 6, 7, 8]])

MaskedArray.compressed()
      Return all the non-masked data as a 1-D array.

   Returns
      data : ndarray
         A new ndarray holding the non-masked data is returned.

   Notes
      The result is not a MaskedArray!

Examples

>>> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<type 'numpy.ndarray'>

MaskedArray.filled(fill_value=None)
      Return a copy of self, with masked values filled with a given value.

   Parameters
      fill_value : scalar, optional
         The value to use for invalid entries (None by default). If None, the fill_value
         attribute of the array is used instead.

   Returns
      filled_array : ndarray
         A copy of self with invalid entries replaced by fill_value (be it the function argument
         or the attribute of self.

   Notes
      The result is not a MaskedArray!

Examples


```python
>>> x = np.ma.array([1,2,3,4,5], mask=[0,0,1,0,1], fill_value=-999)
>>> x.filled()
array([1, 2, -999, 4, -999])
>>> type(x.filled())
<type 'numpy.ndarray'>
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, `filled` returns a matrix:

```python
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.filled()
matrix([[ 1, 999999],
        [999999, 4]])
```

> to another object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.MaskedArray.tofile(fid[, sep, format])</code></td>
<td>Save a masked array to a file in binary format.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.tolist([fill_value])</code></td>
<td>Return the data portion of the masked array as a hierarchical Python list.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.torecords()</code></td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.tostring([fill_value, order])</code></td>
<td>Return the array data as a string containing the raw bytes in the array.</td>
</tr>
</tbody>
</table>

**MaskedArray.tofile**

Save a masked array to a file in binary format.

**Warning:** This function is not implemented yet.

**Raises**

`NotImplementedError`

When `tofile` is called.

**MaskedArray.tolist** *(fill_value=None)*

Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to `fill_value`. If `fill_value` is None, the corresponding entries in the output list will be `None`.

**Parameters**

- `fill_value` : scalar, optional
  The value to use for invalid entries. Default is None.

**Returns**

- `result` : list
  The Python list representation of the masked array.

**Examples**

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]
>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]
```

**MaskedArray.torecords()**

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

```
• the _data field stores the _data part of the array.
• the _mask field stores the _mask part of the array.

Parameters
None

Returns
record : ndarray
A new flexible-type ndarray with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

Notes
A side-effect of transforming a masked array into a flexible ndarray is that meta information (fill_value, ...) will be lost.

Examples
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)

>>> print x
[[1 -- 3]
 [ -- 5 --]
 [7 -- 9]]

>>> print x.toflex()
[[(1, False) (2, True) (3, False)]
 [(4, True) (5, False) (6, True)]
 [(7, False) (8, True) (9, False)]]

MaskedArray.tostring(fill_value=None, order='C')
Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

Parameters
fill_value : scalar, optional
Value used to fill in the masked values. Default is None, in which case MaskedArray.fill_value is used.

order : {'C', 'F', 'A'}, optional
Order of the data item in the copy. Default is ‘C’.
• ‘C’ – C order (row major).
• ‘F’ – Fortran order (column major).
• ‘A’ – Any, current order of array.
• None – Same as ‘A’.

See Also:
ndarray.tostring,tolist,tofile

Notes
As for ndarray.tostring, information about the shape, dtype, etc., but also about fill_value, will be lost.
Examples

```python
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00\x01B\x0f\x00?B\x0f\x00?B\x0f\x00\x04\x00\x00\x00'
```

Pickling and unpickling

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.dump(a, F)</code></td>
<td>Pickle a masked array to a file.</td>
</tr>
<tr>
<td><code>ma.dumps(a)</code></td>
<td>Return a string corresponding to the pickling of a masked array.</td>
</tr>
<tr>
<td><code>ma.load(F)</code></td>
<td>Wrapper around <code>cPickle.load</code> which accepts either a file-like object</td>
</tr>
<tr>
<td><code>ma.loads(strg)</code></td>
<td>Load a pickle from the current string.</td>
</tr>
</tbody>
</table>
```

```python
numpy.ma.dump(a, F)
```

Pickle a masked array to a file.

This is a wrapper around `cPickle.dump`.

**Parameters**

- **a**: MaskedArray
  
The array to be pickled.
- **F**: str or file-like object
  
The file to pickle `a` to. If a string, the full path to the file.

```python
numpy.ma.dumps(a)
```

Return a string corresponding to the pickling of a masked array.

This is a wrapper around `cPickle.dumps`.

**Parameters**

- **a**: MaskedArray
  
The array for which the string representation of the pickle is returned.

```python
numpy.ma.load(F)
```

Wrapper around `cPickle.load` which accepts either a file-like object or a filename.

**Parameters**

- **F**: str or file
  
The file or file name to load.

**See Also:**

`dump`

Pickle an array

**Notes**

This is different from `numpy.load`, which does not use cPickle but loads the NumPy binary `.npy` format.

```python
numpy.ma.loads(strg)
```

Load a pickle from the current string.

The result of `cPickle.loads(strg)` is returned.

**Parameters**

- **strg**: str

366 Chapter 1. Array objects
The string to load.

See Also:

**dumps**

Return a string corresponding to the pickling of a masked array.

## Filling a masked array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.common_fill_value(a, b)</code></td>
<td>Return the common filling value of two masked arrays, if any.</td>
</tr>
<tr>
<td><code>ma.default_fill_value(obj)</code></td>
<td>Return the default fill value for the argument object.</td>
</tr>
<tr>
<td><code>ma.maximum_fill_value(obj)</code></td>
<td>Return the minimum value that can be represented by the dtype of an object.</td>
</tr>
<tr>
<td><code>ma.set_fill_value(a, fill_value)</code></td>
<td>Set the filling value of <code>a</code>, if <code>a</code> is a masked array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.get_fill_value()</code></td>
<td>Return the filling value of the masked array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.set_fill_value(value)</code></td>
<td>Set the filling value of the masked array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.fill_value</code></td>
<td>Filling value.</td>
</tr>
</tbody>
</table>

### numpy.ma.common_fill_value(a, b)

Return the common filling value of two masked arrays, if any.

If `a.fill_value == b.fill_value`, return the fill value, otherwise return None.

**Parameters**

- **a, b**: `MaskedArray`

  The masked arrays for which to compare fill values.

**Returns**

- **fill_value**: scalar or None

  The common fill value, or None.

### Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=3)
>>> y = np.ma.array([0, 1.], fill_value=3)
>>> np.ma.common_fill_value(x, y)
3.0
```

### numpy.ma.default_fill_value(obj)

Return the default fill value for the argument object.

The default filling value depends on the datatype of the input array or the type of the input scalar:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>int</td>
<td>999999</td>
</tr>
<tr>
<td>float</td>
<td>1.e20</td>
</tr>
<tr>
<td>complex</td>
<td>1.e20+0j</td>
</tr>
<tr>
<td>object</td>
<td>‘?’</td>
</tr>
<tr>
<td>string</td>
<td>‘N/A’</td>
</tr>
</tbody>
</table>

**Parameters**

- **obj**: ndarray, dtype or scalar

  The array data-type or scalar for which the default fill value is returned.
Returns

fill_value : scalar

The default fill value.

Examples

>>> np.ma.default_fill_value(1)
999999
>>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
1e+20
>>> np.ma.default_fill_value(np.dtype(complex))
(1e+20+0j)

numpy.ma.maximum_fill_value(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

Parameters

obj : {ndarray, dtype}

An object that can be queried for it’s numeric type.

Returns

val : scalar

The minimum representable value.

Raises

TypeError

If obj isn’t a suitable numeric type.

See Also:

minimum_fill_value

The inverse function.

set_fill_value

Set the filling value of a masked array.

MaskedArray.fill_value

Return current fill value.

Examples

>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648

An array of numeric data can also be passed.

>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
```python
>>> ma.maximum_fill_value(a)
-inf
```

```python
numpy.ma.maximum_fill_value(obj)
```

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

**Parameters**

- `obj` : {ndarray, dtype}
  An object that can be queried for it’s numeric type.

**Returns**

- `val` : scalar
  The minimum representable value.

**Raises**

- `TypeError`
  If `obj` isn’t a suitable numeric type.

**See Also:**

- `minimum_fill_value`
  The inverse function.

- `set_fill_value`
  Set the filling value of a masked array.

- `MaskedArray.fill_value`
  Return current fill value.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.

```python
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

```python
numpy.ma.set_fill_value(a, fill_value)
```

Set the filling value of `a`, if `a` is a masked array.

This function changes the fill value of the masked array `a` in place. If `a` is not a masked array, the function returns silently, without doing anything.

**Parameters**

- `a` : array_like
  Input array.
fill_value : dtype
Filling value. A consistency test is performed to make sure the value is compatible with the dtype of \( a \).

Returns
None
Nothing returned by this function.

See Also:

maximum_fill_value
Return the default fill value for a dtype.

MaskedArray.fill_value
Return current fill value.

MaskedArray.set_fill_value
Equivalent method.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data = [-- -- -- 3 4],
       mask = [ True True True False False],
      fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data = [-- -- -- 3 4],
       mask = [ True True True False False],
      fill_value=-999)

Nothing happens if \( a \) is not a masked array.

``` python
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])
```

MaskedArray.get_fill_value()
Return the filling value of the masked array.

Returns
fill_value : scalar
The filling value.
Examples

```python
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...     np.ma.array([0, 1], dtype=dt).get_fill_value()
...     999999
     999999
     1e+20
     (1e+20+0j)

>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf
```

MaskedArray.set_fill_value(value=None)

Set the filling value of the masked array.

Parameters

- **value**: scalar, optional
  - The new filling value. Default is None, in which case a default based on the data type is used.

See Also:

- **ma.set_fill_value**
  - Equivalent function.

Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931
```

Reset to default:

```python
>>> x.set_fill_value()
>>> x.fill_value
1e+20
```

MaskedArray.fill_value

Filling value.

---

Masked arrays arithmetics

Arithmetics

- **ma.anom(self[, axis, dtype])**
  - Compute the anomalies (deviations from the arithmetic mean) along the given axis.

- **ma.anomalies(self[, axis, dtype])**
  - Compute the anomalies (deviations from the arithmetic mean) along the given axis.

- **ma.average(a[, axis, weights, returned])**
  - Return the weighted average of array over the given axis.

- **ma.conjugate(x[, out])**
  - Return the complex conjugate, element-wise.

- **ma.corrcoef(x[, y, rowvar, bias, ...])**
  - Return correlation coefficients of the input array.
Table 1.95 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.cov(x[, y, rowvar, bias, allow_masked, ddof])</code></td>
<td>Estimate the covariance matrix.</td>
</tr>
<tr>
<td><code>ma.cumsum(self[, axis, dtype, out])</code></td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.cumprod(self[, axis, dtype, out])</code></td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.mean(self[, axis, dtype, out])</code></td>
<td>Returns the average of the array elements.</td>
</tr>
<tr>
<td><code>ma.median(a[, axis, out, overwrite_input])</code></td>
<td>Compute the median along the specified axis.</td>
</tr>
<tr>
<td><code>ma.power(a, b[, third])</code></td>
<td>Returns element-wise base array raised to power from second array.</td>
</tr>
<tr>
<td><code>ma.prod(self[, axis, dtype, out])</code></td>
<td>Return the product of the array elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.std(self[, axis, dtype, out, ddof])</code></td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td><code>ma.sum(self[, axis, dtype, out])</code></td>
<td>Return the sum of the array elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.var(self[, axis, dtype, out, ddof])</code></td>
<td>Compute the variance along the specified axis.</td>
</tr>
</tbody>
</table>

```
# numpy.ma.anom

Computes the anomalies (deviations from the arithmetic mean) along the given axis.

Parameters

- **axis** : int, optional
  
  Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

- **dtype** : dtype, optional
  
  Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

- `mean`

Examples

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
            fill_value = 1e+20)
```

```
# numpy.ma.anomalies

Computes the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

```

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anomalies()
masked_array(data = [-1. 0. 1.],
             mask = False,
            fill_value = 1e+20)
```
Parameters

**axis**: int, optional

Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

**dtype**: dtype, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

`mean`

Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
             mask = False,
             fill_value = 1e+20)
```

`numpy.ma.average(a, axis=None, weights=None, returned=False)`

Return the weighted average of array over the given axis.

Parameters

**a**: array_like

Data to be averaged. Masked entries are not taken into account in the computation.

**axis**: int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

**weights**: array_like, optional

The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of `a` along the given axis) or of the same shape as `a`. If `weights=None`, then all data in `a` are assumed to have a weight equal to one. If `weights` is complex, the imaginary parts are ignored.

**returned**: bool, optional

Flag indicating whether a tuple (result, sum of weights) should be returned as output (True), or just the result (False). Default is False.

Returns

**average, [sum_of_weights]**: (tuple of) scalar or MaskedArray

The average along the specified axis. When returned is True, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is `np.float64` if `a` is of integer type, otherwise it is of the same type as `a`. If returned, `sum_of_weights` is of the same type as `average`.

Examples

```python
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25
```
numpy.ma.conjugate(x[, out]) = <numpy.ma.core._MaskedUnaryOperation instance at 0x272bfc8>
Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters
  x : array_like
      Input value.

Returns
  y : ndarray
      The complex conjugate of x, with same dtype as y.

Examples
>>> np.conjugate(1+2j)
(1-2j)

>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j, 0.-0.j],
       [ 0.-0.j, 1.-1.j]])

numpy.ma.corrcoef(x, y=None, rowvar=True, bias=False, allow_masked=True, ddof=None)
Return correlation coefficients of the input array.

Except for the handling of missing data this function does the same as numpy.corrcoef. For more details and examples, see numpy.corrcoef.

Parameters
  x : array_like
      A 1-D or 2-D array containing multiple variables and observations. Each row of x represents a variable, and each column a single observation of all those variables. Also see rowvar below.
  y : array_like, optional
      An additional set of variables and observations. y has the same shape as x.
  rowvar : bool, optional
      If rowvar is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.
  bias : bool, optional
      Default normalization (False) is by \((N-1)\), where \(N\) is the number of observations given (unbiased estimate). If bias is 1, then normalization is by \(N\). This keyword can be overridden by the keyword ddof in numpy versions >= 1.5.
allow_masked : bool, optional
    If True, masked values are propagated pair-wise: if a value is masked in \( x \), the corresponding value is masked in \( y \). If False, raises an exception.

ddof : {None, int}, optional
    New in version 1.5. If not None normalization is by \((N - \text{ddof})\), where \( N \) is the number of observations; this overrides the value implied by bias. The default value is None.

See Also:

numpy.corrcoef
Equivalent function in top-level NumPy module.

cov
Estimate the covariance matrix.

numpy.ma.cov (x=None, y=None, rowvar=True, bias=False, allow_masked=True, ddof=None)
Estimate the covariance matrix.

Except for the handling of missing data this function does the same as numpy.cov. For more details and examples, see numpy.cov.

By default, masked values are recognized as such. If \( x \) and \( y \) have the same shape, a common mask is allocated: if \( x[i,j] \) is masked, then \( y[i,j] \) will also be masked. Setting allow_masked to False will raise an exception if values are missing in either of the input arrays.

Parameters

\( x \) : array_like
    A 1-D or 2-D array containing multiple variables and observations. Each row of \( x \) represents a variable, and each column a single observation of all those variables. Also see rowvar below.

\( y \) : array_like, optional
    An additional set of variables and observations. \( y \) has the same form as \( x \).

rowvar : bool, optional
    If rowvar is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

bias : bool, optional
    Default normalization (False) is by \((N-1)\), where \( N \) is the number of observations given (unbiased estimate). If bias is True, then normalization is by \( N \). This keyword can be overridden by the keyword ddof in numpy versions >= 1.5.

allow_masked : bool, optional
    If True, masked values are propagated pair-wise: if a value is masked in \( x \), the corresponding value is masked in \( y \). If False, raises a ValueError exception when some values are missing.

ddof : {None, int}, optional
    New in version 1.5. If not None normalization is by \((N - \text{ddof})\), where \( N \) is the number of observations; this overrides the value implied by bias. The default value is None.
Raises

ValueError

Raised if some values are missing and *allow_masked* is False.

See Also:

numpy.cov

```
numpy.ma.cumsum(self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820200>
```

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

*axis*: {None, -1, int}, optional

Axis along which the sum is computed. The default (*axis* = None) is to compute over the flattened array. *axis* may be negative, in which case it counts from the last to the first axis.

*dtype*: {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

*out*: ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

```
cumsum : ndarray.
```

A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned.

Notes

The mask is lost if *out* is not a valid *MaskedArray*!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

```
numpy.ma.cumprod(self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820170>
```

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

*axis*: {None, -1, int}, optional
Axis along which the product is computed. The default \((axis = None)\) is to compute over the flattened array.

dtype : \{None, dtype\}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns
cumprod : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

numpy.ma.mean (self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820440>

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to numpy.mean for the full documentation.

Parameters

a : array_like

Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : dtype, optional

Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

mean : ndarray, see dtype parameter above

If out=None, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

numpy.ma.mean

Equivalent function.

1.7. Masked arrays
**numpy.mean**
Equivalent function on non-masked arrays.

**numpy.ma.average**
Weighted average.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
              mask = [False False True],
              fill_value = 999999)
>>> a.mean()
1.5
```

**numpy.ma.median** *(a, axis=None, out=None, overwrite_input=False)*

Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

- **a**: array_like
  Input array or object that can be converted to an array.

- **axis**: int, optional
  Axis along which the medians are computed. The default (None) is to compute the median along a flattened version of the array.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

- **overwrite_input**: bool, optional
  If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is True, and the input is not already an ndarray, an error will be raised.

**Returns**

- **median**: ndarray
  A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return data-type is float64 for integers and floats smaller than float64, or the input data-type, otherwise.

**See Also:**

- mean

**Notes**

Given a vector \( V \) with \( N \) non masked values, the median of \( V \) is the middle value of a sorted copy of \( V \) (\( Vs \)) - i.e. \( Vs[(N-1)/2], \) when \( N \) is odd, or \( (Vs[N/2 - 1] + Vs[N/2])/2 \) when \( N \) is even.
Examples

>>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)
1.5

>>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
2.5

>>> np.ma.extras.median(x, axis=-1, overwrite_input=True)
masked_array(data = [ 2. 5.],
             mask = False,
             fill_value = 1e+20)

numpy.ma.power(a, b, third=None)

Returns element-wise base array raised to power from second array.

This is the masked array version of numpy.power. For details see numpy.power.

See Also:

numpy.power

Notes

The out argument to numpy.power is not supported, third has to be None.

numpy.ma.prod(self, axis=None, dtype=None, out=None)

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters

axis : {None, int}, optional

Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

Returns

product_along_axis : {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

See Also:

prod
equivalent function

1.7. Masked arrays
Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

`numpy.ma.std(self, axis=None, dtype=None, out=None, ddof=0)`

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- `a`: array_like
  Calculate the standard deviation of these values.

- `axis`: int, optional
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- `dtype`: dtype, optional
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- `out`: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- `ddof`: int, optional
  Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default `ddof` is zero.

- `keepdims`: bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.

Returns

- `standard_deviation`: ndarray, see dtype parameter above.
  If `out` is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

- `var`, `mean`, `nanmean`, `nanstd`, `nanvar`
- `numpy.doc.ufuncs`

Section “Output arguments”
Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean()}^2))} \).

The average squared deviation is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( ddof=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.118033988749895
>>> np.std(a, axis=0)
array([ 1., 1.])
>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, \( \text{std}() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

```
numpy.ma.sum(self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820908>
```

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

Parameters

- **axis**: {None, -1, int}, optional
  - Axis along which the sum is computed. The default (\( axis = \text{None} \)) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  - Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional

1.7. Masked arrays
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

sum_along_axis : MaskedArray or scalar

An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

Examples

>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
[[1 -- 3]
 [-- 5 --]
[7 -- 9]]

>>> print x.sum()
25

>>> print x.sum(axis=1)
[4 5 16]

>>> print x.sum(axis=0)
[8 5 12]

>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>

Parameters

a : array_like

Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

axis : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

out : ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddf : int, optional

“Delta Degrees of Freedom”: the divisor used in the calculation is N − ddf, where N represents the number of elements. By default ddf is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.
Returns

variance : ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise, a reference to
the output array is returned.

See Also:

std, mean, nanmean, nanstd, nanvar

numpy.doc.ufuncs
Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean}())**2) \).

The mean is normally calculated as \( \frac{x.\text{sum}}{N} \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the
divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator
of the variance of a hypothetical infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the
variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and
nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the
input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a
higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1., 1.])
>>> np.var(a, axis=1)
array([ 0.25, 0.25])
```

In single precision, var() can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

MaskedArray.anom (axis=None, dtype=None)

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed
along the given axis.
Parameters

axis : int, optional

Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

mean

Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1, 2, 3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
              mask = False,
              fill_value = 1e+20)
```

MaskedArray.cumprod(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

axis : {None, -1, int}, optional

Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

dtype : {None, dtype}, optional

Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns
cumprod : ndarray

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.
MaskedArray.

cumsum (axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis. The cumulative sum is calculated over
the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result
will be masked at the same locations.

Parameters

axis : {None, -1, int}, optional

Axis along which the sum is computed. The default (axis = None) is to compute over
the flattened array. axis may be negative, in which case it counts from the last to the
first axis.

dtype : {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If
dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a
precision less than that of the default platform integer. In that case, the default platform
integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape and
buffer length as the expected output but the type will be cast if necessary.

Returns
cumsum : ndarray.

A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]

MaskedArray.

mean (axis=None, dtype=None, out=None)

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified
axis. Refer to numpy.mean for the full documentation.

Parameters

a : array_like

Array containing numbers whose mean is desired. If a is not an array, a conversion is
attempted.

axis : int, optional

Axis along which the means are computed. The default is to compute the mean of the
flattened array.

dtype : dtype, optional
Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **mean**: ndarray, see dtype parameter above

  If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also:**

- `numpy.ma.mean`
  Equivalent function.

- `numpy.mean`
  Equivalent function on non-masked arrays.

- `numpy.ma.average`
  Weighted average.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
      mask = [False False True],
    fill_value = 999999)
>>> a.mean()
1.5
```

**MaskedArray.prod** `(axis=None, dtype=None, out=None)`

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

**Parameters**

- **axis**: {None, int}, optional
  
  Axis over which the product is taken. If None is used, then the product is over all the array elements.

- **dtype**: {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of `a`.

- **out**: {None, array}, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **product_along_axis**: {array, scalar}, see dtype parameter above.

  Returns an array whose shape is the same as `a` with the specified axis removed. Returns a 0d array when `a` is 1d or `axis=None`. Returns a reference to the specified output array if specified.
See Also:

`prod`

equivalent function

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

MaskedArray.<code>std</code>(<code>axis=None, dtype=None, out=None, ddof=0</code>)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- <code>a</code>: array_like
  Calculate the standard deviation of these values.

- <code>axis</code>: int, optional
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- <code>dtype</code>: dtype, optional
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- <code>out</code>: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- <code>ddof</code>: int, optional
  Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof \) is zero.

- <code>keepdims</code>: bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \( arr \).

Returns

- <code>standard_deviation</code>: ndarray, see <code>dtype</code> parameter above.
  If <code>out</code> is None, return a new array containing the standard deviation, otherwise return a reference to the output array.
See Also:

var, mean, nanmean, nanstd, nanvar

deprecated

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{mean}(x))^2)} \). 

The average squared deviation is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( ddof=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([1., 1.])
>>> np.std(a, axis=1)
array([0.5, 0.5])
```

In single precision, \( \text{std}() \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

MaskedArray. sum (axis=None, dtype=None, out=None)

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

Parameters

- **axis**: {None, -1, int}, optional

  Axis along which the sum is computed. The default \( axis = \text{None} \) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional

  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision
less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

**out** : {None, ndarray}, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

**sum_along_axis** : MaskedArray or scalar

An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

**Examples**

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
>>> print x.sum()
25
>>> print x.sum(axis=1)
[4 5 16]
>>> print x.sum(axis=0)
[8 5 12]
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type ‘numpy.int64’>
```

MaskedArray.var (axis=0, dtype=0, out=0, ddof=0)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

**a** : array_like

Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

**axis** : int, optional

Axis along which the variance is computed. The default is to compute the variance of the flattened array.

**dtype** : data-type, optional

Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

**out** : ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof** : int, optional

“Delta Degrees of Freedom”: the divisor used in the calculation is N - ddof, where N represents the number of elements. By default ddof is zero.

**keepdims** : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.

**Returns**

- **variance**: ndarray, see dtype parameter above

  If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**See Also**

- `std`, `mean`, `nanmean`, `nanstd`, `nanvar`
- `numpy.doc.ufuncs`  
  Section “Output arguments”

**Notes**

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean})^2) \).

The mean is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, `ddof` is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1.,  1.])
>>> np.var(a, axis=1)
array([ 0.25,  0.25])
```

In single precision, `var()` can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:]=1.0
>>> a[1,:]=0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in `float64` is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
```

\[
\frac{(1-0.55)^2 + (0.1-0.55)^2}{2} = 0.20250000000000001
\]
### Table 1.96 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argmax(a[, axis, fill_value])</code></td>
<td>Function version of the eponymous method.</td>
</tr>
<tr>
<td><code>ma.argmin(a[, axis, fill_value])</code></td>
<td>Returns array of indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.max(obj[, axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.min(obj[, axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.ptp(obj[, axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmax([axis, fill_value, out])</code></td>
<td>Returns array of indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmin([axis, fill_value, out])</code></td>
<td>Return array of indices to the minimum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.max([axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.min([axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ptp([axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e.</td>
</tr>
</tbody>
</table>

**numpy.ma.argmax (a, axis=None, fill_value=None)**

Function version of the eponymous method.

**numpy.ma.argmin (a, axis=None, fill_value=None)**

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.

**Parameters**

- **axis**: {None, integer}
  - If None, the index is into the flattened array, otherwise along the specified axis
- **fill_value**: {var}, optional
  - Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.
- **out**: {None, array}, optional
  - Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

- **index_array**: {integer_array}

**Examples**

```python
gap> a = np.arange(6).reshape(2,3)
gap> a.argmax()
5
gap> a.argmin(0)
array([1, 1, 1])
gap> a.argmax(1)
array([2, 2])
```

**numpy.ma.max (obj, axis=None, out=None, fill_value=None)**

Return the maximum along a given axis.

**Parameters**

- **axis**: {None, int}, optional
  - Axis along which to operate. By default, axis is None and the flattened input is used.
- **out**: array_like, optional
  - Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

---

**1.7. Masked arrays**
fill_value : {var}, optional

Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns
amax : array_like

New array holding the result. If out was specified, out is returned.

See Also:
maximum_fill_value
Returns the maximum filling value for a given datatype.

numpy.ma.min (obj, axis=None, out=None, fill_value=None)
Return the minimum along a given axis.

Parameters
axis : {None, int}, optional
Axis along which to operate. By default, axis is None and the flattened input is used.
out : array_like, optional
Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.
fill_value : {var}, optional
Value used to fill in the masked values. If None, use the output of minimum_fill_value.

Returns
amin : array_like
New array holding the result. If out was specified, out is returned.

See Also:
minimum_fill_value
Returns the minimum filling value for a given datatype.

numpy.ma.ptp (obj, axis=None, out=None, fill_value=None)
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

Parameters
axis : {None, int}, optional
Axis along which to find the peaks. If None (default) the flattened array is used.
out : {None, array_like}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.
fill_value : {var}, optional
Value used to fill in the masked values.

Returns
ptp : ndarray.
A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

`MaskedArray.argmax(axis=None, fill_value=None, out=None)`

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value `fill_value`.

**Parameters**

- `axis` : {None, integer}
  
  If None, the index is into the flattened array, otherwise along the specified axis

- `fill_value` : {var}, optional
  
  Value used to fill in the masked values. If None, the output of `maximum_fill_value(self._data)` is used instead.

- `out` : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

- `index_array` : {integer_array}

**Examples**

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

`MaskedArray.argmin(axis=None, fill_value=None, out=None)`

Return array of indices to the minimum values along the given axis.

**Parameters**

- `axis` : {None, integer}
  
  If None, the index is into the flattened array, otherwise along the specified axis

- `fill_value` : {var}, optional
  
  Value used to fill in the masked values. If None, the output of `minimum_fill_value(self._data)` is used instead.

- `out` : {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

- `{ndarray, scalar}`
  
  If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.
Examples

```python
>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[- --]
 [2 3]]
>>> print x.argmin(axis=0, fill_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill_value=9)
[1 1]
```

MaskedArray.max (axis=None, out=None, fill_value=None)

Return the maximum along a given axis.

Parameters

- `axis` : {None, int}, optional
  Axis along which to operate. By default, `axis` is None and the flattened input is used.

- `out` : array_like, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- `fill_value` : {var}, optional
  Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns

- `amax` : array_like
  New array holding the result. If `out` was specified, `out` is returned.

See Also:

- `maximum_fill_value`

MaskedArray.min (axis=None, out=None, fill_value=None)

Return the minimum along a given axis.

Parameters

- `axis` : {None, int}, optional
  Axis along which to operate. By default, `axis` is None and the flattened input is used.

- `out` : array_like, optional
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- `fill_value` : {var}, optional
  Value used to fill in the masked values. If None, use the output of minimum_fill_value().

Returns

- `amin` : array_like
  New array holding the result. If `out` was specified, `out` is returned.

See Also:
**minimum_fill_value**

Returns the minimum filling value for a given datatype.

**MaskedArray.ptp** *(axis=None, out=None, fill_value=None)*

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

**Parameters**

- **axis**: {None, int}, optional
  
  Axis along which to find the peaks. If None (default) the flattened array is used.

- **out**: {None, array_like}, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

- **fill_value**: {var}, optional
  
  Value used to fill in the masked values.

**Returns**

- **ptp**: ndarray.
  
  A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

**Sorting**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argsort(a[, axis, kind, order, fill_value])</code></td>
<td>Return an ndarray of indices that sort the array along the specified axis.</td>
</tr>
<tr>
<td><code>ma.sort(a[, axis, kind, order, endwith, ...])</code></td>
<td>Sort the array, in-place</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argsort([axis, kind, order, ...])</code></td>
<td>Return an ndarray of indices that sort the array along the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.sort([axis, kind, order, ...])</code></td>
<td>Sort the array, in-place</td>
</tr>
</tbody>
</table>

**numpy.ma.argsort** *(a, axis=None, kind=’quicksort’, order=None, fill_value=None)*

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to `fill_value`.

**Parameters**

- **axis**: int, optional
  
  Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

- **fill_value**: var, optional
  
  Value used to fill the array before sorting. The default is the `fill_value` attribute of the input array.

- **kind**: {'quicksort', 'mergesort', 'heapsort'}, optional
  
  Sorting algorithm.

- **order**: list, optional
  
  When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

- **index_array**: ndarray, int
  
  Array of indices that sort `a` along the specified axis. In other words, `a[index_array]` yields a sorted `a`.  

1.7. Masked arrays 395
See Also:

**sort**
Describes sorting algorithms used.

**lexsort**
Indirect stable sort with multiple keys.

**ndarray.sort**
Inplace sort.

Notes
See **sort** for notes on the different sorting algorithms.

Examples

```python
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```

```
numpy.ma.sort (a, axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)
    Sort the array, in-place
```

Parameters

- **a**: array_like
  Array to be sorted.

- **axis**: int, optional
  Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

- **kind**: {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm. Default is 'quicksort'.

- **order**: list, optional
  When *a* is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

- **endwith**: {True, False}, optional
  Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

- **fill_value**: {var}, optional
  Value used internally for the masked values. If **fill_value** is not None, it supersedes **endwith**.

Returns

- **sorted_array**: ndarray
  Array of the same type and shape as *a*.

See Also:
**ndarray.sort**  
Method to sort an array in-place.

**argsort**  
Indirect sort.

**lexsort**  
Indirect stable sort on multiple keys.

**searchsorted**  
Find elements in a sorted array.

**Notes**

See `sort` for notes on the different sorting algorithms.

**Examples**

```python
>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Default
>>> a.sort()
>>> print a
[1 3 5 -- --]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Put missing values in the front
>>> a.sort(endwith=False)
>>> print a
[-- -- 1 3 5]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # fill_value takes over endwith
>>> a.sort(endwith=False, fill_value=3)
>>> print a
[1 -- -- 3 5]
```

MaskedArray.**argsort**(axis=None, kind='quicksort', order=None, fill_value=None)  
Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to `fill_value`.

**Parameters**

- **axis** : int, optional  
  Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

- **fill_value** : var, optional  
  Value used to fill the array before sorting. The default is the `fill_value` attribute of the input array.

- **kind** : {'quicksort', 'mergesort', 'heapsort'}, optional  
  Sorting algorithm.

- **order** : list, optional  
  When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**Returns**

- **index_array** : ndarray, int
Array of indices that sort $a$ along the specified axis. In other words, $a[index\_array]$ yields a sorted $a$.

**See Also:**

- **sort**
  Describes sorting algorithms used.
- **lexsort**
  Indirect stable sort with multiple keys.
- **ndarray.sort**
  Inplace sort.

**Notes**

See **sort** for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
        mask = [False False True],
       fill_value = 999999)
>>> a.argsort()
array([1, 0, 2])
```

MaskedArray.sort(axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)

Sort the array, in-place

**Parameters**

- **a**: array_like
  - Array to be sorted.
- **axis**: int, optional
  - Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.
- **kind**: {'quicksort', 'mergesort', 'heapsort'}, optional
  - Sorting algorithm. Default is ‘quicksort’.
- **order**: list, optional
  - When $a$ is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.
- **endwith**: {True, False}, optional
  - Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).
- **fill_value**: {var}, optional
  - Value used internally for the masked values. If fill_value is not None, it supersedes endwith.

**Returns**

- **sorted_array**: ndarray
  - Array of the same type and shape as $a$. 398 Chapter 1. Array objects
See Also:

**ndarray.sort**
Method to sort an array in-place.

**argsort**
Indirect sort.

**lexsort**
Indirect stable sort on multiple keys.

**searchsorted**
Find elements in a sorted array.

Notes
See `sort` for notes on the different sorting algorithms.

Examples

```python
>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Default
>>> a.sort()
>>> print a
[1 3 5 -- --]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Put missing values in the front
>>> a.sort(endwith=False)
>>> print a
[-- -- 1 3 5]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # fill_value takes over endwith
>>> a.sort(endwith=False, fill_value=3)
>>> print a
[1 -- -- 3 5]
```

Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.diag(v[, k])</td>
<td>Extract a diagonal or construct a diagonal array.</td>
</tr>
<tr>
<td>ma.dot(a, b[, strict])</td>
<td>Return the dot product of two arrays.</td>
</tr>
<tr>
<td>ma.identity(n[, dtype])</td>
<td>Return the identity array.</td>
</tr>
<tr>
<td>ma.inner(a, b)</td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td>ma.innerproduct(a, b)</td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td>ma.outer(a, b)</td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td>ma.outerproduct(a, b)</td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td>ma.trace(self[, offset, axis1, axis2, ...])</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td>ma.transpose(a[, axes])</td>
<td>Permute the dimensions of an array.</td>
</tr>
<tr>
<td>ma.MaskedArray.trace([offset, axis1, axis2, ...])</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td>ma.MaskedArray.transpose(*axes)</td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

```python
numpy.ma.diag(v, k=0)
```

Extract a diagonal or construct a diagonal array.

This function is the equivalent of `numpy.diag` that takes masked values into account, see `numpy.diag` for details.

1.7. Masked arrays 399
numpy.ma.dot(a, b, strict=False)
Return the dot product of two arrays.

Note: Works only with 2-D arrays at the moment.

This function is the equivalent of numpy.dot that takes masked values into account, see numpy.dot for details.

Parameters
- a, b : ndarray
  Inputs arrays.
- strict : bool, optional
  Whether masked data are propagated (True) or set to 0 (False) for the computation. Default is False. Propagating the mask means that if a masked value appears in a row or column, the whole row or column is considered masked.

See Also:
- numpy.dot
  Equivalent function for ndarrays.

Examples
>>> a = ma.array([[1, 2, 3], [4, 5, 6]], mask=[[1, 0, 0], [0, 0, 0]])
>>> b = ma.array([[1, 2], [3, 4], [5, 6]], mask=[[1, 0], [0, 0], [0, 0]])
>>> np.ma.dot(a, b)
masked_array(data =
[[21 26]
[45 64]],
mask =
[[False False]
[False False]],
fill_value = 999999)

>>> np.ma.dot(a, b, strict=True)
masked_array(data =
[[-- --]
[-- 64]],
mask =
[[ True  True]
[ True False]],
fill_value = 999999)

numpy.ma.identity(n, dtype=None) = <numpy.core._convert2ma instance at 0x2820d40>
Return the identity array.

The identity array is a square array with ones on the main diagonal.

Parameters
- n : int
  Number of rows (and columns) in n x n output.
**dtype**: data-type, optional

Data-type of the output. Defaults to float.

**Returns**

- **out**: ndarray
  
  $n \times n$ array with its main diagonal set to one, and all other elements 0.

**Examples**

```python
>>> np.identity(3)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

numpy.ma.inner(a, b)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

- **a, b**: array_like

  If a and b are nonscalar, their last dimensions of must match.

**Returns**

- **out**: ndarray

  out.shape = a.shape[:-1] + b.shape[:-1]

**Raises**

- **ValueError**

  If the last dimension of a and b has different size.

**See Also**:

tensordot
  
  Sum products over arbitrary axes.

dot
  
  Generalised matrix product, using second last dimension of b.
einsum
  
  Einstein summation convention.

**Notes**

Masked values are replaced by 0.

**Examples**

Ordinary inner product for vectors:

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([0, 1, 0])
>>> np.inner(a, b)
2
```

A multidimensional example:
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [135, 110, 134]])

An example where \( b \) is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

numpy.ma\_.innerproduct\( (a, b) \)
Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

Parameters:
a, b : array_like
   If \( a \) and \( b \) are nonscalar, their last dimensions of must match.

Returns:
out : ndarray
   out.shape = a.shape[:-1] + b.shape[:-1]

Raises:
ValueError
   If the last dimension of \( a \) and \( b \) has different size.

See Also:
tensordot
   Sum products over arbitrary axes.
dot
   Generalised matrix product, using second last dimension of \( b \).
einsum
   Einstein summation convention.

Notes

Masked values are replaced by 0.

Examples

Ordinary inner product for vectors:

```python
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
```

402 Chapter 1. Array objects
An example where $b$ is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

def numpy.ma.outer(a, b)

Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, a_1, \ldots, a_M]\) and \(b = [b_0, b_1, \ldots, b_N]\), the outer product [R48] is:

\[
\begin{bmatrix}
  a_0 \times b_0 & a_0 \times b_1 & \ldots & a_0 \times b_N \\
  a_1 \times b_0 & a_1 \times b_1 & \ldots & a_1 \times b_N \\
  \vdots & \vdots & \ddots & \vdots \\
  a_M \times b_0 & a_M \times b_1 & \ldots & a_M \times b_N
\end{bmatrix}
\]

**Parameters**

- **a**: (M,) array_like
  
  First input vector. Input is flattened if not already 1-dimensional.

- **b**: (N,) array_like
  
  Second input vector. Input is flattened if not already 1-dimensional.

**Returns**

- **out**: (M, N) ndarray
  
  \( \text{out}[i, j] = a[i] \times b[j] \)

**See Also:**

inner, einsum

**Notes**

Masked values are replaced by 0.

**References**

[R48]

**Examples**

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]]

>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j,  0.+2.j],
       [ 0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j,  0.+1.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]],
       dtype=complex)
```
An example using a “vector” of letters:

```python
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
       [b, bb, bbb],
       [c, cc, ccc]], dtype=object)
```

```
numpy.ma.outerproduct(a, b)
Compute the outer product of two vectors.

Given two vectors, a = [a0, a1, ..., aM] and b = [b0, b1, ..., bN], the outer product [R49] is:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    ...    ]
 [ ...    ...    ]
 [aM*b0         aM*bN ]]```

Parameters

a : (M,) array_like
    First input vector. Input is flattened if not already 1-dimensional.

b : (N,) array_like
    Second input vector. Input is flattened if not already 1-dimensional.

Returns

out : (M, N) ndarray
    out[i, j] = a[i] * b[j]

See Also:
inner, einsum

Notes
Masked values are replaced by 0.

References

[R49]

Examples
Make a (very coarse) grid for computing a Mandelbrot set:
>>> r1 = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> r1
array([[[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]],
       [[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]],
       [[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]],
       [[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]],
       [[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]]])

>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]],
       [[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]],
       [[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]],
       [[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]],
       [[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]]]

>>> grid = r1 + im
>>> grid
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]],
       [[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
        [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
        [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
        [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
        [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]]])

An example using a “vector” of letters:

>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
        [b, bb, bbb],
        [c, cc, ccc]], dtype=object)

numpy.ma.trace (self, offset=0, axis1=0, axis2=1, dtype=None, out=None) a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x28209e0>

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

See Also:

numpy.trace

Equivalent function

numpy.ma.transpose (a, axes=None) Permute the dimensions of an array.

This function is exactly equivalent to numpy.transpose.

See Also:

numpy.transpose

Equivalent function in top-level NumPy module.

Examples

>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked

1.7. Masked arrays
masked_array(data =
 [[0 1]
 [2 --]],
   mask =
 [[False False]
 [False  True]],
   fill_value = 999999)

>>> ma.transpose(x)
masked_array(data =
 [[0 2]
 [1 --]],
   mask =
 [[False False]
 [False  True]],
   fill_value = 999999)

MaskedArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)
Return the sum along diagonals of the array.
Refer to numpy.trace for full documentation.

See Also:

numpy.trace
equivalent function

MaskedArray.transpose(*axes)
Returns a view of the array with axes transposed.
For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters
axes : None, tuple of ints, or n ints
  • None or no argument: reverses the order of the axes.
  • tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
  • n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns
out : ndarray
  View of a, with axes suitably permuted.

See Also:

ndarray.T
  Array property returning the array transposed.

Examples
```python
>>> a = np.array([[1, 2], [3, 4]])
```
```text
>>> a
array([[1, 2],
       [3, 4]])
```
```text
>>> a.transpose()
array([[1, 3],
       [2, 4]])
```
```text
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
```
```text
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

Polynomial fit

**numpy.ma.vander**

Generate a Van der Monde matrix.

The columns of the output matrix are decreasing powers of the input vector. Specifically, the \(i\)-th output column is the input vector raised element-wise to the power of \(N - i - 1\). Such a matrix with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

**Parameters**

- **x**: array_like
  
  1-D input array.

- **N**: int, optional
  
  Order of (number of columns in) the output. If \(N\) is not specified, a square array is returned \((N = \text{len}(x))\).

**Returns**

- **out**: ndarray
  
  Van der Monde matrix of order \(N\). The first column is \(x^{(N-1)}\), the second \(x^{(N-2)}\) and so forth.

**Notes**

Masked values in the input array result in rows of zeros.

**Examples**

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```
```python
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

1.7. Masked arrays
\[
\begin{bmatrix}
9 & 3 & 1 \\
25 & 5 & 1 \\
\end{bmatrix}
\]

```python
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1,  1,  1,  1],
       [ 8,  4,  2,  1],
       [27,  9,  3,  1],
       [125, 25,  5,  1]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043
>>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```

```
np.ma.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
Least squares polynomial fit.
Fit a polynomial \( p(x) = p[0] \times x^{deg} + \ldots + p[deg] \) of degree \( deg \) to points \( (x, y) \). Returns a vector of coefficients \( p \) that minimises the squared error.

Parameters
- **x**: array_like, shape (M,)
x-coordinates of the M sample points \( (x[i], y[i]) \).
- **y**: array_like, shape (M,) or (M, K)y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.
- **deg**: int
Degree of the fitting polynomial
- **rcond**: float, optionalRelative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.
- **full**: bool, optionalSwitch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.
- **w**: array_like, shape (M,), optionalweights to apply to the y-coordinates of the sample points.
- **cov**: bool, optionalReturn the estimate and the covariance matrix of the estimate If full is True, then cov is not returned.

Returns
- **p**: ndarray, shape (M,) or (M, K)
Polynomial coefficients, highest power first. If y was 2-D, the coefficients for \(k\)-th data set are in \(p[:,k]\).

**residuals, rank, singular_values, rcond** : present only if \(\text{full} = \text{True}\)

Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of \(\text{rcond}\). For more details, see `linalg.lstsq`.

**V** : ndarray, shape (M,M) or (M,M,K)

The covariance matrix of the polynomial coefficient estimates. The diagonal of this matrix are the variance estimates for each coefficient. If y is a 2-d array, then the covariance matrix for the \(k\)-th data set are in \(V[:,:,k]\)

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if \(\text{full} = \text{False}\).

The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See Also:**

- **polyval**
  Computes polynomial values.

- **linalg.lstsq**
  Computes a least-squares fit.

- **scipy.interpolate.UnivariateSpline**
  Computes spline fits.

**Notes**

Any masked values in x is propagated in y, and vice-versa.

**References**

[R50], [R51]

**Examples**

```python
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])
```

It is convenient to use `poly1d` objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```
High-order polynomials may oscillate wildly:

```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```python
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

Clipping and rounding

- `ma.around` Round an array to the given number of decimals.
- `ma.clip(a, a_min, a_max[, out])` Clip (limit) the values in an array.
- `ma.round(a[, decimals, out])` Return a copy of a, rounded to `decimals` places.
- `ma.MaskedArray.clip(a_min, a_max[, out])` Return an array whose values are limited to `[a_min, a_max]`.
- `ma.MaskedArray.round([decimals, out])` Return `a` with each element rounded to the given number of decimals.

```python
numpy.ma.around = <numpy.ma.core.MaskedUnaryOperation instance at 0x2816440>
```

Round an array to the given number of decimals.

Refer to `around` for full documentation.

See Also:

- `around`
equivalent function

```
numpy.ma.clip(a, a_min, a_max, out=None)
```

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of \([0, 1]\) is specified, values smaller than 0 become 0, and values larger than 1 become 1.

**Parameters**

- **a**: array_like
  Array containing elements to clip.

- **a_min**: scalar or array_like
  Minimum value.

- **a_max**: scalar or array_like
  Maximum value. If **a_min** or **a_max** are array_like, then they will be broadcasted to the shape of **a**.

- **out**: ndarray, optional
  The results will be placed in this array. It may be the input array for in-place clipping. **out** must be of the right shape to hold the output. Its type is preserved.

**Returns**

- **clipped_array**: ndarray
  An array with the elements of **a**, but where values < **a_min** are replaced with **a_min**, and those > **a_max** with **a_max**.

**See Also:**

```
numpy.doc.ufuncs
```

Section “Output arguments”

**Examples**

```
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
```

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
```

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.clip(a, [3,4,1,1,1,4,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

```
numpy.ma.round(a, decimals=0, out=None)
```

Return a copy of **a**, rounded to ‘decimals’ places.

When ‘decimals’ is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and ‘decimals’ is greater than or equal to 0.

**Parameters**

- **decimals**: int
Number of decimals to round to. May be negative.

**out** : array_like

Existing array to use for output. If not given, returns a default copy of a.

**Notes**

If out is given and does not have a mask attribute, the mask of a is lost!

**MaskedArray**.clip(**a_min**, **a_max**, **out**=None)

Return an array whose values are limited to [a_min, a_max].

Refer to numpy.clip for full documentation.

**See Also:**

numpy.clip
equivalent function

**MaskedArray**.round(**decimals**=0, **out**=None)

Return a with each element rounded to the given number of decimals.

Refer to numpy.around for full documentation.

**See Also:**

numpy.around
equivalent function

**Miscellanea**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.allequal(a, b[, fill_value])</td>
<td>Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.</td>
</tr>
<tr>
<td>ma.allclose(a, b[, masked_equal, rtol, atol])</td>
<td>Returns True if two arrays are element-wise equal within a tolerance.</td>
</tr>
<tr>
<td>ma.arange([start[, stop[, step[, dtype]]]])</td>
<td>Return evenly spaced values within a given interval.</td>
</tr>
<tr>
<td>ma.choose(indices, choices[, out, mode])</td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td>ma.ediff1d(arr[, to_end, to_begin])</td>
<td>Compute the differences between consecutive elements of an array.</td>
</tr>
<tr>
<td>ma.indices(dimensions[, dtype])</td>
<td>Return an array representing the indices of a grid.</td>
</tr>
<tr>
<td>ma.where(condition[, x, y])</td>
<td>Return a masked array with elements from x or y, depending on condition.</td>
</tr>
</tbody>
</table>

**numpy.ma.allequal(a, b, fill_value=True)**

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

**Parameters**

- **a, b** : array_like
  - Input arrays to compare.

- **fill_value** : bool, optional
  - Whether masked values in a or b are considered equal (True) or not (False).

**Returns**

- **y** : bool
  - Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See Also:**
all, any, numpy.ma.allclose

Examples

```python
code>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
code>> a
masked_array(data = [10000000000.0 1e-07 --],
code     mask = [False False True],
code     fill_value=1e+20)
code>> b = array([1e10, 1e-7, -42.0])
code>> b
array([ 1.00000000e+10, 1.00000000e-07, -4.20000000e+01])
code>> ma.allequal(a, b, fill_value=False)
False
code>> ma.allequal(a, b)
True
```

numpy.ma.allclose(a, b, masked_equal=True, rtol=1e-05, atol=1e-08)

Returns True if two arrays are element-wise equal within a tolerance.

This function is equivalent to allclose except that masked values are treated as equal (default) or unequal, depending on the masked_equal argument.

Parameters

- `a, b` : array_like
  Input arrays to compare.

- `masked_equal` : bool, optional
  Whether masked values in `a` and `b` are considered equal (True) or not (False). They are considered equal by default.

- `rtol` : float, optional
  Relative tolerance. The relative difference is equal to rtol * b. Default is 1e-5.

- `atol` : float, optional
  Absolute tolerance. The absolute difference is equal to atol. Default is 1e-8.

Returns

- `y` : bool
  Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

See Also:

all, any

numpy.allclose

the non-masked allclose.

Notes

If the following equation is element-wise True, then allclose returns True:

```
absolute(a - b) <= (atol + rtol * absolute(b))
```

Return True if all elements of `a` and `b` are equal subject to given tolerances.
Examples

```python
>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
              mask = [False False True],
              fill_value = 1e+20)
>>> b = ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
False

>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

Masked values are not compared directly.

```python
>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

numpy.ma.apply_along_axis(func1d, axis, arr, *args, **kwargs)

Apply a function to 1-D slices along the given axis.

Execute `func1d(a, *args)` where `func1d` operates on 1-D arrays and `a` is a 1-D slice of `arr` along `axis`.

Parameters

- **func1d**: function
  - This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis.
- **axis**: integer
  - Axis along which `arr` is sliced.
- **arr**: ndarray
  - Input array.
- **args**: any
  - Additional arguments to `func1d`.

Returns

- **apply_along_axis**: ndarray
  - The output array. The shape of `outarr` is identical to the shape of `arr`, except along the `axis` dimension, where the length of `outarr` is equal to the size of the return value of `func1d`. If `func1d` returns a scalar `outarr` will have one fewer dimensions than `arr`.

See Also:

- apply_over_axes
  - Apply a function repeatedly over multiple axes.
Examples

```python
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([ 4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([ 2., 5., 8.])
```

For a function that doesn’t return a scalar, the number of dimensions in `outarr` is the same as `arr`.

```python
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

`numpy.ma.arange([start], stop[, step], dtype=None)`

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including `start` but excluding `stop`). For integer arguments the function is equivalent to the Python built-in `range` function, but returns an `ndarray` rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `linspace` for these cases.

**Parameters**

- `start`: number, optional
  
  Start of interval. The interval includes this value. The default start value is 0.

- `stop`: number
  
  End of interval. The interval does not include this value, except in some cases where `step` is not an integer and floating point round-off affects the length of `out`.

- `step`: number, optional
  
  Spacing between values. For any output `out`, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If `step` is specified, `start` must also be given.

- `dtype`: dtype
  
  The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

**Returns**

- `arange`: ndarray
  
  Array of evenly spaced values.

  For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of `out` being greater than `stop`.

**See Also:**

- `linspace`
  
  Evenly spaced numbers with careful handling of endpoints.
ogrid
Arrays of evenly spaced numbers in N-dimensions.

mgrid
Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples

>>> np.arange(3)
array([0, 1, 2])

>>> np.arange(3.0)
array([ 0., 1., 2.])

>>> np.arange(3,7)
array([3, 4, 5, 6])

>>> np.arange(3,7,2)
array([3, 5])

numpy.ma.choose (indices, choices, out=None, mode='raise')
Use an index array to construct a new array from a set of choices.

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of
the choice arrays. Where a value in a is i, the new array will have the value that choices[i] contains in the same
place.

Parameters

- a : ndarray of ints
  This array must contain integers in [0, n-1], where n is the number of choices.

- choices : sequence of arrays
  Choice arrays. The index array and all of the choices should be broadcastable to the
  same shape.

- out : array, optional
  If provided, the result will be inserted into this array. It should be of the appropriate
  shape and dtype.

- mode : {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices will behave.
  - 'raise' : raise an error
  - 'wrap' : wrap around
  - 'clip' : clip to the range

Returns

merged_array : array

See Also:

choose
equivalent function

Examples

>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data = [3 2 1],

mask = False,
fill_value=999999)

numpy.ma.ediff1d(arr, to_end=None, to_begin=None)

Compute the differences between consecutive elements of an array.

This function is the equivalent of numpy.ediff1d that takes masked values into account, see
numpy.ediff1d for details.

See Also:

numpy.ediff1d
Equivalent function for ndarrays.

numpy.ma.indices(dimensions, dtype=<type 'int'>)

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

Parameters
dimensions : sequence of ints
The shape of the grid.

dtype : dtype, optional
Data type of the result.

Returns
grid : ndarray
The array of grid indices, grid.shape = (len(dimensions),) +
tuple(dimensions).

See Also:
mgrid, meshgrid

Notes
The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if
dimensions is a tuple (r0, ..., rN-1) of length N, the output shape is (N, r0, ..., rN-1).

The subarrays grid[k] contains the N-D array of indices along the k-th axis. Explicitly:
grid[k,i0,i1,...,iN-1] = ik

Examples
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]  # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]  # column indices
array([[0, 1, 2],
       [0, 1, 2]])

The indices can be used as an index into an array.
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])

Note that it would be more straightforward in the above example to extract the required elements directly with x[:2, :3].

`numpy.ma.where(condition, x=None, y=None)`

Return a masked array with elements from x or y, depending on condition.

Returns a masked array, shaped like condition, where the elements are from x when `condition` is True, and from y otherwise. If neither x nor y are given, the function returns a tuple of indices where `condition` is True (the result of `condition.nonzero()`).

**Parameters**

- `condition` : array_like, bool
  
  The condition to meet. For each True element, yield the corresponding element from x, otherwise from y.

- `x`, `y` : array_like, optional
  
  Values from which to choose. x and y need to have the same shape as condition, or be broadcast-able to that shape.

**Returns**

- `out` : MaskedArray or tuple of ndarrays
  
  The resulting masked array if x and y were given, otherwise the result of `condition.nonzero()`.

**See Also:**

`numpy.where`

Equivalent function in the top-level NumPy module.

**Examples**

```python
>>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],
          ... [1, 0, 1],
          ... [0, 1, 0]])
>>> print x
[[0.0 -- 2.0]
 [-- 4.0 --]
 [6.0 -- 8.0]]
>>> np.ma.where(x > 5)  # return the indices where x > 5
(array([2, 2]), array([0, 2]))

>>> print np.ma.where(x > 5, x, -3.1416)
[[[0.0 -- 2.0]
  [-- 4.0 --]
  [6.0 -- 8.0]]
```

### 1.8 The Array Interface
Note: This page describes the numpy-specific API for accessing the contents of a numpy array from other C extensions. PEP 3118 – The Revised Buffer Protocol introduces similar, standardized API to Python 2.6 and 3.0 for any extension module to use. Cython’s buffer array support uses the PEP 3118 API; see the Cython numpy tutorial. Cython provides a way to write code that supports the buffer protocol with Python versions older than 2.6 because it has a backward-compatible implementation utilizing the array interface described here.

version
3

The array interface (sometimes called array protocol) was created in 2005 as a means for array-like Python objects to re-use each other’s data buffers intelligently whenever possible. The homogeneous N-dimensional array interface is a default mechanism for objects to share N-dimensional array memory and information. The interface consists of a Python-side and a C-side using two attributes. Objects wishing to be considered an N-dimensional array in application code should support at least one of these attributes. Objects wishing to support an N-dimensional array in application code should look for at least one of these attributes and use the information provided appropriately.

This interface describes homogeneous arrays in the sense that each item of the array has the same “type”. This type can be very simple or it can be a quite arbitrary and complicated C-like structure.

There are two ways to use the interface: A Python side and a C-side. Both are separate attributes.

1.8.1 Python side

This approach to the interface consists of the object having an __array_interface__ attribute.

__array_interface__

A dictionary of items (3 required and 5 optional). The optional keys in the dictionary have implied defaults if they are not provided.

The keys are:

shape (required)

TUPLE whose elements are the array size in each dimension. Each entry is an integer (a Python int or long). Note that these integers could be larger than the platform “int” or “long” could hold (a Python int is a C long). It is up to the code using this attribute to handle this appropriately; either by raising an error when overflow is possible, or by using Py_LONG_LONG as the C type for the shapes.

typestr (required)

A string providing the basic type of the homogenous array. The basic string format consists of 3 parts: a character describing the byteorder of the data (<: little-endian, >: big-endian, |: not-relevant), a character code giving the basic type of the array, and an integer providing the number of bytes the type uses.

The basic type character codes are:

| t | Bit field (following integer gives the number of bits in the bit field). |
| b | Boolean (integer type where all values are only True or False) |
| i | Integer |
| u | Unsigned integer |
| f | Floating point |
| c | Complex floating point |
| O | Object (i.e. the memory contains a pointer to PyObject) |
| S | String (fixed-length sequence of char) |
| U | Unicode (fixed-length sequence of Py_UNICODE) |
| V | Other (void * – each item is a fixed-size chunk of memory) |
**descr (optional)**

A list of tuples providing a more detailed description of the memory layout for each item in the homogeneous array. Each tuple in the list has two or three elements. Normally, this attribute would be used when `typestr` is `V[0-9]+`, but this is not a requirement. The only requirement is that the number of bytes represented in the `typestr` key is the same as the total number of bytes represented here. The idea is to support descriptions of C-like structs (records) that make up array elements. The elements of each tuple in the list are

1. A string providing a name associated with this portion of the record. This could also be a tuple of `(‘full name’, ‘basic name’)` where basic name would be a valid Python variable name representing the full name of the field.

2. Either a basic-type description string as in `typestr` or another list (for nested records)

3. An optional shape tuple providing how many times this part of the record should be repeated. No repeats are assumed if this is not given. Very complicated structures can be described using this generic interface. Notice, however, that each element of the array is still of the same data-type. Some examples of using this interface are given below.

**Default:** `[('', typestr)]`

**data (optional)**

A 2-tuple whose first argument is an integer (a long integer if necessary) that points to the data-area storing the array contents. This pointer must point to the first element of data (in other words any offset is always ignored in this case). The second entry in the tuple is a read-only flag (true means the data area is read-only).

This attribute can also be an object exposing the buffer interface which will be used to share the data. If this key is not present (or returns `None`), then memory sharing will be done through the buffer interface of the object itself. In this case, the offset key can be used to indicate the start of the buffer. A reference to the object exposing the array interface must be stored by the new object if the memory area is to be secured.

**Default:** `None`

**strides (optional)**

Either `None` to indicate a C-style contiguous array or a Tuple of strides which provides the number of bytes needed to jump to the next array element in the corresponding dimension. Each entry must be an integer (a Python `int` or `long`). As with shape, the values may be larger than can be represented by a C “int” or “long”; the calling code should handle this appropriately, either by raising an error, or by using `Py_LONG_LONG` in C. The default is `None` which implies a C-style contiguous memory buffer. In this model, the last dimension of the array varies the fastest. For example, the default strides tuple for an object whose array entries are 8 bytes long and whose shape is `(10,20,30)` would be `(4800, 240, 8)`

**Default:** `None` (C-style contiguous)

**mask (optional)**

None or an object exposing the array interface. All elements of the mask array should be interpreted only as true or not true indicating which elements of this array are valid. The shape of this object should be “broadcastable” to the shape of the original array.

**Default:** `None` (All array values are valid)

**offset (optional)**

An integer offset into the array data region. This can only be used when data is `None` or returns a buffer object.
Default: 0.

version (required)

An integer showing the version of the interface (i.e. 3 for this version). Be careful not to use this to invalidate objects exposing future versions of the interface.

1.8.2 C-struct access

This approach to the array interface allows for faster access to an array using only one attribute lookup and a well-defined C-structure.

__array_struct__

A PyObject whose voidptr member contains a pointer to a filled PyArrayInterface structure. Memory for the structure is dynamically created and the PyObject is also created with an appropriate destructor so the retriever of this attribute simply has to apply Py_DECREF to the object returned by this attribute when it is finished. Also, either the data needs to be copied out, or a reference to the object exposing this attribute must be held to ensure the data is not freed. Objects exposing the __array_struct__ interface must also not reallocate their memory if other objects are referencing them.

The PyArrayInterface structure is defined in numpy/ndarrayobject.h as:

```c
typedef struct {
  int two; /* contains the integer 2 -- simple sanity check */
  int nd; /* number of dimensions */
  char typekind; /* kind in array --- character code of typestr */
  int itemsize; /* size of each element */
  int flags; /* flags indicating how the data should be interpreted */
  /* must set ARR_HAS_DESCR bit to validate descr */
  Py_intptr_t *shape; /* A length-nd array of shape information */
  Py_intptr_t *strides; /* A length-nd array of stride information */
  void *data; /* A pointer to the first element of the array */
  PyObject *descr; /* NULL or data-description (same as descr key of __array_interface__) -- must set ARR_HAS_DESCR flag or this will be ignored. */
} PyArrayInterface;
```

The flags member may consist of 5 bits showing how the data should be interpreted and one bit showing how the Interface should be interpreted. The data-bits are CONTIGUOUS (0x1), FORTRAN (0x2), ALIGNED (0x100), NOTSWAPPED (0x200), and WRITEABLE (0x400). A final flag ARR_HAS_DESCR (0x800) indicates whether or not this structure has the arrdescr field. The field should not be accessed unless this flag is present.

New since June 16, 2006:

In the past most implementations used the “desc” member of the PyObject itself (do not confuse this with the “descr” member of the PyArrayInterface structure above — they are two separate things) to hold the pointer to the object exposing the interface. This is now an explicit part of the interface. Be sure to own a reference to the object when the PyObject is created using PyObject_FromVoidPtrAndDesc.

1.8.3 Type description examples

For clarity it is useful to provide some examples of the type description and corresponding __array_interface__ ‘desc’ entries. Thanks to Scott Gilbert for these examples:

In every case, the ‘desc’ key is optional, but of course provides more information which may be important for various applications:
* Float data
  
  typestr == 'f4'
  descr == [('','>f4')]

* Complex double
  
  typestr == 'c8'
  descr == [('real','>f4'), ('imag','>f4')]

* RGB Pixel data
  
  typestr == '|V3'
  descr == [('r','|u1'), ('g','|u1'), ('b','|u1')]

* Mixed endian (weird but could happen).
  
  typestr == '|V8' (or '>u8')
  descr == [('big','>i4'), ('little','<i4')]

* Nested structure
  
  struct {
    int ival;
    struct {
      unsigned short sval;
      unsigned char bval;
      unsigned char cval;
    } sub;
  }
  typestr == '|V8' (or '<u8' if you want)
  descr == [('ival','<i4'), ('sub', ['sval','<u2'), ('bval','|u1'), ('cval','|u1') ]]

* Nested array
  
  struct {
    int ival;
    double data[16*4];
  }
  typestr == '|V516'
  descr == [('ival','>i4'), ('data','>f8',(16,4))]

* Padded structure
  
  struct {
    int ival;
    double dval;
  }
  typestr == '|V16'
  descr == [('ival','>i4'), ('dval','>f8')]

It should be clear that any record type could be described using this interface.

1.8.4 Differences with Array interface (Version 2)

The version 2 interface was very similar. The differences were largely aesthetic. In particular:

1. The PyArrayInterface structure had no descr member at the end (and therefore no flag ARR_HAS_DESCR)

2. The desc member of the PyCObject returned from __array_struct__ was not specified. Usually, it was the object exposing the array (so that a reference to it could be kept and destroyed when the C-object was destroyed). Now it must be a tuple whose first element is a string with “PyArrayInterface Version #” and whose second element is the object exposing the array.
3. The tuple returned from `__array_interface__['data']` used to be a hex-string (now it is an integer or a long integer).

4. There was no `__array_interface__` attribute instead all of the keys (except for version) in the `__array_interface__` dictionary were their own attribute: Thus to obtain the Python-side information you had to access separately the attributes:

   • `__array_data__`
   • `__array_shape__`
   • `__array_strides__`
   • `__array_typestr__`
   • `__array_descr__`
   • `__array_offset__`
   • `__array_mask__`

## 1.9 Datetimes and Timedeltas

New in version 1.7.0. Starting in NumPy 1.7, there are core array data types which natively support datetime functionality. The data type is called “datetime64”, so named because “datetime” is already taken by the datetime library included in Python.

---

**Note:** The datetime API is experimental in 1.7.0, and may undergo changes in future versions of NumPy.

### 1.9.1 Basic Datetimes

The most basic way to create datetimes is from strings in ISO 8601 date or datetime format. The unit for internal storage is automatically selected from the form of the string, and can be either a date unit or a time unit. The date units are years (‘Y’), months (‘M’), weeks (‘W’), and days (‘D’), while the time units are hours (‘h’), minutes (‘m’), seconds (‘s’), milliseconds (‘ms’), and some additional SI-prefix seconds-based units.

---

**Example**

A simple ISO date:

```python
gnp.datetime64('2005-02-25')
numpy.datetime64('2005-02-25')
```

Using months for the unit:

```python
gnp.datetime64('2005-02')
numpy.datetime64('2005-02')
```

Specifying just the month, but forcing a ‘days’ unit:

```python
gnp.datetime64('2005-02', 'D')
numpy.datetime64('2005-02-01')
```

Using UTC “Zulu” time:

```python
gnp.datetime64('2005-02-25T03:30Z')
numpy.datetime64('2005-02-24T21:30-0600')
```
ISO 8601 specifies to use the local time zone if none is explicitly given:

```python
>>> np.datetime64('2005-02-25T03:30')
numpy.datetime64('2005-02-25T03:30-0600')
```

When creating an array of datetimes from a string, it is still possible to automatically select the unit from the inputs, by using the datetime type with generic units.

**Example**

```python
>>> np.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64')
array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64[D]')

>>> np.array(['2001-01-01T12:00', '2002-02-03T13:56:03.172'], dtype='datetime64')
array(['2001-01-01T12:00:00.000-0600', '2002-02-03T13:56:03.172-0600'], dtype='datetime64[ms]')
```

The datetime type works with many common NumPy functions, for example `arange` can be used to generate ranges of dates.

**Example**

All the dates for one month:

```python
>>> np.arange('2005-02', '2005-03', dtype='datetime64[D]')
```

The datetime object represents a single moment in time. If two datetimes have different units, they may still be representing the same moment of time, and converting from a bigger unit like months to a smaller unit like days is considered a ‘safe’ cast because the moment of time is still being represented exactly.

**Example**

```python
>>> np.datetime64('2005') == np.datetime64('2005-01-01')
True

>>> np.datetime64('2010-03-14T15Z') == np.datetime64('2010-03-14T15:00:00.00Z')
True
```

An important exception to this rule is between datetimes with *date units* and datetimes with *time units*. This is because this kind of conversion generally requires a choice of timezone and particular time of day on the given date.

**Example**

```python
>>> np.datetime64('2003-12-25', 's')
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
TypeError: Cannot parse "2003-12-25" as unit 's' using casting rule 'same_kind'
```
1.9.2 Datetime and Timedelta Arithmetic

NumPy allows the subtraction of two Datetime values, an operation which produces a number with a time unit. Because NumPy doesn’t have a physical quantities system in its core, the timedelta64 data type was created to complement datetime64.

Datetimes and Timedeltas work together to provide ways for simple datetime calculations.

Example

```python
>>> import numpy as np

>>> np.datetime64('2009-01-01') - np.datetime64('2008-01-01')
numpy.timedelta64(366,'D')

>>> np.datetime64('2009') + np.timedelta64(20, 'D')
numpy.datetime64('2009-01-21')

>>> np.datetime64('2011-06-15T00:00') + np.timedelta64(12, 'h')
numpy.datetime64('2011-06-15T12:00-0500')

>>> np.timedelta64(1,'W') / np.timedelta64(1,'D')
7.0
```

There are two Timedelta units (‘Y’, years and ‘M’, months) which are treated specially, because how much time they represent changes depending on when they are used. While a timedelta day unit is equivalent to 24 hours, there is no way to convert a month unit into days, because different months have different numbers of days.

Example

```python
>>> a = np.timedelta64(1, 'Y')

>>> np.timedelta64(a, 'M')
numpy.timedelta64(12,'M')

>>> np.timedelta64(a, 'D')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot cast NumPy timedelta64 scalar from metadata [Y] to [D] according to the rule ‘same_kind’
```

1.9.3 Datetime Units

The Datetime and Timedelta data types support a large number of time units, as well as generic units which can be coerced into any of the other units based on input data.

Datetimes are always stored based on POSIX time (though having a TAI mode which allows for accounting of leap-seconds is proposed), with a epoch of 1970-01-01T00:00Z. This means the supported dates are always a symmetric interval around the epoch, called “time span” in the table below.

The length of the span is the range of a 64-bit integer times the length of the date or unit. For example, the time span for ‘W’ (week) is exactly 7 times longer than the time span for ‘D’ (day), and the time span for ‘D’ (day) is exactly 24 times longer than the time span for ‘h’ (hour).
Here are the date units:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Time span (relative)</th>
<th>Time span (absolute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>year</td>
<td>+/- 9.2e18 years</td>
<td>[9.2e18 BC, 9.2e18 AD]</td>
</tr>
<tr>
<td>M</td>
<td>month</td>
<td>+/- 7.6e17 years</td>
<td>[7.6e17 BC, 7.6e17 AD]</td>
</tr>
<tr>
<td>W</td>
<td>week</td>
<td>+/- 1.7e17 years</td>
<td>[1.7e17 BC, 1.7e17 AD]</td>
</tr>
<tr>
<td>D</td>
<td>day</td>
<td>+/- 2.5e16 years</td>
<td>[2.5e16 BC, 2.5e16 AD]</td>
</tr>
</tbody>
</table>

And here are the time units:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Time span (relative)</th>
<th>Time span (absolute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>hour</td>
<td>+/- 1.0e15 years</td>
<td>[1.0e15 BC, 1.0e15 AD]</td>
</tr>
<tr>
<td>m</td>
<td>minute</td>
<td>+/- 1.7e13 years</td>
<td>[1.7e13 BC, 1.7e13 AD]</td>
</tr>
<tr>
<td>s</td>
<td>second</td>
<td>+/- 2.9e12 years</td>
<td>[2.9e12 BC, 2.9e12 AD]</td>
</tr>
<tr>
<td>ms</td>
<td>millisecond</td>
<td>+/- 2.9e9 years</td>
<td>[2.9e9 BC, 2.9e9 AD]</td>
</tr>
<tr>
<td>us</td>
<td>microsecond</td>
<td>+/- 2.9e6 years</td>
<td>[2.9e6 BC, 2.9e6 AD]</td>
</tr>
<tr>
<td>ns</td>
<td>nanosecond</td>
<td>+/- 292 years</td>
<td>[1678 AD, 2262 AD]</td>
</tr>
<tr>
<td>ps</td>
<td>picosecond</td>
<td>+/- 106 days</td>
<td>[1969 AD, 1970 AD]</td>
</tr>
<tr>
<td>fs</td>
<td>femtosecond</td>
<td>+/- 2.6 hours</td>
<td>[1969 AD, 1970 AD]</td>
</tr>
<tr>
<td>as</td>
<td>attosecond</td>
<td>+/- 9.2 seconds</td>
<td>[1969 AD, 1970 AD]</td>
</tr>
</tbody>
</table>

### 1.9.4 Business Day Functionality

To allow the datetime to be used in contexts where only certain days of the week are valid, NumPy includes a set of “busday” (business day) functions.

The default for busday functions is that the only valid days are Monday through Friday (the usual business days). The implementation is based on a “weekmask” containing 7 Boolean flags to indicate valid days; custom weekmasks are possible that specify other sets of valid days.

The “busday” functions can additionally check a list of “holiday” dates, specific dates that are not valid days.

The function `busday_offset` allows you to apply offsets specified in business days to datetimes with a unit of ‘D’ (day).

#### Example

```python
>>> np.busday_offset('2011-06-23', 1)
numpy.datetime64('2011-06-24')

>>> np.busday_offset('2011-06-23', 2)
numpy.datetime64('2011-06-27')
```

When an input date falls on the weekend or a holiday, `busday_offset` first applies a rule to roll the date to a valid business day, then applies the offset. The default rule is ‘raise’, which simply raises an exception. The rules most typically used are ‘forward’ and ‘backward’.

#### Example

```python
>>> np.busday_offset('2011-06-25', 2)
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
ValueError: Non-business day date in busday_offset

>>> np.busday_offset('2011-06-25', 0, roll='forward')
numpy.datetime64('2011-06-27')

>>> np.busday_offset('2011-06-25', 2, roll='forward')
numpy.datetime64('2011-06-29')
```
>>> np.busday_offset('2011-06-25', 0, roll='backward')
numpy.datetime64('2011-06-24')

>>> np.busday_offset('2011-06-25', 2, roll='backward')
numpy.datetime64('2011-06-28')

In some cases, an appropriate use of the roll and the offset is necessary to get a desired answer.

**Example**

The first business day on or after a date:

```python
>>> np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21', 'D')

>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22', 'D')
```

The first business day strictly after a date:

```python
>>> np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21', 'D')

>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23', 'D')
```

The function is also useful for computing some kinds of days like holidays. In Canada and the U.S., Mother’s day is on the second Sunday in May, which can be computed with a custom weekmask.

**Example**

```python
>>> np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13', 'D')
```

When performance is important for manipulating many business dates with one particular choice of weekmask and holidays, there is an object `busdaycalendar` which stores the data necessary in an optimized form.

**np.is_busday():**

To test a datetime64 value to see if it is a valid day, use `np.is_busday`.

**Example**

```python
>>> np.is_busday(np.datetime64('2011-07-15'))  # a Friday
True

>>> np.is_busday(np.datetime64('2011-07-16'))  # a Saturday
False

>>> np.is_busday(np.datetime64('2011-07-16'), weekmask='Sat Sun')
True

>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))

>>> np.is_busday(a)
array([ True,  True,  True,  True,  True, False, False], dtype='bool')
```
np.busday_count():

To find how many valid days there are in a specified range of datetime64 dates, use `busday_count`:

Example

```python
>>> np.busday_count(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
5
>>> np.busday_count(np.datetime64('2011-07-18'), np.datetime64('2011-07-11'))
-5
```

If you have an array of datetime64 day values, and you want a count of how many of them are valid dates, you can do this:

Example

```python
>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
>>> np.count_nonzero(np.is_busday(a))
5
```

Custom Weekmasks

Here are several examples of custom weekmask values. These examples specify the “busday” default of Monday through Friday being valid days.

Some examples:

```python
# Positional sequences; positions are Monday through Sunday.
# Length of the sequence must be exactly 7.
weekmask = [1, 1, 1, 1, 1, 0, 0]
# list or other sequence; 0 == invalid day, 1 == valid day
weekmask = "1111100"
# string '0' == invalid day, '1' == valid day
# string abbreviations from this list: Mon Tue Wed Thu Fri Sat Sun
weekmask = "Mon Tue Wed Thu Fri"
# any amount of whitespace is allowed; abbreviations are case-sensitive.
weekmask = "MonTue Wed Thu Fri"
```

1.9.5 Differences Between 1.6 and 1.7 Datetimes

The NumPy 1.6 release includes a more primitive datetime data type than 1.7. This section documents many of the changes that have taken place.

String Parsing

The datetime string parser in NumPy 1.6 is very liberal in what it accepts, and silently allows invalid input without raising errors. The parser in NumPy 1.7 is quite strict about only accepting ISO 8601 dates, with a few convenience extensions. 1.6 always creates microsecond (us) units by default, whereas 1.7 detects a unit based on the format of the string. Here is a comparison.:

```python
# NumPy 1.6.1
>>> np.datetime64('1979-03-22')
1979-03-22 00:00:00
```
# NumPy 1.7.0
>>> np.datetime64('1979-03-22')
numpy.datetime64('1979-03-22')

# NumPy 1.6.1, unit default microseconds
>>> np.datetime64('1979-03-22').dtype
dtype('datetime64[us]')

# NumPy 1.7.0, unit of days detected from string
>>> np.datetime64('1979-03-22').dtype
dtype('<M8[D]')

# NumPy 1.6.1, ignores invalid part of string
>>> np.datetime64('1979-03-2corruptedstring')
1979-03-02 00:00:00

# NumPy 1.7.0, raises error for invalid input
>>> np.datetime64('1979-03-2corruptedstring')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Error parsing datetime string "1979-03-2corruptedstring" at position 8

# NumPy 1.6.1, 'nat' produces today's date
>>> np.datetime64('nat')
2012-04-30 00:00:00

# NumPy 1.7.0, 'nat' produces not-a-time
>>> np.datetime64('nat')
numpy.datetime64('NaT')

# NumPy 1.6.1, 'garbage' produces today's date
>>> np.datetime64('garbage')
2012-04-30 00:00:00

# NumPy 1.7.0, 'garbage' raises an exception
>>> np.datetime64('garbage')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Error parsing datetime string "garbage" at position 0

# NumPy 1.6.1, can't specify unit in scalar constructor
>>> np.datetime64('1979-03-22T19:00', 'h')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function takes at most 1 argument (2 given)

# NumPy 1.7.0, unit in scalar constructor
>>> np.datetime64('1979-03-22T19:00', 'h')
numpy.datetime64('1979-03-22T19:00-0500','h')

# NumPy 1.6.1, reads ISO 8601 strings w/o TZ as UTC
>>> np.array(["1979-03-22T19:00'"], dtype='M8[h]')
array(['1979-03-22 19:00:00'], dtype=datetime64[h])

# NumPy 1.7.0, reads ISO 8601 strings w/o TZ as local (ISO specifies this)
>>> np.array(["1979-03-22T19:00'"], dtype='M8[h]')
array(['1979-03-22T19-0500'], dtype='datetime64[h]')

# NumPy 1.6.1, doesn't parse all ISO 8601 strings correctly
>>> np.array(['1979-03-22T12'], dtype='M8[h]')
array(['1979-03-22 00:00:00'], dtype=datetime64[h])

# NumPy 1.7.0 handles this case correctly
1.9. Datetimes and Timedeltas
Unit Conversion

The 1.6 implementation of datetime does not convert between units correctly:

```python
>>> np.array(['1979-03-22T12'], dtype='M8[h]')
array(['1979-03-22T12-0500'], dtype='datetime64[h]')
>>> np.array(['1979-03-22T12:00'], dtype='M8[h]')
array(['1979-03-22T12-0500'], dtype='datetime64[h]')
```

Datetime Arithmetic

The 1.6 implementation of datetime only works correctly for a small subset of arithmetic operations. Here we show some simple cases:

```python
# NumPy 1.6.1, produces invalid results if units are incompatible
>>> a = np.array(['1979-03-22T12'], dtype='M8[h]')
>>> b = np.array([3*60], dtype='m8[m]')
>>> a + b
array([1970-01-01 00:00:00.080988], dtype=datetim64[us])
# NumPy 1.7.0, promotes to higher-resolution unit
>>> a = np.array(['1979-03-22T12'], dtype='M8[h]')
>>> b = np.array([3*60], dtype='m8[m]')
>>> a + b
array(['1979-03-22T15:00-0500'], dtype='datetime64[m]')
```
```
# NumPy 1.6.1, arithmetic works if everything is microseconds
>>> a = np.array(['1979-03-22T12:00'], dtype='M8[us]')
>>> b = np.array([3*60*60+1000000], dtype='m8[us]')
>>> a + b
array([1979-03-22 15:00:00], dtype=datetim64[us])
# NumPy 1.7.0
>>> a = np.array(['1979-03-22T12:00'], dtype='M8[us]')
>>> b = np.array([3*60*60+1000000], dtype='m8[us]')
>>> a + b
array(['1979-03-22T15:00.080988-0500'], dtype='datetime64[us]')
```
UNIVERSAL FUNCTIONS (UFUNC)

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a ”vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

In Numpy, universal functions are instances of the numpy.ufunc class. Many of the built-in functions are implemented in compiled C code, but ufunc instances can also be produced using the frompyfunc factory function.

2.1 Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs. Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

1. All input arrays with ndim smaller than the input array of largest ndim, have 1’s prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension (the stride will be 0 for that dimension).

Broadcasting is used throughout NumPy to decide how to handle disparately shaped arrays; for example, all arithmetic operations (+, -, *, ...) between ndarrays broadcast the arrays before operation. A set of arrays is called “broadcastable” to the same shape if the above rules produce a valid result, i.e., one of the following is true:

1. The arrays all have exactly the same shape.
2. The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

Example

If a.shape is (5,1), b.shape is (1,6), c.shape is (6,) and d.shape is () so that d is a scalar, then a, b, c, and d are all broadcastable to dimension (5,6); and

- a acts like a (5,6) array where a[:,0] is broadcast to the other columns,
- b acts like a (5,6) array where b[0,:] is broadcast to the other rows,
• $c$ acts like a (1,6) array and therefore like a (5,6) array where $c[:]$ is broadcast to every row, and finally,
• $d$ acts like a (5,6) array where the single value is repeated.

### 2.2 Output type determination

The output of the ufunc (and its methods) is not necessarily an ndarray, if all input arguments are not ndarrays.

All output arrays will be passed to the __array_prepare__ and __array_wrap__ methods of the input (besides ndarrays, and scalars) that defines it and has the highest __array_priority__ of any other input to the universal function. The default __array_priority__ of the ndarray is 0.0, and the default __array_priority__ of a subtype is 1.0. Matrices have __array_priority__ equal to 10.0.

All ufuncs can also take output arguments. If necessary, output will be cast to the data-type(s) of the provided output array(s). If a class with an __array__ method is used for the output, results will be written to the object returned by __array__. Then, if the class also has an __array_prepare__ method, it is called so metadata may be determined based on the context of the ufunc (the context consisting of the ufunc itself, the arguments passed to the ufunc, and the ufunc domain.) The array object returned by __array_prepare__ is passed to the ufunc for computation. Finally, if the class also has an __array_wrap__ method, the returned ndarray result will be passed to that method just before passing control back to the caller.

### 2.3 Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of internal buffers is settable on a per-thread basis. There can be up to $2(n_{\text{inputs}} + n_{\text{outputs}})$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of a buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrectly-typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function

\[ \text{setbufsize}(\text{size}) \]

Set the size of the buffer used in ufuncs.

numpy.setbufsize(size)

Set the size of the buffer used in ufuncs.

**Parameters**

- **size**: int

  Size of buffer.

### 2.4 Error handling

Universal functions can trip special floating-point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. Error handling is controlled on a per-thread basis, and can be configured using the functions

\[ \text{seterr}([\text{all, divide, over, under, invalid}]) \]

Set how floating-point errors are handled.

\[ \text{seterrcall}([\text{func}]) \]

Set the floating-point error callback function or log object.
numpy.seterr (all=None, divide=None, over=None, under=None, invalid=None)

Set how floating-point errors are handled.

Note that operations on integer scalar types (such as int16) are handled like floating point, and are affected by these settings.

**Parameters**

- **all**: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  
  Set treatment for all types of floating-point errors at once:
  
  • ignore: Take no action when the exception occurs.
  • warn: Print a RuntimeWarning (via the Python warnings module).
  • raise: Raise a FloatingPointError.
  • call: Call a function specified using the seterrcall function.
  • print: Print a warning directly to stdout.
  • log: Record error in a Log object specified by seterrcall.

  The default is not to change the current behavior.

- **divide**: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional

  Treatment for division by zero.

- **over**: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional

  Treatment for floating-point overflow.

- **under**: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional

  Treatment for floating-point underflow.

- **invalid**: {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional

  Treatment for invalid floating-point operation.

**Returns**

- **old_settings**: dict

  Dictionary containing the old settings.

**See Also:**

- **seterrcall**

  Set a callback function for the ‘call’ mode.

- **geterr, geterrcall, errstate**

**Notes**

The floating-point exceptions are defined in the IEEE 754 standard [1]:

- Division by zero: infinite result obtained from finite numbers.
- Overflow: result too large to be expressed.
- Underflow: result so close to zero that some precision was lost.
- Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.
NumPy Reference, Release 1.8.1

Examples
>>> old_settings = np.seterr(all=’ignore’) #seterr to known value
>>> np.seterr(over=’raise’)
{’over’: ’ignore’, ’divide’: ’ignore’, ’invalid’: ’ignore’,
’under’: ’ignore’}
>>> np.seterr(**old_settings) # reset to default
{’over’: ’raise’, ’divide’: ’ignore’, ’invalid’: ’ignore’, ’under’: ’ignore’}
>>> np.int16(32000) * np.int16(3)
30464
>>> old_settings = np.seterr(all=’warn’, over=’raise’)
>>> np.int16(32000) * np.int16(3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FloatingPointError: overflow encountered in short_scalars
>>> old_settings = np.seterr(all=’print’)
>>> np.geterr()
{’over’: ’print’, ’divide’: ’print’, ’invalid’: ’print’, ’under’: ’print’}
>>> np.int16(32000) * np.int16(3)
Warning: overflow encountered in short_scalars
30464

numpy.seterrcall(func)
Set the floating-point error callback function or log object.
There are two ways to capture floating-point error messages. The first is to set the error-handler to ‘call’, using
seterr. Then, set the function to call using this function.
The second is to set the error-handler to ‘log’, using seterr. Floating-point errors then trigger a call to the
‘write’ method of the provided object.
Parameters
func : callable f(err, flag) or object with write method
Function to call upon floating-point errors (‘call’-mode) or object whose ‘write’ method
is used to log such message (‘log’-mode).
The call function takes two arguments. The first is the type of error (one of “divide”,
“over”, “under”, or “invalid”), and the second is the status flag. The flag is a byte, whose
least-significant bits indicate the status:
[0 0 0 0 invalid over under invalid]

In other words, flags = divide + 2*over + 4*under + 8*invalid.
If an object is provided, its write method should take one argument, a string.
Returns
h : callable, log instance or None
The old error handler.
See Also:
seterr, geterr, geterrcall
Examples
Callback upon error:

434

Chapter 2. Universal functions (ufunc)


```python
>>> def err_handler(type, flag):
...    print "Floating point error (%s), with flag %s" % (type, flag)
... 
>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf, Inf, Inf])

>>> np.seterrcall(saved_handler)
<function err_handler at 0x...>
>>> np.seterr(**save_err)
{'over': 'call', 'divide': 'call', 'invalid': 'call', 'under': 'call'}

Log error message:

```python
>>> class Log(object):
...    def write(self, msg):
...        print "LOG: %s" % msg
...

>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in divide
array([ Inf, Inf, Inf])

>>> np.seterrcall(saved_handler)
<__main__.Log object at 0x...>
>>> np.seterr(**save_err)
{'over': 'log', 'divide': 'log', 'invalid': 'log', 'under': 'log'}
```

### 2.5 Casting Rules

**Note:** In NumPy 1.6.0, a type promotion API was created to encapsulate the mechanism for determining output types. See the functions `result_type`, `promote_types`, and `min_scalar_type` for more details.

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the `.types` attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (character codes are used in said output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast “safely.” The first one it finds in its internal list of loops is selected and performed, after all necessary type casting. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer (which is user settable).
**Note:** Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating-point and integer values. See `ldexp` for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast "safely" to another data type. The answer to this question can be determined in Python with a function call: `can_cast(fromtype, totype)`. The Figure below shows the results of this call for the 24 internally supported types on the author's 64-bit system. You can generate this table for your system with the code given in the Figure.

**Figure**

Code segment showing the “can cast safely” table for a 32-bit system.

```python
def print_table(ntypes):
    print 'X',
    for char in ntypes: print char,
    print
    for row in ntypes:
        print row,
        for col in ntypes:
            print int(np.can_cast(row, col)),
        print

>>> print_table(np.typecodes['All'])
X ? b h i l q p B H I L Q P e f d g F D G S U V O M m
? 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

You should note that, while included in the table for completeness, the ‘S’, ‘U’, and ‘V’ types cannot be operated on by ufuncs. Also, note that on a 32-bit system the integer types may have different sizes, resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot “upcast” an array unless the scalar is of a fundamentally different kind of data (i.e., under a different hierarchy in the data-type hierarchy) than the array. This rule enables you to use scalar constants in your code (which, as Python types, are interpreted
accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.

## 2.6 ufunc

### 2.6.1 Optional keyword arguments

All ufuncs take optional keyword arguments. Most of these represent advanced usage and will not typically be used.

- **out**
  
  New in version 1.6. The first output can provided as either a positional or a keyword parameter.

- **where**
  
  New in version 1.7. Accepts a boolean array which is broadcast together with the operands. Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

- **casting**
  
  New in version 1.6. Provides a policy for what kind of casting is permitted. For compatibility with previous versions of NumPy, this defaults to ‘unsafe’. May be ‘no’, ‘equiv’, ‘safe’, ‘same_kind’, or ‘unsafe’. See `can_cast` for explanations of the parameter values.

  In a future version of numpy, this argument will default to ‘same_kind’. As part of this transition, starting in version 1.7, ufuncs will produce a DeprecationWarning for calls which are allowed under the ‘unsafe’ rules, but not under the ‘same_kind’ rules.

- **order**
  
  New in version 1.6. Specifies the calculation iteration order/memory layout of the output array. Defaults to ‘K’. ‘C’ means the output should be C-contiguous, ‘F’ means F-contiguous, ‘A’ means F-contiguous if the inputs are F-contiguous and not also not C-contiguous, C-contiguous otherwise, and ‘K’ means to match the element ordering of the inputs as closely as possible.

- **dtype**
  
  New in version 1.6. Overrides the dtype of the calculation and output arrays. Similar to `sig`.

- **subok**
  
  New in version 1.6. Defaults to true. If set to false, the output will always be a strict array, not a subtype.

- **sig**
  
  Either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to provide a specific signature for the 1-d loop to use in the underlying calculation. If the loop specified does not exist for the ufunc, then a TypeError is raised. Normally, a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This keyword argument lets you bypass that search and choose a particular loop. A list of available signatures is provided by the `types` attribute of the ufunc object.

- **extobj**
  
  a list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error callback function. Normally, these values are looked up in a thread-specific dictionary. Passing them here circumvents that look up and uses the low-level specification provided for the error mode. This may be
useful, for example, as an optimization for calculations requiring many ufunc calls on small arrays in a loop.

### 2.6.2 Attributes

There are some informational attributes that universal functions possess. None of the attributes can be set.

<table>
<thead>
<tr>
<th>Doc</th>
<th>A docstring for each ufunc. The first part of the docstring is dynamically generated from the number of outputs, the name, and the number of inputs. The second part of the docstring is provided at creation time and stored with the ufunc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the ufunc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>ufunc.nin</code></th>
<th>The number of inputs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ufunc.nout</code></td>
<td>The number of outputs.</td>
</tr>
<tr>
<td><code>ufunc.nargs</code></td>
<td>The number of arguments.</td>
</tr>
<tr>
<td><code>ufunc.ntypes</code></td>
<td>The number of types.</td>
</tr>
<tr>
<td><code>ufunc.types</code></td>
<td>Returns a list with types grouped input-&gt;output.</td>
</tr>
<tr>
<td><code>ufunc.identity</code></td>
<td>The identity value.</td>
</tr>
</tbody>
</table>

**ufunc.nin**

The number of inputs.

Data attribute containing the number of arguments the ufunc treats as input.

**Examples**

```python
>>> np.add.nin
2
>>> np.multiply.nin
2
>>> np.power.nin
2
>>> np.exp.nin
1
```

**ufunc.nout**

The number of outputs.

Data attribute containing the number of arguments the ufunc treats as output.

**Notes**

Since all ufuncs can take output arguments, this will always be (at least) 1.

**Examples**

```python
>>> np.add.nout
1
>>> np.multiply.nout
1
>>> np.power.nout
1
>>> np.exp.nout
1
```

**ufunc.nargs**

The number of arguments.
Data attribute containing the number of arguments the ufunc takes, including optional ones.

Notes
Typically this value will be one more than what you might expect because all ufuncs take the optional “out” argument.

Examples
```python
gnp.add.nargs
3
gnp.multiply.nargs
3
gnp.power.nargs
3
gnp.exp.nargs
2
```

ufunc.ntypes
The number of types.
The number of numerical NumPy types - of which there are 18 total - on which the ufunc can operate.

See Also:
numpy.ufunc.types

Examples
```python
gnp.add.ntypes
18
gnp.multiply.ntypes
18
gnp.power.ntypes
17
gnp.exp.ntypes
7
gnp.remainder.ntypes
14
```

ufunc.types
Returns a list with types grouped input->output.
Data attribute listing the data-type “Domain-Range” groupings the ufunc can deliver. The data-types are given using the character codes.

See Also:
numpy.ufunc.ntypes

Examples
```python
gnp.add.types
gnp.multiply.types
```
```python
>>> np.power.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
'O0->O']

>>> np.exp.types
['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']

>>> np.remainder.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'O0->O']
```

`ufunc.identity`

The identity value.

Data attribute containing the identity element for the ufunc, if it has one. If it does not, the attribute value is None.

**Examples**

```python
>>> np.add.identity
0

>>> np.multiply.identity
1

>>> np.power.identity
1

>>> print np.exp.identity
None
```

### 2.6.3 Methods

All ufuncs have four methods. However, these methods only make sense on ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a `ValueError`. The reduce-like methods all take an `axis` keyword and a `dtype` keyword, and the arrays must all have dimension $>= 1$. The `axis` keyword specifies the axis of the array over which the reduction will take place and may be negative, but must be an integer. The `dtype` keyword allows you to manage a very common problem that arises when naively using `{op}.reduce`. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The `dtype` keyword allows you to alter the data type over which the reduction takes place (and therefore the type of the output). Thus, you can ensure that the output is a data type with precision large enough to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no `dtype` is given for a reduction on the “add” or “multiply” operations, then if the input type is an integer (or Boolean) data-type and smaller than the size of the `int_` data type, it will be internally upcast to the `int_` (or `uint`) data-type.

Ufuncs also have a fifth method that allows in place operations to be performed using fancy indexing. No buffering is used on the dimensions where fancy indexing is used, so the fancy index can list an item more than once and the operation will be performed on the result of the previous operation for that item.

```python
ufunc.reduce(a[, axis, dtype, out, keepdims])  # Reduces a’s dimension by one, by applying ufunc along one axis.
ufunc.accumulate(array[, axis, dtype, out])   # Accumulate the result of applying the operator to all elements.
ufunc.reduceat(a, indices[, axis, dtype, out]) # Performs a (local) reduce with specified slices over a single axis.
ufunc.outer(A, B)                              # Apply the ufunc op to all pairs (a, b) with a in A and b in B.
ufunc.at(a, indices[, b])                      # Performs unbuffered in place operation on operand ‘a’ for elements specified by
```

**ufunc.reduce**  
(a, axis=0, dtype=None, out=None, keepdims=False)
Reduces a’s dimension by one, by applying ufunc along one axis.

Let a.shape = (N_0, ..., N_i, ..., N_{M-1}). Then ufunc.reduce(a, axis = i)[k_0, ..., k_{i-1}, k_{i+1}, ..., k_{M-1}] = the result of iterating j over range(N_i), cumulatively applying ufunc to each a[k_0, ..., k_{i-1}, j, k_{i+1}, ..., k_{M-1}]. For a one-dimensional array, reduce produces results equivalent to:

```python
r = op.identity # op = ufunc
for i in range(len(A)):
    r = op(r, A[i])
return r
```

For example, add.reduce() is equivalent to sum().

**Parameters**

- `a`: array_like
  - The array to act on.
- `axis`: None or int or tuple of ints, optional
  - Axis or axes along which a reduction is performed. The default (axis = 0) is perform a reduction over the first dimension of the input array. axis may be negative, in which case it counts from the last to the first axis. New in version 1.7.0. If this is None, a reduction is performed over all the axes. If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

For operations which are either not commutative or not associative, doing a reduction over multiple axes is not well-defined. The ufuncs do not currently raise an exception in this case, but will likely do so in the future.

- `dtype`: data-type code, optional
  - The type used to represent the intermediate results. Defaults to the data-type of the output array if this is provided, or the data-type of the input array if no output array is provided.

- `out`: ndarray, optional
  - A location into which the result is stored. If not provided, a freshly-allocated array is returned.

- `keepdims`: bool, optional
  - If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

**Returns**

- `r`: ndarray
  - The reduced array. If `out` was supplied, `r` is a reference to it.

**Examples**

```python
>>> np.multiply.reduce([2, 3, 5])
30
```

A multi-dimensional array example:

```python
>>> X = np.arange(8).reshape((2,2,2))
>>> X
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
```
>>> np.add.reduce(X, 0)
array([[ 4,  6],
      [ 8, 10]])

>>> np.add.reduce(X)  # confirm: default axis value is 0
array([[ 4,  6],
      [ 8, 10]])

>>> np.add.reduce(X, 1)
array([[ 2,  4],
      [10, 12]])

>>> np.add.reduce(X, 2)
array([[ 1,  5],
      [ 9, 13]])

ufunc.accumulate(array, axis=0, dtype=None, out=None)
Accumulate the result of applying the operator to all elements.

For a one-dimensional array, accumulate produces results equivalent to:

```python
r = np.empty(len(A))
t = op.identity  # op = the ufunc being applied to A's elements
for i in range(len(A)):
    t = op(t, A[i])
    r[i] = t
return r
```

For example, add.accumulate() is equivalent to np.cumsum().

For a multi-dimensional array, accumulate is applied along only one axis (axis zero by default; see Examples below) so repeated use is necessary if one wants to accumulate over multiple axes.

Parameters

- **array**: array_like
  - The array to act on.

- **axis**: int, optional
  - The axis along which to apply the accumulation; default is zero.

- **dtype**: data-type code, optional
  - The data-type used to represent the intermediate results. Defaults to the data-type of the output array if such is provided, or the the data-type of the input array if no output array is provided.

- **out**: ndarray, optional
  - A location into which the result is stored. If not provided a freshly-allocated array is returned.

Returns

- **r**: ndarray
  - The accumulated values. If `out` was supplied, `r` is a reference to `out`.

Examples

1-D array examples:

```python
>>> np.add.accumulate([2, 3, 5])
array([ 2,  5, 10])
```
```python
>>> np.multiply.accumulate([2, 3, 5])
array([ 2,  6, 30])

2-D array examples:

```python
>>> I = np.eye(2)
```
```python
>>> I
array([[ 1., 0.],
       [ 0., 1.]])
```

Accumulate along axis 0 (rows), down columns:

```python
>>> np.add.accumulate(I, 0)
array([[ 1., 0.],
       [ 1., 1.]])
```
```python
>>> np.add.accumulate(I)  # no axis specified == axis zero
array([[ 1., 0.],
       [ 1., 1.]])
```

Accumulate along axis 1 (columns), through rows:

```python
>>> np.add.accumulate(I, 1)
array([[ 1., 1.],
       [ 0., 1.]])
```

`ufunc.reduceat(a, indices, axis=0, dtype=None, out=None)`

Performs a (local) reduce with specified slices over a single axis.

For `i` in `range(len(indices))`, `reduceat` computes `ufunc.reduce(a[indices[i]:indices[i+1]])`, which becomes the i-th generalized “row” parallel to `axis` in the final result (i.e., in a 2-D array, for example, if `axis = 0`, it becomes the i-th row, but if `axis = 1`, it becomes the i-th column). There are two exceptions to this:

- when `i = len(indices) - 1` (so for the last index), `indices[i+1] = a.shape[axis]`.
- if `indices[i] >= indices[i + 1]`, the i-th generalized “row” is simply `a[indices[i]]`.

The shape of the output depends on the size of `indices`, and may be larger than `a` (this happens if `len(indices) > a.shape[axis]`).

**Parameters**

- `a`: array_like
  The array to act on.

- `indices`: array_like
  Paired indices, comma separated (not colon), specifying slices to reduce.

- `axis`: int, optional
  The axis along which to apply the reduceat.

- `dtype`: data-type code, optional
  The type used to represent the intermediate results. Defaults to the data type of the output array if this is provided, or the data type of the input array if no output array is provided.

- `out`: ndarray, optional
  A location into which the result is stored. If not provided a freshly-allocated array is returned.
Returns

\[ r : \text{ndarray} \]

The reduced values. If \textit{out} was supplied, \( r \) is a reference to \textit{out}.

Notes

A descriptive example:

If \( a \) is 1-D, the function \textit{ufunc.accumulate}(a) is the same as \textit{ufunc.reduceat}(a, indices)[::2] where \textit{indices} is range(len(array) - 1) with a zero placed in every other element: \textit{indices} = zeros(2 * len(a) - 1), indices[1::2] = range(1, len(a)).

Don’t be fooled by this attribute’s name: \textit{reduceat}(a) is not necessarily smaller than \( a \).

Examples

To take the running sum of four successive values:

```python
>>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
array([ 6, 10, 14, 18])
```

A 2-D example:

```python
>>> x = np.linspace(0, 15, 16).reshape(4,4)
>>> x
array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
```

# reduce such that the result has the following five rows:
# [row1 + row2 + row3]
# [row4]
# [row2]
# [row3]
# [row1 + row2 + row3 + row4]

```python
>>> np.add.reduceat(x, [0, 3, 1, 2, 0])
array([[12., 15., 18., 21.],
        [12., 13., 14., 15.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [24., 28., 32., 36.]])
```

# reduce such that result has the following two columns:
# [col0 * col2 * col3, col4]

```python
>>> np.multiply.reduceat(x, [0, 3, 1, 2, 0])
array([[ 0.,  3.],
        [120.,  7.],
        [ 720., 11.],
        [2184., 15.]])
```

\textit{ufunc.outer}(A, B)

Apply the \textit{ufunc} \textit{op} to all pairs (a, b) with a in \( A \) and b in \( B \).

Let \( M = A.\text{ndim}, N = B.\text{ndim} \). Then the result, \( C \), of \textit{op.outer}(A, B) is an array of dimension \( M + N \).
such that:

\[ C[i_0, \ldots, i_{M-1}, j_0, \ldots, j_{N-1}] = op(A[i_0, \ldots, i_{M-1}], B[j_0, \ldots, j_{N-1}]) \]

For \( A \) and \( B \) one-dimensional, this is equivalent to:

```python
r = empty(len(A), len(B))
for i in range(len(A)):
    for j in range(len(B)):
        r[i, j] = op(A[i], B[j])  # op = ufunc in question
```

**Parameters**

- **A**: array_like
  - First array
- **B**: array_like
  - Second array

**Returns**

- **r**: ndarray
  - Output array

**See Also:**

numpy.outer

**Examples**

```python
>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])
```

A multi-dimensional example:

```python
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1, 2, 3, 4]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
array([[[1, 2, 3, 4],
        [2, 4, 6, 8]],
        [3, 6, 9, 12]],
        [[4, 8, 12, 16]],
        [[5, 10, 15, 20]],
        [[6, 12, 18, 24]])
```

**ufunc.at (a, indices, b=None)**

Performs unbuffered in place operation on operand ‘a’ for elements specified by ‘indices’. For addition ufunc, this method is equivalent to \( a[indices] += b \), except that results are accumulated for elements that are indexed more than once. For example, \( a[[0,0]] += 1 \) will only increment the first element once because of buffering, whereas \( add.at(a, [0,0], 1) \) will increment the first element twice. New in version 1.8.0.
**Parameters**

- **a**: array_like
  The array to perform in place operation on.

- **indices**: array_like or tuple
  Array like index object or slice object for indexing into first operand. If first operand has multiple dimensions, indices can be a tuple of array like index objects or slice objects.

- **b**: array_like
  Second operand for ufuncs requiring two operands. Operand must be broadcastable over first operand after indexing or slicing.

**Examples**

Set items 0 and 1 to their negative values:

```python
gnp.array([1, 2, 3, 4])
gnp.negative.at(a, [0, 1])
gprint(a)
array([-1, -2, 3, 4])
```

Increment items 0 and 1, and increment item 2 twice:

```python
gnp.add.at(a, [0, 1, 2, 2], 1)
gprint(a)
array([2, 3, 5, 4])
```

Add items 0 and 1 in first array to second array, and store results in first array:

```python
gnp.array([1, 2, 3, 4])
gnp.add.at(a, [0, 1], b)
gprint(a)
array([2, 4, 3, 4])
```

**Warning**: A reduce-like operation on an array with a data-type that has a range “too small” to handle the result will silently wrap. One should use `dtype` to increase the size of the data-type over which reduction takes place.

### 2.7 Available ufuncs

There are currently more than 60 universal functions defined in `numpy` on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (e.g., `np.add(a, b)` is called internally when `a + b` is written and `a` or `b` is a ndarray). Nevertheless, you may still want to use the ufunc call in order to use the optional output argument(s) to place the output(s) in an object (or objects) of your choice.

Recall that each ufunc operates element-by-element. Therefore, each ufunc will be described as if acting on a set of scalar inputs to return a set of scalar outputs.

**Note**: The ufunc still returns its output(s) even if you use the optional output argument(s).
2.7.1 Math operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x1, x2[, out])</code></td>
<td>Add arguments element-wise.</td>
</tr>
<tr>
<td><code>subtract(x1, x2[, out])</code></td>
<td>Subtract arguments, element-wise.</td>
</tr>
<tr>
<td><code>multiply(x1, x2[, out])</code></td>
<td>Multiply arguments element-wise.</td>
</tr>
<tr>
<td><code>divide(x1, x2[, out])</code></td>
<td>Divide arguments element-wise.</td>
</tr>
<tr>
<td><code>logaddexp(x1, x2[, out])</code></td>
<td>Logarithm of the sum of exponentiations of the inputs.</td>
</tr>
<tr>
<td><code>logaddexp2(x1, x2[, out])</code></td>
<td>Logarithm of the sum of exponentiations of the inputs in base-2.</td>
</tr>
<tr>
<td><code>true_divide(x1, x2[, out])</code></td>
<td>Returns a true division of the inputs, element-wise.</td>
</tr>
<tr>
<td><code>floor_divide(x1, x2[, out])</code></td>
<td>Return the largest integer smaller or equal to the division of the inputs.</td>
</tr>
<tr>
<td><code>negative(x[, out])</code></td>
<td>Numerical negative, element-wise.</td>
</tr>
<tr>
<td><code>power(x1, x2[, out])</code></td>
<td>First array elements raised to powers from second array, element-wise.</td>
</tr>
<tr>
<td><code>mod(x1, x2[, out])</code></td>
<td>Return element-wise remainder of division.</td>
</tr>
<tr>
<td><code>fmod(x1, x2[, out])</code></td>
<td>Return the element-wise remainder of division.</td>
</tr>
<tr>
<td><code>absolute(x[, out])</code></td>
<td>Calculate the absolute value element-wise.</td>
</tr>
<tr>
<td><code>rint(x[, out])</code></td>
<td>Round elements of the array to the nearest integer.</td>
</tr>
<tr>
<td><code>sign(x[, out])</code></td>
<td>Returns an element-wise indication of the sign of a number.</td>
</tr>
<tr>
<td><code>conj(x[, out])</code></td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td><code>exp(x[, out])</code></td>
<td>Calculate the exponential of all elements in the input array.</td>
</tr>
<tr>
<td><code>exp2(x[, out])</code></td>
<td>Calculate $2^p$ for all $p$ in the input array.</td>
</tr>
<tr>
<td><code>log(x[, out])</code></td>
<td>Natural logarithm, element-wise.</td>
</tr>
<tr>
<td><code>log2(x[, out])</code></td>
<td>Base-2 logarithm of $x$.</td>
</tr>
<tr>
<td><code>log10(x[, out])</code></td>
<td>Return the base 10 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td><code>expm1(x[, out])</code></td>
<td>Calculate $\exp(x) - 1$ for all elements in the array.</td>
</tr>
<tr>
<td><code>log1p(x[, out])</code></td>
<td>Return the natural logarithm of one plus the input array, element-wise.</td>
</tr>
<tr>
<td><code>sqrt(x[, out])</code></td>
<td>Return the positive square-root of an array, element-wise.</td>
</tr>
<tr>
<td><code>square(x[, out])</code></td>
<td>Return the element-wise square of the input.</td>
</tr>
<tr>
<td><code>reciprocal(x[, out])</code></td>
<td>Return the reciprocal of the argument, element-wise.</td>
</tr>
<tr>
<td><code>ones_like(a[, dtype, order, subok])</code></td>
<td>Return an array of ones with the same shape and type as a given array.</td>
</tr>
</tbody>
</table>

**Tip:** The optional output arguments can be used to help you save memory for large calculations. If your arrays are large, complicated expressions can take longer than absolutely necessary due to the creation and (later) destruction of temporary calculation spaces. For example, the expression $G = a * b + c$ is equivalent to $t1 = A * B; G = T1 + C; del t1$. It will be more quickly executed as $G = A * B; add(G, C, G)$ which is the same as $G = A * B; G += C$.

2.7.2 Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is $180^\circ / \pi$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sin(x[, out])</code></td>
<td>Trigonometric sine, element-wise.</td>
</tr>
<tr>
<td><code>cos(x[, out])</code></td>
<td>Cosine element-wise.</td>
</tr>
<tr>
<td><code>tan(x[, out])</code></td>
<td>Compute tangent element-wise.</td>
</tr>
<tr>
<td><code>arcsin(x[, out])</code></td>
<td>Inverse sine, element-wise.</td>
</tr>
<tr>
<td><code>arccos(x[, out])</code></td>
<td>Trigonometric inverse cosine, element-wise.</td>
</tr>
<tr>
<td><code>arctan(x[, out])</code></td>
<td>Trigonometric inverse tangent, element-wise.</td>
</tr>
<tr>
<td><code>arctan2(x1, x2[, out])</code></td>
<td>Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.</td>
</tr>
</tbody>
</table>
Table 2.6 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hypot(x1, x2[, out])</code></td>
<td>Given the “legs” of a right triangle, return its hypotenuse.</td>
</tr>
<tr>
<td><code>sinh(x[, out])</code></td>
<td>Hyperbolic sine, element-wise.</td>
</tr>
<tr>
<td><code>cosh(x[, out])</code></td>
<td>Hyperbolic cosine, element-wise.</td>
</tr>
<tr>
<td><code>tanh(x[, out])</code></td>
<td>Compute hyperbolic tangent element-wise.</td>
</tr>
<tr>
<td><code>arcsinh(x[, out])</code></td>
<td>Inverse hyperbolic sine element-wise.</td>
</tr>
<tr>
<td><code>arccosh(x[, out])</code></td>
<td>Inverse hyperbolic cosine, element-wise.</td>
</tr>
<tr>
<td><code>arctanh(x[, out])</code></td>
<td>Inverse hyperbolic tangent element-wise.</td>
</tr>
<tr>
<td><code>deg2rad(x[, out])</code></td>
<td>Convert angles from degrees to radians.</td>
</tr>
<tr>
<td><code>rad2deg(x[, out])</code></td>
<td>Convert angles from radians to degrees.</td>
</tr>
</tbody>
</table>

**2.7.3 Bit-twiddling functions**

These function all require integer arguments and they manipulate the bit-pattern of those arguments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bitwise_and(x1, x2[, out])</code></td>
<td>Compute the bit-wise AND of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_or(x1, x2[, out])</code></td>
<td>Compute the bit-wise OR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_xor(x1, x2[, out])</code></td>
<td>Compute the bit-wise XOR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>invert(x[, out])</code></td>
<td>Compute bit-wise inversion, or bit-wise NOT, element-wise.</td>
</tr>
<tr>
<td><code>left_shift(x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the left.</td>
</tr>
<tr>
<td><code>right_shift(x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the right.</td>
</tr>
</tbody>
</table>

**2.7.4 Comparison functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>greater(x1, x2[, out])</code></td>
<td>Return the truth value of (x1 &gt; x2) element-wise.</td>
</tr>
<tr>
<td><code>greater_equal(x1, x2[, out])</code></td>
<td>Return the truth value of (x1 &gt;= x2) element-wise.</td>
</tr>
<tr>
<td><code>less(x1, x2[, out])</code></td>
<td>Return the truth value of (x1 &lt; x2) element-wise.</td>
</tr>
<tr>
<td><code>less_equal(x1, x2[, out])</code></td>
<td>Return the truth value of (x1 &lt;= x2) element-wise.</td>
</tr>
<tr>
<td><code>not_equal(x1, x2[, out])</code></td>
<td>Return (x1 != x2) element-wise.</td>
</tr>
<tr>
<td><code>equal(x1, x2[, out])</code></td>
<td>Return (x1 == x2) element-wise.</td>
</tr>
</tbody>
</table>

**Warning:** Do not use the Python keywords `and` and `or` to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators `&` and `|` instead.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>logical_and(x1, x2[, out])</code></td>
<td>Compute the truth value of x1 AND x2 element-wise.</td>
</tr>
<tr>
<td><code>logical_or(x1, x2[, out])</code></td>
<td>Compute the truth value of x1 OR x2 element-wise.</td>
</tr>
<tr>
<td><code>logical_xor(x1, x2[, out])</code></td>
<td>Compute the truth value of x1 XOR x2, element-wise.</td>
</tr>
<tr>
<td><code>logical_not(x[, out])</code></td>
<td>Compute the truth value of NOT x element-wise.</td>
</tr>
</tbody>
</table>

**Warning:** The bit-wise operators `&` and `|` are the proper way to perform element-by-element array comparisons. Be sure you understand the operator precedence: `(a > 2) & (a < 5)` is the proper syntax because `a > 2 & a < 5` will result in an error due to the fact that `a > 2` is evaluated first.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maximum(x1, x2[, out])</code></td>
<td>Element-wise maximum of array elements.</td>
</tr>
</tbody>
</table>
Tip: The Python function `max()` will find the maximum over a one-dimensional array, but it will do so using a slower sequence interface. The reduce method of the maximum ufunc is much faster. Also, the `max()` method will not give answers you might expect for arrays with greater than one dimension. The reduce method of minimum also allows you to compute a total minimum over an array.

\[
\text{minimum}(x1, x2[, out]) \quad \text{Element-wise minimum of array elements.}
\]

Warning: the behavior of `maximum(a, b)` is different than that of `max(a, b)`. As a ufunc, `maximum(a, b)` performs an element-by-element comparison of `a` and `b` and chooses each element of the result according to which element in the two arrays is larger. In contrast, `max(a, b)` treats the objects `a` and `b` as a whole, looks at the (total) truth value of `a > b` and uses it to return either `a` or `b` (as a whole). A similar difference exists between `minimum(a, b)` and `min(a, b)`.

\[
fmax(x1, x2[, out]) \quad \text{Element-wise maximum of array elements.}
\]

\[
fmin(x1, x2[, out]) \quad \text{Element-wise minimum of array elements.}
\]

### 2.7.5 Floating functions

Recall that all of these functions work element-by-element over an array, returning an array output. The description details only a single operation.

- `isreal(x)` Returns a bool array, where True if input element is real.
- `iscomplex(x)` Returns a bool array, where True if input element is complex.
- `isfinite(x[, out])` Test element-wise for finiteness (not infinity or not Not a Number).
- `isinf(x[, out])` Test element-wise for positive or negative infinity.
- `isnan(x[, out])` Test element-wise for NaN and return result as a boolean array.
- `signbit(x[, out])` Returns element-wise True where signbit is set (less than zero).
- `copysign(x1, x2[, out])` Change the sign of `x1` to that of `x2`, element-wise.
- `nextafter(x1, x2[, out])` Return the next floating-point value after `x1` towards `x2`, element-wise.
- `ldexp(x1, x2[, out])` Returns `x1 * 2**x2`, element-wise.
- `frexp(x[, out1, out2])` Decompose the elements of `x` into mantissa and twos exponent.
- `fmod(x1, x2[, out])` Return the element-wise remainder of division.
- `floor(x[, out])` Return the floor of the input, element-wise.
- `ceil(x[, out])` Return the ceiling of the input, element-wise.
- `trunc(x[, out])` Return the truncated value of the input, element-wise.

### 2.7. Available ufuncs
ROUTINES

In this chapter routine docstrings are presented, grouped by functionality. Many docstrings contain example code, which demonstrates basic usage of the routine. The examples assume that NumPy is imported with:

```python
>>> import numpy as np
```

A convenient way to execute examples is the `%doctest_mode` mode of IPython, which allows for pasting of multi-line examples and preserves indentation.

### 3.1 Array creation routines

**See Also:**

*Array creation*

#### 3.1.1 Ones and zeros

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty</code></td>
<td>Return a new array of given shape and type, without initializing entries.</td>
</tr>
<tr>
<td><code>empty_like</code></td>
<td>Return a new array with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>eye</code></td>
<td>Return a 2-D array with ones on the diagonal and zeros elsewhere.</td>
</tr>
<tr>
<td><code>identity</code></td>
<td>Return the identity array.</td>
</tr>
<tr>
<td><code>ones</code></td>
<td>Return an array of ones with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>ones_like</code></td>
<td>Return an array of ones with the same shape and type as a given array.</td>
</tr>
<tr>
<td><code>zeros</code></td>
<td>Return a new array of given shape and type, filled with zeros.</td>
</tr>
<tr>
<td><code>zeros_like</code></td>
<td>Return an array of zeros with the same shape and type as a given array.</td>
</tr>
</tbody>
</table>

```python
def numpy.empty(shape, dtype=float, order='C')
    Return a new array of given shape and type, without initializing entries.
```

**Parameters**

- **shape**: int or tuple of int
  - Shape of the empty array
- **dtype**: data-type, optional
  - Desired output data-type.
- **order**: {‘C’, ‘F’}, optional

```python
def numpy.eye(N, M=None, k=0, dtype=float)
    Return a 2-D array with ones on the diagonal and zeros elsewhere.
```

```python
def numpy.identity(n, dtype=float)
    Return the identity array.
```

```python
def numpy.ones(shape, dtype=float, order='C')
    Return an array of ones with the same shape and type as a given array.
```

```python
def numpy.zeros(shape, dtype=float, order='C')
    Return a new array of given shape and type, filled with zeros.
```

```python
def numpy.ones_like(a, dtype=None, order=None, subok=True)
    Return an array of ones with the same shape and type as a given array.
```

```python
def numpy.zeros_like(a, dtype=None, order=None, subok=True)
    Return an array of zeros with the same shape and type as a given array.
```
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See Also:
empty_like, zeros, ones

Notes
eempty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples
>>> np.empty([2, 2])
array([[-9.74499359e+001, 6.69583040e-309],
      [ 2.13182611e-314, 3.06959433e-309]]) #random

>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
      [ 496041986, 19249760]]) #random

numpy.empty_like(a, dtype=None, order='K', subok=True)
Return a new array with the same shape and type as a given array.

Parameters
    a : array_like
        The shape and data-type of a define these same attributes of the returned array.

dtype : data-type, optional
        New in version 1.6.0. Overrides the data type of the result.

order : {'C', 'F', 'A', or 'K'}, optional
        New in version 1.6.0. Overrides the memory layout of the result. ‘C’ means C-order, ‘F’
        means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means
        match the layout of a as closely as possible.

subok : bool, optional.
        If True, then the newly created array will use the sub-class type of ‘a’, otherwise it will
        be a base-class array. Defaults to True.

Returns
    out : ndarray
        Array of uninitialized (arbitrary) data with the same shape and type as a.

See Also:
ones_like
Return an array of ones with shape and type of input.

zeros_like
Return an array of zeros with shape and type of input.

empty
Return a new uninitialized array.

ones
Return a new array setting values to one.
zeros
Return a new array setting values to zero.

Notes
This function does not initialize the returned array; to do that use zeros_like or ones_like instead. It may be marginally faster than the functions that do set the array values.

Examples
>>> a = ([1, 2, 3], [4, 5, 6])
# a is array-like
>>> np.empty_like(a)
array([[[-1073741821, -1073741821, 3],
        [0, 0, -1073741821]]])

numpy.eye(N, M=None, k=0, dtype=’float’)
Return a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters
N : int
   Number of rows in the output.
M : int, optional
   Number of columns in the output. If None, defaults to N.
k : int, optional
   Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.
dtype : data-type, optional
   Data-type of the returned array.

Returns
I : ndarray of shape (N,M)
   An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

See Also:
identity
(almost) equivalent function
diag
diagonal 2-D array from a 1-D array specified by the user.

Examples
>>> np.eye(2, dtype=int)
array([[1, 0],
       [0, 1]])

   # references:
numpy.identity(n, dtype=None)
Return the identity array.

The identity array is a square array with ones on the main diagonal.

Parameters
- n : int
  Number of rows (and columns) in \( n \times n \) output.
- dtype : data-type, optional
  Data-type of the output. Defaults to float.

Returns
- out : ndarray
  \( n \times n \) array with its main diagonal set to one, and all other elements 0.

Examples
>>> np.identity(3)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])

numpy.ones(shape, dtype=None, order='C')
Return a new array of given shape and type, filled with ones.

Parameters
- shape : int or sequence of ints
  Shape of the new array, e.g., (2, 3) or 2.
- dtype : data-type, optional
  The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.
- order : {'C', 'F'}, optional
  Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns
- out : ndarray
  Array of ones with the given shape, dtype, and order.

See Also:
zeros, ones_like

Examples
>>> np.ones(5)
array([ 1., 1., 1., 1., 1.])

>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])

>>> np.ones((2, 1))
array([[ 1.],
       [ 1.]])
```python
>>> s = (2,2)
>>> np.ones(s)
array([[ 1.,  1.],
       [ 1.,  1.]])
```

**numpy.ones_like** *(a, dtype=None, order='K', subok=True)*

Return an array of ones with the same shape and type as a given array.

**Parameters**

- `a` : array_like
  
The shape and data-type of `a` define these same attributes of the returned array.

- `dtype` : data-type, optional
  
  New in version 1.6.0. Overrides the data type of the result.

- `order` : {'C', 'F', 'A', or 'K'}, optional
  
  New in version 1.6.0. Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if `a` is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of `a` as closely as possible.

- `subok` : bool, optional.
  
  If True, then the newly created array will use the sub-class type of ‘a’, otherwise it will be a base-class array. Defaults to True.

**Returns**

- `out` : ndarray
  
  Array of ones with the same shape and type as `a`.

**See Also:**

- `zeros_like`
  
  Return an array of zeros with shape and type of input.

- `empty_like`
  
  Return an empty array with shape and type of input.

- `zeros`
  
  Return a new array setting values to zero.

- `ones`
  
  Return a new array setting values to one.

- `empty`
  
  Return a new uninitialized array.

**Examples**

```python
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.ones_like(x)
array([[1, 1, 1],
       [1, 1, 1]])
```
```python
>>> y = np.arange(3, dtype=np.float)
>>> y
array([ 0., 1., 2.])
>>> np.ones_like(y)
array([ 1., 1., 1.])
```

`numpy.zeros(shape, dtype=float, order=’C’)`

Return a new array of given shape and type, filled with zeros.

**Parameters**

`shape` : int or sequence of ints
Shape of the new array, e.g., (2, 3) or 2.

`dtype` : data-type, optional
The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

`order` : {‘C’, ‘F’}, optional
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

`out` : ndarray
Array of zeros with the given shape, dtype, and order.

**See Also:**

`zeros_like`  
Return an array of zeros with shape and type of input.

`ones_like`  
Return an array of ones with shape and type of input.

`empty_like`  
Return an empty array with shape and type of input.

`ones`  
Return a new array setting values to one.

`empty`  
Return a new uninitialized array.

**Examples**

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])

>>> s = (2,2)
>>> np.zeros(s)
array([[ 0., 0.],
       [ 0., 0.]])
```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
dtype=[('x', '<i4'), ('y', '<i4')])

numpy.zeros_like(a, dtype=None, order='K', subok=True)
Return an array of zeros with the same shape and type as a given array.

Parameters
   a : array_like
       The shape and data-type of a define these same attributes of the returned array.

dtype : data-type, optional
       New in version 1.6.0. Overrides the data type of the result.

order : {'C', 'F', 'A', or 'K'}, optional
       New in version 1.6.0. Overrides the memory layout of the result. ‘C’ means C-order,
       ‘F’ means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means
       match the layout of a as closely as possible.

subok : bool, optional.
       If True, then the newly created array will use the sub-class type of ‘a’, otherwise it will
       be a base-class array. Defaults to True.

Returns
   out : ndarray
       Array of zeros with the same shape and type as a.

See Also:
ones_like
Return an array of ones with shape and type of input.

empty_like
Return an empty array with shape and type of input.

zeros
Return a new array setting values to zero.

ones
Return a new array setting values to one.

empty
Return a new uninitialized array.

Examples
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.zeros_like(x)
array([[0, 0, 0],
       [0, 0, 0]])
```python
>>> y = np.arange(3, dtype=np.float)
>>> y
array([ 0., 1., 2.])
>>> np.zeros_like(y)
array([ 0., 0., 0.])
```

### 3.1.2 From existing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array(object[, dtype, copy, order, subok, ndmin])</code></td>
<td>Create an array.</td>
</tr>
<tr>
<td><code>asarray(a[, dtype, order])</code></td>
<td>Convert the input to an array.</td>
</tr>
<tr>
<td><code>asanyarray(a[, dtype, order])</code></td>
<td>Convert the input to an array, but pass ndarray subclasses through.</td>
</tr>
<tr>
<td><code>ascontiguousarray(a[, dtype])</code></td>
<td>Return a contiguous array in memory (C order).</td>
</tr>
<tr>
<td><code>asmatrix(data[, dtype])</code></td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td><code>copy(a[, order])</code></td>
<td>Return an array copy of the given object.</td>
</tr>
<tr>
<td><code>frombuffer(buffer[, dtype, count, offset])</code></td>
<td>Interpret a buffer as a 1-dimensional array.</td>
</tr>
<tr>
<td><code>fromfile(file[, dtype, count, sep])</code></td>
<td>Construct an array from data in a text or binary file.</td>
</tr>
<tr>
<td><code>fromfunction(function, shape, **kwargs)</code></td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td><code>fromiter(iterable, dtype[, count])</code></td>
<td>Create a new 1-dimensional array from an iterable object.</td>
</tr>
<tr>
<td><code>fromstring(string[, dtype, count, sep])</code></td>
<td>A new 1-D array initialized from raw binary or text data in a string.</td>
</tr>
<tr>
<td><code>loadtxt(fname[, dtype, comments, delimiter, ...])</code></td>
<td>Load data from a text file.</td>
</tr>
</tbody>
</table>

```python
def numpy.array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0):
    Create an array.
```

**Parameters**

- `object`: array_like
  An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

- `dtype`: data-type, optional
  The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

- `copy`: bool, optional
  If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (dtype, order, etc.).

  Specify the order of the array. If order is ‘C’ (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is ‘F’, then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is ‘A’, then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

- `subok`: bool, optional
  If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

- `ndmin`: int, optional
  ...
Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

Returns

out : ndarray

An array object satisfying the specified requirements.

See Also:

empty, empty_like, zeros, zeros_like, ones, ones_like, fill

Examples

```python
gnp.array([1, 2, 3])
a(1, 2, 3)
```

Upcasting:

```python
gnp.array([1, 2, 3.0])
a(1, 2, 3.0)
```

More than one dimension:

```python
gnp.array([[1, 2], [3, 4]])
a([[1, 2], [3, 4]])
```

Minimum dimensions 2:

```python
gnp.array([1, 2, 3], ndmin=2)
a([[1, 2, 3]])
```

Type provided:

```python
gnp.array([1, 2, 3], dtype=complex)
a([1.0, 2.0, 3.0])
```

Data-type consisting of more than one element:

```python
x = gnp.array([(1, 2), (3, 4)], dtype=[('a', '<i4'), ('b', '<i4')])
x['a']
a([1, 2])
```

Creating an array from sub-classes:

```python
gnp.array(np.mat('1 2; 3 4'))
a([[1, 2], [3, 4]])
```

```python
gnp.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2], [3, 4]])
```

`numpy.asarray(a, dtype=None, order=None)`

Convert the input to an array.

Parameters

- a : array_like
  
  Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.
**dtype**: data-type, optional

By default, the data-type is inferred from the input data.

**order**: {'C', 'F'}, optional

Whether to use row-major ('C') or column-major ('F' for FORTRAN) memory representation. Defaults to 'C'.

**Returns**

**out**: ndarray

Array interpretation of a. No copy is performed if the input is already an ndarray. If a is a subclass of ndarray, a base class ndarray is returned.

**See Also:**

*asanyarray*

Similar function which passes through subclasses.

*ascontiguousarray*

Convert input to a contiguous array.

*asfarray*

Convert input to a floating point ndarray.

*asfortranarray*

Convert input to an ndarray with column-major memory order.

*asarray_chkfinite*

Similar function which checks input for NaNs and Infs.

*fromiter*

Create an array from an iterator.

*fromfunction*

Construct an array by executing a function on grid positions.

**Examples**

Convert a list into an array:

```python
>>> a = [1, 2]
>>> np.asarray(a)
array([1, 2])
```

Existing arrays are not copied:

```python
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

If **dtype** is set, array is copied only if dtype does not match:

```python
>>> a = np.array([1, 2], dtype=np.float32)
>>> np.asarray(a, dtype=np.float32) is a
True
>>> np.asarray(a, dtype=np.float64) is a
False
```

Contrary to **asanyarray**, ndarray subclasses are not passed through:
```python
>>> issubclass(np.matrix, np.ndarray)
True
>>> a = np.matrix([[1, 2]])
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
```

`numpy.asanyarray(a, dtype=None, order=None)`  
Convert the input to an ndarray, but pass ndarray subclasses through.

**Parameters**

- `a`: array_like
  - Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists, and ndarrays.
- `dtype`: data-type, optional
  - By default, the data-type is inferred from the input data.
- `order`: {'C', 'F'}, optional
  - Whether to use row-major ('C') or column-major ('F') memory representation. Defaults to 'C'.

**Returns**

- `out`: ndarray or an ndarray subclass
  - Array interpretation of `a`. If `a` is an ndarray or a subclass of ndarray, it is returned as-is and no copy is performed.

**See Also:**

- `asarray`
  - Similar function which always returns ndarrays.
- `ascontiguousarray`
  - Convert input to a contiguous array.
- `asfarray`
  - Convert input to a floating point ndarray.
- `asfortranarray`
  - Convert input to an ndarray with column-major memory order.
- `asarray_chkfinite`
  - Similar function which checks input for NaNs and Infs.
- `fromiter`
  - Create an array from an iterator.
- `fromfunction`
  - Construct an array by executing a function on grid positions.

**Examples**

Convert a list into an array:

```python
>>> a = [1, 2]
>>> np.asanyarray(a)
array([1, 2])
```
Instances of `ndarray` subclasses are passed through as-is:

```python
>>> a = np.matrix([[1, 2]])
>>> np.asanyarray(a) is a
True
```

`numpy.ascontiguousarray(a, dtype=None)`

Return a contiguous array in memory (C order).

**Parameters**
- `a`: array_like
  - Input array.
- `dtype`: str or dtype object, optional
  - Data-type of returned array.

**Returns**
- `out`: ndarray
  - Contiguous array of same shape and content as `a`, with type `dtype` if specified.

**See Also:**
- `asfortranarray`  
  Convert input to an ndarray with column-major memory order.
- `require`  
  Return an ndarray that satisfies requirements.
- `ndarray.flags`  
  Information about the memory layout of the array.

**Examples**

```python
>>> x = np.arange(6).reshape(2,3)
>>> np.ascontiguousarray(x, dtype=np.float32)
array([[ 0., 1., 2.],
       [ 3., 4., 5.]], dtype=float32)
>>> x.flags['C_CONTIGUOUS']
True
```

`numpy.asmatrix(data, dtype=None)`

Interpret the input as a matrix.

Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

**Parameters**
- `data`: array_like
  - Input data.

**Returns**
- `mat`: matrix
  - `data` interpreted as a matrix.

**Examples**

```python
>>> x = np.array([[1, 2], [3, 4]])
```
>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
         [3, 4]])

numpy.copy(a, order='K')
   Return an array copy of the given object.

   Parameters
   a : array_like
       Input data.

   order : {'C', 'F', 'A', 'K'}, optional
       Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’
       means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a
       as closely as possible. (Note that this function and :meth:ndarray.copy are very similar,
       but have different default values for their order= arguments.)

   Returns
   arr : ndarray
       Array interpretation of a.

   Notes
   This is equivalent to

>>> np.array(a, copy=True)

Examples
Create an array x, with a reference y and a copy z:

>>> x = np.array([1, 2, 3])

>>> y = x

>>> z = np.copy(x)

Note that, when we modify x, y changes, but not z:

>>> x[0] = 10

>>> x[0] == y[0]
True

>>> x[0] == z[0]
False

numpy.frombuffer(buffer, dtype=float, count=-1, offset=0)
   Interpret a buffer as a 1-dimensional array.

   Parameters
   buffer : buffer_like
       An object that exposes the buffer interface.

   dtype : data-type, optional
       Data-type of the returned array; default: float.

   count : int, optional

3.1. Array creation routines
Number of items to read. –1 means all data in the buffer.

offset : int, optional

Start reading the buffer from this offset; default: 0.

Notes

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```python
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

Examples

```python
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
dtype='|S1')
```

numpy.fromfile (file, dtype=float, count=-1, sep='')
Construct an array from data in a text or binary file.

A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the tofile method can be read using this function.

Parameters

- file : file or str
  Open file object or filename.

- dtype : data-type
  Data type of the returned array. For binary files, it is used to determine the size and byte-order of the items in the file.

- count : int
  Number of items to read. –1 means all items (i.e., the complete file).

- sep : str
  Separator between items if file is a text file. Empty ("") separator means the file should be treated as binary. Spaces (" ") in the separator match zero or more whitespace characters. A separator consisting only of spaces must match at least one whitespace.

See Also:
load, save, ndarray.tofile

loadtxt
More flexible way of loading data from a text file.

Notes

Do not rely on the combination of tofile and fromfile for data storage, as the binary files generated are are not platform independent. In particular, no byte-order or data-type information is saved. Data can be stored in the platform independent .npy format using save and load instead.
Examples

Construct an ndarray:

```python
>>> dt = np.dtype([('time', [('min', int), ('sec', int)]),
                 ('temp', float)])
>>> x = np.zeros((1,), dtype=dt)
>>> x['time']['min'] = 10; x['temp'] = 98.25
>>> x
array([(10, 0), 98.25]),
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

Save the raw data to disk:

```python
>>> import os
>>> fname = os.tmpnam()
>>> x.tofile(fname)
```

Read the raw data from disk:

```python
>>> np.fromfile(fname, dtype=dt)
array([(10, 0), 98.25]),
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

The recommended way to store and load data:

```python
>>> np.save(fname, x)
>>> np.load(fname + '.npy')
array([(10, 0), 98.25]),
      dtype=[('time', [('min', '<i4'), ('sec', '<i4')]), ('temp', '<f8')])
```

**numpy.fromfunction** *(function, shape, **kwargs)*

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value `fn(x, y, z)` at coordinate `(x, y, z)`.

**Parameters**

- **function**: callable
  - The function is called with N parameters, where N is the rank of `shape`. Each parameter represents the coordinates of the array varying along a specific axis. For example, if `shape` were `(2, 2)`, then the parameters in turn be `(0, 0), (0, 1), (1, 0), (1, 1)`.

- **shape**: (N,) tuple of ints
  - Shape of the output array, which also determines the shape of the coordinate arrays passed to `function`.

- **dtype**: data-type, optional
  - Data-type of the coordinate arrays passed to `function`. By default, `dtype` is float.

**Returns**

- **fromfunction**: any
  - The result of the call to `function` is passed back directly. Therefore the shape of `fromfunction` is completely determined by `function`. If `function` returns a scalar value, the shape of `fromfunction` would match the `shape` parameter.

**See Also**:

- `indices`, `meshgrid`
Notes

Keywords other than `dtype` are passed to `function`.

Examples

```python
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)
```

```python
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

`numpy.fromiter(iterable, dtype, count=-1)`

Create a new 1-dimensional array from an iterable object.

Parameters

- **iterable**: iterable object
  
  An iterable object providing data for the array.

- **dtype**: data-type
  
  The data-type of the returned array.

- **count**: int, optional
  
  The number of items to read from `iterable`. The default is -1, which means all data is read.

Returns

- **out**: ndarray
  
  The output array.

Notes

Specify `count` to improve performance. It allows `fromiter` to pre-allocate the output array, instead of resizing it on demand.

Examples

```python
>>> iterable = (x*x for x in range(5))
```

```python
>>> np.fromiter(iterable, np.float)
array([ 0.,  1.,  4.,  9., 16.])
```

`numpy.fromstring(string, dtype=float, count=-1, sep='')`

A new 1-D array initialized from raw binary or text data in a string.

Parameters

- **string**: str
  
  A string containing the data.

- **dtype**: data-type, optional
  
  The data type of the array; default: float. For binary input data, the data must be in exactly this format.

- **count**: int, optional
Read this number of `dtype` elements from the data. If this is negative (the default), the count will be determined from the length of the data.

```
sep : str, optional
If not provided or, equivalently, the empty string, the data will be interpreted as binary data; otherwise, as ASCII text with decimal numbers. Also in this latter case, this argument is interpreted as the string separating numbers in the data; extra whitespace between elements is also ignored.
```

Returns

```
arr : ndarray
The constructed array.
```

Raises

```
ValueError
If the string is not the correct size to satisfy the requested `dtype` and `count`.
```

See Also:

`frombuffer`, `fromfile`, `fromiter`

Examples

```
>>> np.fromstring(\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
```
```
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
```
```
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
```
```
>>> np.fromstring(\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

`numpy.loadtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0)`

```
Load data from a text file.
```

```
Each row in the text file must have the same number of values.
```

Parameters

```
fname : file or str
File, filename, or generator to read. If the filename extension is .gz or .bz2, the file is first decompressed. Note that generators should return byte strings for Python 3k.
```
```
dtype : data-type, optional
Data-type of the resulting array; default: float. If this is a record data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.
```
```
comments : str, optional
The character used to indicate the start of a comment; default: ‘#’.
```
```
delimiter : str, optional
The string used to separate values. By default, this is any whitespace.
```
```
converters : dict, optional
```
A dictionary mapping column number to a function that will convert that column to a float. E.g., if column 0 is a date string: converters = (0: datestr2num). Converters can also be used to provide a default value for missing data (but see also genfromtxt): converters = {3: lambda s: float(s.strip() or 0)}. Default: None.

skiprows : int, optional

   Skip the first skiprows lines; default: 0.

usecols : sequence, optional

   Which columns to read, with 0 being the first. For example, usecols = (1,4,5) will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.

unpack : bool, optional

   If True, the returned array is transposed, so that arguments may be unpacked using x, y, z = loadtxt(...). When used with a record data-type, arrays are returned for each field. Default is False.

ndmin : int, optional

   The returned array will have at least ndmin dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2. New in version 1.6.0.

Returns

   out : ndarray

   Data read from the text file.

See Also:

load, fromstring, fromregex

genfromtxt

   Load data with missing values handled as specified.

scipy.io.loadmat

   reads MATLAB data files

Notes

   This function aims to be a fast reader for simply formatted files. The genfromtxt function provides more sophisticated handling of, e.g., lines with missing values.

Examples

>>> from StringIO import StringIO  # StringIO behaves like a file object
>>> c = StringIO("0 1
2 3")
>>> np.loadtxt(c)
array([[ 0., 1.],
       [ 2., 3.]])

>>> d = StringIO("M 21 72
F 35 58")
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
... 'formats': ('S1', 'i4', 'f4')})
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])
NumPy Reference, Release 1.8.1

```python
>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter='\'', usecols=(0, 2), unpack=True)
```

```
array([ 1., 3.])
array([ 2., 4.])
```

3.1.3 Creating record arrays (numpy.rec)

---

**Note:** numpy.rec is the preferred alias for numpy.core.records.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>core.records.array</code></td>
<td>Construct a record array from a wide-variety of objects.</td>
</tr>
<tr>
<td><code>core.records.fromarrays</code></td>
<td>create a record array from a (flat) list of arrays</td>
</tr>
<tr>
<td><code>core.records.fromrecords</code></td>
<td>create a recarray from a list of records in text form</td>
</tr>
<tr>
<td><code>core.records.fromstring</code></td>
<td>create a (read-only) record array from binary data contained in</td>
</tr>
<tr>
<td><code>core.records.fromfile</code></td>
<td>Create an array from binary file data</td>
</tr>
</tbody>
</table>

**numpy.core.records.array**

```
(\obj, dtype=None, shape=None, ...) | Construct a record array from a wide-variety of objects.
```

**numpy.core.records.fromarrays**

```
arrayList[, dtype, ...]) | create a record array from a (flat) list of arrays |
```

**numpy.core.records.fromrecords**

```
recList[, dtype, ...]) | create a recarray from a list of records in text form |
```

```
... \r=fromrecords([[2.3,'abc']]*100000)
```

```
... \r.col1
```

```
array([2.0, 2.0, 2.0])
```

---

3.1. Array creation routines 469

```python
>>> r=fromrecords([[456,'dbe',1.2],[2,'de',1.3]],... names='col1,col2,col3')
```

```
print r[0]
```

```
(456, 'dbe', 1.2)
```

```
>>> r.col1
```
NumPy Reference, Release 1.8.1

array([456,  2])
>>> r.col2
chararray(['dbe', 'de'],
dtype='|S3')
>>> import pickle
>>> print pickle.loads(pickle.dumps(r))
[(456, 'dbe', 1.2) (2, 'de', 1.3)]

numpy.core.records.fromstring(datastring, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=False, byteorder=None)
create a (read-only) record array from binary data contained in a string

numpy.core.records.fromfile(fd, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=False, byteorder=None)
Create an array from binary file data
If file is a string then that file is opened, else it is assumed to be a file object.

3.1.4 Creating character arrays (numpy.char)

Note: numpy.char is the preferred alias for numpy.core.defchararray.

core.defchararray.array(obj[, itemsize, ...]) Create a chararray.
core.defchararray.asarray(obj[, itemsize, ...]) Convert the input to a chararray, copying the data only if necessary.

numpy.core.defchararray.array(obj, itemsize=None, copy=True, unicode=None, order=None)
Create a chararray.

Note: This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type string_ or unicode_ and use the free functions in numpy.char for fast vectorized string operations instead.

Versus a regular Numpy array of type str or unicode, this class adds the following functionality:
1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. Vectorized string operations are provided as methods (e.g. `str.endswith`) and infix operators (e.g. `+`, `*`, `%`)

**Parameters**

- **obj**: array of str or unicode-like
- **itemsize**: int, optional
  
  `itemsize` is the number of characters per scalar in the resulting array. If `itemsize` is `None`, and `obj` is an object array or a Python list, the `itemsize` will be automatically determined. If `itemsize` is provided and `obj` is of type str or unicode, then the `obj` string will be chunked into `itemsize` pieces.

- **copy**: bool, optional
  
  If true (default), then the object is copied. Otherwise, a copy will only be made if `__array__` returns a copy, if `obj` is a nested sequence, or if a copy is needed to satisfy any of the other requirements (`itemsize`, `unicode`, `order`, etc.).

- **unicode**: bool, optional
  
  When true, the resulting *chararray* can contain Unicode characters, when false only 8-bit characters. If `unicode` is `None` and `obj` is one of the following:

  - a *chararray*,
  - an ndarray of type str or unicode
  - a Python str or unicode object,

  then the `unicode` setting of the output array will be automatically determined.

- **order**: {'C', 'F', 'A'}, optional
  
  Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

### numpy.core.defchararray.asarray(obj, itemsize=None, unicode=None, order=None)

Convert the input to a *chararray*, copying the data only if necessary.

Versus a regular Numpy array of type str or unicode, this class adds the following functionality:

1. values automatically have whitespace removed from the end when indexed
2. comparison operators automatically remove whitespace from the end when comparing values
3. vectorized string operations are provided as methods (e.g. `str.endswith`) and infix operators (e.g. `+`, `*`, `%`)

**Parameters**

- **obj**: array of str or unicode-like
- **itemsize**: int, optional
  
  `itemsize` is the number of characters per scalar in the resulting array. If `itemsize` is `None`, and `obj` is an object array or a Python list, the `itemsize` will be automatically determined. If `itemsize` is provided and `obj` is of type str or unicode, then the `obj` string will be chunked into `itemsize` pieces.

- **unicode**: bool, optional
When true, the resulting chararray can contain Unicode characters, when false only 8-bit characters. If unicode is None and obj is one of the following:

- a chararray,
- an ndarray of type str or ‘unicode’
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

**order**: {'C', 'F'}, optional

Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest).

### 3.1.5 Numerical ranges

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arange([start, stop[, step[, dtype]])</td>
<td>Return evenly spaced values within a given interval.</td>
</tr>
<tr>
<td>linspace(start, stop[, num, endpoint, retstep])</td>
<td>Return evenly spaced numbers over a specified interval.</td>
</tr>
<tr>
<td>logspace(start, stop[, num, endpoint, base])</td>
<td>Return numbers spaced evenly on a log scale.</td>
</tr>
<tr>
<td>meshgrid(*xi, **kwargs)</td>
<td>Return coordinate matrices from two or more coordinate vectors.</td>
</tr>
<tr>
<td>mgrid</td>
<td>nd_grid instance which returns a dense multi-dimensional “meshgrid”.</td>
</tr>
<tr>
<td>ogrid</td>
<td>nd_grid instance which returns an open multi-dimensional “meshgrid”.</td>
</tr>
</tbody>
</table>

numpy.arange ([start], stop[, step[, dtype=None]])

Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

**Parameters**

- **start**: number, optional
  Start of interval. The interval includes this value. The default start value is 0.

- **stop**: number
  End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.

- **step**: number, optional
  Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1. If step is specified, start must also be given.

- **dtype**: dtype
  The type of the output array. If dtype is not given, infer the data type from the other input arguments.

**Returns**

- **arange**: ndarray
Array of evenly spaced values.

For floating point arguments, the length of the result is \( \text{ceil}((\text{stop} - \text{start})/\text{step}) \). Because of floating point overflow, this rule may result in the last element of \( \text{out} \) being greater than \( \text{stop} \).

See Also:

**linspace**
   
   Evenly spaced numbers with careful handling of endpoints.

**ogrid**
   
   Arrays of evenly spaced numbers in N-dimensions.

**mgrid**
   
   Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples

```python
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0., 1., 2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

**numpy.linspace**(start, stop, num=50, endpoint=True, retstep=False)

Return evenly spaced numbers over a specified interval.

Returns **num** evenly spaced samples, calculated over the interval [start, stop].

The endpoint of the interval can optionally be excluded.

Parameters

**start** : scalar
   
   The starting value of the sequence.

**stop** : scalar
   
   The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.

**num** : int, optional
   
   Number of samples to generate. Default is 50.

**endpoint** : bool, optional
   
   If True, stop is the last sample. Otherwise, it is not included. Default is True.

**retstep** : bool, optional
   
   If True, return (samples, step), where step is the spacing between samples.

Returns

**samples** : ndarray
   
   There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False).
step : float (only if retstep is True)

Size of spacing between samples.

See Also:

arange

Similar to linspace, but uses a step size (instead of the number of samples).

logspace

Samples uniformly distributed in log space.

Examples

```python
>>> np.linspace(2.0, 3.0, num=5)
array([ 2. , 2.25, 2.5 , 2.75, 3. ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([ 2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([ 2. , 2.25, 2.5 , 2.75, 3. ]), 0.25)
```

Graphical illustration:

```python
>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

```python
numpy.logspace(start, stop, num=50, endpoint=True, base=10.0)
```

Return numbers spaced evenly on a log scale.
In linear space, the sequence starts at $\text{base} \, \text{**} \, \text{start}$ (base to the power of start) and ends with $\text{base} \, \text{**} \, \text{stop}$ (see endpoint below).

**Parameters**

- `start`: float
  
  $\text{base} \, \text{**} \, \text{start}$ is the starting value of the sequence.

- `stop`: float
  
  $\text{base} \, \text{**} \, \text{stop}$ is the final value of the sequence, unless endpoint is False. In that case, $\text{num} + 1$ values are spaced over the interval in log-space, of which all but the last (a sequence of length num) are returned.

- `num`: integer, optional
  
  Number of samples to generate. Default is 50.

- `endpoint`: boolean, optional
  
  If true, `stop` is the last sample. Otherwise, it is not included. Default is True.

- `base`: float, optional
  
  The base of the log space. The step size between the elements in $\ln(\text{samples}) / \ln(\text{base})$ (or $\log_{\text{base}}(\text{samples})$) is uniform. Default is 10.0.

**Returns**

- `samples`: ndarray
  
  num samples, equally spaced on a log scale.

**See Also:**

- `arange`
  
  Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

- `linspace`
  
  Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

**Notes**

Logspace is equivalent to the code

```python
>>> y = np.logspace(start, stop, num=num, endpoint=endpoint)
...
>>> power(base, y)
...
```

**Examples**

```python
>>> np.logspace(2.0, 3.0, num=4)
array([  100.,  215.443469,  464.15888336, 1000.        ])
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
array([  100.0000000 ,  177.82794101,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
array([  4. ,  5.0396842 ,  6.34960421,  8.        ])
```

Graphical illustration:
```python
>>> import matplotlib.pyplot as plt
>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

```
numpy.meshgrid(*xi, **kwargs)

Return coordinate matrices from two or more coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Parameters

- x1, x2,..., xn : array_like
  1-D arrays representing the coordinates of a grid.

- indexing : {'xy', 'ij'}, optional
  Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.

- sparse : bool, optional
  If True a sparse grid is returned in order to conserve memory. Default is False.

- copy : bool, optional
  If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that sparse=False, copy=False will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.

Returns

- X1, X2,..., XN : ndarray
  ```
For vectors \( x_1, x_2, \ldots, x_n \) with lengths \( N_1 = \text{len}(x_1), \ldots, N_n = \text{len}(x_n) \), return \((N_1, N_2, \ldots, N_n)\) shaped arrays if indexing='ij' or \((N_2, N_1, \ldots, N_n)\) shaped arrays if indexing='xy' with the elements of \( x_i \) repeated to fill the matrix along the first dimension for \( x_1 \), the second for \( x_2 \) and so on.

See Also:

- `index_tricks.mgrid`
  Construct a multi-dimensional “meshgrid” using indexing notation.

- `index_tricks.ogrid`
  Construct an open multi-dimensional “meshgrid” using indexing notation.

Notes

This function supports both indexing conventions through the indexing keyword argument. Giving the string ‘ij’ returns a meshgrid with matrix indexing, while ‘xy’ returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length \( M \) and \( N \), the outputs are of shape \((N, M)\) for ‘xy’ indexing and \((M, N)\) for ‘ij’ indexing. In the 3-D case with inputs of length \( M \), \( N \) and \( P \), outputs are of shape \((N, M, P)\) for ‘xy’ indexing and \((M, N, P)\) for ‘ij’ indexing. The difference is illustrated by the following code snippet:

```python
oxv, yv = meshgrid(x, y, sparse=False, indexing='ij')
for i in range(nx):
    for j in range(ny):
        # treat xv[i,j], yv[i,j]
xv, yv = meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
    for j in range(ny):
        # treat xv[j,i], yv[j,i]
```

In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

Examples

```python
 >>> nx, ny = (3, 2)
 >>> x = np.linspace(0, 1, nx)
 >>> y = np.linspace(0, 1, ny)
 >>> xv, yv = meshgrid(x, y)
 >>> xv
 array([[ 0. , 0.5, 1. ],
        [ 0. , 0.5, 1. ]])
 >>> yv
 array([[ 0. , 0. , 0. ],
        [ 1. , 1. , 1. ]])
 >>> xv, yv = meshgrid(x, y, sparse=True)  # make sparse output arrays
 >>> xv
 array([[ 0. , 0.5, 1. ]])
 >>> yv
 array([[ 0. ],
        [ 1. ]])
```

`meshgrid` is very useful to evaluate functions on a grid.

```python
 >>> x = np.arange(-5, 5, 0.1)
 >>> y = np.arange(-5, 5, 0.1)
 >>> xx, yy = meshgrid(x, y, sparse=True)
 >>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
 >>> h = plt.contourf(x,y,z)
```

3.1. Array creation routines
An instance of `numpy.lib.index_tricks.nd_grid` which returns a dense (or fleshed out) mesh-grid when indexed, so that each returned argument has the same shape. The dimensions and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is **inclusive**.

**Returns**

mesh-grid ndarrays all of the same dimensions

**See Also:**

- `numpy.lib.index_tricks.nd_grid`
  class of `ogrid` and `mgrid` objects
- `numpy.lib.index_tricks.ogrid`
  like `mgrid` but returns open (not fleshed out) mesh grids
- `r_`
  array concatenator

**Examples**

```python
>>> np.mgrid[0:5,0:5]
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]],
      [[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])

>>> np.mgrid[-1:1:5j]
array([-1. , -0.5, 0. , 0.5, 1. ])
```

An instance of `numpy.lib.index_tricks.nd_grid` which returns an open (i.e. not fleshed out) meshgrid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is **inclusive**.

**Returns**

mesh-grid ndarrays with only one dimension ≠ 1

**See Also:**

- `np.lib.index_tricks.nd_grid`
  class of `ogrid` and `mgrid` objects
**mgrid**

like `ogrid` but returns dense (or fleshed out) mesh grids

**r_**

array concatenator

**Examples**

```python
>>> from numpy import ogrid
>>> ogrid[-1:1:5j]
array([-1.0, -0.5, 0.0, 0.5, 1.0])
>>> ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

### 3.1.6 Building matrices

- **`diag(v[, k])`**
  Extract a diagonal or construct a diagonal array.

- **`diagflat(v[, k])`**
  Create a two-dimensional array with the flattened input as a diagonal.

- **`tril(N[, M, k, dtype])`**
  An array with ones at and below the given diagonal and zeros elsewhere.

- **`triu(m[, k])`**
  Lower triangle of an array.

- **`vander(x[, N])`**
  Generate a Van der Monde matrix.

```python
numpy.diag(v, k=0)
```

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for `numpy.diagonal` if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

**Parameters**

- **v : array_like**
  If `v` is a 2-D array, return a copy of its `k`-th diagonal. If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th diagonal.

- **k : int, optional**
  Diagonal in question. The default is 0. Use `k>0` for diagonals above the main diagonal, and `k<0` for diagonals below the main diagonal.

**Returns**

- **out : ndarray**
  The extracted diagonal or constructed diagonal array.

**See Also:**

- **`diagonal`**
  Return specified diagonals.

- **`diagflat`**
  Create a 2-D array with the flattened input as a diagonal.
trace
Sum along diagonals.

triu
Upper triangle of an array.

tril
Lower triangle of an array.

Examples
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])

>>> np.diag(x, k=1)
array([1, 5])

>>> np.diag(x, k=-1)
array([3, 7])

>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])

numpy.diagflat(v, k=0)
Create a two-dimensional array with the flattened input as a diagonal.

Parameters
v : array_like
Input data, which is flattened and set as the k-th diagonal of the output.

k : int, optional
Diagonal to set; 0, the default, corresponds to the “main” diagonal, a positive (negative)
k giving the number of the diagonal above (below) the main.

Returns
out : ndarray
The 2-D output array.

See Also:
diag
MATLAB work-alike for 1-D and 2-D arrays.

diagonal
Return specified diagonals.
trace
Sum along diagonals.
Examples

```python
>>> np.diagflat([[1, 2], [3, 4]])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```python
>>> np.diagflat([1, 2], 1)
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
```

numpy.tri(N, M=None, k=0, dtype=<type 'float'>)

An array with ones at and below the given diagonal and zeros elsewhere.

Parameters

N : int

Number of rows in the array.

M : int, optional

Number of columns in the array. By default, M is taken equal to N.

k : int, optional

The sub-diagonal at and below which the array is filled. k = 0 is the main diagonal, while k < 0 is below it, and k > 0 is above. The default is 0.

dtype : dtype, optional

Data type of the returned array. The default is float.

Returns

tri : ndarray of shape (N, M)

Array with its lower triangle filled with ones and zero elsewhere; in other words T[i, j] == 1 for i <= j + k, 0 otherwise.

Examples

```python
>>> np.tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1]])
```

```python
>>> np.tri(3, 5, -1)
array([[ 0., 0., 0., 0., 0.],
       [ 1., 0., 0., 0., 0.],
       [ 1., 1., 0., 0., 0.]])
```

numpy.tril(m, k=0)

Lower triangle of an array.

Return a copy of an array with elements above the k-th diagonal zeroed.

Parameters

m : array_like, shape (M, N)

Input array.

k : int, optional
Diagonal above which to zero elements. $k = 0$ (the default) is the main diagonal, $k < 0$ is below it and $k > 0$ is above.

**Returns**

*tril*: ndarray, shape (M, N)

Lower triangle of $m$, of same shape and data-type as $m$.

**See Also:**

*triu*

same thing, only for the upper triangle

**Examples**

```python
>>> np.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[0, 0, 0],
       [4, 0, 0],
       [7, 8, 0],
       [10, 11, 12]])
```

```
numpy.triu(m, k=0)
```

Upper triangle of an array.

Return a copy of a matrix with the elements below the $k$-th diagonal zeroed.

Please refer to the documentation for *tril* for further details.

**See Also:**

*tril*

lower triangle of an array

**Examples**

```python
>>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[1, 2, 3],
       [4, 5, 6],
       [0, 8, 9],
       [0, 0, 12]])
```

```
numpy.vander(x, N=None)
```

Generate a Van der Monde matrix.

The columns of the output matrix are decreasing powers of the input vector. Specifically, the $i$-th output column is the input vector raised element-wise to the power of $N - i - 1$. Such a matrix with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

**Parameters**

$x$ : array_like

1-D input array.

$N$ : int, optional

Order of (number of columns in) the output. If $N$ is not specified, a square array is returned ($N = \text{len}(x)$).

**Returns**

$out$ : ndarray
Van der Monde matrix of order \( N \). The first column is \( x^{(N-1)} \), the second \( x^{(N-2)} \) and so forth.

**Examples**

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
      [ 4,  2,  1],
      [ 9,  3,  1],
      [25,  5,  1]])

>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
      [ 4,  2,  1],
      [ 9,  3,  1],
      [25,  5,  1]])
```

```python
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1,  1,  1,  1],
      [ 8,  4,  2,  1],
      [27,  9,  3,  1],
      [125, 25,  5,  1]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043

>>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```

### 3.1.7 The Matrix class

- `mat(data[, dtype])` Interpret the input as a matrix.
- `bmat(obj[, ldict, gdict])` Build a matrix object from a string, nested sequence, or array.

```
numpy.mat (data, dtype=None)
Interpret the input as a matrix.
```

Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

**Parameters**

- `data` : array_like
  Input data.

**Returns**

- `mat` : matrix
  
  `data` interpreted as a matrix.
Examples

```python
>>> x = np.array([[1, 2], [3, 4]])
>>> m = np.asmatrix(x)
>>> x[0,0] = 5
>>> m
matrix([[5, 2],
        [3, 4]])
```

**numpy.bmat**(obj, ldict=None, gdict=None)
Build a matrix object from a string, nested sequence, or array.

**Parameters**

- **obj**: str or array_like
  
  Input data. Names of variables in the current scope may be referenced, even if `obj` is a string.

**Returns**

- **out**: matrix
  
  Returns a matrix object, which is a specialized 2-D array.

**See Also:**
matrix

**Examples**

```python
>>> A = np.mat('1 1; 1 1')
>>> B = np.mat('2 2; 2 2')
>>> C = np.mat('3 4; 5 6')
>>> D = np.mat('7 8; 9 0')

All the following expressions construct the same block matrix:

```python
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```
`copyto`Copies values from one array to another, broadcasting as necessary.

`numpy.copyto (dst, src[, casting, where, preservena])`
Copies values from one array to another, broadcasting as necessary.

Raises a TypeError if the `casting` rule is violated, and if `where` is provided, it selects which elements to copy. New in version 1.7.0.

**Parameters**
- **dst**: ndarray
  The array into which values are copied.
- **src**: array_like
  The array from which values are copied.
- **casting**: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
  Controls what kind of data casting may occur when copying.
  - 'no' means the data types should not be cast at all.
  - 'equiv' means only byte-order changes are allowed.
  - 'safe' means only casts which can preserve values are allowed.
  - 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
  - 'unsafe' means any data conversions may be done.
- **where**: array_like of bool, optional
  A boolean array which is broadcasted to match the dimensions of `dst`, and selects elements to copy from `src` to `dst` wherever it contains the value True.
- **preservena**: bool, optional
  If set to True, leaves any NA values in `dst` untouched. This is similar to the “hard mask” feature in `numpy.ma`.

### 3.2.2 Changing array shape

<table>
<thead>
<tr>
<th><code>reshape(a, newshape[, order])</code></th>
<th>Gives a new shape to an array without changing its data.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ravel(a[, order])</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>ndarray.flat</code></td>
<td>A 1-D iterator over the array.</td>
</tr>
<tr>
<td><code>ndarray.flatten([order])</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
</tbody>
</table>

`numpy.reshape (a, newshape, order=’C’)`
Gives a new shape to an array without changing its data.

**Parameters**
- **a**: array_like
  Array to be reshaped.
- **newshape**: int or tuple of ints
  The new shape should be compatible with the original shape. If an integer, then the
result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

**order**: {'C', 'F', 'A'}, optional

Read the elements of a using this index order, and place the elements into the reshaped array using this index order. ‘C’ means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. ‘F’ means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the ‘C’ and ‘F’ options take no account of the memory layout of the underlying array, and only refer to the order of indexing. ‘A’ means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise.

**Returns**

**reshaped_array**: ndarray

This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the memory layout (C- or Fortran- contiguous) of the returned array.

**See Also**

`ndarray.reshape`

Equivalent method.

**Notes**

It is not always possible to change the shape of an array without copying the data. If you want an error to be raise if the data is copied, you should assign the new shape to the shape attribute of the array:

```python
>>> a = np.zeros((10, 2))  
# A transpose make the array non-contiguous  
>>> b = a.T  
# Taking a view makes it possible to modify the shape without modifying the  
# initial object.  
>>> c = b.view()  
>>> c.shape = (20)  
AttributeError: incompatible shape for a non-contiguous array
```

The **order** keyword gives the index ordering both for fetching the values from a, and then placing the values into the output array. For example, let’s say you have an array:

```python
>>> a = np.arange(6).reshape((3, 2))  
>>> a  
array([[0, 1],  
        [2, 3],  
        [4, 5]])
```

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```python
>>> np.reshape(a, (2, 3)) # C-like index ordering  
array([[0, 1, 2],  
        [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape  
array([[0, 1, 2],  
        [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering  
array([[0, 4, 3],  
        [2, 1, 5]])
```
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
       [1, 1, 5]])

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
>>> np.reshape(a, (3, -1))  # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

**numpy.ravel** *(a, order='C')*

Return a flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

**Parameters**

- **a**: array_like

  Input array. The elements in `a` are read in the order specified by `order`, and packed as a 1-D array.

- **order**: {'C', 'F', 'A', 'K'}, optional

  The elements of `a` are read using this index order. ‘C’ means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. ‘F’ means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the ‘C’ and ‘F’ options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. ‘A’ means to read the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise. ‘K’ means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, ‘C’ index order is used.

**Returns**

- **1d_array**: ndarray

  Output of the same dtype as `a`, and of shape `(a.size,)`.

**See Also**:

- **ndarray.flat**

  1-D iterator over an array.

- **ndarray.flatten**

  1-D array copy of the elements of an array in row-major order.

**Notes**

In C-like (row-major) order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for Fortran-like, or column-major, index ordering.
Examples

It is equivalent to `reshape(-1, order=order)`.

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print np.ravel(x)
[1 2 3 4 5 6]

>>> print x.reshape(-1)
[1 2 3 4 5 6]

>>> print np.ravel(x, order='F')
[1 4 2 5 3 6]
```

When `order` is ‘A’, it will preserve the array’s ‘C’ or ‘F’ ordering:

```python
>>> print np.ravel(x.T)
[1 4 2 5 3 6]
>>> print np.ravel(x.T, order='A')
[1 2 3 4 5 6]
```

When `order` is ‘K’, it will preserve orderings that are neither ‘C’ nor ‘F’, but won’t reverse axes:

```python
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])

>>> a.ravel(order='C')
array([2, 1, 0])

>>> a.ravel(order='K')
array([2, 1, 0])
```

`ndarray.flat`

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See Also:

`flatten`

Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])

>>> x.flat[3]
```
An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1, 4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

**ndarray.flaten** *(order='C')*

Return a copy of the array collapsed into one dimension.

**Parameters**

- **order**: {'C', 'F', 'A'}, optional
  Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from `a`. The default is 'C'.

**Returns**

- **y**: ndarray
  A copy of the input array, flattened to one dimension.

**See Also:**

- **ravel**
  Return a flattened array.

- **flat**
  A 1-D flat iterator over the array.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

### 3.2.3 Transpose-like operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rollaxis</td>
<td>Roll the specified axis backwards, until it lies in a given position.</td>
</tr>
<tr>
<td>swapaxes</td>
<td>Interchange two axes of an array.</td>
</tr>
<tr>
<td>ndarray.T</td>
<td>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td>transpose</td>
<td>Permute the dimensions of an array.</td>
</tr>
</tbody>
</table>

3.2. Array manipulation routines 489
numpy.

**rollaxis** *(a, axis, start=0)*
Roll the specified axis backwards, until it lies in a given position.

**Parameters**

- **a**: ndarray
  Input array.
- **axis**: int
  The axis to roll backwards. The positions of the other axes do not change relative to one another.
- **start**: int, optional
  The axis is rolled until it lies before this position. The default, 0, results in a “complete” roll.

**Returns**

- **res**: ndarray
  Output array.

**See Also:**

- **roll**
  Roll the elements of an array by a number of positions along a given axis.

**Examples**

```python
>>> a = np.ones((3,4,5,6))
>>> np.rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> np.rollaxis(a, 2).shape
(5, 3, 4, 6)
>>> np.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

numpy.

**swapaxes** *(a, axis1, axis2)*
Interchange two axes of an array.

**Parameters**

- **a**: array_like
  Input array.
- **axis1**: int
  First axis.
- **axis2**: int
  Second axis.

**Returns**

- **a_swapped**: ndarray
  If `a` is an ndarray, then a view of `a` is returned; otherwise a new array is created.

**Examples**

```python
>>> x = np.array([[1,2,3]])
>>> np.swapaxes(x,0,1)
array([[1],
```
```python
>>> x = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
>>> x
array([[[ 0,  1],
        [ 2,  3]],
       [[ 4,  5],
        [ 6,  7]]])

>>> np.swapaxes(x,0,2)
array([[[ 0,  4],
        [ 2,  6]],
       [[ 1,  5],
        [ 3,  7]]])
```

`ndarray.T`

Same as `self.transpose()`, except that `self` is returned if `self.ndim < 2`.

**Examples**

```python
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])

>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
```

```python
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])

>>> x.T
array([ 1.,  2.,  3.,  4.])
```

**numpy.transpose(a, axes=None)**

Permute the dimensions of an array.

**Parameters**

- `a`: array_like
  - Input array.

- `axes`: list of ints, optional
  - By default, reverse the dimensions, otherwise permute the axes according to the values given.

**Returns**

- `p`: ndarray
  - `a` with its axes permuted. A view is returned whenever possible.

See Also:

- `rollaxis`

**Examples**

```python
>>> x = np.arange(4).reshape((2,2))
>>> x
```

3.2. Array manipulation routines
array([[0, 1],
       [2, 3]])

>>> np.transpose(x)
array([[0, 2],
       [1, 3]])

>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)

3.2.4 Changing number of dimensions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atleast_1d(*arys)</code></td>
<td>Convert inputs to arrays with at least one dimension.</td>
</tr>
<tr>
<td><code>atleast_2d(*arys)</code></td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td><code>atleast_3d(*arys)</code></td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td><code>broadcast</code></td>
<td>Produce an object that mimics broadcasting.</td>
</tr>
<tr>
<td><code>broadcast_arrays(*args)</code></td>
<td>Broadcast any number of arrays against each other.</td>
</tr>
<tr>
<td><code>expand_dims(a, axis)</code></td>
<td>Expand the shape of an array.</td>
</tr>
<tr>
<td><code>squeeze(a[, axis])</code></td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
</tbody>
</table>

numpy.atleast_1d(*arys)
Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

Parameters

arys1, arys2, ... : array_like
One or more input arrays.

Returns

ret : ndarray
An array, or sequence of arrays, each with a.ndim >= 1. Copies are made only if necessary.

See Also:

atleast_2d, atleast_3d

Examples

>>> np.atleast_1d(1.0)
array([ 1.])

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0., 1., 2.],
       [ 3., 4., 5.],
       [ 6., 7., 8.]])

>>> np.atleast_1d(x) is x
True

>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]

492 Chapter 3. Routines
NumPy Reference, Release 1.8.1

numpy.atleast_2d(*arys)

View inputs as arrays with at least two dimensions.

    Parameters
        arys1, arys2, ... : array_like
            One or more array-like sequences. Non-array inputs are converted to arrays. Arrays
            that already have two or more dimensions are preserved.

    Returns
        res, res2, ... : ndarray
            An array, or tuple of arrays, each with a.ndim >= 2. Copies are avoided where
            possible, and views with two or more dimensions are returned.

See Also:
    atleast_1d, atleast_3d

Examples

>>> np.atleast_2d(3.0)
array([[3.0]])

>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[0., 1., 2.]])

>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]

numpy.atleast_3d(*arys)

View inputs as arrays with at least three dimensions.

    Parameters
        arys1, arys2, ... : array_like
            One or more array-like sequences. Non-array inputs are converted to arrays. Arrays
            that already have three or more dimensions are preserved.

    Returns
        res1, res2, ... : ndarray
            An array, or tuple of arrays, each with a.ndim >= 3. Copies are avoided where
            possible, and views with three or more dimensions are returned. For example, a 1-D
            array of shape (N,) becomes a view of shape (1, N, 1), and a 2-D array of shape
            (M, N) becomes a view of shape (M, N, 1).

See Also:
    atleast_1d, atleast_2d

Examples

>>> np.atleast_3d(3.0)
array([[[3.0]]])

>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x
True

>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print arr, arr.shape
...
[[1]
 [2]]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)

class numpy.broadcast
    Produce an object that mimics broadcasting.

    Parameters
    ----------
    in1, in2, ... : array_like
        Input parameters.

    Returns
    -------
    b : broadcast object
        Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has shape and nd properties, and may be used as an iterator.

Examples

Manually adding two vectors, using broadcasting:

>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)

>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[ 5.,  6.,  7.],
       [ 6.,  7.,  8.],
       [ 7.,  8.,  9.]])

Compare against built-in broadcasting:

>>> x + y
array([[ 5,  6,  7],
       [ 6,  7,  8],
       [ 7,  8,  9]])

Attributes

    index current index in broadcasted result
    iters tuple of iterators along self’s “components.”
    shape Shape of broadcasted result.
    size Total size of broadcasted result.
broadcast.index
  current index in broadcasted result

Examples
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (1, 5), (1, 6))
>>> b.index
3

broadcast.iters
  tuple of iterators along self’s “components.”

Returns a tuple of numpy.flatiter objects, one for each “component” of self.

See Also:
  numpy.flatiter

Examples
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> row, col = b.iters
>>> row.next(), col.next()
(1, 4)

broadcast.shape
  Shape of broadcasted result.

Examples
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.shape
(3, 3)

broadcast.size
  Total size of broadcasted result.

Examples
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.size
9

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>x.next() -&gt; the next value, or raise StopIteration</td>
</tr>
<tr>
<td>reset()</td>
<td>Reset the broadcasted result’s iterator(s).</td>
</tr>
</tbody>
</table>
broadcast.next
   x.next() -> the next value, or raise StopIteration

broadcast.reset()
   Reset the broadcasted result’s iterator(s).

Parameters
   None

Returns
   None

Examples

>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> b.next(), b.next(), b.next()
((1, 4), (2, 4), (3, 4))
>>> b.index
3
>>> b.reset()
>>> b.index
0

numpy.broadcast_arrays(*args)
   Broadcast any number of arrays against each other.

Parameters
   *args: array_likes
      The arrays to broadcast.

Returns
   broadcasted: list of arrays
      These arrays are views on the original arrays. They are typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location. If you need to write to the arrays, make copies first.

Examples

>>> x = np.array([[1, 2, 3]])
>>> y = np.array([[1], [2], [3]])
>>> np.broadcast_arrays(x, y)
[array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]), array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]

Here is a useful idiom for getting contiguous copies instead of non-contiguous views.

>>> [np.array(a) for a in np.broadcast_arrays(x, y)]
[array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]), array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]
**numpy.expand_dims** *(a, axis)*

Expand the shape of an array.

Insert a new axis, corresponding to a given position in the array shape.

**Parameters**

- **a**: array_like
  - Input array.

- **axis**: int
  - Position (amongst axes) where new axis is to be inserted.

**Returns**

- **res**: ndarray
  - Output array. The number of dimensions is one greater than that of the input array.

**See Also:**

doc.indexing, `atleast_1d`, `atleast_2d`, `atleast_3d`

**Examples**

```python
def x = np.array([1,2])
>>> x.shape
(2,)
```

The following is equivalent to `x[np.newaxis, :]` or `x[np.newaxis]`:

```python
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)
```

```python
>>> y = np.expand_dims(x, axis=1)  # Equivalent to x[:,newaxis]
>>> y
array([[1],
        [2]])
>>> y.shape
(2, 1)
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```python
>>> np.newaxis is None
True
```

**numpy.squeeze** *(a, axis=None)*

Remove single-dimensional entries from the shape of an array.

**Parameters**

- **a**: array_like
  - Input data.

- **axis**: None or int or tuple of ints, optional
  - New in version 1.7.0. Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

**Returns**

- **squeezed**: ndarray
  - Output array with single-dimensional entries removed.
The input array, but with with all or a subset of the dimensions of length 1 removed. This is always \( a \) itself or a view into \( a \).

**Examples**

```python
>>> x = np.array([[0], [1], [2]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=(2,)).shape
(1, 3)
```

3.2.5 Changing kind of array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asarray(a[, dtype, order])</code></td>
<td>Convert the input to an array.</td>
</tr>
<tr>
<td><code>asanyarray(a[, dtype, order])</code></td>
<td>Convert the input to an ndarray, but pass ndarray subclasses through.</td>
</tr>
<tr>
<td><code>asmatrix(data[, dtype])</code></td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td><code>asfarray(a[, dtype])</code></td>
<td>Return an array converted to a float type.</td>
</tr>
<tr>
<td><code>asfortranarray(a[, dtype])</code></td>
<td>Return an array laid out in Fortran order in memory.</td>
</tr>
<tr>
<td><code>asscalar(a)</code></td>
<td>Convert an array of size 1 to its scalar equivalent.</td>
</tr>
<tr>
<td><code>require(a[, dtype, requirements])</code></td>
<td>Return an ndarray of the provided type that satisfies requirements.</td>
</tr>
</tbody>
</table>

**numpy.asarray(a, dtype=None, order=None)**

Convert the input to an array.

**Parameters**

- `a`: array_like
  
  Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples of tuples, tuples of lists and ndarrays.

- `dtype`: data-type, optional
  
  By default, the data-type is inferred from the input data.

- `order`: {'C', 'F'}, optional
  
  Whether to use row-major (‘C’) or column-major (‘F’ for FORTRAN) memory representation. Defaults to ‘C’.

**Returns**

- `out`: ndarray
  
  Array interpretation of \( a \). No copy is performed if the input is already an ndarray. If \( a \) is a subclass of ndarray, a base class ndarray is returned.

**See Also:**

- `asanyarray`
  
  Similar function which passes through subclasses.

- `ascontiguousarray`
  
  Convert input to a contiguous array.

- `asfarray`
  
  Convert input to a floating point ndarray.
**asfortranarray**  
Convert input to an ndarray with column-major memory order.

**asarray_chkfinite**  
Similar function which checks input for NaNs and Infs.

**fromiter**  
Create an array from an iterator.

**fromfunction**  
Construct an array by executing a function on grid positions.

**Examples**

Convert a list into an array:

```python
>>> a = [1, 2]
>>> np.asarray(a)
array([1, 2])
```

Existing arrays are not copied:

```python
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

If `dtype` is set, array is copied only if dtype does not match:

```python
>>> a = np.array([1, 2], dtype=np.float32)
>>> np.asarray(a, dtype=np.float32) is a
True
>>> np.asarray(a, dtype=np.float64) is a
False
```

Contrary to `asanyarray`, ndarray subclasses are not passed through:

```python
>>> issubclass(np.matrix, np.ndarray)
True
>>> a = np.matrix([[1, 2]])
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
```

**numpy.asanyarray**(a, dtype=None, order=None)

Convert the input to an ndarray, but pass ndarray subclasses through.

**Parameters**

- `a` : array_like
  Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists, and ndarrays.

- `dtype` : data-type, optional
  By default, the data-type is inferred from the input data.

- `order` : {'C', 'F'}, optional
  Whether to use row-major ('C') or column-major ('F') memory representation. Defaults to 'C'.

---

**3.2. Array manipulation routines**
Returns

**out** : ndarray or an ndarray subclass

Array interpretation of `a`. If `a` is an ndarray or a subclass of ndarray, it is returned as-is and no copy is performed.

See Also:

- **asarray**
  Similar function which always returns ndarrays.
- **ascontiguousarray**
  Convert input to a contiguous array.
- **asfarray**
  Convert input to a floating point ndarray.
- **asfortranarray**
  Convert input to an ndarray with column-major memory order.
- **asarray_chkfinite**
  Similar function which checks input for NaNs and Infs.
- **fromiter**
  Create an array from an iterator.
- **fromfunction**
  Construct an array by executing a function on grid positions.

Examples

Convert a list into an array:

```python
g >> a = [1, 2]
g >> np.asanyarray(a)
g    array([1, 2])
```

Instances of ndarray subclasses are passed through as-is:

```python
g >> a = np.matrix([1, 2])
g >> np.asanyarray(a) is a
```

### numpy.asmatrix

**asmatrix**(data, dtype=None)

Interpret the input as a matrix.

Unlike `matrix`, `asmatrix` does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

Parameters

- **data** : array_like
  Input data.

Returns

- **mat** : matrix
  `data` interpreted as a matrix.

Examples

```python
g >> x = np.array([[1, 2], [3, 4]])
```
```python
>>> m = np.asmatrix(x)

>>> x[0,0] = 5

>>> m
matrix([[5, 2],
        [1, 4]])
```

**numpy.asfarray** *(a, dtype='float64')*

Return an array converted to a float type.

**Parameters**

- `a`: array_like
  
  The input array.

- `dtype`: str or dtype object, optional
  
  Float type code to coerce input array `a`. If `dtype` is one of the ‘int’ dtypes, it is replaced with float64.

**Returns**

- `out`: ndarray
  
  The input `a` as a float `ndarray`.

**Examples**

```python
>>> np.asfarray([2, 3])
array([ 2., 3.])

>>> np.asfarray([2, 3], dtype='float')
array([ 2., 3.])

>>> np.asfarray([2, 3], dtype='int8')
array([ 2., 3.])
```

**numpy.asfortranarray** *(a, dtype=None)*

Return an array laid out in Fortran order in memory.

**Parameters**

- `a`: array_like
  
  Input array.

- `dtype`: str or dtype object, optional
  
  By default, the data-type is inferred from the input data.

**Returns**

- `out`: ndarray
  
  The input `a` in Fortran, or column-major, order.

**See Also:**

- `ascontiguousarray`
  
  Convert input to a contiguous (C order) array.

- `asanyarray`
  
  Convert input to an `ndarray` with either row or column-major memory order.

- `require`
  
  Return an `ndarray` that satisfies requirements.
**ndarray.flags**

Information about the memory layout of the array.

**Examples**

```python
>>> x = np.arange(6).reshape(2,3)
>>> y = np.asfortranarray(x)
>>> x.flags['F_CONTIGUOUS']
False
>>> y.flags['F_CONTIGUOUS']
True
```

**numpy.asscalar(a)**

Convert an array of size 1 to its scalar equivalent.

**Parameters**

- `a`: ndarray
  
  Input array of size 1.

**Returns**

- `out`: scalar
  
  Scalar representation of `a`. The output data type is the same type returned by the input’s `item` method.

**Examples**

```python
>>> np.asscalar(np.array([24]))
24
```

**numpy.require(a, dtype=None, requirements=None)**

Return an ndarray of the provided type that satisfies requirements.

This function is useful to be sure that an array with the correct flags is returned for passing to compiled code (perhaps through ctypes).

**Parameters**

- `a`: array_like
  
  The object to be converted to a type-and-requirement-satisfying array.

- `dtype`: data-type
  
  The required data-type, the default data-type is float64).

- `requirements`: str or list of str
  
  The requirements list can be any of the following
  
  - ‘F_CONTIGUOUS’ (‘F’) - ensure a Fortran-contiguous array
  - ‘C_CONTIGUOUS’ (‘C’) - ensure a C-contiguous array
  - ‘ALIGNED’ (‘A’) - ensure a data-type aligned array
  - ‘WRITEABLE’ (‘W’) - ensure a writable array
  - ‘OWNDATA’ (‘O’) - ensure an array that owns its own data

**See Also:**

- **asarray**
  
  Convert input to an ndarray.
asanyarray
Convert to an ndarray, but pass through ndarray subclasses.

ascontiguousarray
Convert input to a contiguous array.

asfortranarray
Convert input to an ndarray with column-major memory order.

ndarray.flags
Information about the memory layout of the array.

Notes
The returned array will be guaranteed to have the listed requirements by making a copy if needed.

Examples
```python
>>> x = np.arange(6).reshape(2,3)
>>> x.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

>>> y = np.require(x, dtype=np.float32, requirements=['A', 'O', 'W', 'F'])
>>> y.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

3.2.6 Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_stack</td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td>concatenate</td>
<td>Join a sequence of arrays together.</td>
</tr>
<tr>
<td>dstack</td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td>hstack</td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td>vstack</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

numpy.column_stack(tup)
Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters
- tup : sequence of 1-D or 2-D arrays.

Returns
- stacked : 2-D array
The array formed by stacking the given arrays.

See Also:

hstack, vstack, concatenate

Notes

This function is equivalent to np.vstack(tup).T.

Examples

```python
g = np.array((1, 2, 3))
h = np.array((2, 3, 4))
np.column_stack((g, h))
```

Output:

```
array([[1, 2],
       [2, 3],
       [3, 4]])
```

Parameters

- `a1, a2, ...`: sequence of array_like
  - The arrays must have the same shape, except in the dimension corresponding to axis (the first, by default).

  - **axis**: int, optional
    - The axis along which the arrays will be joined. Default is 0.

Returns

- `res`: ndarray
  - The concatenated array.

See Also:

- `ma.concatenate` (a1, a2, ...), axis=0)
  - Join a sequence of arrays together.

- `numpy.concatenate` (a1, a2, ...)
  - Concatenate function that preserves input masks.

- `array_split`
  - Split an array into multiple sub-arrays of equal or near-equal size.

- `split`
  - Split array into a list of multiple sub-arrays of equal size.

- `hstack`
  - Split array into multiple sub-arrays horizontally (column wise)

- `vsplit`
  - Split array into multiple sub-arrays vertically (row wise)

- `dsplit`
  - Split array into multiple sub-arrays along the 3rd axis (depth).

- `hstack`
  - Stack arrays in sequence horizontally (column wise)

- `vstack`
  - Stack arrays in sequence vertically (row wise)
**dstack**

Stack arrays in sequence depth wise (along third dimension)

**Notes**

When one or more of the arrays to be concatenated is a MaskedArray, this function will return a MaskedArray object instead of an ndarray, but the input masks are *not* preserved. In cases where a MaskedArray is expected as input, use the `ma.concatenate` function from the masked array module instead.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
```

This function will not preserve masking of MaskedArray inputs.

```python
>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
             fill_value = 999999)
>>> b
array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
             fill_value = 999999)
```

**numpy.dstack(tup)**

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by `dsplit`. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

**Parameters**

- **tup**: sequence of arrays

  Arrays to stack. All of them must have the same shape along all but the third axis.

**Returns**

- **stacked**: ndarray

  The array formed by stacking the given arrays.

**See Also**:

- **vstack**: Stack along first axis.
numpy.hsplit(tup)

Parameters

tup : sequence of ndarrays

All arrays must have the same shape along all but the second axis.

Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

vstack

Stack arrays in sequence vertically (row wise).

dstack

Stack arrays in sequence depth wise (along third axis).

Notes

Equivalent to np.concatenate(tup, axis=1)
Examples

```python
>>> a = np.array((1, 2, 3))
>>> b = np.array((2, 3, 4))
>>> np.hstack((a, b))
array([1, 2, 3, 2, 3, 4])
```

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

numpy.vstack(tup)

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

Parameters

tup : sequence of ndarrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

concatenate
Join a sequence of arrays together.

vsplit
Split array into a list of multiple sub-arrays vertically.

Notes

Equivalent to np.concatenate(tup, axis=0) if tup contains arrays that are at least 2-dimensional.

Examples

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a, b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```
3.2.7 Splitting arrays

**array_split** *(ary, indices_or_sections[, axis])*  
Split an array into multiple sub-arrays.

Please refer to the *split* documentation. The only difference between these functions is that *array_split* allows *indices_or_sections* to be an integer that does not equally divide the axis.

**See Also:**

*split*

Split array into multiple sub-arrays of equal size.

**Examples**

```python
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
[array([ 0., 1., 2.]), array([ 3., 4., 5.]), array([ 6., 7.])]
```

**numpy.array_split** *(ary, indices_or_sections, axis=0)*  
Split an array into multiple sub-arrays.

Please refer to the *split* documentation. *numpy.array_split* allows *indices_or_sections* to be an integer that does not equally divide the axis.

**See Also:**

*split*

Split array into multiple sub-arrays of equal size.

**Examples**

```python
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> x
array([[[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.]],
       [[ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]]])
>>> np.dsplit(x, 2)
[array([[[ 0.,  1.],
        [ 4.,  5.]],
       [[ 8.,  9.],
        [12., 13.]]),
    array([[[ 2.,  3.],
        [ 6.,  7.]],
       [[10., 11.]]])
```
numpy.hsplit (ary, indices_or_sections)
Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the split documentation. hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.

See Also:

split
Split an array into multiple sub-arrays of equal size.

Examples

```python
>>> x = np.arange(16.0).reshape(4, 4)
array([[ 0.,  1.,  2.,  3.],
[ 4.,  5.,  6.,  7.],
[ 8.,  9., 10., 11.],
[12., 13., 14., 15.]])
```
```python
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
[ 4.,  5.],
[ 8.,  9.],
[12., 13.]]),
array([[ 2.,  3.],
[ 6.,  7.],
[10., 11.],
[14., 15.]]))
```
```python
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
[ 4.,  5.,  6.],
[ 8.,  9., 10.],
[12., 13., 14.]]),
array([[ 3.],
[ 7.],
[11.],
[15.]]),
array([], dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```python
>>> x = np.arange(8.0).reshape(2, 2, 2)
array([[ [ 0.,  1.],
[ 2.,  3.]],
[ [ 4.,  5.],
[ 6.,  7.]]])
```
```python
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
[ 4.,  5.,  6.],
[ 8.,  9., 10.],
[12., 13., 14.]]),
array([[ 3.],
[ 7.],
[11.],
[15.]]),
array([], dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.]]),
 array([[ 2.,  3.],
        [ 6.,  7.]]])

numpy.split(ary, indices_or_sections, axis=0)
    Split an array into multiple sub-arrays.

    Parameters
    =========
    ary : ndarray
        Array to be divided into sub-arrays.

    indices_or_sections : int or 1-D array
        If indices_or_sections is an integer, N, the array will be divided into N equal arrays
        along axis. If such a split is not possible, an error is raised.

        If indices_or_sections is a 1-D array of sorted integers, the entries indicate where along
        axis the array is split. For example, [2, 3] would, for axis=0, result in
        • ary[:2]
        • ary[2:3]
        • ary[3:]

        If an index exceeds the dimension of the array along axis, an empty sub-array is returned
        correspondingly.

    axis : int, optional
        The axis along which to split, default is 0.

    Returns
    =======
    sub-arrays : list of ndarrays
        A list of sub-arrays.

    Raises
    ======
    ValueError
        If indices_or_sections is given as an integer, but a split does not result in equal division.

    See Also:
    =========
    array_split
        Split an array into multiple sub-arrays of equal or near-equal size. Does not raise an exception if an equal
        division cannot be made.

    hsplit
        Split array into multiple sub-arrays horizontally (column-wise).

    vsplit
        Split array into multiple sub-arrays vertically (row wise).

    dsplit
        Split array into multiple sub-arrays along the 3rd axis (depth).

    concatenate
        Join arrays together.

    hstack
        Stack arrays in sequence horizontally (column wise).
vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

Examples

```python
g = np.arange(9.0)
>>> np.vsplit(g, 3)
[array([ 0.,  1.,  2.]),
 array([ 3.,  4.,  5.]),
 array([ 6.,  7.,  8.])]
```

```python
g = np.arange(8.0)
>>> np.vsplit(g, [3, 5, 6, 10])
[array([ 0.,  1.,  2.]),
 array([ 3.,  4.]),
 array([ 5.]),
 array([ 6.,  7.]),
 array([], dtype=float64)]
```

numpy.vsplit(ary, indices_or_sections)
Split an array into multiple sub-arrays vertically (row-wise).

Please refer to the split documentation. vsplit is equivalent to split with axis=0 (default), the array is always split along the first axis regardless of the array dimension.

See Also:

split
Split an array into multiple sub-arrays of equal size.

Examples

```python
g = np.arange(16.0).reshape(4, 4)
>>> g
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```

```python
g = np.arange(16.0).reshape(4, 4)
>>> np.vsplit(g, 2)
[array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]]),
 array([[ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]]))
```

```python
g = np.arange(16.0).reshape(4, 4)
>>> np.vsplit(g, np.array([3, 6]))
[array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]],
 array([[12., 13., 14., 15.]]),
 array([]), dtype=float64)]
```

With a higher dimensional array the split is still along the first axis.

```python
g = np.arange(8.0).reshape(2, 2, 2)
>>> g
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
```

3.2. Array manipulation routines
3.2.8 Tiling arrays

```python
>>> np.vsplit(x, 2)
[array([[ 0., 1.],
        [ 2., 3.]]),
 array([[ 4., 5.],
        [ 6., 7.]]])
```

**tile**

Construct an array by repeating A the number of times given by reps.

```
numpy.tile(A, reps)
```

Construct an array by repeating A the number of times given by reps.

If `reps` has length `d`, the result will have dimension of `max(d, A.ndim)`. If `A.ndim < d`, A is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication. If this is not the desired behavior, promote A to d-dimensions manually before calling this function. If `A.ndim > d`, `reps` is promoted to `A.ndim` by pre-pending 1’s to it. Thus for an A of shape (2, 3, 4, 5), a `reps` of (2, 2) is treated as (1, 1, 2, 2).

**Parameters**

- **A**: array_like
  - The input array.
- **reps**: array_like
  - The number of repetitions of A along each axis.

**Returns**

- **c**: ndarray
  - The tiled output array.

See Also:

**repeat**

Repeat elements of an array.

**Examples**

```python
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (2, 1, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])

>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```
```python
>>> np.tile(b, (2, 1))
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

```python
numpy.repeat(a, repeats, axis=None)
Repeat elements of an array.

Parameters
a : array_like
Input array.
repeats : {int, array of ints}
The number of repetitions for each element. repeats is broadcasted to fit the shape of
the given axis.
axis : int, optional
The axis along which to repeat values. By default, use the flattened input array, and
return a flat output array.

Returns
repeated_array : ndarray
Output array which has the same shape as a, except along the given axis.

See Also:
tile
Tile an array.

Examples
```python
>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
>>> np.repeat(x, [1, 2], axis=0)
array([[1, 2],
       [3, 4],
       [3, 4]])
```

3.2.9 Adding and removing elements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete(arr, obj[, axis])</td>
<td>Return a new array with sub-arrays along an axis deleted.</td>
</tr>
<tr>
<td>insert(arr, obj, values[, axis])</td>
<td>Insert values along the given axis before the given indices.</td>
</tr>
<tr>
<td>append(arr, values[, axis])</td>
<td>Append values to the end of an array.</td>
</tr>
<tr>
<td>resize(a, new_shape)</td>
<td>Return a new array with the specified shape.</td>
</tr>
<tr>
<td>trim_zeros([filt[, trim]])</td>
<td>Trim the leading and/or trailing zeros from a 1-D array or sequence.</td>
</tr>
<tr>
<td>unique(ar[, return_index, return_inverse])</td>
<td>Find the unique elements of an array.</td>
</tr>
</tbody>
</table>
NumPy Reference, Release 1.8.1

numpy.delete(arr, obj, axis=None)
Return a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by arr[obj].

Parameters
arr : array_like
Input array.
obj : slice, int or array of ints
Indicate which sub-arrays to remove.
axis : int, optional
The axis along which to delete the subarray defined by obj. If axis is None, obj is applied to the flattened array.

Returns
out : ndarray
A copy of arr with the elements specified by obj removed. Note that delete does not occur in-place. If axis is None, out is a flattened array.

See Also:
insert
Insert elements into an array.
append
Append elements at the end of an array.

Notes
Often it is preferable to use a boolean mask. For example:

```python
>>> mask = np.ones(len(arr), dtype=bool)
>>> mask[[0,2,4]] = False
>>> result = arr[mask,...]
```

Is equivalent to np.delete(arr, [0,2,4], axis=0), but allows further use of mask.

Examples
```python
>>> arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> arr
array([[ 1, 2, 3, 4],
      [ 5, 6, 7, 8],
      [ 9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[ 1, 3, 4],
      [ 5, 7, 8],
      [ 9, 11, 12]])
>>> np.delete(arr, np.s_[::2], 1)
array([[ 2, 4],
      [ 6, 8],
      [10, 12]])
>>> np.delete(arr, [1,3,5], None)
array([[ 1, 3, 5, 7, 8, 9, 10, 11, 12]])
```
numpy.insert(arr, obj, values, axis=None)
Insert values along the given axis before the given indices.
Parameters

arr : array_like

Input array.

obj : int, slice or sequence of ints

Object that defines the index or indices before which values is inserted. New in version 1.8.0. Support for multiple insertions when obj is a single scalar or a sequence with one element (similar to calling insert multiple times).

values : array_like

Values to insert into arr. If the type of values is different from that of arr, values is converted to the type of arr. values should be shaped so that arr[...,obj,...] = values is legal.

axis : int, optional

Axis along which to insert values. If axis is None then arr is flattened first.

Returns

out : ndarray

A copy of arr with values inserted. Note that insert does not occur in-place: a new array is returned. If axis is None, out is a flattened array.

See Also:

append

Append elements at the end of an array.

concatenate

Join a sequence of arrays together.

delete

Delete elements from an array.

Notes

Note that for higher dimensional inserts obj=0 behaves very different from obj=[0] just like arr[:,0,:]=values is different from arr[:,[0],:]=values.

Examples

>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])

Difference between sequence and scalars: >>> np.insert(a, [1], [[1],[2],[3]], axis=1) array([[1, 1, 1],
         [2, 2, 2],
         [3, 3, 3]])
```python
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
... np.insert(a, [1], [[1],[2],[3]], axis=1))
True

>>> b = a.flatten()
>>> b
array([1, 1, 2, 2, 3, 3])

>>> np.insert(b, [2, 2], [5, 6])
array([1, 1, 5, 6, 2, 2, 3, 3])

>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, 2, 6, 2, 3, 3])

>>> np.insert(b, [2, 2], [7.13, False])  # type casting
array([1, 1, 7, 0, 2, 2, 3, 3])

>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)

>>> np.insert(x, idx, 999, axis=1)
array([[ 0, 999,  1,  2, 999,  3],
       [ 4, 999,  5,  6, 999,  7]])
```

### numpy.append(arr, values, axis=None)

Append values to the end of an array.

**Parameters**

- **arr**: array_like
  
  Values are appended to a copy of this array.

- **values**: array_like
  
  These values are appended to a copy of `arr`. It must be of the correct shape (the same shape as `arr`, excluding `axis`). If `axis` is not specified, `values` can be any shape and will be flattened before use.

- **axis**: int, optional
  
  The axis along which `values` are appended. If `axis` is not given, both `arr` and `values` are flattened before use.

**Returns**

- **append**: ndarray
  
  A copy of `arr` with `values` appended to `axis`. Note that `append` does not occur in-place: a new array is allocated and filled. If `axis` is None, `out` is a flattened array.

**See Also:**

- **insert**
  
  Insert elements into an array.

- **delete**
  
  Delete elements from an array.

**Examples**

```python
>>> np.append([[1, 2, 3], [[4, 5, 6], [7, 8, 9]]])
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

When `axis` is specified, `values` must have the correct shape.
```python
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
Traceback (most recent call last):
...
ValueError: arrays must have same number of dimensions
```

```python
numpy.resize(a, new_shape)
```

Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copies of `a`. Note that this behavior is different from `a.resize(new_shape)` which fills with zeros instead of repeated copies of `a`.

**Parameters**

- `a`: array_like
  - Array to be resized.
- `new_shape`: int or tuple of int
  - Shape of resized array.

**Returns**

- `reshaped_array`: ndarray
  - The new array is formed from the data in the old array, repeated if necessary to fill out the required number of elements. The data are repeated in the order that they are stored in memory.

**See Also:**

- `ndarray.resize`
  - resize an array in-place.

**Examples**

```python
>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(1,4))
array([[0, 1, 2, 3]])
>>> np.resize(a,(2,4))
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

```python
numpy.trim_zeros(filt, trim='fb')
```

Trim the leading and/or trailing zeros from a 1-D array or sequence.

**Parameters**

- `filt`: 1-D array or sequence
  - Input array.
- `trim`: str, optional
  - A string with ‘f’ representing trim from front and ‘b’ to trim from back. Default is ‘fb’, trim zeros from both front and back of the array.

**Returns**

- `trimmed`: 1-D array or sequence
  - The result of trimming the input. The input data type is preserved.
Examples

```python
>>> a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
>>> np.trim_zeros(a)
array([1, 2, 3, 0, 2, 1])
```

```python
>>> np.trim_zeros(a, 'b')
array([0, 0, 0, 1, 2, 3, 0, 2, 1])
```

The input data type is preserved, list/tuple in means list/tuple out.

```python
>>> np.trim_zeros([0, 1, 2, 0])
[1, 2]
```

```python
numpy.unique(ar, return_index=False, return_inverse=False)
```

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.

**Parameters**

- `ar` : array_like
  Input array. This will be flattened if it is not already 1-D.

- `return_index` : bool, optional
  If True, also return the indices of `ar` that result in the unique array.

- `return_inverse` : bool, optional
  If True, also return the indices of the unique array that can be used to reconstruct `ar`.

**Returns**

- `unique` : ndarray
  The sorted unique values.

- `unique_indices` : ndarray, optional
  The indices of the first occurrences of the unique values in the (flattened) original array. Only provided if `return_index` is True.

- `unique_inverse` : ndarray, optional
  The indices to reconstruct the (flattened) original array from the unique array. Only provided if `return_inverse` is True.

**See Also:**

`numpy.lib.arraysetops`

Module with a number of other functions for performing set operations on arrays.

**Examples**

```python
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
```

```python
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the indices of the original array that give the unique values:
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
      dtype='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
      dtype='|S1')

Reconstruct the input array from the unique values:

```python
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

### 3.2.10 Rearranging elements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fliplr(m)</code></td>
<td>Flip array in the left/right direction.</td>
</tr>
<tr>
<td><code>flipud(m)</code></td>
<td>Flip array in the up/down direction.</td>
</tr>
<tr>
<td><code>reshape(a, newshape[, order])</code></td>
<td>Gives a new shape to an array without changing its data.</td>
</tr>
<tr>
<td><code>roll(a, shift[, axis])</code></td>
<td>Roll array elements along a given axis.</td>
</tr>
<tr>
<td><code>rot90(m[, k])</code></td>
<td>Rotate an array by 90 degrees in the counter-clockwise direction.</td>
</tr>
</tbody>
</table>

#### `numpy.fliplr(m)`

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters**

- `m`: array_like
  - Input array.

**Returns**

- `f`: ndarray
  - A view of `m` with the columns reversed. Since a view is returned, this operation is $O(1)$.

**See Also:**

- `flipud`
  - Flip array in the up/down direction.

- `rot90`
  - Rotate array counterclockwise.

**Notes**

Equivalent to `A[:,::-1]`. Does not require the array to be two-dimensional.
Examples

```python
>>> A = np.diag([1., 2., 3.])
```
```python
>>> A
array([[ 1., 0., 0.],
       [ 0., 2., 0.],
       [ 0., 0., 3.]])
```
```python
>>> np.fliplr(A)
array([[ 0., 0., 1.],
       [ 0., 2., 0.],
       [ 3., 0., 0.]])
```
```python
>>> A = np.random.randn(2, 3, 5)
```n```python
>>> np.all(np.fliplr(A)==A[:,::-1,...])
```
True
```

*numpy.flipud(m)*

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

**Parameters**

- **m**: array_like
  - Input array.

**Returns**

- **out**: array_like
  - A view of m with the rows reversed. Since a view is returned, this operation is \(O(1)\).

**See Also**

- **fliplr**: Flip array in the left/right direction.
- **rot90**: Rotate array counterclockwise.

**Notes**

Equivalent to \(A[:,:,:-1,\ldots]\). Does not require the array to be two-dimensional.

**Examples**

```python
>>> A = np.diag([1.0, 2, 3])
```n```python
>>> A
array([[ 1., 0., 0.],
       [ 0., 2., 0.],
       [ 0., 0., 3.]])
```
```python
>>> np.flipud(A)
array([[ 0., 0., 3.],
       [ 0., 2., 0.],
       [ 1., 0., 0.]])
```
```python
>>> A = np.random.randn(2, 3, 5)
```n```python
>>> np.all(np.flipud(A)==A[::-1,...])
```
True
```
```python
>>> np.flipud([1,2])
array([2, 1])
```

numpy.reshape(a, newshape, order='C')

Gives a new shape to an array without changing its data.

Parameters

- `a` : array_like
  Array to be reshaped.

- `newshape` : int or tuple of ints
  The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

- `order` : {'C', 'F', 'A'}, optional
  Read the elements of `a` using this index order, and place the elements into the reshaped array using this index order. ‘C’ means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. ‘F’ means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the ‘C’ and ‘F’ options take no account of the memory layout of the underlying array, and only refer to the order of indexing. ‘A’ means to read / write the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise.

Returns

- `reshaped_array` : ndarray
  This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the memory layout (C- or Fortran- contiguous) of the returned array.

See Also:

- ndarray.reshape
  Equivalent method.

Notes

It is not always possible to change the shape of an array without copying the data. If you want an error to be raise if the data is copied, you should assign the new shape to the shape attribute of the array:

```python
>>> a = np.zeros((10, 2))
# A transpose make the array non-contiguous
>>> b = a.T
# Taking a view makes it possible to modify the shape without modifying the # initial object.
>>> c = b.view()
>>> c.shape = (20)
AttributeError: incompatible shape for a non-contiguous array
```

The `order` keyword gives the index ordering both for fetching the values from `a`, and then placing the values into the output array. For example, let’s say you have an array:

```python
>>> a = np.arange(6).reshape((3, 2))
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

3.2. Array manipulation routines
You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```python
>>> np.reshape(a, (2, 3))  # C-like index ordering
array([[0, 1, 2],
        [3, 4, 5]]

>>> np.reshape(np.ravel(a), (2, 3))  # equivalent to C ravel then C reshape
array([[0, 1, 2],
        [3, 4, 5]]

>>> np.reshape(a, (2, 3), order='F')  # Fortran-like index ordering
array([[0, 4, 3],
        [2, 1, 5]]

>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
        [2, 1, 5]])
```

**Examples**

```python
>>> a = np.array([[1, 2, 3],
                [4, 5, 6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])

>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])

>>> np.reshape(a, (3,-1))  # the unspecified value is inferred to be 2
array([[1, 2],
        [3, 4],
        [5, 6]])
```

**numpy.roll**(a, shift, axis=None)

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

**Parameters**

- **a**: array_like
  - Input array.

- **shift**: int
  - The number of places by which elements are shifted.

- **axis**: int, optional
  - The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

**Returns**

- **res**: ndarray
  - Output array, with the same shape as `a`.

**See Also:**

**rollaxis**

Roll the specified axis backwards, until it lies in a given position.

**Examples**
>>> x = np.arange(10)
>>> np.roll(x, 2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])

>>> x2 = np.reshape(x, (2,5))
>>> x2
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

>>> np.roll(x2, 1)
array([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])

>>> np.roll(x2, 1, axis=0)
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])

>>> np.roll(x2, 1, axis=1)
array([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])

numpy.rot90(m, k=1)

Rotate an array by 90 degrees in the counter-clockwise direction.

The first two dimensions are rotated; therefore, the array must be at least 2-D.

Parameters

m : array_like
    Array of two or more dimensions.

k : integer
    Number of times the array is rotated by 90 degrees.

Returns

y : ndarray
    Rotated array.

See Also:

fliplr

Flip an array horizontally.

flipud

Flip an array vertically.

Examples

>>> m = np.array([[1, 2], [3, 4]], int)
>>> m
array([[1, 2],
       [3, 4]])

>>> np.rot90(m)
array([[2, 4],
       [1, 3]])

>>> np.rot90(m, 2)
array([[4, 3],
       [2, 1]])
3.3 Binary operations

3.3.1 Elementwise bit operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bitwise_and(x1, x2[, out])</code></td>
<td>Compute the bit-wise AND of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_or(x1, x2[, out])</code></td>
<td>Compute the bit-wise OR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>bitwise_xor(x1, x2[, out])</code></td>
<td>Compute the bit-wise XOR of two arrays element-wise.</td>
</tr>
<tr>
<td><code>invert(x[, out])</code></td>
<td>Compute bit-wise inversion, or bit-wise NOT, element-wise.</td>
</tr>
<tr>
<td><code>left_shift(x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the left.</td>
</tr>
<tr>
<td><code>right_shift(x1, x2[, out])</code></td>
<td>Shift the bits of an integer to the right.</td>
</tr>
</tbody>
</table>

numpy.bitwise_and(x1, x2[, out]) = <ufunc ‘bitwise_and’>

Compute the bit-wise AND of two arrays element-wise.

Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator &.

**Parameters**

- `x1, x2`: array_like
  
  Only integer and boolean types are handled.

**Returns**

- `out`: array_like
  
  Result.

**See Also:**

- `logical_and`, `bitwise_or`, `bitwise_xor`

**binary_repr**

Return the binary representation of the input number as a string.

**Examples**

The number 13 is represented by 00001101. Likewise, 17 is represented by 00010001. The bit-wise AND of 13 and 17 is therefore 00000000, or 1:

```python
>>> np.bitwise_and(13, 17)
1
```

```python
>>> np.bitwise_and(14, 13)
12
>>> np.bitwise_and([14, 3], 13)
array([12, 1])
```

```python
>>> np.bitwise_and([11,7], [4,25])
array([0, 1])
```

```python
>>> np.bitwise_and(np.array([2,5,255]), np.array([3,14,16]))
array([ 2,  4, 16])
```

```python
>>> np.bitwise_and([True, True], [False, True])
array([False, True], dtype=bool)
```

numpy.bitwise_or(x1, x2[, out]) = <ufunc ‘bitwise_or’>

Compute the bit-wise OR of two arrays element-wise.
Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator |.

Parameters

- **x1, x2**: array_like
  
  Only integer and boolean types are handled.

- **out**: ndarray, optional
  
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

- **out**: array_like
  
  Result.

See Also:

- `logical_or`, `bitwise_and`, `bitwise_xor`

- `binary_repr`

  Return the binary representation of the input number as a string.

Examples

The number 13 has the binary representation 00001101. Likewise, 16 is represented by 00010000. The bit-wise OR of 13 and 16 is then 000111011, or 29:

```python
>>> np.bitwise_or(13, 16)
29
>>> np.binary_repr(29)
'11101'
```

```python
>>> np.bitwise_or(32, 2)
34
>>> np.bitwise_or([33, 4], 1)
array([33, 5])
>>> np.bitwise_or([33, 4], [1, 2])
array([33, 6])
```

```python
>>> np.bitwise_or(np.array([2, 5, 255]), np.array([4, 4, 4]))
array([ 6,  5, 255])
>>> np.array([2, 5, 255]) | np.array([4, 4, 4])
array([ 6,  5, 255])
>>> np.bitwise_or(np.array([2, 5, 255, 2147483647L], dtype=np.int32),
... np.array([4, 4, 4, 2147483647L], dtype=np.int32))
array([ 6,  5, 255, 2147483647])
>>> np.bitwise_or([True, True], [False, True])
array([True, True], dtype=bool)
```

`numpy.bitwise_xor(x1, x2[, out]) = <ufunc 'bitwise_xor'>`

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ^.

Parameters

- **x1, x2**: array_like
  
  Only integer and boolean types are handled.
NumPy Reference, Release 1.8.1

Returns
out : array_like
Result.

See Also:
logical_xor, bitwise_and, bitwise_or

binary_repr
Return the binary representation of the input number as a string.

Examples
The number 13 is represented by 00001101. Likewise, 17 is represented by 00010001. The bit-wise XOR of 13 and 17 is therefore 00011100, or 28:

```python
>>> np.bitwise_xor(13, 17)
28
>>> np.binary_repr(28)
'11100'
```

```python
>>> np.bitwise_xor(31, 5)
26
>>> np.bitwise_xor([31,3], 5)
array([26, 6])
```

```python
>>> np.bitwise_xor([31,3], [5,6])
array([26, 5])
>>> np.bitwise_xor([True, True], [False, True])
array([ True, False], dtype=bool)
```

numpy.invert(x[, out]) = <ufunc 'invert'>
Compute bit-wise inversion, or bit-wise NOT, element-wise.

Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ~.

For signed integer inputs, the two’s complement is returned. In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers [R32]. A N-bit two’s-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1} - 1$.

Parameters
x1 : array_like
Only integer and boolean types are handled.

Returns
out : array_like
Result.

See Also:
bitwise_and, bitwise_or, bitwise_xor, logical_not

binary_repr
Return the binary representation of the input number as a string.
Notes

bitwise_not is an alias for invert:

```python
>>> np.bitwise_not is np.invert
True
```

References

[R32]

Examples

We’ve seen that 13 is represented by 00001101. The invert or bit-wise NOT of 13 is then:

```python
>>> np.invert(np.array([13], dtype=uint8))
array([242], dtype=uint8)
>>> np.binary_repr(x, width=8)
'00001101'
>>> np.binary_repr(242, width=8)
'11110010'
```

The result depends on the bit-width:

```python
>>> np.invert(np.array([13], dtype=uint16))
array([65522], dtype=uint16)
>>> np.binary_repr(x, width=16)
'0000000000001101'
>>> np.binary_repr(65522, width=16)
'1111111111110010'
```

When using signed integer types the result is the two’s complement of the result for the unsigned type:

```python
>>> np.invert(np.array([13], dtype=int8))
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)
'11110010'
```

Booleans are accepted as well:

```python
>>> np.invert(array([True, False]))
array([False, True], dtype=bool)
```

numpy.left_shift(x1, x2[, out]) = <ufunc 'left_shift'>

Shift the bits of an integer to the left.

Bits are shifted to the left by appending x2 0s at the right of x1. Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying x1 by \(2^{x2}\).

Parameters

- x1 : array_like of integer type
  - Input values.
- x2 : array_like of integer type
  - Number of zeros to append to x1. Has to be non-negative.

Returns

- out : array of integer type
  - Return x1 with bits shifted x2 times to the left.

See Also:
**right_shift**
Shift the bits of an integer to the right.

**binary_repr**
Return the binary representation of the input number as a string.

### Examples

```python
>>> np.binary_repr(5)
'101'
>>> np.left_shift(5, 2)
20
>>> np.binary_repr(20)
'10100'

>>> np.left_shift(5, [1,2,3])
array([10, 20, 40])
```

numpy.right_shift(x1, x2[, out]) = <ufunc 'right_shift'>
Shift the bits of an integer to the right.

Bits are shifted to the right x2. Because the internal representation of numbers is in binary format, this operation is equivalent to dividing x1 by \(2^{x2}\).

**Parameters**

- **x1**: array_like, int
  Input values.

- **x2**: array_like, int
  Number of bits to remove at the right of x1.

**Returns**

- **out**: ndarray, int
  Return x1 with bits shifted x2 times to the right.

**See Also:**

**left_shift**
Shift the bits of an integer to the left.

**binary_repr**
Return the binary representation of the input number as a string.

### Examples

```python
>>> np.binary_repr(10)
'1010'
>>> np.right_shift(10, 1)
5
>>> np.binary_repr(5)
'101'

>>> np.right_shift(10, [1,2,3])
array([5, 2, 1])
```

### 3.3.2 Bit packing
**packbits**(myarray[, axis])  Packs the elements of a binary-valued array into bits in a uint8 array.

**unpackbits**(myarray[, axis])  Unpacks elements of a uint8 array into a binary-valued output array.

numpy.packbits (myarray, axis=None)

Packs the elements of a binary-valued array into bits in a uint8 array.

The result is padded to full bytes by inserting zero bits at the end.

**Parameters**

- **myarray** : array_like
  
  An integer type array whose elements should be packed to bits.

- **axis** : int, optional
  
  The dimension over which bit-packing is done. None implies packing the flattened array.

**Returns**

- **packed** : ndarray
  
  Array of type uint8 whose elements represent bits corresponding to the logical (0 or nonzero) value of the input elements. The shape of packed has the same number of dimensions as the input (unless axis is None, in which case the output is 1-D).

**See Also:**

**unpackbits**

Unpacks elements of a uint8 array into a binary-valued output array.

**Examples**

```python
>>> a = np.array([[1,0,1],
...                [0,1,0],
...                [1,1,0],
...                [0,0,1]])
>>> b = np.packbits(a, axis=-1)
>>> b
array([[160],[64]], dtype=uint8)
```

Note that in binary 160 = 1010 0000, 64 = 0100 0000, 192 = 1100 0000, and 32 = 0010 0000.

numpy.unpackbits (myarray, axis=None)

Unpacks elements of a uint8 array into a binary-valued output array.

Each element of myarray represents a bit-field that should be unpacked into a binary-valued output array. The shape of the output array is either 1-D (if axis is None) or the same shape as the input array with unpacking done along the axis specified.

**Parameters**

- **myarray** : ndarray, uint8 type
  
  Input array.

- **axis** : int, optional
  
  Unpacks along this axis.

**Returns**

- **unpacked** : ndarray, uint8 type

3.3. Binary operations 529
The elements are binary-valued (0 or 1).

See Also:

packbits
Packs the elements of a binary-valued array into bits in a uint8 array.

Examples

```python
>>> a = np.array([[2], [7], [23]], dtype=np.uint8)
```

```text
array([[ 2],
       [ 7],
       [23]], dtype=uint8)
```

```python
>>> b = np.unpackbits(a, axis=1)
```

```text
array([[0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)
```

### 3.3.3 Output formatting

**numpy.binary_repr(num[, width])**  
Return the binary representation of the input number as a string.

**numpy.binary_repr(num, width=None)**  
Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two’s complement of the number is returned, with respect to that width.

In a two’s-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers \[R16\]. A N-bit two’s-complement system can represent every integer in the range \(-2^{N-1} + 1\) to \(+2^{N-1} - 1\).

**Parameters**

- **num** : int
  Only an integer decimal number can be used.

- **width** : int, optional
  The length of the returned string if `num` is positive, the length of the two’s complement if `num` is negative.

**Returns**

- **bin** : str
  Binary representation of `num` or two’s complement of `num`.

See Also:

**base_repr**
Return a string representation of a number in the given base system.

**Notes**

`binary_repr` is equivalent to using `base_repr` with base 2, but about 25x faster.
3.4 String operations

This module provides a set of vectorized string operations for arrays of type `numpy.string_` or `numpy.unicode_`. All of them are based on the string methods in the Python standard library.

### 3.4.1 String operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x1, x2)</td>
<td>Return element-wise string concatenation for two arrays of str or unicode.</td>
</tr>
<tr>
<td>multiply(a, i)</td>
<td>Return (a * i), that is string multiple concatenation, element-wise.</td>
</tr>
<tr>
<td>mod(a, values)</td>
<td>Return (a % i), that is pre-Python 2.6 string formatting.</td>
</tr>
<tr>
<td>capitalize(a)</td>
<td>Return a copy of a with only the first character of each element capitalized.</td>
</tr>
<tr>
<td>center(a, width[, fillchar])</td>
<td>Return a copy of a with its elements centered in a string of length width.</td>
</tr>
<tr>
<td>decode(a[, encoding, errors])</td>
<td>Calls str.decode element-wise.</td>
</tr>
<tr>
<td>encode(a[, encoding, errors])</td>
<td>Calls str.encode element-wise.</td>
</tr>
<tr>
<td>join(sep, seq)</td>
<td>Return a string which is the concatenation of the strings in the sequence sep.</td>
</tr>
<tr>
<td>ljust(a, width[, fillchar])</td>
<td>Return an array with the elements of a left-justified in a string of length width.</td>
</tr>
<tr>
<td>lower(a)</td>
<td>Return an array with the elements converted to lowercase.</td>
</tr>
<tr>
<td>lstrip(a[, chars])</td>
<td>For each element in a, return a copy with the leading characters removed.</td>
</tr>
<tr>
<td>partition(a, sep)</td>
<td>Partition each element in a around sep.</td>
</tr>
<tr>
<td>replace(a, old, new[, count])</td>
<td>For each element in a, return a copy of the string with all occurrences of substring old replaced.</td>
</tr>
<tr>
<td>rjust(a, width[, fillchar])</td>
<td>Return an array with the elements of a right-justified in a string of length width.</td>
</tr>
<tr>
<td>rpartition(a, sep)</td>
<td>Partition (split) each element around the right-most separator.</td>
</tr>
<tr>
<td>rsplit(a[, sep, maxsplit])</td>
<td>For each element in a, return a list of the words in the string, using sep as the delimiter string.</td>
</tr>
<tr>
<td>rstrip(a[, chars])</td>
<td>For each element in a, return a copy with the trailing characters removed.</td>
</tr>
<tr>
<td>split(a[, sep, maxsplit])</td>
<td>For each element in a, return a list of the words in the string, using sep as the delimiter string.</td>
</tr>
<tr>
<td>splitlines(a[, keepends])</td>
<td>For each element in a, return a list of the lines in the element, breaking at line boundaries.</td>
</tr>
<tr>
<td>strip(a[, chars])</td>
<td>For each element in a, return a copy with the leading and trailing characters removed.</td>
</tr>
<tr>
<td>swapcase(a)</td>
<td>Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa.</td>
</tr>
<tr>
<td>title(a)</td>
<td>Return element-wise title cased version of string or unicode.</td>
</tr>
<tr>
<td>translate(a, table[, deletechars])</td>
<td>For each element in a, return a copy of the string where all characters occurring in the optional deletechars argument are removed.</td>
</tr>
<tr>
<td>upper(a)</td>
<td>Return an array with the elements converted to uppercase.</td>
</tr>
<tr>
<td>zfill(a, width)</td>
<td>Return the numeric string left-filled with zeros</td>
</tr>
</tbody>
</table>
numpy.core.defchararray.add(x1, x2)
Return element-wise string concatenation for two arrays of str or unicode.

Arrays x1 and x2 must have the same shape.

Parameters
x1 : array_like of str or unicode
Input array.
x2 : array_like of str or unicode
Input array.

Returns
add : ndarray
Output array of string_ or unicode_, depending on input types of the same shape as x1 and x2.

numpy.core.defchararray.multiply(a, i)
Return (a * i), that is string multiple concatenation, element-wise.

Values in i of less than 0 are treated as 0 (which yields an empty string).

Parameters
a : array_like of str or unicode
i : array_like of ints

Returns
out : ndarray
Output array of str or unicode, depending on input types

numpy.core.defchararray.mod(a, values)
Return (a % i), that is pre-Python 2.6 string formatting (interpolation), element-wise for a pair of array_likes of str or unicode.

Parameters
a : array_like of str or unicode
values : array_like of values

These values will be element-wise interpolated into the string.

Returns
out : ndarray
Output array of str or unicode, depending on input types

See Also:
str.__mod__

numpy.core.defchararray.capitalize(a)
Return a copy of a with only the first character of each element capitalized.

Calls str.capitalize element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters
a : array_like of str or unicode

Input array of strings to capitalize.
Returns

out : ndarray

Output array of str or unicode, depending on input types

See Also:

str.capitalize

Examples

```python
c = np.array(['a1b2', '1b2a', 'b2a1', '2a1b'], 'S4'); c
dtype='|S4')
np.char.capitalize(c)
dtype='|S4')
```

numpy.core.defchararray.center(a, width, fillchar=' ')

Return a copy of a with its elements centered in a string of length width.

Calls str.center element-wise.

Parameters

- a : array_like of str or unicode
- width : int
  The length of the resulting strings
- fillchar : str or unicode, optional
  The padding character to use (default is space).

Returns

out : ndarray

Output array of str or unicode, depending on input types

See Also:

str.center

numpy.core.defchararray.decode(a, encoding=None, errors=None)

Calls str.decode element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the codecs module.

Parameters

- a : array_like of str or unicode
- encoding : str, optional
  The name of an encoding
- errors : str, optional
  Specifies how to handle encoding errors

Returns

out : ndarray

See Also:

str.decode
Notes

The type of the result will depend on the encoding specified.

Examples

```python
>>> c = np.array(['aAaAaA', ' aA ', 'abBAbba'])
>>> c
array(['aAaAaA', ' aA ', 'abBAbba'],
      dtype='|S7')
>>> np.char.encode(c, encoding='cp037')
array(['\x81\xc1\x81\xc1\x81\xc1', '@@\x81\xc1@@',
       '\x81\x82\xc2\xc1\xc2\x82\x81'],
      dtype='|S7')
```

`numpy.core.defchararray.encode(a, encoding=None, errors=None)`

Calls `str.encode` element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the codecs module.

Parameters

- `a` : array_like of str or unicode
- `encoding` : str, optional
  The name of an encoding
- `errors` : str, optional
  Specifies how to handle encoding errors

Returns

- `out` : ndarray

See Also:

- `str.encode`

Notes

The type of the result will depend on the encoding specified.

`numpy.core.defchararray.join(sep, seq)`

Return a string which is the concatenation of the strings in the sequence `seq`.

Calls `str.join` element-wise.

Parameters

- `sep` : array_like of str or unicode
- `seq` : array_like of str or unicode

Returns

- `out` : ndarray
  Output array of str or unicode, depending on input types

See Also:

- `str.join`

`numpy.core.defchararray.ljust(a, width, fillchar='')`

Return an array with the elements of `a` left-justified in a string of length `width`.

Calls `str.ljust` element-wise.
Parameters

- `a`: array_like of str or unicode
- `width`: int
  - The length of the resulting strings
- `fillchar`: str or unicode, optional
  - The character to use for padding

Returns

- `out`: ndarray
  - Output array of str or unicode, depending on input type

See Also:

- `str.ljust`
- `numpy.core.defchararray.lower(a)`
  - Return an array with the elements converted to lowercase.
  - Call `str.lower` element-wise.
  - For 8-bit strings, this method is locale-dependent.

Parameters

- `a`: array_like, {str, unicode}

Returns

- `out`: ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See Also:

- `str.lower`

Examples

```python
>>> c = np.array(['A1B C', '1BCA', 'BCA1']); c
array(['A1B C', '1BCA', 'BCA1'],
      dtype='|S5')
>>> np.char.lower(c)
array(['a1b c', '1bca', 'bca1'],
      dtype='|S5')
```

- `numpy.core.defchararray.lstrip(a, chars=None)`
  - For each element in `a`, return a copy with the leading characters removed.
  - Calls `str.lstrip` element-wise.

Parameters

- `a`: array-like, {str, unicode}
  - Input array.
- `chars`: {str, unicode}, optional
  - The `chars` argument is a string specifying the set of characters to be removed. If omitted or None, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix; rather, all combinations of its values are stripped.
NumPy Reference, Release 1.8.1

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type

See Also:

str.lstrip

Examples

>>> c = np.array(['aAaAaA', ' aA ', 'abABba'])
>>> c
array(['aAaAaA', ' aA ', 'abABba'],
      dtype='|S7')

The ‘a’ variable is unstripped from c[1] because whitespace leading.

>>> np.char.lstrip(c, 'a')
array(['AaAaA', ' aA ', 'bABba'],
      dtype='|S7')

>>> np.char.lstrip(c, 'A') # leaves c unchanged
array(['aAaAaA', ' aA ', 'abABba'],
      dtype='|S7')

>>> (np.char.lstrip(c, '') == np.char.lstrip(c, '')).all()  # XXX: is this a regression? this line now returns False
... # np.char.lstrip(c,'') does not modify c at all.
True

>>> (np.char.lstrip(c, '') == np.char.lstrip(c, None)).all()  # XXX: is this a regression? this line now returns False
True

numpy.core.defchararray.partition(a, sep)

Partition each element in a around sep.

Calls str.partition element-wise.

For each element in a, split the element as the first occurrence of sep, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

Parameters

a : array_like, {str, unicode}

Input array

sep : {str, unicode}

Separator to split each string element in a.

Returns

out : ndarray, {str, unicode}

Output array of str or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

See Also:

str.partition

numpy.core.defchararray.replace(a, old, new, count=None)

For each element in a, return a copy of the string with all occurrences of substring old replaced by new.

Calls str.replace element-wise.
Parameters
  a : array-like of str or unicode
  old, new : str or unicode
  count : int, optional
    If the optional argument count is given, only the first count occurrences are replaced.

Returns
  out : ndarray
    Output array of str or unicode, depending on input type

See Also:
  str.replace

numpy.core.defchararray.rjust (a, width, fillchar=' ')
  Return an array with the elements of a right-justified in a string of length width.
  Calls str.rjust element-wise.

Parameters
  a : array_like of str or unicode
  width : int
    The length of the resulting strings
  fillchar : str or unicode, optional
    The character to use for padding

Returns
  out : ndarray
    Output array of str or unicode, depending on input type

See Also:
  str.rjust

numpy.core.defchararray.rpartition (a, sep)
  Partition (split) each element around the right-most separator.
  Calls str.rpartition element-wise.

For each element in a, split the element as the last occurrence of sep, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

Parameters
  a : array_like of str or unicode
    Input array
  sep : str or unicode
    Right-most separator to split each element in array.

Returns
  out : ndarray
    Output array of string or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.
See Also:

str.rpartition

numpy.core.defchararray.rsplit (a, sep=None, maxsplit=None)

For each element in a, return a list of the words in the string, using sep as the delimiter string.

Calls str.rsplit element-wise.

Except for splitting from the right, rsplit behaves like split.

Parameters

a : array_like of str or unicode

sep : str or unicode, optional
    If sep is not specified or None, any whitespace string is a separator.

maxsplit : int, optional
    If maxsplit is given, at most maxsplit splits are done, the rightmost ones.

Returns

out : ndarray
    Array of list objects

See Also:

str.rsplit, split

numpy.core.defchararray.rstrip (a, chars=None)

For each element in a, return a copy with the trailing characters removed.

Calls str.rstrip element-wise.

Parameters

a : array-like of str or unicode

chars : str or unicode, optional
    The chars argument is a string specifying the set of characters to be removed. If omitted
    or None, the chars argument defaults to removing whitespace. The chars argument is
    not a suffix; rather, all combinations of its values are stripped.

Returns

out : ndarray
    Output array of str or unicode, depending on input type

See Also:

str.rstrip

Examples

>>> c = np.array(['AaaAaA', 'abABBa'], dtype='S7'); c
array(['AaaAaA', 'abABBa'],
      dtype='|S7')
>>> np.char.rstrip(c, 'a')
array(['AaaAaA', 'abABBa'],
      dtype='|S7')
>>> np.char.rstrip(c, 'A')
array(['AaaAaA', 'abABBa'],
      dtype='|S7')
numpy.core.defchararray.split \((a, \text{sep}=\text{None}, \text{maxsplit}=\text{None})\)
For each element in \(a\), return a list of the words in the string, using \(sep\) as the delimiter string.
Calls \(\text{str}.rsplit\) element-wise.

**Parameters**
- \(a\) : array_like of str or unicode
- \(\text{sep}\) : str or unicode, optional
  If \(\text{sep}\) is not specified or \(\text{None}\), any whitespace string is a separator.
- \(\text{maxsplit}\) : int, optional
  If \(\text{maxsplit}\) is given, at most \(\text{maxsplit}\) splits are done.

**Returns**
- \(\text{out}\) : ndarray
  Array of list objects

**See Also:**
- \(\text{str}.split, \text{rsplit}\)

numpy.core.defchararray.splitlines \((a, \text{keepends}=\text{None})\)
For each element in \(a\), return a list of the lines in the element, breaking at line boundaries.
Calls \(\text{str}.splitlines\) element-wise.

**Parameters**
- \(a\) : array_like of str or unicode
- \(\text{keepends}\) : bool, optional
  Line breaks are not included in the resulting list unless \(\text{keepends}\) is given and true.

**Returns**
- \(\text{out}\) : ndarray
  Array of list objects

**See Also:**
- \(\text{str}.splitlines\)

numpy.core.defchararray.strip \((a, \text{chars}=\text{None})\)
For each element in \(a\), return a copy with the leading and trailing characters removed.
Calls \(\text{str}.strip\) element-wise.

**Parameters**
- \(a\) : array-like of str or unicode
- \(\text{chars}\) : str or unicode, optional
  The \(\text{chars}\) argument is a string specifying the set of characters to be removed. If omitted or \(\text{None}\), the \(\text{chars}\) argument defaults to removing whitespace. The \(\text{chars}\) argument is not a prefix or suffix; rather, all combinations of its values are stripped.

**Returns**
- \(\text{out}\) : ndarray
  Output array of str or unicode, depending on input type

**See Also:**
- \(\text{str}.strip\)
Examples

```python
>>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
>>> c
array(['aAaAaA', ' aA ', 'abBABba'],
      dtype='|S7')
>>> np.char.strip(c)
array(['aAaAaA', 'aA ', 'abBABba'],
      dtype='|S7')
>>> np.char.strip(c, 'a')  # 'a' unstripped from c[1] because whitespace leads
array(['AaAaA', ' aA ', 'bBABb'],
      dtype='|S7')
>>> np.char.strip(c, 'A')  # 'A' unstripped from c[1] because (unprinted) ws trails
array(['aAaAa', ' aA ', 'abBABba'],
      dtype='|S7')
```

`numpy.core.defchararray.swapcase(a)`
Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa.

Calls `str.swapcase` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

- `a` : array_like, {str, unicode}
  Input array.

Returns

- `out` : ndarray, {str, unicode}
  Output array of str or unicode, depending on input type

See Also:

- `str.swapcase`

Examples

```python
>>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
      dtype='|S5')
>>> np.char.swapcase(c)
array(['A1b C', '1B cA', 'B cA1', 'Ca1B'],
      dtype='|S5')
```

`numpy.core.defchararray.title(a)`
Return element-wise title cased version of string or unicode.

Title case words start with uppercase characters, all remaining cased characters are lowercase.

Calls `str.title` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

- `a` : array_like, {str, unicode}
  Input array.

Returns

- `out` : ndarray
  Output array of str or unicode, depending on input type

See Also:

- `str.title`

Examples

```python
>>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
      dtype='|S5')
>>> np.char.title(c)
array(['A1B C', '1B cA', 'B cA1', 'Ca1B'],
      dtype='|S5')
```
See Also:

\texttt{str.title}

\textbf{Examples}

\begin{verbatim}
>>> c = np.array(['a1b c', 'lb ca', 'b ca', 'calb'], 'S5'); c
array(['a1b c', 'lb ca', 'b ca', 'calb'],
dtype='|S5')

>>> np.char.title(c)
array(['A1B C', '1B Ca', 'B Ca1', 'Ca1B'],
dtype='|S5')
\end{verbatim}

\texttt{numpy.core.defchararray.\texttt{translate}(a, table, deletechars=None)}

For each element in \(a\), return a copy of the string where all characters occurring in the optional argument \texttt{deletechars} are removed, and the remaining characters have been mapped through the given translation table.

Calls \texttt{str.translate} element-wise.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{a} : array-like of str or unicode
  \item \texttt{table} : str of length 256
  \item \texttt{deletechars} : str
\end{itemize}

\textbf{Returns}

\begin{itemize}
  \item \texttt{out} : ndarray
\end{itemize}

Output array of str or unicode, depending on input type

See Also:

\texttt{str.translate}

\texttt{numpy.core.defchararray.\texttt{upper}(a)}

Return an array with the elements converted to uppercase.

Calls \texttt{str.upper} element-wise.

For 8-bit strings, this method is locale-dependent.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{a} : array_like, \{str, unicode\}
\end{itemize}

\textbf{Returns}

\begin{itemize}
  \item \texttt{out} : ndarray, \{str, unicode\}
\end{itemize}

Output array of str or unicode, depending on input type

See Also:

\texttt{str.upper}

\textbf{Examples}

\begin{verbatim}
>>> c = np.array(['alb c', 'lbc a', 'bcal']); c
array(['alb c', 'lbc a', 'bcal'],
dtype='|S5')

>>> np.char.upper(c)
array(['ALB C', 'LBC A', 'BCAL'],
dtype='|S5')
\end{verbatim}
numpy.core.defchararray.zfill(a, width)
Return the numeric string left-filled with zeros
Calls str.zfill element-wise.

Parameters
  a : array_like, {str, unicode}
    Input array.
  width : int
    Width of string to left-fill elements in a.

Returns
  out : ndarray, {str, unicode}
    Output array of str or unicode, depending on input type

See Also:
  str.zfill

3.4.2 Comparison
Unlike the standard numpy comparison operators, the ones in the char module strip trailing whitespace characters before performing the comparison.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal(x1, x2)</td>
<td>Return (x1 == x2) element-wise.</td>
</tr>
<tr>
<td>not_equal(x1, x2)</td>
<td>Return (x1 != x2) element-wise.</td>
</tr>
<tr>
<td>greater_equal(x1, x2)</td>
<td>Return (x1 &gt;= x2) element-wise.</td>
</tr>
<tr>
<td>less_equal(x1, x2)</td>
<td>Return (x1 &lt;= x2) element-wise.</td>
</tr>
<tr>
<td>greater(x1, x2)</td>
<td>Return (x1 &gt; x2) element-wise.</td>
</tr>
<tr>
<td>less(x1, x2)</td>
<td>Return (x1 &lt; x2) element-wise.</td>
</tr>
</tbody>
</table>

numpy.core.defchararray.equal(x1, x2)
Return (x1 == x2) element-wise.

Unlike numpy.equal, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
  x1, x2 : array_like of str or unicode
    Input arrays of the same shape.

Returns
  out : {ndarray, bool}
    Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
  not_equal, greater_equal, less_equal, greater, less

numpy.core.defchararray.not_equal(x1, x2)
Return (x1 != x2) element-wise.

Unlike numpy.not_equal, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.
Parameters
   x1, x2 : array_like of str or unicode
   Input arrays of the same shape.

Returns
   out : {ndarray, bool}
   Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   equal, greater_equal, less_equal, greater, less

numpy.core.defchararray.greater_equal(x1, x2)
   Return (x1 >= x2) element-wise.

   Unlike numpy.greater_equal, this comparison is performed by first stripping whitespace characters from
   the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
   Input arrays of the same shape.

Returns
   out : {ndarray, bool}
   Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   equal, not_equal, less_equal, greater, less

numpy.core.defchararray.less_equal(x1, x2)
   Return (x1 <= x2) element-wise.

   Unlike numpy.less_equal, this comparison is performed by first stripping whitespace characters from the
   end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
   Input arrays of the same shape.

Returns
   out : {ndarray, bool}
   Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
   equal, not_equal, greater_equal, greater, less

numpy.core.defchararray.greater(x1, x2)
   Return (x1 > x2) element-wise.

   Unlike numpy.greater, this comparison is performed by first stripping whitespace characters from the end
   of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
   x1, x2 : array_like of str or unicode
   Input arrays of the same shape.

Returns
   out : {ndarray, bool}
Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
- `equal`
- `not_equal`
- `greater_equal`
- `less_equal`
- `less`

```
numpy.core.defchararray.less(x1, x2)
```

Return (x1 < x2) element-wise.

Unlike `numpy.greater`, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

Parameters
- `x1, x2` : array_like of str or unicode

Input arrays of the same shape.

Returns
- `out` : {ndarray, bool}

Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
- `equal`
- `not_equal`
- `greater_equal`
- `less_equal`
- `greater`

### 3.4.3 String information

**count(a, sub[, start, end])**
Returns an array with the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`.

Calls `str.count` element-wise.

Parameters
- `a` : array_like of str or unicode
- `sub` : str or unicode
  The substring to search for.
- `start, end` : int, optional
  Optional arguments `start` and `end` are interpreted as slice notation to specify the range in which to count.

```
numpy.core.defchararray.count(a, sub, start=0, end=None)
```
Returns

out : ndarray
    Output array of ints.

See Also:

str.count

Examples

>>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
>>> c
array(['aAaAaA', ' aA ', 'abBABba'],
      dtype='|S7')
>>> np.char.count(c, 'A')
array([3, 1, 1])
>>> np.char.count(c, 'aA')
array([3, 1, 0])
>>> np.char.count(c, 'A', start=1, end=4)
array([2, 1, 1])
>>> np.char.count(c, 'A', start=1, end=3)
array([1, 0, 0])

numpy.core.defchararray.find(a, sub=0, end=None)
    For each element, return the lowest index in the string where substring sub is found.
    Calls str.find element-wise.
    For each element, return the lowest index in the string where substring sub is found, such that sub is contained in the range [start, end].

    Parameters
    a : array_like of str or unicode
    sub : str or unicode
    start, end : int, optional
        Optional arguments start and end are interpreted as in slice notation.

    Returns
    out : ndarray or int
        Output array of ints. Returns -1 if sub is not found.

    See Also:

str.find

numpy.core.defchararray.index(a, sub=0, end=None)
    Like find, but raises ValueError when the substring is not found.
    Calls str.index element-wise.

    Parameters
    a : array_like of str or unicode
    sub : str or unicode
    start, end : int, optional

    Returns
    out : ndarray
        Output array of ints. Returns -1 if sub is not found.
See Also:

```
find, str.find
```

```
numpy.core.defchararray.isalpha(a)
```

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

Calls `str.isalpha` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

```
a : array_like of str or unicode
```

Returns

```
out : ndarray
```

Output array of bools

See Also:

```
str.isalpha
```

```
numpy.core.defchararray.isdecimal(a)
```

For each element, return True if there are only decimal characters in the element.

Calls `unicode.isdecimal` element-wise.

Decimal characters include digit characters, and all characters that that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

Parameters

```
a : array_like, unicode
```

Input array.

Returns

```
out : ndarray, bool
```

Array of booleans identical in shape to `a`.

See Also:

```
unicode.isdecimal
```

```
numpy.core.defchararray.isdigit(a)
```

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

Calls `str.isdigit` element-wise.

For 8-bit strings, this method is locale-dependent.

Parameters

```
a : array_like of str or unicode
```

Returns

```
out : ndarray
```

Output array of bools

See Also:

```
str.isdigit
```
numpy.core.defchararray.islower(a)
    Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

    Calls str.islower element-wise.

    For 8-bit strings, this method is locale-dependent.

    Parameters
        a : array_like of str or unicode

    Returns
        out : ndarray
            Output array of bools

    See Also:
        str.islower

numpy.core.defchararray.isnumeric(a)
    For each element, return True if there are only numeric characters in the element.

    Calls unicode.isnumeric element-wise.

    Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH.

    Parameters
        a : array_like, unicode
            Input array.

    Returns
        out : ndarray, bool
            Array of booleans of same shape as a.

    See Also:
        unicode.isnumeric

numpy.core.defchararray.isspace(a)
    Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

    Calls str.isspace element-wise.

    For 8-bit strings, this method is locale-dependent.

    Parameters
        a : array_like of str or unicode

    Returns
        out : ndarray
            Output array of bools

    See Also:
        str.isspace

numpy.core.defchararray.istitle(a)
    Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

    Call str.istitle element-wise.

3.4. String operations
For 8-bit strings, this method is locale-dependent.

**Parameters**

- `a` : array_like of str or unicode

**Returns**

- `out` : ndarray
  Output array of bools

**See Also:**

- `str.istitle`
- `numpy.core.defchararray.isupper(a)`
  Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

Call `str.isupper` element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

- `a` : array_like of str or unicode

**Returns**

- `out` : ndarray
  Output array of bools

**See Also:**

- `str.isupper`
- `numpy.core.defchararray.rfind(a, sub, start=0, end=None)`
  For each element in `a`, return the highest index in the string where substring `sub` is found, such that `sub` is contained within `[start, end]`.

Calls `str.rfind` element-wise.

**Parameters**

- `a` : array-like of str or unicode
- `sub` : str or unicode
- `start, end` : int, optional
  Optional arguments `start` and `end` are interpreted as in slice notation.

**Returns**

- `out` : ndarray
  Output array of ints. Return -1 on failure.

**See Also:**

- `str.rfind`
- `numpy.core.defchararray.rindex(a, sub, start=0, end=None)`
  Like `rfind`, but raises `ValueError` when the substring `sub` is not found.

Calls `str.rindex` element-wise.

**Parameters**

- `a` : array-like of str or unicode
- `sub` : str or unicode
start, end : int, optional

Returns
out : ndarray
Output array of ints.

See Also:
rfind, str.rindex

numpy.core.defchararray.startswith(a, prefix, start=0, end=None)
Returns a boolean array which is True where the string element in a starts with prefix, otherwise False.
Calls str.startswith element-wise.

Parameters
a : array_like of str or unicode
prefix : str
start, end : int, optional
With optional start, test beginning at that position. With optional end, stop comparing at that position.

Returns
out : ndarray
Array of booleans

See Also:
str.startswith

3.4.4 Convenience class

class numpy.core.defchararray.chararray
Provides a convenient view on arrays of string and unicode values.

Note: The chararray class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of dtype object_, string_ or unicode_, and use the free functions in the numpy.char module for fast vectorized string operations.

Versus a regular Numpy array of type str or unicode, this class adds the following functionality:
1.values automatically have whitespace removed from the end when indexed
2.comparison operators automatically remove whitespace from the end when comparing values
3.vectorized string operations are provided as methods (e.g. endswith) and infix operators (e.g. "+", "*", "%")
chararrays should be created using numpy.char.array or numpy.char.asarray, rather than this constructor directly.
This constructor creates the array, using buffer (with offset and strides) if it is not None. If buffer

3.4. String operations
is None, then constructs a new array with strides in “C order”, unless both len(shape) >= 2 and order=’Fortran’, in which case strides is in “Fortran order”.

**Parameters**

- **shape**: tuple
  Shape of the array.

- **itemsize**: int, optional
  Length of each array element, in number of characters. Default is 1.

- **unicode**: bool, optional
  Are the array elements of type unicode (True) or string (False). Default is False.

- **buffer**: int, optional
  Memory address of the start of the array data. Default is None, in which case a new array is created.

- **offset**: int, optional
  Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

- **strides**: array_like of ints, optional
  Strides for the array (see ndarray.strides for full description). Default is None.

- **order**: {'C', 'F'}, optional
  The order in which the array data is stored in memory: ‘C’ -> “row major” order (the default), ‘F’ -> “column major” (Fortran) order.

**Examples**

```python
guar = np.chararray((3, 3))
guar[:] = ‘a’
guar
chararray(['a', 'a', 'a'], ['a', 'a', 'a'], ['a', 'a', 'a'], dtype='|S1')
guar = np.chararray(guar.shape, itemsize=5)
guar[:] = ‘abc’
guar
chararray(['abc', 'abc', 'abc'], ['abc', 'abc', 'abc'], ['abc', 'abc', 'abc'], dtype='|S5')
```

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Same as self.transpose(), except that self is returned if self.ndim &lt; 2.</td>
</tr>
<tr>
<td>base</td>
<td>Base object if memory is from some other object.</td>
</tr>
<tr>
<td>ctypes</td>
<td>An object to simplify the interaction of the array with the ctypes module.</td>
</tr>
<tr>
<td>data</td>
<td>Python buffer object pointing to the start of the array’s data.</td>
</tr>
<tr>
<td>dtype</td>
<td>Data-type of the array’s elements.</td>
</tr>
<tr>
<td>flags</td>
<td>Information about the memory layout of the array.</td>
</tr>
<tr>
<td>flat</td>
<td>A T-D iterator over the array.</td>
</tr>
</tbody>
</table>
Table 3.27 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imag</td>
<td>The imaginary part of the array.</td>
</tr>
<tr>
<td>itemsize</td>
<td>Length of one array element in bytes.</td>
</tr>
<tr>
<td>nbytes</td>
<td>Total bytes consumed by the elements of the array.</td>
</tr>
<tr>
<td>ndim</td>
<td>Number of array dimensions.</td>
</tr>
<tr>
<td>real</td>
<td>The real part of the array.</td>
</tr>
<tr>
<td>shape</td>
<td>Tuple of array dimensions.</td>
</tr>
<tr>
<td>size</td>
<td>Number of elements in the array.</td>
</tr>
<tr>
<td>strides</td>
<td>Tuple of bytes to step in each dimension when traversing an array.</td>
</tr>
</tbody>
</table>

```python
>>> x = np.array([[1., 2.], [3., 4.]])
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
```

```python
>>> x = np.array([1., 2., 3., 4.])
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

**chararray.T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

**Examples**

```python
>>> x = np.array([[1., 2.],[3.,4.]])
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
```

**chararray.base**

Base object if memory is from some other object.

**Examples**

The base of an array that owns its memory is None:

```python
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
>>> y = x[2:]
>>> y.base is x
True
```

**chararray.ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

None

**Returns**

- c : Python object
  - Possessing attributes data, shape, strides, etc.
See Also:

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

• **data**: A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

• **shape (c_intp*self.ndim)**: A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype(‘p’) on this platform. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

• **strides (c_intp*self.ndim)**: A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

• **data_as(obj)**: Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

• **shape_as(obj)**: Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

• **strides_as(obj)**: Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling (a+b).ctypes.data_as(ctypes.c_void_p) returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either c=a+b or ct=(a+b).ctypes. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
```
chararray.data
Python buffer object pointing to the start of the array’s data.

chararray.dtype
Data-type of the array’s elements.

Parameters
None

Returns
d : numpy dtype object

See Also:
numpy.dtype

Examples
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype(‘int32’)
>>> type(x.dtype)
<type ‘numpy.dtype’>

chararray.flags
Information about the memory layout of the array.

Notes
The flags object can be accessed dictionary-like (as in a.flags[‘WRITEABLE’]), or by using lowercase attribute names (as in a.flags.writeable). Short flag names are only supported in dictionary access.

Only the UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling ndarray.setflags.

The array flags cannot be set arbitrarily:

• UPDATEIFCOPY can only be set False.
• ALIGNED can only be set True if the data is truly aligned.
• WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension arr.strides[dim] may be arbitrary if arr.shape[dim] == 1 or the array has no elements. It does not generally hold that self.strides[-1] == self.itemsize for C-style contiguous arrays or self.strides[0] == self.itemsize for Fortran-style contiguous arrays is true.
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CONTIGUOUS (C)</td>
<td>The data is in a single, C-style contiguous segment.</td>
</tr>
<tr>
<td>F_CONTIGUOUS (F)</td>
<td>The data is in a single, Fortran-style contiguous segment.</td>
</tr>
<tr>
<td>OWN_DATA (O)</td>
<td>The array owns the memory it uses or borrows it from another object.</td>
</tr>
<tr>
<td>WRITEABLE (W)</td>
<td>The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a Runtime Exception.</td>
</tr>
<tr>
<td>ALIGNED (A)</td>
<td>The data and all elements are aligned appropriately for the hardware.</td>
</tr>
<tr>
<td>UPDATEIFCOPY (U)</td>
<td>This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.</td>
</tr>
<tr>
<td>FNC</td>
<td>F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FORC</td>
<td>F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).</td>
</tr>
<tr>
<td>BEHAVED (B)</td>
<td>ALIGNED and WRITEABLE.</td>
</tr>
<tr>
<td>CARRAY (CA)</td>
<td>BEHAVED and C_CONTIGUOUS.</td>
</tr>
<tr>
<td>FARRAY (FA)</td>
<td>BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.</td>
</tr>
</tbody>
</table>

**chararray.flat**

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

**See Also:**

**flatten**

Return a copy of the array collapsed into one dimension.

**Examples**

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```
>>> x.T.flat[3]
5
>>> type(x.flat)
<type 'numpy.flatiter'>

An assignment example:

```python
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

**chararray.imag**
The imaginary part of the array.

**Examples**
```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

**chararray.itemsize**
Length of one array element in bytes.

**Examples**
```python
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

**chararray.nbytes**
Total bytes consumed by the elements of the array.

**Notes**
Does not include memory consumed by non-element attributes of the array object.

**Examples**
```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

**chararray.ndim**
Number of array dimensions.

**Examples**
```python
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
```
```
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3

chararray.real
The real part of the array.

See Also:

numpy.real
equivalent function

Examples
```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

chararray.shape
Tuple of array dimensions.

Notes
May be used to “reshape” the array, as long as this would not require a change in the total number of elements

Examples
```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

chararray.size
Number of elements in the array.

Equivalent to np.prod(a.shape), i.e., the product of the array’s dimensions.

Examples
```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```
chararray\texttt{.strides}
Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element \((i[0], i[1], \ldots, i[n])\) in an array \(a\) is:

\[
\text{offset} = \sum (\text{np.array}(i) \times a.\text{strides})
\]

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

See Also:
numpy.lib.stride_tricks.as_strided

Notes
Imagine an array of 32-bit integers (each 4 bytes):

\[
x = \text{np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]], \text{dtype}=\text{np.int32})}
\]

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array \(x\) will be \((20, 4)\).

Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17
```

```python
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

Methods

\begin{itemize}
\item \texttt{astype}([\text{dtype}, \text{order}, \text{casting}, \text{subok}, \text{copy}]) Copy of the array, cast to a specified type.
\item \texttt{copy}([\text{order}]) Return a copy of the array.
\end{itemize}
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count(sub[, start, end])</code></td>
<td>Returns an array with the number of non-overlapping occurrences of substring <code>sub</code> in <code>self</code>.</td>
</tr>
<tr>
<td><code>decode([encoding, errors])</code></td>
<td>Calls <code>str.decode</code> element-wise.</td>
</tr>
<tr>
<td><code>dump(file)</code></td>
<td>Dump a pickle of the array to the specified file.</td>
</tr>
<tr>
<td><code>dumps()</code></td>
<td>Returns the pickled string of the array.</td>
</tr>
<tr>
<td><code>encode([encoding, errors])</code></td>
<td>Calls <code>str.encode</code> element-wise.</td>
</tr>
<tr>
<td><code>endswith(suffix[, start, end])</code></td>
<td>Returns a boolean array which is True where the string element ends with <code>suffix</code>.</td>
</tr>
<tr>
<td><code>expandtabs(tabsize)</code></td>
<td>Return a copy of each string element where all tab characters are replaced by one or more spaces.</td>
</tr>
<tr>
<td><code>fill(value)</code></td>
<td>Fill the array with the specified scalar value.</td>
</tr>
<tr>
<td><code>find(sub[, start, end])</code></td>
<td>For each element, return the lowest index in the string where substring <code>sub</code> is found.</td>
</tr>
<tr>
<td><code>flatten(order)</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>getfield(dtype[, offset])</code></td>
<td>Returns a field of the given array as a certain type.</td>
</tr>
<tr>
<td><code>index(sub[, start, end])</code></td>
<td>Like <code>find</code>, but raises <code>ValueError</code> when the substring is not found.</td>
</tr>
<tr>
<td><code>isalnum()</code></td>
<td>Returns true for each element if all characters in the string are alphanumeric and there are at least one character.</td>
</tr>
<tr>
<td><code>isalpha()</code></td>
<td>Returns true for each element if all characters in the string are alphabetic and there is at least one character.</td>
</tr>
<tr>
<td><code>isdecimal()</code></td>
<td>For each element in <code>self</code>, return True if there are only decimal digits and there is at least one character.</td>
</tr>
<tr>
<td><code>isdigit()</code></td>
<td>Returns true for each element if all characters in the string are digits and there is at least one character.</td>
</tr>
<tr>
<td><code>islower()</code></td>
<td>Returns true for each element if all cased characters in the string are lowercase and there is at least one character.</td>
</tr>
<tr>
<td><code>isspace()</code></td>
<td>Returns true for each element if there are only whitespace characters in the string and there is at least one character.</td>
</tr>
<tr>
<td><code>isnumeric()</code></td>
<td>For each element in <code>self</code>, return True if there are only numeric characters and there is at least one character.</td>
</tr>
<tr>
<td><code>isupper()</code></td>
<td>Returns true for each element if all cased characters in the string are uppercase and there is at least one character.</td>
</tr>
<tr>
<td><code>item(*args)</code></td>
<td>Copy an element of an array to a standard Python scalar and return it.</td>
</tr>
<tr>
<td><code>join(seq)</code></td>
<td>Return a string which is the concatenation of the strings in the sequence <code>seq</code>.</td>
</tr>
<tr>
<td><code>ljust(width[, fillchar])</code></td>
<td>Return an array with the elements of <code>self</code> left-justified in a string of length <code>width</code>.</td>
</tr>
<tr>
<td><code>lower()</code></td>
<td>Return an array with the elements of <code>self</code> converted to lowercase.</td>
</tr>
<tr>
<td><code>lstrip(chars)</code></td>
<td>For each element in <code>self</code>, return a copy with the leading characters removed.</td>
</tr>
<tr>
<td><code>nonzero()</code></td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td><code>put(indices, values[, mode])</code></td>
<td>Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.</td>
</tr>
<tr>
<td><code>ravel(order)</code></td>
<td>Return a flattened array.</td>
</tr>
<tr>
<td><code>repeat(repeats[, axis])</code></td>
<td>Repeat elements of an array.</td>
</tr>
<tr>
<td><code>replace(old, new[, count])</code></td>
<td>For each element in <code>self</code>, return a copy of the string with all occurrences of substring <code>old</code> replaced by substring <code>new</code>.</td>
</tr>
<tr>
<td><code>reshape(shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>resize(new_shape[, refcheck])</code></td>
<td>Change shape and size of array in-place.</td>
</tr>
<tr>
<td><code>rfind(sub[, start, end])</code></td>
<td>For each element in <code>self</code>, return the highest index in the string where substring <code>sub</code> is found.</td>
</tr>
<tr>
<td><code>rindex(sub[, start, end])</code></td>
<td>Like <code>rfind</code>, but raises <code>ValueError</code> when the substring <code>sub</code> is not found.</td>
</tr>
<tr>
<td><code>rjust(width[, fillchar])</code></td>
<td>Return an array with the elements of <code>self</code> right-justified in a string of length <code>width</code>.</td>
</tr>
<tr>
<td><code>rsplit(sep[, maxsplit])</code></td>
<td>For each element in <code>self</code>, return a list of the words in the string, using <code>sep</code> as the delimiter.</td>
</tr>
<tr>
<td><code>rstrip(chars)</code></td>
<td>For each element in <code>self</code>, return a copy with the trailing characters removed.</td>
</tr>
<tr>
<td><code>searchsorted(v[, side, sorter])</code></td>
<td>Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.</td>
</tr>
<tr>
<td><code>setfield(val, dtype[, offset])</code></td>
<td>Put a value into a specified place in a field defined by a data-type.</td>
</tr>
<tr>
<td><code>setflags([write, align, uic])</code></td>
<td>Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.</td>
</tr>
<tr>
<td><code>sort([axis, kind, order])</code></td>
<td>Sort an array, in-place.</td>
</tr>
<tr>
<td><code>split(sep, maxsplit)</code></td>
<td>For each element in <code>self</code>, return a list of the words in the string, using <code>sep</code> as the delimiter.</td>
</tr>
<tr>
<td><code>splitlines([keepends])</code></td>
<td>For each element in <code>self</code>, return a list of the lines in the element, breaking at line boundaries.</td>
</tr>
<tr>
<td><code>squeeze(axis)</code></td>
<td>Remove single-dimensional entries from the shape of <code>a</code>.</td>
</tr>
<tr>
<td><code>startswith(prefix[, start, end])</code></td>
<td>Returns a boolean array which is True where the string element starts with <code>prefix</code>.</td>
</tr>
<tr>
<td><code>strip(chars)</code></td>
<td>For each element in <code>self</code>, return a copy with the leading and trailing characters removed.</td>
</tr>
<tr>
<td><code>swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with <code>axis1</code> and <code>axis2</code> interchanged.</td>
</tr>
<tr>
<td><code>swapcase()</code></td>
<td>For each element in <code>self</code>, return a copy of the string with uppercase characters converted to lowercase.</td>
</tr>
<tr>
<td><code>take(indices[, axis, out, mode])</code></td>
<td>Return an array formed from the elements of <code>a</code> at the given indices.</td>
</tr>
</tbody>
</table>
For each element in `self`, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

Write array to a file as text or binary (default).

Return the array as a (possibly nested) list.

Construct a Python string containing the raw data bytes in the array.

For each element in `self`, return a copy of the string where all characters occurring in the optional argument `deletechars` are removed, and the remaining characters have been mapped through the given translation table.

Returns a view of the array with axes transposed.

Return an array with the elements of `self` converted to uppercase.

New view of array with the same data.

Return the numeric string left-filled with zeros in a string of length `width`.

chararray.astype (dtype, order=’K’, casting=’unsafe’, subok=True, copy=True)
Copy of the array, cast to a specified type.

Parameters

dtype : str or dtype
Typecode or data-type to which the array is cast.

Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.

• ‘no’ means the data types should not be cast at all.

• ‘equiv’ means only byte-order changes are allowed.

• ‘safe’ means only casts which can preserve values are allowed.

• ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.

• ‘unsafe’ means any data conversions may be done.

subok : bool, optional
If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy : bool, optional
By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t : ndarray

Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), `arr_t` is a new array of the same shape as the input array, with dtype, order given by `dtype`, `order`.

Raises
ComplexWarning

3.4. String operations
When casting from complex to float or int, To avoid this, one should use a.real.astype(t).

**Examples**

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

`chararray.copy(order='C')`

Return a copy of the array.

**Parameters**

`order`: {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their order= arguments.)

**See Also:**

`numpy.copy`, `numpy.copyto`

**Examples**

```python
>>> x = np.array([[1,2,3],[4,5,6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

`chararray.count(sub, start=0, end=None)`

Returns an array with the number of non-overlapping occurrences of substring sub in the range [start, end].

**See Also:**

`char.count`

`chararray.decode(encoding=None, errors=None)`

Calls `str.decode` element-wise.

**See Also:**

`char.decode`
chararray.\texttt{dump} (file)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

\begin{verbatim}
Parameters
file : str
A string naming the dump file.
\end{verbatim}

chararray.\texttt{dumps}()
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

\begin{verbatim}
Parameters
None
\end{verbatim}

chararray.\texttt{encode} (encoding=None, errors=None)
Calls \texttt{str.encode} element-wise.

\begin{verbatim}
See Also:
char.encode
\end{verbatim}

chararray.\texttt{endswith} (suffix, start=0, end=None)
Returns a boolean array which is \texttt{True} where the string element in \texttt{self} ends with \texttt{suffix}, otherwise \texttt{False}.

\begin{verbatim}
See Also:
char.endswith
\end{verbatim}

chararray.\texttt{expandtabs} (tabsize=8)
Return a copy of each string element where all tab characters are replaced by one or more spaces.

\begin{verbatim}
See Also:
char.expandtabs
\end{verbatim}

chararray.\texttt{fill} (value)
Fill the array with a scalar value.

\begin{verbatim}
Parameters
value : scalar
All elements of \texttt{a} will be assigned this value.
\end{verbatim}

\textbf{Examples}

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
```

```
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1., 1.])
```

chararray.\texttt{find} (sub, start=0, end=None)
For each element, return the lowest index in the string where substring \texttt{sub} is found.

\begin{verbatim}
See Also:
char.find
\end{verbatim}

chararray.\texttt{flatten} (order='C')
Return a copy of the array collapsed into one dimension.

\section*{3.4. String operations}
Parameters

order : {'C', 'F', 'A'}, optional

Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from a. The default is ‘C’.

Returns

y : ndarray

A copy of the input array, flattened to one dimension.

See Also:

ravel
Return a flattened array.

flat
A 1-D flat iterator over the array.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

chararray.getfield(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype : str or dtype

The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset : int

Number of bytes to skip before beginning the element view.

Examples

```python
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[ 1.+1.j, 0.+0.j],
       [ 0.+0.j, 2.+4.j]])
>>> x.getfield(np.float64)
array([[ 1., 0.],
       [ 0., 2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```python
>>> x.getfield(np.float64, offset=8)
array([[ 1., 0.],
       [ 0., 4.]])
```
chararray.index \( \text{(sub, start=0, end=None)} \)
Like `find`, but raises `ValueError` when the substring is not found.

**See Also:**
char.index

chararray.isalnum()
Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

**See Also:**
char.isalnum

chararray.isalpha()
Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

**See Also:**
char.isalpha

chararray.isdecimal()
For each element in `self`, return True if there are only decimal characters in the element.

**See Also:**
char.isdecimal

chararray.isdigit()
Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

**See Also:**
char.isdigit

chararray.islower()
Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

**See Also:**
char.islower

chararray.isnumeric()
For each element in `self`, return True if there are only numeric characters in the element.

**See Also:**
char.isnumeric

chararray.isspace()
Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

**See Also:**
char.isspace

chararray.istitle()
Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

**See Also:**
char.istitle

chararray.isupper()

Returns true for each element if all cased characters in the string are uppercase and there is at least one character. False otherwise.

See Also:
char.isupper

chararray.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters
*args : Arguments (variable number and type)
  • none: in this case, the method only works for arrays with one element (a.size == 1), which element is copied into a standard Python scalar object and returned.
  • int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
  • tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns
z : Standard Python scalar object
    A copy of the specified element of the array as a suitable Python scalar

Notes
When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python’s optimized math.

Examples
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3

chararray.join(seq)

Return a string which is the concatenation of the strings in the sequence seq.

See Also:
char.join
chararray\texttt{.ljust}(width,\texttt{fillchar}=\textquote{\textasciitilde})

Return an array with the elements of \texttt{self} left-justified in a string of length \texttt{width}.

\textbf{See Also:}

\texttt{char.ljust}

chararray\texttt{.lower}()

Return an array with the elements of \texttt{self} converted to lowercase.

\textbf{See Also:}

\texttt{char.lower}

chararray\texttt{.lstrip}(\texttt{chars}=\texttt{None})

For each element in \texttt{self}, return a copy with the leading characters removed.

\textbf{See Also:}

\texttt{char.lstrip}

chararray\texttt{.nonzero}()

Return the indices of the elements that are non-zero.

Refer to \texttt{numpy.nonzero} for full documentation.

\textbf{See Also:}

\texttt{numpy.nonzero}

equivalent function

chararray\texttt{.put}(\texttt{indices},\texttt{values},\texttt{mode}=\textquote{\texttt{raise}})

Set \texttt{a.flat[n]} = \texttt{values[n]} for all \texttt{n} in \texttt{indices}.

Refer to \texttt{numpy.put} for full documentation.

\textbf{See Also:}

\texttt{numpy.put}

equivalent function

chararray\texttt{.ravel}(\texttt{order})

Return a flattened array.

Refer to \texttt{numpy.ravel} for full documentation.

\textbf{See Also:}

\texttt{numpy.ravel}

equivalent function

\texttt{ndarray.flat}

a flat iterator on the array.

chararray\texttt{.repeat}(\texttt{repeats},\texttt{axis}=\texttt{None})

Repeat elements of an array.

Refer to \texttt{numpy.repeat} for full documentation.

\textbf{See Also:}

\texttt{numpy.repeat}

equivalent function
chararray.replace(old, new, count=None)
For each element in self, return a copy of the string with all occurrences of substring old replaced by new.

See Also:
char.replace

chararray.reshape(shape, order='C')
Returns an array containing the same data with a new shape.

Refer to numpy.reshape for full documentation.

See Also:
numpy.reshape
equivalent function

chararray.resize(new_shape, refcheck=True)
Change shape and size of array in-place.

Parameters
new_shape : tuple of ints, or n ints
Shape of resized array.
refcheck : bool, optional
If False, reference count will not be checked. Default is True.

Returns
None

Raises
ValueError
If a does not own its own data or references or views to it exist, and the data memory
must be changed.

SystemError
If the order keyword argument is specified. This behaviour is a bug in NumPy.

See Also:
resize
Return a new array with the specified shape.

Notes
This reallocates space for the data area if necessary.
Only contiguous arrays (data elements consecutive in memory) can be resized.
The purpose of the reference count check is to make sure you do not use this array as a buffer for another
Python object and then reallocate the memory. However, reference counts can increase in other ways so if
you are sure that you have not shared the memory for this array with another Python object, then you may
safely set refcheck to False.

Examples
Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-
shaped:
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a
array([[0, 1],
       [2, 3]])

>>> a.resize((2, 1))
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a
array([[0, 1],
       [2, 3]])

>>> a.resize((2, 1))
array([[0],
       [2]])

Enlarging an array: as above, but missing entries are filled with zeros:

>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn’t have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])

Referencing an array prevents resizing...

>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
  ...
ValueError: cannot resize an array that has been referenced ...

Unless refcheck is False:

>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])

chararray.rfind(sub, start=0, end=None)
For each element in self, return the highest index in the string where substring sub is found, such that sub is contained within [start, end].

See Also:
char.rfind

chararray.rindex(sub, start=0, end=None)
Like rfind, but raises ValueError when the substring sub is not found.

See Also:
char.rindex

chararray.rjust(width, fillchar=’ ’)
Return an array with the elements of self right-justified in a string of length width.

See Also:
char.rjust

chararray.rsplit(sep=None, maxsplit=None)
For each element in self, return a list of the words in the string, using sep as the delimiter string.

See Also:
char.rsplit
chararray.rstrip(chars=None)
For each element in self, return a copy with the trailing characters removed.

See Also:
char.rsplit

chararray.searchsorted(v, side='left', sorter=None)
Find indices where elements of v should be inserted in a to maintain order.
For full documentation, see numpy.searchsorted

See Also:

numpy.searchsorted
equivalent function

chararray.setfield(val, dtype, offset=0)
Put a value into a specified place in a field defined by a data-type.
Place val into a’s field defined by dtype and beginning offset bytes into the field.

Parameters
val : object
    Value to be placed in field.
dtype : dtype object
    Data-type of the field in which to place val.
offset : int, optional
    The number of bytes into the field at which to place val.

Returns
None

See Also:
getfield

Examples
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[ 1., 0., 0.],
[ 0., 1., 0.],
[ 0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
[3, 3, 3],
[3, 3, 3]])
>>> x
array([[ 1.00000000e+000, 1.48219694e-323, 1.48219694e-323],
[ 1.48219694e-323, 1.0000000000e+000, 1.48219694e-323],
[ 1.48219694e-323, 1.48219694e-323, 1.0000000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1., 0., 0.],
chararray.setflags(write=None, align=None, uic=None)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by a (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write : bool, optional
    Describes whether or not a can be written to.

align : bool, optional
    Describes whether or not a is aligned properly for its type.

uic : bool, optional
    Describes whether or not a is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

All flags can be accessed using their first (upper case) letter as well as the full name.

Examples

```python
>>> y
array([[ 3.,  1.,  7.],
       [ 2.,  0.,  0.],
       [ 8.,  5.,  9.]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
UPDATEIFCOPY : False
```
chararray.sort (axis=-1, kind='quicksort', order=None)
Sort an array, in-place.

Parameters
axis : int, optional
    Axis along which to sort. Default is -1, which means sort along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm. Default is 'quicksort'.

order : list, optional
    When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:
numpy.sort
    Return a sorted copy of an array.
argsort
    Indirect sort.
lexsort
    Indirect stable sort on multiple keys.
searchsorted
    Find elements in sorted array.
partition
    Partial sort.

Notes
See sort for notes on the different sorting algorithms.

Examples
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [3, 1]])

Use the order keyword to specify a field to use when sorting a structured array:

>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
chararray.split(sep=None, maxsplit=None)
For each element in self, return a list of the words in the string, using sep as the delimiter string.

See Also:
char.split

cchararray.splitlines(keepends=None)
For each element in self, return a list of the lines in the element, breaking at line boundaries.

See Also:
char.splitlines

cchararray.squeeze(axis=None)
Remove single-dimensional entries from the shape of a.
Refer to numpy.squeeze for full documentation.

See Also:
numpy.squeeze
equivalent function

cchararray.startswith(prefix, start=0, end=None)
Returns a boolean array which is True where the string element in self starts with prefix, otherwise False.

See Also:
char.startswith

cchararray.strip(chars=None)
For each element in self, return a copy with the leading and trailing characters removed.

See Also:
char.strip

cchararray.swapaxes(axis1, axis2)
Return a view of the array with axis1 and axis2 interchanged.
Refer to numpy.swapaxes for full documentation.

See Also:
numpy.swapaxes
equivalent function

cchararray.swapcase()
For each element in self, return a copy of the string with uppercase characters converted to lowercase and vice versa.

See Also:
char.swapcase

cchararray.take(indices, axis=None, out=None, mode='raise')
Return an array formed from the elements of a at the given indices.
Refer to numpy.take for full documentation.

See Also:
**numpy.take**

Equivalent function

**chararray.title()**

For each element in `self`, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

**See Also:**

`char.title`

**chararray.tofile** *(`fid`, `sep``=``""`, `format``=``"%s"`)*

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of `a`. The data produced by this method can be recovered using the function fromfile().

**Parameters**

- `fid`: file or str
  
  An open file object, or a string containing a filename.

- `sep`: str
  
  Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tostring())`.

- `format`: str
  
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

**Notes**

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

**chararray.tolist()**

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

**Parameters**

- `none`

**Returns**

- `y`: list
  
  The possibly nested list of array elements.

**Notes**

The array may be recreated, `a = np.array(a.tolist())`.

**Examples**

```python
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
```
>>> list(a)
[array([[1, 2]], [3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]

cchararray.tostring(order='C')
Construct a Python string containing the raw data bytes in the array.

Constructs a Python string showing a copy of the raw contents of data memory. The string can be produced in either ‘C’ or ‘Fortran’, or ‘Any’ order (the default is ‘C’-order). ‘Any’ order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means ‘Fortran’ order.

Parameters
order : {‘C’, ‘F’, None}, optional
Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns
s : str
A Python string exhibiting a copy of a’s raw data.

Examples
>>> x = np.array([[0, 1], [2, 3]])
>>> x.tostring()
'\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x03\x00\x00\x00'
>>> x.tostring('C') == x.tostring()
True
>>> x.tostring('F')
'\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x03\x00\x00\x00'

cchararray.translate(table, deletechars=None)
For each element in self, return a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table.

See Also:
char.translate
cchararray.transpose(*axes)
Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters
axes : None, tuple of ints, or n ints
- None or no argument: reverses the order of the axes.
- tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
- n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)
Returns

out : ndarray

View of $a$, with axes suitably permuted.

See Also:

ndarray.T

Array property returning the array transposed.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

chararray.upper()

Return an array with the elements of self converted to uppercase.

See Also:

char.upper

chararray.view(dtype=None, type=None)

New view of array with the same data.

Parameters

dtype : data-type or ndarray sub-class, optional

Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as $a$. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the type parameter).

type : Python type, optional

Type of the returned view, e.g., ndarray or matrix. Again, the default None results in type preservation.

Notes

a.view() is used two different ways:

a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For a.view(some_dtype), if some_dtype has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view
cannot be predicted just from the superficial appearance of a (shown by `print(a)`). It also depends on exactly how a is stored in memory. Therefore if a is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

**Examples**

```python
>>> x = np.array([[1, 2]], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:
```n
```python
defprint(y)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Creating a view on a structured array so it can be used in calculations
```python
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([ 2., 3.])
```

Making changes to the view changes the underlying array
```python
>>> xv[0,1] = 20
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a record array:
```python
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:
```python
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:
```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=('width', np.int16), ('length', np.int16))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:
chararray.zfill(width)
   Return the numeric string left-filled with zeros in a string of length width.

   See Also:
   char.zfill

3.5 C-Types Foreign Function Interface (numpy.ctypeslib)

numpy.ctypeslib.as_array(obj, shape=None)
   Create a numpy array from a ctypes array or a ctypes POINTER. The numpy array shares the memory with the
cypes object.
   The size parameter must be given if converting from a ctypes POINTER. The size parameter is ignored if
converting from a ctypes array

numpy.ctypeslib.as_ctypes(obj)
   Create and return a ctypes object from a numpy array. Actually anything that exposes the __array_interface__
is accepted.

numpy.ctypeslib.ctypes_load_library(*args, **kwds)
   ctypes_load_library is deprecated, use load_library instead!

numpy.ctypeslib.load_library(libname, loader_path)

numpy.ctypeslib.ndpointer(dtype=None, ndim=None, shape=None, flags=None)
   Array-checking restype/argtypes.
   An ndpointer instance is used to describe an ndarray in restypes and argtypes specifications. This approach
is more flexible than using, for example, POINTER(c_double), since several restrictions can be specified,
which are verified upon calling the ctypes function. These include data type, number of dimensions, shape and
flags. If a given array does not satisfy the specified restrictions, a TypeError is raised.

   Parameters
   dtype : data-type, optional
      Array data-type.
   ndim : int, optional
      Number of array dimensions.
   shape : tuple of ints, optional
      Array shape.
   flags : str or tuple of str
      Array flags; may be one or more of:
      • C_CONTIGUOUS / C / CONTIGUOUS
      • F_CONTIGUOUS / F / FORTRAN
      • OWndata / O
      • WRITEABLE / W
      • ALIGNED / A
      • UPDATEIFCOPY / U
Returns

klass : ndarray type object

A type object, which is an _ndtp object containing dtype, ndim, shape and flags information.

 Raises

TypeError

If a given array does not satisfy the specified restrictions.

Examples

```python
>>> clib.somefunc.argtypes = [np.ctypeslib.ndpointer(dtype=np.float64,
... ndim=1,
... flags='C_CONTIGUOUS')]
...

>>> clib.somefunc(np.array([1, 2, 3], dtype=np.float64))
...
```

3.6 Datetime Support Functions

3.6.1 Business Day Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>busdaycalendar</td>
<td>A business day calendar object that efficiently stores information defining valid days.</td>
</tr>
<tr>
<td>is_busday</td>
<td>Calculates which of the given dates are valid days, and which are not.</td>
</tr>
<tr>
<td>busday_offset</td>
<td>First adjusts the date to fall on a valid day according to the roll rule, then applies offsets.</td>
</tr>
<tr>
<td>busday_count</td>
<td>Counts the number of valid days between begindates and enddates, not including the enddates.</td>
</tr>
</tbody>
</table>

The default valid days are Monday through Friday (“business days”). A busdaycalendar object can be specified with any set of weekly valid days, plus an optional “holiday” dates that always will be invalid.

Once a busdaycalendar object is created, the weekmask and holidays cannot be modified. New in version 1.7.0.

Parameters

- **weekmask** : str or array_like of bool, optional
  A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,0,0]; a length-seven string, like ‘111100’; or a string like “Mon Tue Wed Thu Fri”, made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

- **holidays** : array_like of datetime64[D], optional
  An array of dates to consider as invalid dates, no matter which weekday they fall upon. Holiday dates may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

Returns

- **out** : busdaycalendar
  A business day calendar object containing the specified weekmask and holidays values.
See Also:

**is_busday**
Returns a boolean array indicating valid days.

**busday_offset**
Applies an offset counted in valid days.

**busday_count**
Counts how many valid days are in a half-open date range.

Examples

```python
>>> # Some important days in July
... bdd = np.busdaycalendar(
...     holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
>>> # Default is Monday to Friday weekdays
... bdd.weekmask
array([True, True, True, True, True, False, False], dtype='bool')
>>> # Any holidays already on the weekend are removed
... bdd.holidays
array(['2011-07-01', '2011-07-04'], dtype='datetime64[D]')
```

Attributes

```python
busdaycalendar.weekmask
A copy of the seven-element boolean mask indicating valid days.

busdaycalendar.holidays
A copy of the holiday array indicating additional invalid days.
```

Note: once a busdaycalendar object is created, you cannot modify the weekmask or holidays. The attributes return copies of internal data.

numpy.is_busday(dates, weekmask='1111100', holidays=None, busdaycal=None, out=None)
Calculates which of the given dates are valid days, and which are not. New in version 1.7.0.

Parameters

dates: array_like of datetime64[D]
The array of dates to process.

weekmask: str or array_like of bool, optional
A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

holidays: array_like of datetime64[D], optional
An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

busdaycal: busdaycalendar, optional
A `busdaycalendar` object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

**out** : array of bool, optional

If provided, this array is filled with the result.

**Returns**

**out** : array of bool

An array with the same shape as `dates`, containing True for each valid day, and False for each invalid day.

**See Also:**

`busdaycalendar`

An object that specifies a custom set of valid days.

`busday_offset`

Applies an offset counted in valid days.

`busday_count`

Counts how many valid days are in a half-open date range.

**Examples**

```python
def np.is_busday(dates, holidays=None):
    if isinstance(dates, list):
        dates = np.array(dates, dtype='datetime64[D]')
    dates = np.busdaycalendar(dates, holidays=holidays)
    return dates
```

```python
>>> np.is_busday(['2011-07-01', '2011-07-02', '2011-07-18'],
array([False, False, True], dtype='bool')
```

**numpy.busday_offset**

First adjusts the date to fall on a valid day according to the `roll` rule, then applies offsets to the given dates counted in valid days. New in version 1.7.0.

**Parameters**

**dates** : array_like of datetime64[D]

The array of dates to process.

**offsets** : array_like of int

The array of offsets, which is broadcast with `dates`.

**roll** : {'raise', 'nat', 'forward', 'following', 'backward', 'preceding', 'modifiedfollowing', 'modifiedpreceding'}, optional

How to treat dates that do not fall on a valid day. The default is ‘raise’.

- ‘raise’ means to raise an exception for an invalid day.
- ‘nat’ means to return a NaT (not-a-time) for an invalid day.
- ‘forward’ and ‘following’ mean to take the first valid day later in time.
- ‘backward’ and ‘preceding’ mean to take the first valid day earlier in time.
- ‘modifiedfollowing’ means to take the first valid day later in time unless it is across a Month boundary, in which case to take the first valid day earlier in time.
- ‘modifiedpreceding’ means to take the first valid day earlier in time unless it is across a Month boundary, in which case to take the first valid day later in time.

**weekmask** : str or array_like of bool, optional
A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like ‘1111000’; or a string like “Mon Tue Wed Thu Fri”, made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

**holidays** : array_like of datetime64[D], optional

An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

**busdaycal** : busdaycalendar, optional

A *busdaycalendar* object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

**out** : array of datetime64[D], optional

If provided, this array is filled with the result.

**Returns**

**out** : array of datetime64[D]

An array with a shape from broadcasting *dates* and *offsets* together, containing the dates with offsets applied.

**See Also:**

**busdaycalendar**
An object that specifies a custom set of valid days.

**is_busday**
Returns a boolean array indicating valid days.

**busday_count**
Counts how many valid days are in a half-open date range.

**Examples**

```python
>>> # First business day in October 2011 (not accounting for holidays)
... np.busday_offset('2011-10', 0, roll='forward')
numpy.datetime64('2011-10-03','D')

>>> # Last business day in February 2012 (not accounting for holidays)
... np.busday_offset('2012-03', -1, roll='forward')
numpy.datetime64('2012-02-29','D')

>>> # Third Wednesday in January 2011
... np.busday_offset('2011-01', 2, roll='forward', weekmask='Wed')
numpy.datetime64('2011-01-19','D')

>>> # 2012 Mother’s Day in Canada and the U.S.
... np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13','D')

>>> # First business day on or after a date
... np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21','D')

>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22','D')

>>> # First business day after a date
... np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21','D')
```
```python
>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23','D')
```

**numpy.busday_count** (*begindates*, *enddates*, *weekmask='1111100'*, *holidays=[], busdaycal=None, out=None*)

Counts the number of valid days between *begindates* and *enddates*, not including the day of *enddates*.

If *enddates* specifies a date value that is earlier than the corresponding *begindates* date value, the count will be negative. New in version 1.7.0.

**Parameters**

*begindates* : array_like of datetime64[D]

The array of the first dates for counting.

*enddates* : array_like of datetime64[D]

The array of the end dates for counting, which are excluded from the count themselves.

*weekmask* : str or array_like of bool, optional

A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like `[1,1,1,1,0,0,0]`; a length-seven string, like ‘1111100’; or a string like “Mon Tue Wed Thu Fri”, made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

*holidays* : array_like of datetime64[D], optional

An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

*busdaycal* : busdaycalendar, optional

A busdaycalendar object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

*out* : array of int, optional

If provided, this array is filled with the result.

**Returns**

*out* : array of int

An array with a shape from broadcasting *begindates* and *enddates* together, containing the number of valid days between the begin and end dates.

**See Also:**

*busdaycalendar*

An object that specifies a custom set of valid days.

*is_busday*

Returns a boolean array indicating valid days.

*busday_offset*

Applies an offset counted in valid days.

**Examples**
>>> # Number of weekdays in January 2011
... np.busday_count('2011-01', '2011-02')
21
>>> # Number of weekdays in 2011
... np.busday_count('2011', '2012')
260
>>> # Number of Saturdays in 2011
... np.busday_count('2011', '2012', weekmask='Sat')
53

3.7 Data type routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>can_cast</code></td>
<td>Returns True if cast between data types can occur according to the</td>
</tr>
<tr>
<td><code>promote_types</code></td>
<td>Returns the data type with the smallest size and smallest scalar kind to</td>
</tr>
<tr>
<td><code>min_scalar_type</code></td>
<td>For scalar <code>a</code>, returns the data type with the smallest size and smallest</td>
</tr>
<tr>
<td><code>result_type</code></td>
<td>Returns the type that results from applying the NumPy</td>
</tr>
<tr>
<td><code>common_type</code></td>
<td>Return a scalar type which is common to the input arrays.</td>
</tr>
<tr>
<td><code>obj2sctype</code></td>
<td>Return the scalar dtype or NumPy equivalent of Python type of an object.</td>
</tr>
</tbody>
</table>

**`numpy.can_cast(from, totype, casting = 'safe')`**

Returns True if cast between data types can occur according to the casting rule. If from is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an integer.

**Parameters**

- **from**: dtype, dtype specifier, scalar, or array
  
  Data type, scalar, or array to cast from.

- **totype**: dtype or dtype specifier
  
  Data type to cast to.

- **casting**: {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
  
  Controls what kind of data casting may occur.

  - ‘no’ means the data types should not be cast at all.
  - ‘equiv’ means only byte-order changes are allowed.
  - ‘safe’ means only casts which can preserve values are allowed.
  - ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
  - ‘unsafe’ means any data conversions may be done.

**Returns**

- **out**: bool
  
  True if cast can occur according to the casting rule.

**See Also**

- `dtype`, `result_type`
Examples

Basic examples

>>> np.can_cast(np.int32, np.int64)
True
>>> np.can_cast(np.float64, np.complex)
True
>>> np.can_cast(np.complex, np.float)
False

>>> np.can_cast('i8', 'f8')
True
>>> np.can_cast('i8', 'f4')
False
>>> np.can_cast('i4', 'S4')
True

Casting scalars

>>> np.can_cast(100, 'i1')
True
>>> np.can_cast(150, 'i1')
False
>>> np.can_cast(150, 'u1')
True

>>> np.can_cast(3.5e100, np.float32)
False
>>> np.can_cast(1000.0, np.float32)
True

Array scalar checks the value, array does not

>>> np.can_cast(np.array(1000.0), np.float32)
True
>>> np.can_cast(np.array([1000.0]), np.float32)
False

Using the casting rules

>>> np.can_cast('i8', 'i8', 'no')
True
>>> np.can_cast('<i8', '>i8', 'no')
False

>>> np.can_cast('<i8', '>i8', 'equiv')
True
>>> np.can_cast('<i4', '>i8', 'equiv')
False

>>> np.can_cast('<i4', '>i8', 'safe')
True
>>> np.can_cast('<i8', '>i4', 'safe')
False

>>> np.can_cast('<i8', '>i4', 'same_kind')
True
>>> np.can_cast('<i8', '>u4', 'same_kind')
False
>>> np.can_cast('i8', '>u4', 'unsafe')
True

numpy.promote_types(type1, type2)
Returns the data type with the smallest size and smallest scalar kind to which both type1 and type2 may be safely cast. The returned data type is always in native byte order.

This function is symmetric and associative.

Parameters

- **type1**: dtype or dtype specifier
  - First data type.
- **type2**: dtype or dtype specifier
  - Second data type.

Returns

- **out**: dtype
  - The promoted data type.

See Also:

result_type, dtype, can_cast

Notes

New in version 1.6.0.

Examples

```python
>>> np.promote_types('f4', 'f8')
dtype('float64')

>>> np.promote_types('i8', 'f4')
dtype('float64')

>>> np.promote_types('>i8', '<c8')
dtype('complex128')

>>> np.promote_types('i1', 'S8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: invalid type promotion
```

numpy.min_scalar_type(a)
For scalar a, returns the data type with the smallest size and smallest scalar kind which can hold its value. For non-scalar array a, returns the vector’s dtype unmodified.

Floating point values are not demoted to integers, and complex values are not demoted to floats.

Parameters

- **a**: scalar or array_like
  - The value whose minimal data type is to be found.

Returns

- **out**: dtype
  - The minimal data type.
See Also:

result_type, promote_types, dtype, can_cast

Notes
New in version 1.6.0.

Examples

```python
>>> np.min_scalar_type(10)
dtype('uint8')

>>> np.min_scalar_type(-260)
dtype('int16')

>>> np.min_scalar_type(3.1)
dtype('float16')

>>> np.min_scalar_type(1e50)
dtype('float64')

>>> np.min_scalar_type(np.arange(4, dtype='f8'))
dtype('float64')
```

`numpy.result_type(*arrays_and_dtypes)`
Returns the type that results from applying the NumPy type promotion rules to the arguments.

Type promotion in NumPy works similarly to the rules in languages like C++, with some slight differences. When both scalars and arrays are used, the array’s type takes precedence and the actual value of the scalar is taken into account.

For example, calculating 3*a, where a is an array of 32-bit floats, intuitively should result in a 32-bit float output. If the 3 is a 32-bit integer, the NumPy rules indicate it can’t convert losslessly into a 32-bit float, so a 64-bit float should be the result type. By examining the value of the constant, ‘3’, we see that it fits in an 8-bit integer, which can be cast losslessly into the 32-bit float.

Parameters
arrays_and_dtypes : list of arrays and dtypes

The operands of some operation whose result type is needed.

Returns
out : dtype

The result type.

See Also:
dtype, promote_types, min_scalar_type, can_cast

Notes
New in version 1.6.0. The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex) the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the arrays, the data types are combined with promote_types to produce the return value.

Otherwise, min_scalar_type is called on each array, and the resulting data types are all combined with promote_types to produce the return value.
The set of int values is not a subset of the uint values for types with the same number of bits, something not reflected in `min_scalar_type`, but handled as a special case in `result_type`.

**Examples**

```python
code python
define
>>> np.result_type(3, np.arange(7, dtype='i1'))
dtype('int8')

>>> np.result_type('i4', 'c8')
dtype('complex128')

>>> np.result_type(3.0, -2)
dtype('float64')
```

```
np.common_type(*arrays)
Return a scalar type which is common to the input arrays.

The return type will always be an inexact (i.e. floating point) scalar type, even if all the arrays are integer arrays. If one of the inputs is an integer array, the minimum precision type that is returned is a 64-bit floating point dtype.

All input arrays can be safely cast to the returned dtype without loss of information.

**Parameters**

array1, array2, ... : ndarrays
Input arrays.

**Returns**

out : data type code
Data type code.

**See Also:**
dtype, mintypecode
```

```
np.obj2sctype(rep, default=None)
Return the scalar dtype or NumPy equivalent of Python type of an object.

**Parameters**

rep : any
The object of which the type is returned.

default : any, optional
If given, this is returned for objects whose types can not be determined. If not given, None is returned for those objects.

**Returns**

dtype : dtype or Python type
The data type of rep.
```
See Also:

sctype2char, issctype, issubscctype, issubdtype, maximum_sctype

Examples

>>> np.obj2sctype(np.int32)
<type 'numpy.int32'>

>>> np.obj2sctype(np.array([1., 2.]))
<type 'numpy.float64'>

>>> np.obj2sctype(np.array([1.j]))
<type 'numpy.complex128'>

>>> np.obj2sctype(dict)
<type 'numpy.object_'>

>>> np.obj2sctype('string')
<type 'numpy.string_'>

>>> np.obj2sctype(1, default=list)
<type 'list'>

3.7.1 Creating data types

<table>
<thead>
<tr>
<th>dtype</th>
<th>Create a data type object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>format_parser</td>
<td>Class to convert formats, names, titles description to a dtype.</td>
</tr>
</tbody>
</table>

class numpy.dtype

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

Parameters

obj

Object to be converted to a data type object.

align : bool, optional

Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be True only if obj is a dictionary or a comma-separated string. If a struct dtype is being created, this also sets a sticky alignment flag isalignedstruct.

copy : bool, optional

Make a new copy of the data-type object. If False, the result may just be a reference to a built-in data-type object.

See Also:

result_type

Examples

Using array-scalar type:

>>> np.dtype(np.int16)
dtype('int16')
Record, one field name ‘f1’, containing int16:

```python
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Record, one field named ‘f1’, in itself containing a record with one field:

```python
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Record, two fields: the first field contains an unsigned int, the second an int32:

```python
>>> np.dtype([('f1', np.uint), ('f2', np.int32)])
dtype([('f1', '<u4'), ('f2', '<i4')])
```

Using array-protocol type strings:

```python
>>> np.dtype(('a','f8'),('b','S10'))
dtype([('a', '<f8'), ('b', '|S10')])
```

Using comma-separated field formats. The shape is (2,3):

```python
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. int is a fixed type, 3 the field’s shape. void is a flexible type, here of size 10:

```python
>>> np.dtype([('hello',(np.int,3)),('world',np.void,10)])
dtype([('hello', '<i4', 3), ('world', '|V10')])
```

Subdivide int16 into 2 int8’s, called x and y. 0 and 1 are the offsets in bytes:

```python
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype(('<i2', [('x', '|i1'), ('y', '|i1')]))
```

Using dictionaries. Two fields named ‘gender’ and ‘age’:

```python
>>> np.dtype({'names':['gender','age'], 'formats':[S1',np.uint8]})
dtype([('gender', '|S1'), ('age', '|u1')])
```

Offsets in bytes, here 0 and 25:

```python
>>> np.dtype({'surname':('S25',0), 'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

**Attributes**

- `base`  
- `descr`  
- `fields`  
- `hasobject`  
- `isalignedstruct`  
- `isbuiltin`  
- `isnative`  
- `metadata`  
- `name`  
- `names`  
- `shape`
**Table 3.33 – continued from previous page**

<table>
<thead>
<tr>
<th>str</th>
<th>The array-protocol typestring of this data-type object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>subdtype</td>
<td>Tuple <code>(item_dtype, shape)</code> if this dtype describes a sub-array, and</td>
</tr>
</tbody>
</table>

**dtype.base**

**dtype.descr**

Array-interface compliant full description of the data-type.

The format is that required by the `descr` key in the `__array_interface__` attribute.

**dtype.fields**

Dictionary of named fields defined for this data type, or None.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

`(dtype, offset[, title])`

If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the `ndarray.getfield` and `ndarray.setfield` methods.

See Also:

`ndarray.getfield, ndarray.setfield`

**Examples**

```python
gt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
gt = print dt.fields
gt = {'grades': (dtype((float64, (2,))), 16), 'name': (dtype('|S16'), 0)}
```

**dtype.hasobject**

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won’t.

**dtype.isalignedstruct**

Boolean indicating whether the dtype is a struct which maintains field alignment. This flag is sticky, so when combining multiple structs together, it is preserved and produces new dtypes which are also aligned.

**dtype.isbuiltin**

Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

| 0 | if this is a structured array type, with fields |
| 1 | if this is a dtype compiled into numpy (such as ints, floats etc) |
| 2 | if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See `user.user-defined-data-types` in the Numpy manual. |

**Examples**

3.7. Data type routines
```python
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.isbuiltin
0

dtype.isnative
    Boolean indicating whether the byte order of this dtype is native to the platform.

dtype.metadata

dtype.name
    A bit-width name for this data-type.

    Un-sized flexible data-type objects do not have this attribute.

dtype.names
    Ordered list of field names, or None if there are no fields.

    The names are ordered according to increasing byte offset. This can be used, for example, to walk through
    all of the named fields in offset order.

    Examples
    >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
    >>> dt.names
    ('name', 'grades')

dtype.shape
    Shape tuple of the sub-array if this data type describes a sub-array, and () otherwise.

dtype.str
    The array-protocol typestring of this data-type object.

dtype.subdtype
    Tuple (item_dtype, shape) if this dtype describes a sub-array, and None otherwise.

    The shape is the fixed shape of the sub-array described by this data type, and item_dtype the data type of
    the array.

    If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by shape are
    tacked on to the end of the retrieved array.

Methods

    newbyteorder([new_order])  Return a new dtype with a different byte order.


dtype.newbyteorder (new_order=‘S’)
    Return a new dtype with a different byte order.

    Changes are also made in all fields and sub-arrays of the data type.

    Parameters
    new_order : string, optional

        Byte order to force; a value from the byte order specifications below. The default value
('S') results in swapping the current byte order. new_order codes can be any of:

* 'S' - swap dtype from current to opposite endian
* {'<', 'L'} - little endian
* {'>', 'B'} - big endian
* {'=', 'N'} - native order
* {'|', 'I'} - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of new_order for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

Returns

new_dtype : dtype

New dtype object with the given change to the byte order.

Notes

Changes are also made in all fields and sub-arrays of the data type.

Examples

```python
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

class numpy.format_parser(formats, names, titles, aligned=False, byteorder=None)

Class to convert formats, names, titles description to a dtype.

After constructing the format_parser object, the dtype attribute is the converted data-type: dtype = format_parser(formats, names, titles).dtype

Parameters

- formats : str or list of str

The format description, either specified as a string with comma-separated format descriptions in the form 'f8, i4, a5', or a list of format description strings in the
form ['f8', 'i4', 'a5'].

names : str or list/tuple of str

The field names, either specified as a comma-separated string in the form 'col1, col2, col3', or as a list or tuple of strings in the form ['col1', 'col2', 'col3']. An empty list can be used, in that case default field names ('f0', 'f1', ...) are used.

titles : sequence

Sequence of title strings. An empty list can be used to leave titles out.

aligned : bool, optional

If True, align the fields by padding as the C-compiler would. Default is False.

byteorder : str, optional

If specified, all the fields will be changed to the provided byte-order. Otherwise, the default byte-order is used. For all available string specifiers, see dtype.newbyteorder.

See Also:
dtype, typename, sctype2char

Examples

```python
>>> np.format_parser([‘f8’, ‘i4’, ‘a5’], [‘col1’, ‘col2’, ‘col3’], ...
[‘T1’, ‘T2’, ‘T3’]).dtype
dtype([(‘T1’, ‘col1’), ‘<f8’), ((‘T2’, ‘col2’), ‘<i4’),
((‘T3’, ‘col3’), ‘|S5’)])
```

Names and/or titles can be empty lists. If titles is an empty list, titles will simply not appear. If names is empty, default field names will be used.

```python
>>> np.format_parser([‘f8’, ‘i4’, ‘a5’], [‘col1’, ‘col2’, ‘col3’], ...
[[], []).dtype
dtype([‘col1’, ‘<f8’), (‘col2’, ‘<i4’), (‘col3’, ‘|S5’)])
>>> np.format_parser([‘f8’, ‘i4’, ‘a5’], [‘f0’, ‘<f8’), (‘f1’, ‘<i4’), (‘f2’, ‘|S5’)])
```

Attributes

dtype (dtype) The converted data-type.

3.7.2 Data type information

<table>
<thead>
<tr>
<th>finfo</th>
<th>Machine limits for floating point types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>iinfo (type)</td>
<td>Machine limits for integer types.</td>
</tr>
<tr>
<td>MachAr([float_conv, int_conv, …])</td>
<td>Diagnosing machine parameters.</td>
</tr>
</tbody>
</table>

class numpy.finfo

Machine limits for floating point types.

Parameters

dtype : float, dtype, or instance
Kind of floating point data-type about which to get information.

**See Also:**

**MachAr**
The implementation of the tests that produce this information.

**iinfo**
The equivalent for integer data types.

**Notes**
For developers of NumPy: do not instantiate this at the module level. The initial calculation of these parameters is expensive and negatively impacts import times. These objects are cached, so calling `finfo()` repeatedly inside your functions is not a problem.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eps</code></td>
<td>(float) The smallest representable positive number such that $1.0 + \text{eps} \neq 1.0$. Type of <code>eps</code> is an appropriate floating point type.</td>
</tr>
<tr>
<td><code>epsneg</code></td>
<td>(floating point number of the appropriate type) The smallest representable positive number such that $1.0 - \text{epsneg} \neq 1.0$.</td>
</tr>
<tr>
<td><code>iexp</code></td>
<td>(int) The number of bits in the exponent portion of the floating point representation.</td>
</tr>
<tr>
<td><code>machar</code></td>
<td>(MachAr) The object which calculated these parameters and holds more detailed information.</td>
</tr>
<tr>
<td><code>machep</code></td>
<td>(int) The exponent that yields <code>eps</code>.</td>
</tr>
<tr>
<td><code>max</code></td>
<td>(floating point number of the appropriate type) The largest representable number.</td>
</tr>
<tr>
<td><code>maxexp</code></td>
<td>(int) The smallest positive power of the base (2) that causes overflow.</td>
</tr>
<tr>
<td><code>min</code></td>
<td>(floating point number of the appropriate type) The smallest representable number, typically $-\text{max}$.</td>
</tr>
<tr>
<td><code>minexp</code></td>
<td>(int) The most negative power of the base (2) consistent with there being no leading 0’s in the mantissa.</td>
</tr>
<tr>
<td><code>negep</code></td>
<td>(int) The exponent that yields <code>epsneg</code>.</td>
</tr>
<tr>
<td><code>nexp</code></td>
<td>(int) The number of bits in the exponent including its sign and bias.</td>
</tr>
<tr>
<td><code>nmant</code></td>
<td>(int) The number of bits in the mantissa.</td>
</tr>
<tr>
<td><code>precision</code></td>
<td>(int) The approximate number of decimal digits to which this kind of float is precise.</td>
</tr>
<tr>
<td><code>resolution</code></td>
<td>(floating point number of the appropriate type) The approximate decimal resolution of this type, i.e., $10^{\text{precision}}$.</td>
</tr>
<tr>
<td><code>tiny</code></td>
<td>(float) The smallest positive usable number. Type of <code>tiny</code> is an appropriate floating point type.</td>
</tr>
</tbody>
</table>

```python
class numpy.iinfo(type)
```
Machine limits for integer types.

**Parameters**

- `type` : integer type, dtype, or instance
  
The kind of integer data type to get information about.

**See Also:**

**finfo**
The equivalent for floating point data types.

**Examples**

With types:
>>> ii16 = np.iinfo(np.int16)
>>> ii16.min
-32768
>>> ii16.max
32767

>>> ii32 = np.iinfo(np.int32)
>>> ii32.min
-2147483648
>>> ii32.max
2147483647

With instances:

>>> ii32 = np.iinfo(np.int32(10))
>>> ii32.min
-2147483648
>>> ii32.max
2147483647

Attributes

<table>
<thead>
<tr>
<th>min</th>
<th>Minimum value of given dtype.</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>Maximum value of given dtype.</td>
</tr>
</tbody>
</table>

iinfo.min

Minimum value of given dtype.

iinfo.max

Maximum value of given dtype.

class numpy.MachAr (float_conv=<type 'float'>, int_conv=<type 'int'>, float_to_float=<type 'float'>, float_to_str=<function <lambda> at 0x248ab90>, title='Python floating point number')

Diagnosing machine parameters.

Parameters

float_conv : function, optional

Function that converts an integer or integer array to a float or float array. Default is float.

int_conv : function, optional

Function that converts a float or float array to an integer or integer array. Default is int.

float_to_float : function, optional

Function that converts a float array to float. Default is float. Note that this does not seem to do anything useful in the current implementation.

float_to_str : function, optional

Function that converts a single float to a string. Default is lambda v:'%24.16e' %v.

title : str, optional

Title that is printed in the string representation of MachAr.

See Also:
**finfo**
Machine limits for floating point types.

**iinfo**
Machine limits for integer types.

**References**

[R1]

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibeta</td>
<td>(int) Radix in which numbers are represented.</td>
</tr>
<tr>
<td>it</td>
<td>(int) Number of base-( \beta ) digits in the floating point mantissa ( M ).</td>
</tr>
<tr>
<td>machep</td>
<td>(int) Exponent of the smallest (most negative) power of ( \beta ) that, added to 1.0, gives something different from 1.0.</td>
</tr>
<tr>
<td>eps</td>
<td>(float) Floating-point number ( \beta^{\text{machep}} ) (floating point precision).</td>
</tr>
<tr>
<td>negep</td>
<td>(int) Exponent of the smallest power of ( \beta ) that, substracted from 1.0, gives something different from 1.0.</td>
</tr>
<tr>
<td>epsneg</td>
<td>(float) Floating-point number ( \beta^{\text{negep}} ).</td>
</tr>
<tr>
<td>iexp</td>
<td>(int) Number of bits in the exponent (including its sign and bias).</td>
</tr>
<tr>
<td>minexp</td>
<td>(int) Smallest (most negative) power of ( \beta ) consistent with there being no leading zeros in the mantissa.</td>
</tr>
<tr>
<td>xmin</td>
<td>(float) Floating point number ( \beta^{\text{minexp}} ) (the smallest [in magnitude] usable floating value).</td>
</tr>
<tr>
<td>maxexp</td>
<td>(int) Smallest (positive) power of ( \beta ) that causes overflow.</td>
</tr>
<tr>
<td>xmax</td>
<td>(float) ((1-\text{epsneg}) \times \beta^{\text{maxexp}} ) (the largest [in magnitude] usable floating value).</td>
</tr>
<tr>
<td>irnd</td>
<td>(int) In ( \text{range}(6) ), information on what kind of rounding is done in addition, and on how underflow is handled.</td>
</tr>
<tr>
<td>ngrd</td>
<td>(int) Number of ‘guard digits’ used when truncating the product of two mantissas to fit the representation.</td>
</tr>
<tr>
<td>epsilon</td>
<td>(float) Same as ( \text{eps} ).</td>
</tr>
<tr>
<td>tiny</td>
<td>(float) Same as ( \text{xmin} ).</td>
</tr>
<tr>
<td>huge</td>
<td>(float) Same as ( \text{xmax} ).</td>
</tr>
<tr>
<td>precision</td>
<td>(float) (- \text{int}(-\log_{10}(\text{eps})))</td>
</tr>
<tr>
<td>resolution</td>
<td>(float) (- 10^{\text{precision}})</td>
</tr>
</tbody>
</table>

### 3.7.3 Data type testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>issctype(rep)</td>
<td>Determines whether the given object represents a scalar data-type.</td>
</tr>
<tr>
<td>issubdtype(arg1, arg2)</td>
<td>Returns True if first argument is a typecode lower/equal in type hierarchy.</td>
</tr>
<tr>
<td>issubsctype(arg1, arg2)</td>
<td>Determine if the first argument is a subclass of the second argument.</td>
</tr>
<tr>
<td>issubclass_(arg1, arg2)</td>
<td>Determine if a class is a subclass of a second class.</td>
</tr>
<tr>
<td>find_common_type(array_types, scalar_types)</td>
<td>Determine common type following standard coercion rules.</td>
</tr>
</tbody>
</table>

**numpy.issctype(rep)**
Determine whether the given object represents a scalar data-type.

**Parameters**

- rep : any
If `rep` is an instance of a scalar dtype, True is returned. If not, False is returned.

```
Returns
out : bool

Boolean result of check whether `rep` is a scalar dtype.
```

See Also:

`issubactype`, `issubdtype`, `obj2sctype`, `sctype2char`

Examples

```python
>>> np.issctype(np.int32)
True
>>> np.issctype(list)
False
>>> np.issctype(1.1)
False

Strings are also a scalar type:

```python
>>> np.issctype(np.dtype('str'))
True
```

```
numpy.issubdtype(arg1, arg2)
Returns True if first argument is a typecode lower/equal in type hierarchy.

Parameters
arg1, arg2 : dtype_like
    dtype or string representing a typecode.

Returns
out : bool
```

See Also:

`issubactype`, `issubclass_`

```
numpy.core.numerictypes
    Overview of numpy type hierarchy.
```

Examples

```python
>>> np.issubdtype('S1', str)
True
>>> np.issubdtype(np.float64, np.float32)
False
```

```
numpy.issubsctype(arg1, arg2)
Determine if the first argument is a subclass of the second argument.

Parameters
arg1, arg2 : dtype or dtype specifier
    Data-types.

Returns
out : bool
    The result.
```
See Also:

isscttype, issubdtype, obj2sctype

Examples

```python
>>> np.issubsctype('S8', str)
True
>>> np.issubsctype(np.array([1]), np.int)
True
>>> np.issubsctype(np.array([1]), np.float)
False
```

numpy.issubclass_(arg1, arg2)
Determine if a class is a subclass of a second class.

issubclass_ is equivalent to the Python built-in issubclass, except that it returns False instead of raising a TypeError is one of the arguments is not a class.

Parameters
arg1 : class
Input class. True is returned if arg1 is a subclass of arg2.
arg2 : class or tuple of classes.
Input class. If a tuple of classes, True is returned if arg1 is a subclass of any of the tuple elements.

Returns
out : bool
Whether arg1 is a subclass of arg2 or not.

See Also:

issubsctype, issubdtype, issctype

Examples

```python
>>> np.issubclass_(np.int32, np.int)
True
>>> np.issubclass_(np.int32, np.float)
False
```

numpy.find_common_type(array_types, scalar_types)
Determine common type following standard coercion rules.

Parameters
array_types : sequence
A list of dtypes or dtype convertible objects representing arrays.
scalar_types : sequence
A list of dtypes or dtype convertible objects representing scalars.

Returns
datatype : dtype
The common data type, which is the maximum of array_types ignoring scalar_types, unless the maximum of scalar_types is of a different kind (dtype.kind). If the kind is not understood, then None is returned.
See Also:

dtype, common_type, can_cast, mintypecode

Examples

>>> np.find_common_type([], [np.int64, np.float32, np.complex])
dtype('complex128')

>>> np.find_common_type([np.int64, np.float32], [])
dtype('float64')

The standard casting rules ensure that a scalar cannot up-cast an array unless the scalar is of a fundamentally
different kind of data (i.e. under a different hierarchy in the data type hierarchy) then the array:

>>> np.find_common_type([np.float32], [np.int64, np.float64])
dtype('float32')

Complex is of a different type, so it up-casts the float in the array_types argument:

>>> np.find_common_type([np.float32], [np.complex])
dtype('complex128')

Type specifier strings are convertible to dtypes and can therefore be used instead of dtypes:

>>> np.find_common_type(['f4', 'f4', 'i4'], ['c8'])
dtype('complex128')

3.7.4 Miscellaneous

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typename(char)</td>
<td>Return a description for the given data type code.</td>
</tr>
<tr>
<td>sctype2char(sctype)</td>
<td>Return the string representation of a scalar dtype.</td>
</tr>
<tr>
<td>mintypecode(typechars[, typeset, default])</td>
<td>Return the character for the minimum-size type to which given types can be safely cast.</td>
</tr>
</tbody>
</table>

numpy typename(char)

Return a description for the given data type code.

Parameters

char : str

Data type code.

Returns

out : str

Description of the input data type code.

See Also:

dtype, typecodes

Examples

>>> typechars = ['S1', '?', 'B', 'D', 'G', 'F', 'I', 'H', 'L', 'O', 'Q', ...
... 'S', 'U', 'V', 'b', 'd', 'g', 'f', 'i', 'h', 'l', 'q']

>>> for typechar in typechars:
...     print typechar, ': ', np.typename(typechar)
...
S1 : character
numpy.sctype2char(sctype)

Return the string representation of a scalar dtype.

Parameters
    sctype : scalar dtype or object
        If a scalar dtype, the corresponding string character is returned. If an object,
        sctype2char tries to infer its scalar type and then return the corresponding string
        character.

Returns
    typechar : str
        The string character corresponding to the scalar type.

Raises
    ValueError
        If sctype is an object for which the type can not be inferred.

See Also:
    obj2sctype, issctype, issubsctype, mintypecode

Examples

>>> for sctype in [np.int32, np.float, np.complex, np.string_, np.ndarray]:
...     print np.sctype2char(sctype)
...     l
d
D
S
O

>>> x = np.array([1., 2-1.j])
>>> np.sctype2char(x)
'D'
>>> np.sctype2char(list)
'O'
numpy.mintypecode(typechars, typeset='GDFgdf', default='d')

Return the character for the minimum-size type to which given types can be safely cast.

The returned type character must represent the smallest size dtype such that an array of the returned type can handle the data from an array of all types in typechars (or if typechars is an array, then its dtype.char).

Parameters

- **typechars**: list of str or array_like
  If a list of strings, each string should represent a dtype. If array_like, the character representation of the array dtype is used.

- **typeset**: str or list of str, optional
  The set of characters that the returned character is chosen from. The default set is ‘GDFgdf’.

- **default**: str, optional
  The default character, this is returned if none of the characters in typechars matches a character in typeset.

Returns

- **typechar**: str
  The character representing the minimum-size type that was found.

See Also:
dtype, sctype2char, maximum_sctype

Examples

```python
>>> np.mintypecode(['d', 'f', 'S'])
'd'
>>> x = np.array([1.1, 2-3.j])
>>> np.mintypecode(x)
'D'
>>> np.mintypecode('abceh', default='G')
'G'
```

### 3.8 Optionally Scipy-accelerated routines (numpy.dual)

Aliases for functions which may be accelerated by Scipy.

Scipy can be built to use accelerated or otherwise improved libraries for FFTs, linear algebra, and special functions. This module allows developers to transparently support these accelerated functions when scipy is available but still support users who have only installed Numpy.

#### 3.8.1 Linear algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky(a)</td>
<td>Cholesky decomposition.</td>
</tr>
<tr>
<td>det(a)</td>
<td>Compute the determinant of an array.</td>
</tr>
<tr>
<td>eig(a)</td>
<td>Compute the eigenvalues and right eigenvectors of a square array.</td>
</tr>
<tr>
<td>eigh(a[, UPL0])</td>
<td>Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.</td>
</tr>
</tbody>
</table>
### 3.8.2 FFT

- `fft(a[, n, axis])`: Compute the one-dimensional discrete Fourier Transform.
- `fft2(a[, s, axes])`: Compute the 2-dimensional discrete Fourier Transform.
- `fftn(a[, s, axes])`: Compute the N-dimensional discrete Fourier Transform.
- `ifft(a[, n, axis])`: Compute the one-dimensional inverse discrete Fourier Transform.
- `ifft2(a[, s, axes])`: Compute the 2-dimensional inverse discrete Fourier Transform.
- `ifftn(a[, s, axes])`: Compute the N-dimensional inverse discrete Fourier Transform.

### 3.8.3 Other

- `i0(x)`: Modified Bessel function of the first kind, order 0.

### 3.9 Mathematical functions with automatic domain (numpy.emath)

**Note:** *numpy.emath* is a preferred alias for *numpy.lib.scimath*, available after *numpy* is imported.

Wrapper functions to more user-friendly calling of certain math functions whose output data-type is different than the input data-type in certain domains of the input.

For example, for functions like *log* with branch cuts, the versions in this module provide the mathematically valid answers in the complex plane:

```python
>>> import math
>>> from numpy.lib import scimath
>>> scimath.log(-math.exp(1)) == (1+1j*math.pi)
True
```

Similarly, *sqrt*, other base logarithms, *power* and trig functions are correctly handled. See their respective docstrings for specific examples.

### 3.10 Floating point error handling

#### 3.10.1 Setting and getting error handling

- `seterr([all, divide, over, under, invalid])`: Set how floating-point errors are handled.

Continued on next page
Table 3.42 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>geterr()</code></td>
<td>Get the current way of handling floating-point errors.</td>
</tr>
<tr>
<td><code>seterrcall(func)</code></td>
<td>Set the floating-point error callback function or log object.</td>
</tr>
<tr>
<td><code>geterrcall()</code></td>
<td>Return the current callback function used on floating-point errors.</td>
</tr>
<tr>
<td><code>errstate(**kwargs)</code></td>
<td>Context manager for floating-point error handling.</td>
</tr>
</tbody>
</table>

```python
numpy.seterr(all=None, divide=None, over=None, under=None, invalid=None)
```

Set how floating-point errors are handled.

Note that operations on integer scalar types (such as `int16`) are handled like floating point, and are affected by these settings.

**Parameters**

- **all** : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  - Set treatment for all types of floating-point errors at once:
    - ignore: Take no action when the exception occurs.
    - warn: Print a `RuntimeWarning` (via the Python `warnings` module).
    - raise: Raise a `FloatingPointError`.
    - call: Call a function specified using the `seterrcall` function.
    - print: Print a warning directly to `stdout`.
    - log: Record error in a Log object specified by `seterrcall`.

The default is not to change the current behavior.

- **divide** : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  - Treatment for division by zero.

- **over** : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  - Treatment for floating-point overflow.

- **under** : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  - Treatment for floating-point underflow.

- **invalid** : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
  - Treatment for invalid floating-point operation.

**Returns**

- `old_settings` : dict
  - Dictionary containing the old settings.

**See Also:**

- `seterrcall`  
  - Set a callback function for the ‘call’ mode.

- `geterr`, `geterrcall`, `errstate`
Notes
The floating-point exceptions are defined in the IEEE 754 standard [1]:

- Division by zero: infinite result obtained from finite numbers.
- Overflow: result too large to be expressed.
- Underflow: result so close to zero that some precision was lost.
- Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.

Examples
>>> old_settings = np.seterr(all='ignore')  # seterr to known value
>>> np.seterr(over='raise')
{'over': 'ignore', 'divide': 'ignore', 'invalid': 'ignore',
 'under': 'ignore'}
>>> np.seterr(**old_settings)  # reset to default
{'over': 'raise', 'divide': 'ignore', 'invalid': 'ignore', 'under': 'ignore'}

>>> np.int16(32000) * np.int16(3)
30464
>>> np.geterr()  # geterr to known value
{'over': 'warn', 'divide': 'warn', 'invalid': 'warn',
 'under': 'ignore'}

FloatingPointError: overflow encountered in short_scalars

>>> np.int16(32000) / np.int16(3)
array([ NaN, 1., 1.])

numpy.geterr()
Get the current way of handling floating-point errors.

Returns
res : dict
A dictionary with keys “divide”, “over”, “under”, and “invalid”, whose values are from the strings “ignore”, “print”, “log”, “warn”, “raise”, and “call”. The keys represent possible floating-point exceptions, and the values define how these exceptions are handled.

See Also:
geterrcall, seterr, seterrcall

Notes
For complete documentation of the types of floating-point exceptions and treatment options, see seterr.

Examples
>>> np.geterr()
{'over': 'warn', 'divide': 'warn', 'invalid': 'warn',
 'under': 'ignore'}
>>> np.arange(3.) / np.arange(3.)
aarray([ NaN, 1., 1.])
numpy.seterrcall(func)

Set the floating-point error callback function or log object.

There are two ways to capture floating-point error messages. The first is to set the error-handler to 'call', using seterr. Then, set the function to call using this function.

The second is to set the error-handler to 'log', using seterr. Floating-point errors then trigger a call to the 'write' method of the provided object.

Parameters
   func : callable f(err, flag) or object with write method

   Function to call upon floating-point errors ('call'-mode) or object whose 'write' method is used to log such message ('log'-mode).

   The call function takes two arguments. The first is the type of error (one of “divide”, “over”, “under”, or “invalid”), and the second is the status flag. The flag is a byte, whose least-significant bits indicate the status:

   [0 0 0 0 invalid over under invalid]

   In other words, flags = divide + 2*over + 4*under + 8*invalid.

   If an object is provided, its write method should take one argument, a string.

Returns
   h : callable, log instance or None

   The old error handler.

See Also:
   seterr, geterr, geterrcall

Examples

Callback upon error:

>>> def err_handler(type, flag):
...     print "Floating point error ($s), with flag $s" % (type, flag)
...     ...

>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')

Floating point error (divide by zero), with flag 1
array([ Inf, Inf, Inf])

nb.seterrcall(saved_handler)
<function err_handler at 0x...>

np.seterr(**save_err)
{'over': 'call', 'divide': 'call', 'invalid': 'call', 'under': 'call'}
Log error message:

```python
>>> class Log(object):
...     def write(self, msg):
...         print "LOG: %s" % msg
...

>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')

>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in divide
array([ Inf, Inf, Inf])

>>> np.seterrcall(saved_handler)
<__main__.Log object at 0x...>
>>> np.seterr(**save_err)
{'over': 'log', 'divide': 'log', 'invalid': 'log', 'under': 'log'}
```

`numpy.geterrcall()`

Return the current callback function used on floating-point errors.

When the error handling for a floating-point error (one of “divide”, “over”, “under”, or “invalid”) is set to ‘call’ or ‘log’, the function that is called or the log instance that is written to is returned by `geterrcall`. This function or log instance has been set with `seterrcall`.

**Returns**

`errobj`: callable, log instance or None

The current error handler. If no handler was set through `seterrcall`, `None` is returned.

**See Also:**

`seterrcall`, `seterr`, `geterr`

**Notes**

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

**Examples**

```python
>>> np.geterrcall()  # we did not yet set a handler, returns None
```

```python
>>> oldsettings = np.seterr(all='call')
>>> def err_handler(type, flag):
...     print "Floating point error ($s), with flag $s" % (type, flag)
>>> oldhandler = np.seterrcall(err_handler)
>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([ Inf, Inf, Inf])

>>> cur_handler = np.geterrcall()
>>> cur_handler is err_handler
True
```

`class numpy.errstate(**kwargs)`

Context manager for floating-point error handling.
Using an instance of `errstate` as a context manager allows statements in that context to execute with a known error handling behavior. Upon entering the context the error handling is set with `seterr` and `seterrcall`, and upon exiting it is reset to what it was before.

**Parameters**

`kwargs` : {`divide`, `over`, `under`, `invalid`}

Keyword arguments. The valid keywords are the possible floating-point exceptions. Each keyword should have a string value that defines the treatment for the particular error. Possible values are {'ignore', 'warn', 'raise', 'call', 'print', 'log'}.

**See Also:**

`seterr`, `geterr`, `seterrcall`, `geterrcall`

**Notes**

The `with` statement was introduced in Python 2.5, and can only be used there by importing it: from `__future__` import `with_statement`. In earlier Python versions the `with` statement is not available.

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

**Examples**

```python
>>> from __future__ import with_statement  # use 'with' in Python 2.5
>>> olderr = np.seterr(all='ignore')  # Set error handling to known state.

>>> np.arange(3) / 0.
array([ NaN, Inf, Inf])

with np.errstate(divide='warn):
    np.arange(3) / 0.

__main__:2: RuntimeWarning: divide by zero encountered in divide
array([ NaN, Inf, Inf])

>>> np.sqrt(-1)
nan

with np.errstate(invalid='raise'):
    np.sqrt(-1)

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FloatingPointError: invalid value encountered in sqrt
```

Outside the context the error handling behavior has not changed:

```python
>>> np.geterr()
{'over': 'warn', 'divide': 'warn', 'invalid': 'warn', 'under': 'ignore'}
```

### 3.10.2 Internal functions

- `seterrobj(errobj)` Set the object that defines floating-point error handling.
- `geterrobj()` Return the current object that defines floating-point error handling.

```python
numpy.seterrobj(errobj)
```

Set the object that defines floating-point error handling.
The error object contains all information that defines the error handling behavior in Numpy. `seterrobj` is used internally by the other functions that set error handling behavior (`seterr`, `seterrcall`).

**Parameters**

`errobj` : list

The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with

- 0: ‘ignore’
- 1: ‘warn’
- 2: ‘raise’
- 3: ‘call’
- 4: ‘print’
- 5: ‘log’

**See Also:**

`geterrobj`, `seterr`, `geterr`, `seterrcall`, `geterrcall`, `getbufsize`, `setbufsize`

**Notes**

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

**Examples**

```python
>>> old_errobj = np.geterrobj()  # first get the defaults
>>> old_errobj
[10000, 0, None]

>>> def err_handler(type, flag):
...     print "Floating point error ($s), with flag $s" % (type, flag)
...

>>> new_errobj = [20000, 12, err_handler]
>>> np.seterrobj(new_errobj)
>>> np.base_repr(12, 8)  # int for divide=4 ('print') and over=1 ('warn')
'14'
>>> np.geterr()  # 'over': 'warn', 'divide': 'print', 'invalid': 'ignore', 'under': 'ignore'
>>> np.geterrcall() is err_handler
True
```

`numpy.geterrobj()`

Return the current object that defines floating-point error handling.

The error object contains all information that defines the error handling behavior in Numpy. `geterrobj` is used internally by the other functions that get and set error handling behavior (`geterr`, `seterr`, `geterrcall`, `seterrcall`).

**Returns**

`errobj` : list
The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with

- 0: ‘ignore’
- 1: ‘warn’
- 2: ‘raise’
- 3: ‘call’
- 4: ‘print’
- 5: ‘log’

See Also:
seterrobj, seterr, geterr, seterrcall, geterrcall, getbufsize, setbufsize

Notes
For complete documentation of the types of floating-point exceptions and treatment options, see seterr.

Examples
>>> np.geterrobj()  # first get the defaults
[10000, 0, None]

>>> def err_handler(type, flag):
...    print "Floating point error ($s), with flag $s" % (type, flag)
...

>>> old_bufsize = np.setbufsize(20000)
>>> old_err = np.seterr(divide='raise')
>>> old_handler = np.seterrcall(err_handler)
>>> np.geterrobj()
[20000, 2, <function err_handler at 0x91dcaac>]

>>> old_err = np.seterr(all='ignore')
>>> np.base_repr(np.geterrobj()[1], 8)
'0'

>>> old_err = np.seterr(divide='warn', over='log', under='call',
invalid='print')
>>> np.base_repr(np.geterrobj()[1], 8)
'4351'

3.11 Discrete Fourier Transform (numpy.fft)

3.11.1 Standard FFTs

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft(a, n, axis)</td>
<td>Compute the one-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td>ifft(a, n, axis)</td>
<td>Compute the one-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td>fft2(a, s, axes)</td>
<td>Compute the 2-dimensional discrete Fourier Transform</td>
</tr>
</tbody>
</table>
Table 3.44 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifft2</td>
<td>Compute the 2-dimensional inverse discrete Fourier Transform.</td>
</tr>
<tr>
<td>fftn</td>
<td>Compute the N-dimensional discrete Fourier Transform.</td>
</tr>
<tr>
<td>ifftn</td>
<td>Compute the N-dimensional inverse discrete Fourier Transform.</td>
</tr>
</tbody>
</table>

```
numpy.fft.fftn(a[, s, axes])
```

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional \( n \)-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

**Parameters**

- `a`: array_like
  Input array, can be complex.

- `n`: int, optional
  Length of the transformed axis of the output. If \( n \) is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If \( n \) is not given, the length of the input (along the axis specified by `axis`) is used.

- `axis`: int, optional
  Axis over which to compute the FFT. If not given, the last axis is used.

**Returns**

- `out`: complex ndarray
  The truncated or zero-padded input, transformed along the axis indicated by `axis`, or the last one if `axis` is not specified.

**Raises**

- `IndexError`
  if `axes` is larger than the last axis of `a`.

**See Also**:

- `numpy.fft`
  for definition of the DFT and conventions used.

- `ifft`
  The inverse of `fftn`.

- `fft2`
  The two-dimensional FFT.

- `fftn`
  The \( n \)-dimensional FFT.

- `rfftn`
  The \( n \)-dimensional FFT of real input.

- `fftfreq`
  Frequency bins for given FFT parameters.

**Notes**

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when \( n \) is a power of 2, and the transform is therefore most efficient for these sizes.

3.11. Discrete Fourier Transform (numpy.fft)
The DFT is defined, with the conventions used in this implementation, in the documentation for the `numpy.fft` module.

**References**

[CT]

**Examples**

```python
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([ -3.44505240e-16 +1.14383329e-17j,
     8.00000000e+00 -5.71092652e-15j,
     2.33482938e-16 +1.22460635e-16j,
     1.64863782e-15 +1.77635684e-15j,
     9.95839695e-17 +2.33482938e-16j,
     0.00000000e+00 +1.66837030e-15j,
     1.14383329e-17 +1.22460635e-16j,
    -1.64863782e-15 +1.77635684e-15j])

>>> import matplotlib.pyplot as plt

>>> t = np.arange(256)

>>> sp = np.fft.fft(np.sin(t))

>>> freq = np.fft.fftfreq(t.shape[-1])

>>> plt.plot(freq, sp.real, freq, sp.imag)

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the `numpy.fft` documentation.

**`numpy.fft.ifft(a, n=None, axis=-1)`**

Compute the one-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the one-dimensional `n`-point discrete Fourier transform computed by `fft`. In other words, `ifft(fft(a)) == a` to within numerical accuracy. For a general description of the algorithm and definitions, see `numpy.fft`.

The input should be ordered in the same way as is returned by `fft`, i.e., `a[0]` should contain the zero frequency term, `a[1:n/2+1]` should contain the positive-frequency terms, and `a[n/2+1:]` should contain the negative-frequency terms, in order of decreasingly negative frequency. See `numpy.fft` for details.

**Parameters**

- **a**: array_like
  
  Input array, can be complex.

- **n**: int, optional

  Length of the transformed axis of the output. If `n` is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If `n` is not given, the length of the input (along the axis specified by `axis`) is used. See notes about padding issues.

- **axis**: int, optional

  Axis over which to compute the inverse DFT. If not given, the last axis is used.

**Returns**

- **out**: complex ndarray

  The truncated or zero-padded input, transformed along the axis indicated by `axis`, or the last one if `axis` is not specified.
Raises

IndexError

If axes is larger than the last axis of a.

See Also:

numpy.fft

An introduction, with definitions and general explanations.

fft

The one-dimensional (forward) FFT, of which ifft is the inverse

ifft2

The two-dimensional inverse FFT.

ifftn

The n-dimensional inverse FFT.

Notes

If the input parameter n is larger than the size of the input, the input is padded by appending zeros at the end.
Even though this is the common approach, it might lead to surprising results. If a different padding is desired, it must be performed before calling ifft.

Examples

```python
>>> np.fft.ifft([0, 4, 0, 0])
array([ 1.+0.j, 0.+1.j, -1.+0.j, 0.-1.j])
```

Create and plot a band-limited signal with random phases:

```python
>>> import matplotlib.pyplot as plt
>>> t = np.arange(400)
>>> n = np.zeros((400,), dtype=complex)
>>> n[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
>>> s = np.fft.ifft(n)
>>> plt.plot(t, s.real, 'b-', t, s.imag, 'r--')
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.legend(('real', 'imaginary'))
<matplotlib.legend.Legend object at 0x...>
>>> plt.show()
```
NumPy.fft.fft2(a, s=None, axes=(-2, -1))

Compute the 2-dimensional discrete Fourier Transform

This function computes the $n$-dimensional discrete Fourier Transform over any axes in an $M$-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

**Parameters**

- **a**: array_like
  Input array, can be complex

- **s**: sequence of ints, optional
  Shape (length of each transformed axis) of the output ($s[0]$ refers to axis 0, $s[1]$ to axis 1, etc.). This corresponds to $n$ for `fft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if $s$ is not given, the shape of the input (along the axes specified by $axes$) is used.

- **axes**: sequence of ints, optional
  Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in $axes$ means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

**Returns**

- **out**: complex ndarray
  The truncated or zero-padded input, transformed along the axes indicated by $axes$, or the last two axes if $axes$ is not given.

**Raises**

- **ValueError**
  If $s$ and $axes$ have different length, or $axes$ not given and $\text{len}(s) \neq 2$.

- **IndexError**
  If an element of $axes$ is larger than than the number of axes of $a$.

**See Also:**
**numpy.fft**

Overall view of discrete Fourier transforms, with definitions and conventions used.

**ifft2**

The inverse two-dimensional FFT.

**fft**

The one-dimensional FFT.

**fftn**

The $n$-dimensional FFT.

**fftshift**

Shifts zero-frequency terms to the center of the array. For two-dimensional input, swaps first and third quadrants, and second and fourth quadrants.

**Notes**

`ifft2` is just `fftn` with a different default for `axes`.

The output, analogously to `fft`, contains the term for zero frequency in the low-order corner of the transformed axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of the axes, in order of decreasingly negative frequency.

See `fftn` for details and a plotting example, and `numpy.fft` for definitions and conventions used.

**Examples**

```python
>>> a = np.mgrid[:5, :5][0]
>>> np.fft.fft2(a)
array([[ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 5.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [10.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [15.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [20.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]])
```

```python
numpy.fft.ifft2 (a, s=None, axes=(-2, -1))
```

Compute the 2-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the 2-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifft2(fft2(a)) == a` to within numerical accuracy. By default, the inverse transform is computed over the last two axes of the input array.

The input, analogously to `ifft`, should be ordered in the same way as is returned by `fft2`, i.e. it should have the term for zero frequency in the low-order corner of the two axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of both axes, in order of decreasingly negative frequency.

**Parameters**

- `a`: array_like
  Input array, can be complex.

- `s`: sequence of ints, optional
  Shape (length of each axis) of the output ($s[0]$ refers to axis 0, $s[1]$ to axis 1, etc.). This corresponds to $n$ for `ifft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if $s$ is not given, the shape of the input (along the axes specified by `axes`) is used. See notes for issue on `ifft` zero padding.

3.11. Discrete Fourier Transform (`numpy.fft`)
axes : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last two axes are used. A repeated
index in axes means the transform over that axis is performed multiple times. A one-
element sequence means that a one-dimensional FFT is performed.

Returns
out : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by axes, or
the last two axes if axes is not given.

Raises
ValueError

If s and axes have different length, or axes not given and len(s) != 2.

IndexError

If an element of axes is larger than than the number of axes of a.

See Also:

numpy.fft

Overall view of discrete Fourier transforms, with definitions and conventions used.

fft2

The forward 2-dimensional FFT, of which ifft2 is the inverse.

ifftn

The inverse of the n-dimensional FFT.

fft

The one-dimensional FFT.

ifft

The one-dimensional inverse FFT.

Notes

ifft2 is just ifftn with a different default for axes.

See ifftn for details and a plotting example, and numpy.fft for definition and conventions used.

Zero-padding, analogously with ifft, is performed by appending zeros to the input along the specified dimen-
sion. Although this is the common approach, it might lead to surprising results. If another form of zero padding
is desired, it must be performed before ifft2 is called.

Examples

>>> a = 4 * np.eye(4)

>>> np.fft.ifft2(a)
array([ [ 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [ 0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
       [ 0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
       [ 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j]])

numpy.fft.fftn(a, s=None, axes=None)

Compute the N-dimensional discrete Fourier Transform.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-
dimensional array by means of the Fast Fourier Transform (FFT).
Parameters

- **a**: array_like
  - Input array, can be complex.
- **s**: sequence of ints, optional
  - Shape (length of each transformed axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). This corresponds to n for fft(x, n). Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if s is not given, the shape of the input (along the axes specified by axes) is used.
- **axes**: sequence of ints, optional
  - Axes over which to compute the FFT. If not given, the last len(s) axes are used, or all axes if s is also not specified. Repeated indices in axes means that the transform over that axis is performed multiple times.

Returns

- **out**: complex ndarray
  - The truncated or zero-padded input, transformed along the axes indicated by axes, or by a combination of s and a, as explained in the parameters section above.

Raises

- **ValueError**
  - If s and axes have different length.
- **IndexError**
  - If an element of axes is larger than than the number of axes of a.

See Also:

- **numpy.fft**
  - Overall view of discrete Fourier transforms, with definitions and conventions used.
- **ifftn**
  - The inverse of fftn, the inverse n-dimensional FFT.
- **fft**
  - The one-dimensional FFT, with definitions and conventions used.
- **rfftn**
  - The n-dimensional FFT of real input.
- **fft2**
  - The two-dimensional FFT.
- **fftshift**
  - Shifts zero-frequency terms to centre of array

Notes

The output, analogously to fft, contains the term for zero frequency in the low-order corner of all axes, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

See numpy.fft for details, definitions and conventions used.
Examples

```python
>>> a = np.mgrid[:3, :3, :3][0]
>>> np.fft.fftn(a, axes=(1, 2))
array([[ 0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[ 0.+0.j,  0.+0.j,  0.+0.j],
       [ 9.+0.j,  0.+0.j,  0.+0.j]],
       [ [18.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [ 0.+0.j,  0.+0.j,  0.+0.j]])
```

```python
>>> np.fft.fftn(a, (2, 2), axes=(0, 1))
array([[ 2.+0.j,  2.+0.j,  2.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[-2.+0.j, -2.+0.j, -2.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j]])
```

```python
>>> import matplotlib.pyplot as plt
>>> [X, Y] = np.meshgrid(2 * np.pi * np.arange(200) / 12,
...                        2 * np.pi * np.arange(200) / 34)
>>> S = np.sin(X) + np.cos(Y) + np.random.uniform(0, 1, X.shape)
>>> FS = np.fft.fftn(S)
>>> plt.imshow(np.log(np.abs(np.fft.fftshift(FS))**2))
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Parameters

- **a**: array_like
  - Input array, can be complex.

- **s**: sequence of ints, optional
  - Shape (length of each transformed axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). This corresponds to n for ifft(x, n). Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If s is not given, the shape of the input (along the axes specified by axes) is used. See notes for issue on ifft zero padding.

- **axes**: sequence of ints, optional
  - Axes over which to compute the IFFT. If not given, the last len(s) axes are used, or all axes if s is also not specified. Repeated indices in axes means that the inverse transform over that axis is performed multiple times.

Returns

- **out**: complex ndarray
  - The truncated or zero-padded input, transformed along the axes indicated by axes, or by a combination of s or a, as explained in the parameters section above.

Raises

- **ValueError**
  - If s and axes have different length.

- **IndexError**
  - If an element of axes is larger than the number of axes of a.

See Also:

- **numpy.fft**
  - Overall view of discrete Fourier transforms, with definitions and conventions used.

- **fftn**
  - The forward n-dimensional FFT, of which ifftn is the inverse.

- **ifft**
  - The one-dimensional inverse FFT.

- **ifft2**
  - The two-dimensional inverse FFT.

- **ifftshift**
  - Undoes fftshift, shifts zero-frequency terms to beginning of array.

Notes

See numpy.fft for definitions and conventions used.

Zero-padding, analogously with ifft, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before ifftn is called.

Examples
```python
>>> a = np.eye(4)
>>> np.fft.ifftn(np.fft.fftn(a, axes=(0,)), axes=(1,))
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
```

Create and plot an image with band-limited frequency content:

```python
>>> import matplotlib.pyplot as plt

>>> n = np.zeros((200,200), dtype=complex)
>>> n[60:80, 20:40] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20, 20)))
>>> im = np.fft.ifftn(n).real
>>> plt.imshow(im)
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```

### 3.11.2 Real FFTs

- **`rfft(a[, n, axis])`** Compute the one-dimensional discrete Fourier Transform for real input.
- **`irfft(a[, n, axis])`** Compute the inverse of the n-point DFT for real input.
- **`rfft2(a[, s, axes])`** Compute the 2-dimensional FFT of a real array.
- **`irfft2(a[, s, axes])`** Compute the 2-dimensional inverse FFT of a real array.
- **`rfftn(a[, s, axes])`** Compute the N-dimensional discrete Fourier Transform for real input.
- **`irfftn(a[, s, axes])`** Compute the inverse of the N-dimensional FFT of real input.

```python
numpy.fft.rfft (a, n=None, axis=-1)    
```

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional $n$-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

**Parameters**

- **a** : array_like
Input array

**n**: int, optional

Number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input (along the axis specified by *axis*) is used.

**axis**: int, optional

Axis over which to compute the FFT. If not given, the last axis is used.

**Returns**

**out**: complex ndarray

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. If *n* is even, the length of the transformed axis is \((n/2)+1\). If *n* is odd, the length is \((n+1)/2\).

**Raises**

**IndexError**

If *axis* is larger than the last axis of *a*.

**See Also:**

- **numpy.fft**: For definition of the DFT and conventions used.
- **irfft**: The inverse of **rfft**.
- **fft**: The one-dimensional FFT of general (complex) input.
- **fftn**: The *n*-dimensional FFT.
- **rfftn**: The *n*-dimensional FFT of real input.

**Notes**

When the DFT is computed for purely real input, the output is Hermite-symmetric, i.e. the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. This function does not compute the negative frequency terms, and the length of the transformed axis of the output is therefore \(n//2+1\).

When \(A = \text{rfft}(a)\) and *fs* is the sampling frequency, \(A[0]\) contains the zero-frequency term \(0*\text{fs}\), which is real due to Hermitian symmetry.

If \(n\) is even, \(A[-1]\) contains the term representing both positive and negative Nyquist frequency \((+\text{fs}/2 \text{ and } -\text{fs}/2)\), and must also be purely real. If \(n\) is odd, there is no term at \(\text{fs}/2\); \(A[-1]\) contains the largest positive frequency \((\text{fs}/2*(n-1)/n)\), and is complex in the general case.

If the input \(a\) contains an imaginary part, it is silently discarded.

**Examples**

```python
>>> np.fft.fft([0, 1, 0, 0])
array([ 1.+0.j, 0.-1.j, -1.+0.j, 0.+1.j])
```

```python
>>> np.fft.rfft([0, 1, 0, 0])
array([ 1.+0.j, 0.-1.j, -1.+0.j])
```
Notice how the final element of the \texttt{fft} output is the complex conjugate of the second element, for real input. For \texttt{rfft}, this symmetry is exploited to compute only the non-negative frequency terms.

\texttt{numpy.fft.irfft} (\texttt{a}, \texttt{n=None}, \texttt{axis=-1})

Compute the inverse of the \texttt{n}-point DFT for real input.

This function computes the inverse of the one-dimensional \texttt{n}-point discrete Fourier Transform of real input computed by \texttt{rfft}. In other words, \texttt{irfft(rfft(a), len(a)) == a} to within numerical accuracy. (See Notes below for why \texttt{len(a)} is necessary here.)

The input is expected to be in the form returned by \texttt{rfft}, i.e. the real zero-frequency term followed by the complex positive frequency terms in order of increasing frequency. Since the discrete Fourier Transform of real input is Hermite-symmetric, the negative frequency terms are taken to be the complex conjugates of the corresponding positive frequency terms.

**Parameters**

\texttt{a} : array_like

The input array.

\texttt{n} : int, optional

Length of the transformed axis of the output. For \texttt{n} output points, \texttt{n//2+1} input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If \texttt{n} is not given, it is determined from the length of the input (along the axis specified by \texttt{axis}).

\texttt{axis} : int, optional

Axis over which to compute the inverse FFT.

**Returns**

\texttt{out} : ndarray

The truncated or zero-padded input, transformed along the axis indicated by \texttt{axis}, or the last one if \texttt{axis} is not specified. The length of the transformed axis is \texttt{n}, or, if \texttt{n} is not given, \texttt{2* (m-1)} where \texttt{m} is the length of the transformed axis of the input. To get an odd number of output points, \texttt{n} must be specified.

**Raises**

\texttt{IndexError}

If \texttt{axis} is larger than the last axis of \texttt{a}.

**See Also:**

\texttt{numpy.fft}

For definition of the DFT and conventions used.

\texttt{rfft}

The one-dimensional FFT of real input, of which \texttt{irfft} is inverse.

\texttt{fft}

The one-dimensional FFT.

\texttt{irfft2}

The inverse of the two-dimensional FFT of real input.

\texttt{irfftn}

The inverse of the \texttt{n}-dimensional FFT of real input.
Notes

Returns the real valued \( n \)-point inverse discrete Fourier transform of \( a \), where \( a \) contains the non-negative frequency terms of a Hermite-symmetric sequence. \( n \) is the length of the result, not the input.

If you specify an \( n \) such that \( a \) must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to \( m \) points via Fourier interpolation by:
\[
a_{\text{resamp}} = \text{irfft}(\text{rfft}(a), m).
\]

Examples

```python
>>> np.fft.ifft([1, -1j, -1, 1j])
array([ 0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j])
```

```python
>>> np.fft.irfft([1, -1j, -1])
array([ 0., 1., 0., 0.])
```

Notice how the last term in the input to the ordinary \texttt{ifft} is the complex conjugate of the second term, and the output has zero imaginary part everywhere. When calling \texttt{irfft}, the negative frequencies are not specified, and the output array is purely real.

**numpy.fft.rfft2** \((a, s=\text{None}, \text{axes}=(-2, -1))\)

Compute the 2-dimensional FFT of a real array.

**Parameters**

- \( a \) : array
  
  Input array, taken to be real.

- \( s \) : sequence of ints, optional
  
  Shape of the FFT.

- \( \text{axes} \) : sequence of ints, optional
  
  Axes over which to compute the FFT.

**Returns**

- \( \text{out} \) : ndarray
  
  The result of the real 2-D FFT.

**See Also:**

**rfftn**

Compute the N-dimensional discrete Fourier Transform for real input.

**Notes**

This is really just \texttt{rfftn} with different default behavior. For more details see \texttt{rfftn}.

**numpy.fft.irfft2** \((a, s=\text{None}, \text{axes}=(-2, -1))\)

Compute the 2-dimensional inverse FFT of a real array.

**Parameters**

- \( a \) : array_like
  
  The input array.

- \( s \) : sequence of ints, optional
  
  Shape of the inverse FFT.

- \( \text{axes} \) : sequence of ints, optional
  
  The axes over which to compute the inverse fft. Default is the last two axes.
Returns

out : ndarray

The result of the inverse real 2-D FFT.

See Also:

irfftn

Compute the inverse of the N-dimensional FFT of real input.

Notes

This is really irfftn with different defaults. For more details see irfftn.

numpy.fft.rfftn(a, s=None, axes=None)

Compute the N-dimensional discrete Fourier Transform for real input.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional real array by means of the Fast Fourier Transform (FFT). By default, all axes are transformed, with the real transform performed over the last axis, while the remaining transforms are complex.

Parameters

a : array_like

Input array, taken to be real.

s : sequence of ints, optional

Shape (length along each transformed axis) to use from the input. (s[0] refers to axis 0, s[1] to axis 1, etc.). The final element of s corresponds to n for rfft(x, n), while for the remaining axes, it corresponds to n for fft(x, n). Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if s is not given, the shape of the input (along the axes specified by axes) is used.

axes : sequence of ints, optional

Axes over which to compute the FFT. If not given, the last len(s) axes are used, or all axes if s is also not specified.

Returns

out : complex ndarray

The truncated or zero-padded input, transformed along the axes indicated by axes, or by a combination of s and a, as explained in the parameters section above. The length of the last axis transformed will be s[-1]//2+1, while the remaining transformed axes will have lengths according to s, or unchanged from the input.

Raises

ValueError

If s and axes have different length.

IndexError

If an element of axes is larger than than the number of axes of a.

See Also:

irfftn

The inverse of rfftn, i.e. the inverse of the n-dimensional FFT of real input.
**fft**

The one-dimensional FFT, with definitions and conventions used.

**rfft**

The one-dimensional FFT of real input.

**fftn**

The n-dimensional FFT.

**rfft2**

The two-dimensional FFT of real input.

**Notes**

The transform for real input is performed over the last transformation axis, as by `rfft`, then the transform over the remaining axes is performed as by `fftn`. The order of the output is as for `rfft` for the final transformation axis, and as for `fftn` for the remaining transformation axes.

See `fft` for details, definitions and conventions used.

**Examples**

```python
>>> a = np.ones((2, 2, 2))
>>> np.fft.rfftn(a)
array([[8.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j]])

>>> np.fft.rfftn(a, axes=(2, 0))
array([[4.+0.j, 0.+0.j],
       [4.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j]])
```

**numpy.fft.irfftn(a, s=None, axes=None)**

Compute the inverse of the N-dimensional FFT of real input.

This function computes the inverse of the N-dimensional discrete Fourier Transform for real input over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `irfftn(rfftn(a), a.shape) == a` to within numerical accuracy. (The `a.shape` is necessary like `len(a)` is for `irfft`, and for the same reason.)

The input should be ordered in the same way as is returned by `rfftn`, i.e. as for `irfft` for the final transformation axis, and as for `irfftn` along all the other axes.

**Parameters**

- **a**: array_like
  
  Input array.

- **s**: sequence of ints, optional
  
  Shape (length of each transformed axis) of the output (s[0] refers to axis 0, s[1] to axis 1, etc.). s is also the number of input points used along this axis, except for the last axis, where s[-1]/2+1 points of the input are used. Along any axis, if the shape indicated by s is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If s is not given, the shape of the input (along the axes specified by `axes`) is used.

- **axes**: sequence of ints, optional
Axes over which to compute the inverse FFT. If not given, the last \textit{len(s)} axes are used, or all axes if \textit{s} is also not specified. Repeated indices in \textit{axes} means that the inverse transform over that axis is performed multiple times.

**Returns**

\textbf{out} : ndarray

The truncated or zero-padded input, transformed along the axes indicated by \textit{axes}, or by a combination of \textit{s} or \textit{a}, as explained in the parameters section above. The length of each transformed axis is as given by the corresponding element of \textit{s}, or the length of the input in every axis except for the last one if \textit{s} is not given. In the final transformed axis the length of the output when \textit{s} is not given is \(2 \times (m-1)\) where \(m\) is the length of the final transformed axis of the input. To get an odd number of output points in the final axis, \textit{s} must be specified.

**Raises**

\textbf{ValueError}

If \textit{s} and \textit{axes} have different length.

\textbf{IndexError}

If an element of \textit{axes} is larger than than the number of axes of \textit{a}.

**See Also:**

\textbf{rfft2}

The forward \(n\)-dimensional FFT of real input, of which \textbf{ifftn} is the inverse.

\textbf{fft}

The one-dimensional FFT, with definitions and conventions used.

\textbf{irfft}

The inverse of the one-dimensional FFT of real input.

\textbf{irfft2}

The inverse of the two-dimensional FFT of real input.

**Notes**

See \textbf{fft} for definitions and conventions used.

See \textbf{rfft} for definitions and conventions used for real input.

**Examples**

```python
>>> a = np.zeros((3, 2, 2))
>>> a[0, 0, 0] = 3 * 2 * 2
>>> np.fft.irfft2(a)
array([[[ 1.,  1.],
       [ 1.,  1.]],
       [[ 1.,  1.],
       [ 1.,  1.]],
       [[ 1.,  1.],
       [ 1.,  1.]]])
```

### 3.11.3 Hermitian FFTs

Continued on next page
Table 3.46 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hfft(a[, n, axis])</code></td>
<td>Compute the FFT of a signal whose spectrum has Hermitian symmetry.</td>
</tr>
<tr>
<td><code>ihfft(a[, n, axis])</code></td>
<td>Compute the inverse FFT of a signal whose spectrum has Hermitian symmetry.</td>
</tr>
</tbody>
</table>

```python
numpy.fft.hfft(a, n=None, axis=-1)
```

Compute the FFT of a signal whose spectrum has Hermitian symmetry.

**Parameters**
- `a` : array_like
  The input array.
- `n` : int, optional
  The length of the FFT.
- `axis` : int, optional
  The axis over which to compute the FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.

**Returns**
- `out` : ndarray
  The transformed input.

**See Also:**
- `rfft`
  Compute the one-dimensional FFT for real input.
- `ihfft`
  The inverse of `hfft`.

**Notes**

`hfft/ihfft` are a pair analogous to `rfft/irfft`, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's `hfft` for which you must supply the length of the result if it is to be odd: `ihfft(hfft(a), len(a)) == a`, within numerical accuracy.

**Examples**

```python
>>> signal = np.array([[1, 1.j], [-1.j, 2]])
>>> np.conj(signal.T) - signal  # check Hermitian symmetry
array([[ 0.-0.j, 0.+0.j],
       [ 0.+0.j, 0.-0.j]])
>>> freq_spectrum = np.fft.hfft(signal)
```

```python
>>> freq_spectrum
array([[ 1., 1.],
       [ 2., -2.]])
```

```python
numpy.fft.ihfft(a, n=None, axis=-1)
```

Compute the inverse FFT of a signal whose spectrum has Hermitian symmetry.

**Parameters**
- `a` : array_like
  Input array.
- `n` : int, optional
Length of the inverse FFT.

axis : int, optional

Axis over which to compute the inverse FFT, assuming Hermitian symmetry of the spectrum. Default is the last axis.

Returns

out : ndarray

The transformed input.

See Also:

hfft, ifft

Notes

hfft/ihfft are a pair analogous to rfft/irfft, but for the opposite case: here the signal is real in the frequency domain and has Hermite symmetry in the time domain. So here it's hfft for which you must supply the length of the result if it is to be odd: ihfft(hfft(a), len(a)) == a, within numerical accuracy.

3.11.4 Helper routines

numpy.fft.fftfreq(n[, d]) Return the Discrete Fourier Transform sample frequencies.

The returned float array f contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length n and a sample spacing d:

\[
f = \begin{cases} 
0, 1, \ldots, \frac{n}{2}-1, -\frac{n}{2}, \ldots, -1 \end{cases} \frac{\text{d}\times n}{d} \quad \text{if } n \text{ is even} \\
0, 1, \ldots, \frac{(n-1)}{2}, -\frac{(n-1)}{2}, \ldots, -1 \times n}{d} \quad \text{if } n \text{ is odd}
\]

Parameters

n : int

Window length.

d : scalar, optional

Sample spacing (inverse of the sampling rate). Defaults to 1.

Returns

f : ndarray

Array of length n containing the sample frequencies.

Examples

>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
```python
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0. , 1.25, 2.5 , 3.75, -5. , -3.75, -2.5 , -1.25])
```

`numpy.fft.rfftfreq(n, d=1.0)`

Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft).

The returned float array `f` contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length `n` and a sample spacing `d`:

\[
\begin{align*}
f &= \{0, 1, \ldots, n/2-1, n/2\} / (d*n) & \text{if } n \text{ is even} \\
f &= \{0, 1, \ldots, (n-1)/2-1, (n-1)/2\} / (d*n) & \text{if } n \text{ is odd}
\end{align*}
\]

Unlike `fftfreq` (but like `scipy.fftpack.rfftfreq`) the Nyquist frequency component is considered to be positive.

**Parameters**
- `n` : int
  Window length.
- `d` : scalar, optional
  Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns**
- `f` : ndarray
  Array of length `n//2 + 1` containing the sample frequencies.

**Examples**

```python
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
>>> fourier = np.fft.rfft(signal)
>>> n = signal.size
>>> sample_rate = 100
>>> freq = np.fft.fftfreq(n, d=1./sample_rate)
>>> freq
array([ 0., 10., 20., 30., 40., -50., -40., -30., -20., -10.])
>>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
>>> freq
array([ 0., 10., 20., 30., 40., 50.])
```

`numpy.fft.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

**Parameters**
- `x` : array_like
  Input array.
- `axes` : int or shape tuple, optional
  Axes over which to shift. Default is None, which shifts all axes.

**Returns**
- `y` : ndarray

### 3.11. Discrete Fourier Transform (numpy.fft)
The shifted array.

See Also:

iftshift
  The inverse of fftshift.

Examples

```python
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2.,  3.,  4., -5., -4., -3., -2., -1.])
```

```python
>>> np.fft.ifftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```python
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
        [ 3.,  4., -4.],
        [-3., -2., -1.]])
```

```python
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2.,  0.,  1.],
     [-4.,  3.,  4.],
     [-1., -3., -2.]])
```

numppy.fft.iftshift(x, axes=\text{None})

The inverse of fftshift.

Parameters

- `x`: array_like
  Input array.

- `axes`: int or shape tuple, optional
  Axes over which to calculate. Defaults to None, which shifts all axes.

Returns

- `y`: ndarray
  The shifted array.

See Also:

fftshift
  Shift zero-frequency component to the center of the spectrum.

Examples

```python
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
        [ 3.,  4., -4.],
        [-3., -2., -1.]])
```

```python
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0.,  1.,  2.],
        [ 3.,  4., -4.],
        [-3., -2., -1.]])
```
3.11.5 Background information

Fourier analysis is fundamentally a method for expressing a function as a sum of periodic components, and for recovering the function from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT). The DFT has become a mainstay of numerical computing in part because of a very fast algorithm for computing it, called the Fast Fourier Transform (FFT), which was known to Gauss (1805) and was brought to light in its current form by Cooley and Tukey [CT]. Press et al. [NR] provide an accessible introduction to Fourier analysis and its applications.

Because the discrete Fourier transform separates its input into components that contribute at discrete frequencies, it has a great number of applications in digital signal processing, e.g., for filtering, and in this context the discretized input to the transform is customarily referred to as a signal, which exists in the time domain. The output is called a spectrum or transform and exists in the frequency domain.

3.11.6 Implementation details

There are many ways to define the DFT, varying in the sign of the exponent, normalization, etc. In this implementation, the DFT is defined as

\[ A_k = \sum_{m=0}^{n-1} a_m \exp\left\{ -2\pi i \frac{mk}{n} \right\} \quad k = 0, \ldots, n - 1. \]

The DFT is in general defined for complex inputs and outputs, and a single-frequency component at linear frequency \( f \) is represented by a complex exponential \( a_m = \exp\left\{ 2\pi ifm\Delta t \right\} \), where \( \Delta t \) is the sampling interval.

The values in the result follow so-called “standard” order: If \( A = \text{fft}(a, n) \), then \( A[0] \) contains the zero-frequency term (the mean of the signal), which is always purely real for real inputs. Then \( A[1:n/2] \) contains the positive-frequency terms, and \( A[n/2+1:] \) contains the negative-frequency terms, in order of decreasingly negative frequency. For an even number of input points, \( A[n/2] \) represents both positive and negative Nyquist frequency, and is also purely real for real input. For an odd number of input points, \( A[(n-1)/2] \) contains the largest positive frequency, while \( A[(n+1)/2] \) contains the largest negative frequency. The routine \( \text{np.fft.fftfreq}(n) \) returns an array giving the frequencies of corresponding elements in the output. The routine \( \text{np.fft.fftfreq}(A) \) shifts transforms and their frequencies to put the zero-frequency components in the middle, and \( \text{np.fft.ifftshift}(A) \) undoes that shift.

When the input \( a \) is a time-domain signal and \( A = \text{fft}(a) \), \( \text{np.abs}(A) \) is its amplitude spectrum and \( \text{np.abs}(A)^2 \) is its power spectrum. The phase spectrum is obtained by \( \text{np.angle}(A) \).

The inverse DFT is defined as

\[ a_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left\{ 2\pi i \frac{mk}{n} \right\} \quad m = 0, \ldots, n - 1. \]

It differs from the forward transform by the sign of the exponential argument and the normalization by \( 1/n \).

3.11.7 Real and Hermitian transforms

When the input is purely real, its transform is Hermitian, i.e., the component at frequency \( f_k \) is the complex conjugate of the component at frequency \( -f_k \), which means that for real inputs there is no information in the negative frequency components that is not already available from the positive frequency components. The family of \( \text{rfft} \) functions is designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components.
up to and including the Nyquist frequency. Thus, \( n \) input points produce \( n/2+1 \) complex output points. The inverses of this family assumes the same symmetry of its input, and for an output of \( n \) points uses \( n/2+1 \) input points.

Correspondingly, when the spectrum is purely real, the signal is Hermitian. The `hfft` family of functions exploits this symmetry by using \( n/2+1 \) complex points in the input (time) domain for \( n \) real points in the frequency domain.

In higher dimensions, FFTs are used, e.g., for image analysis and filtering. The computational efficiency of the FFT means that it can also be a faster way to compute large convolutions, using the property that a convolution in the time domain is equivalent to a point-by-point multiplication in the frequency domain.

### 3.11.8 Higher dimensions

In two dimensions, the DFT is defined as

\[
A_{kl} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{mn} \exp \left\{ -2\pi i \left( \frac{mk}{M} + \frac{nl}{N} \right) \right\}, \quad k = 0, \ldots, M - 1; \quad l = 0, \ldots, N - 1,
\]

which extends in the obvious way to higher dimensions, and the inverses in higher dimensions also extend in the same way.

### 3.11.9 References

### 3.11.10 Examples

For examples, see the various functions.

### 3.12 Financial functions

#### 3.12.1 Simple financial functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fv(rate, nper, pmt, pv[, when])</code></td>
<td>Compute the future value.</td>
</tr>
<tr>
<td><code>pv(rate, nper, pmt[, fv, when])</code></td>
<td>Compute the present value.</td>
</tr>
<tr>
<td><code>npv(rate, values)</code></td>
<td>Returns the NPV (Net Present Value) of a cash flow series.</td>
</tr>
<tr>
<td><code>pmt(rate, nper, pv[, fv, when])</code></td>
<td>Compute the payment against loan principal.</td>
</tr>
<tr>
<td><code>ppmt(rate, per, nper, pv[, fv, when])</code></td>
<td>Compute the payment against loan principal.</td>
</tr>
<tr>
<td><code>ipmt(rate, per, nper, pv[, fv, when])</code></td>
<td>Compute the interest portion of a payment.</td>
</tr>
<tr>
<td><code>irr(values)</code></td>
<td>Return the Internal Rate of Return (IRR).</td>
</tr>
<tr>
<td><code>mirr(values, finance_rate, reinvest_rate)</code></td>
<td>Modified internal rate of return.</td>
</tr>
<tr>
<td><code>nper(rate, pmt, pv[, fv, when])</code></td>
<td>Compute the number of periodic payments.</td>
</tr>
<tr>
<td><code>rate(nper, pmt, pv[, fv, when[, guess, tol, ...]])</code></td>
<td>Compute the rate of interest per period.</td>
</tr>
</tbody>
</table>

```python
numpy.fv(rate, nper, pmt, pv[, when='end'])
```

`Compute the future value.`

**Given:**

- a present value, `pv`

---

630 Chapter 3. Routines
• an interest rate compounded once per period, of which there are
• nper total
• a (fixed) payment, pmt, paid either
• at the beginning (when = {'begin', 1}) or the end (when = {'end', 0}) of each period

Return:
the value at the end of the nper periods

Parameters
rate : scalar or array_like of shape(M, )
Rate of interest as decimal (not per cent) per period
nper : scalar or array_like of shape(M, )
Number of compounding periods
pmt : scalar or array_like of shape(M, )
Payment
pv : scalar or array_like of shape(M, )
Present value
when : {{'begin', 1}, {'end', 0}}, {string, int}, optional
When payments are due (‘begin’ (1) or ‘end’ (0)). Defaults to {‘end’, 0}.

Returns
out : ndarray
Future values. If all input is scalar, returns a scalar float. If any input is array_like, returns future values for each input element. If multiple inputs are array_like, they all must have the same shape.

Notes
The future value is computed by solving the equation:

fv + pv*(1+rate)**nper +
pmt*(1 + rate*when)/rate*{(1 + rate)**nper - 1} == 0

or, when rate == 0:
fv + pv + pmt * nper == 0

References
[WRW]

Examples
What is the future value after 10 years of saving $100 now, with an additional monthly savings of $100. Assume the interest rate is 5% (annually) compounded monthly?

>>> np.fv(0.05/12, 10*12, -100, -100)
15692.928894335748
By convention, the negative sign represents cash flow out (i.e. money not available today). Thus, saving $100 a month at 5% annual interest leads to $15,692.93 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let’s compare different interest rates from the example above.

```python
>>> a = np.array((0.05, 0.06, 0.07))/12
>>> np.fv(a, 10*12, -100, -100)
array([ 15692.92889434, 16569.87435405, 17509.44688102])
```

numpy.\texttt{pv} (\texttt{rate}, \texttt{nper}, \texttt{pmt}, \texttt{fv}=0.0, \texttt{when}='end')

Compute the present value.

**Given:**

- a future value, \texttt{fv}
- an interest \texttt{rate} compounded once per period, of which there are
- \texttt{nper} total
- a (fixed) payment, \texttt{pmt}, paid either
- at the beginning (\texttt{when} = {'begin', 1}) or the end (\texttt{when} = {'end', 0}) of each period

**Return:**

the value now

**Parameters**

- \texttt{rate} : array_like
  Rate of interest (per period)
- \texttt{nper} : array_like
  Number of compounding periods
- \texttt{pmt} : array_like
  Payment
- \texttt{fv} : array_like, optional
  Future value
- \texttt{when} : {{‘begin’, 1}, {‘end’, 0}}, {string, int}, optional
  When payments are due (‘begin’ (1) or ‘end’ (0))

**Returns**

- \texttt{out} : ndarray, float
  Present value of a series of payments or investments.

**Notes**

The present value is computed by solving the equation:

\[
fv + pv*(1 + rate)**nper + pmt*(1 + rate*when)/rate*(1 + rate)**nper - 1 = 0
\]

or, when \texttt{rate} = 0:
\[ fv + pv + pmt \times nper = 0 \]

for \( pv \), which is then returned.

**References**

[WRW]

**Examples**

What is the present value (e.g., the initial investment) of an investment that needs to total $15692.93 after 10 years of saving $100 every month? Assume the interest rate is 5\% (annually) compounded monthly.

```python
>>> np.pv(0.05/12, 10*12, -100, 15692.93)
-100.00067131625819
```

By convention, the negative sign represents cash flow out (i.e., money not available today). Thus, to end up with $15,692.93 in 10 years saving $100 a month at 5\% annual interest, one’s initial deposit should also be $100.

If any input is array_like, \( pv \) returns an array of equal shape. Let’s compare different interest rates in the example above:

```python
>>> a = np.array((0.05, 0.04, 0.03))/12
>>> np.pv(a, 10*12, -100, 15692.93)
array([-100.00067132,  -649.26771385,  -1273.78633713])
```

So, to end up with the same $15692.93 under the same $100 per month “savings plan,” for annual interest rates of 4\% and 3\%, one would need initial investments of $649.27 and $1273.79, respectively.

**numpy.\texttt{npv}(rate, values)**

Returns the NPV (Net Present Value) of a cash flow series.

**Parameters**

- **rate** : scalar
  The discount rate.

- **values** : array_like, shape(M,)
  The values of the time series of cash flows. The (fixed) time interval between cash flow “events” must be the same as that for which \( rate \) is given (i.e., if \( rate \) is per year, then precisely a year is understood to elapse between each cash flow event). By convention, investments or “deposits” are negative, income or “withdrawals” are positive; \( values \) must begin with the initial investment, thus \( values[0] \) will typically be negative.

**Returns**

- **out** : float
  The NPV of the input cash flow series \( values \) at the discount \( rate \).

**Notes**

Returns the result of: [G51]

\[
\sum_{t=0}^{M-1} \frac{values_t}{(1 + rate)^t}
\]

**References**

[G51]
Examples

```python
>>> np.npv(0.281, [-100, 39, 59, 55, 20])
-0.0084785916384548798
```

(Compare with the Example given for numpy.lib.financial.irr)

```python
numpy.pmt(rate, nper, pv, fv=0, when='end')
```

Compute the payment against loan principal plus interest.

**Given:**

- a present value, `pv` (e.g., an amount borrowed)
- a future value, `fv` (e.g., 0)
- an interest **rate** compounded once per period, of which there are
- `nper` total
- and (optional) specification of whether payment is made at the beginning (`when = ‘begin’, 1`) or the end (`when = ‘end’, 0`) of each period

**Return:**

the (fixed) periodic payment.

**Parameters**

- `rate`: array_like
  Rate of interest (per period)
- `nper`: array_like
  Number of compounding periods
- `pv`: array_like
  Present value
- `fv`: array_like (optional)
  Future value (default = 0)
- `when`: {'begin', 1}, {'end', 0}, {string, int}
  When payments are due (‘begin’ (1) or ‘end’ (0))

**Returns**

- `out`: ndarray
  Payment against loan plus interest. If all input is scalar, returns a scalar float. If any input is array_like, returns payment for each input element. If multiple inputs are array_like, they all must have the same shape.

**Notes**

The payment is computed by solving the equation:

\[
fv + pv \times (1 + rate)^{nper} + \\
pmt \times (1 + rate \times when) / rate \times ((1 + rate)^{nper} - 1) = 0
\]

or, when `rate == 0:`
\[fv + pv + pmt \times nper = 0\] for \(pmt\).

Note that computing a monthly mortgage payment is only one use for this function. For example, \(pmt\) returns the periodic deposit one must make to achieve a specified future balance given an initial deposit, a fixed, periodically compounded interest rate, and the total number of periods.

References

[WRW]

Examples

What is the monthly payment needed to pay off a $200,000 loan in 15 years at an annual interest rate of 7.5%?

```python
>>> np.pmt(0.075/12, 12*15, 200000)
-1854.0247200054619
```

In order to pay-off (i.e., have a future-value of 0) the $200,000 obtained today, a monthly payment of $1,854.02 would be required. Note that this example illustrates usage of \(fv\) having a default value of 0.

```
numpy.ppmt(rate, per, nper, pv, fv=0.0, when='end')
```

Compute the payment against loan principal.

**Parameters**

- **rate**: array_like
  Rate of interest (per period)
- **per**: array_like, int
  Amount paid against the loan changes. The \(per\) is the period of interest.
- **nper**: array_like
  Number of compounding periods
- **pv**: array_like
  Present value
- **fv**: array_like, optional
  Future value
- **when**: {{‘begin’, 1}, {‘end’, 0}}, {string, int}
  When payments are due (‘begin’ (1) or ‘end’ (0))

**See Also:**

- `pmt`, `pv`, `ipmt`

```
numpy.ipmt(rate, per, nper, pv, fv=0.0, when='end')
```

Compute the interest portion of a payment.

**Parameters**

- **rate**: scalar or array_like of shape(M, )
  Rate of interest as decimal (not per cent) per period
- **per**: scalar or array_like of shape(M, )
  Interest paid against the loan changes during the life or the loan. The \(per\) is the payment period to calculate the interest amount.
nper : scalar or array_like of shape(M, )
    Number of compounding periods
pv : scalar or array_like of shape(M, )
    Present value
fv : scalar or array_like of shape(M, ), optional
    Future value
when : {
    'begin', 1
    , 'end', 0
    }, {string, int}, optional
    When payments are due (‘begin’ (1) or ‘end’ (0)). Defaults to {‘end’, 0}.

Returns
out : ndarray
    Interest portion of payment. If all input is scalar, returns a scalar float. If any input
    is array_like, returns interest payment for each input element. If multiple inputs are
    array_like, they all must have the same shape.

See Also:
ppmt, pmt, pv

Notes
The total payment is made up of payment against principal plus interest.
pmt = ppmt + ipmt

Examples
What is the amortization schedule for a 1 year loan of $2500 at 8.24% interest per year compounded monthly?

```python
>>> principal = 2500.00
```

The ‘per’ variable represents the periods of the loan. Remember that financial equations start the period count
at 1!

```python
>>> per = np.arange(1*12) + 1
>>> ipmt = np.ipmt(0.0824/12, per, 1*12, principal)
>>> ppmt = np.ppmt(0.0824/12, per, 1*12, principal)
```

Each element of the sum of the ‘ipmt’ and ‘ppmt’ arrays should equal ‘pmt’.

```python
>>> pmt = np.pmt(0.0824/12, 1*12, principal)
>>> np.allclose(ipmt + ppmt, pmt)
```

```python
True
```

```python
>>> fmt = '{0:2d} {1:8.2f} {2:8.2f} {3:8.2f}
```

```python
>>> for payment in per:
...     index = payment - 1
...     principal = principal + ppmt[index]
...     print fmt.format(payment, ppmt[index], ipmt[index], principal)
```

```
1 -200.58 -17.17 2299.42
2 -201.96 -15.79 2097.46
3 -203.35 -14.40 1894.11
4 -204.74 -13.01 1689.37
5 -206.15 -11.60 1483.22
6 -207.56 -10.18 1275.66
7 -208.99 -8.76 1066.67
```
numpy.irr(values)

Return the Internal Rate of Return (IRR).

This is the “average” periodically compounded rate of return that gives a net present value of 0.0; for a more complete explanation, see Notes below.

Parameters
values : array_like, shape(N)

Input cash flows per time period. By convention, net “deposits” are negative and net “withdrawals” are positive. Thus, for example, at least the first element of values, which represents the initial investment, will typically be negative.

Returns
out : float

Internal Rate of Return for periodic input values.

Notes
The IRR is perhaps best understood through an example (illustrated using np.irr in the Examples section below). Suppose one invests 100 units and then makes the following withdrawals at regular (fixed) intervals: 39, 59, 55, 20. Assuming the ending value is 0, one’s 100 unit investment yields 173 units; however, due to the combination of compounding and the periodic withdrawals, the “average” rate of return is neither simply 0.73/4 nor \((1.73)^{0.25}-1\). Rather, it is the solution (for \(r\)) of the equation:

\[-100 + \frac{39}{1 + r} + \frac{59}{(1 + r)^2} + \frac{55}{(1 + r)^3} + \frac{20}{(1 + r)^4} = 0\]

In general, for \(values = [v_0, v_1, ... v_M]\), irr is the solution of the equation: [G32]

\[\sum_{t=0}^{M} \frac{v_t}{(1 + irr)^t} = 0\]

References
[G32]

Examples
>>> round(irr([-100, 39, 59, 55, 20]), 5)
0.28095
>>> round(irr([-100, 0, 0, 74]), 5)
-0.0955
>>> round(irr([-100, 100, 0, -7]), 5)
-0.0833
>>> round(irr([-100, 100, 0, 7]), 5)
0.06206
>>> round(irr([-5, 10.5, 1, -8, 1]), 5)
0.0886

(Compare with the Example given for numpy.lib.financial.npv)

numpy.mirr(values, finance_rate, reinvest_rate)
Modified internal rate of return.

Parameters
values : array_like
Cash flows (must contain at least one positive and one negative value) or nan is returned.
The first value is considered a sunk cost at time zero.

finance_rate : scalar
Interest rate paid on the cash flows

reinvest_rate : scalar
Interest rate received on the cash flows upon reinvestment

Returns
out : float
Modified internal rate of return

numpy.nper(rate, pmt, pv, fv=0, when='end')
Compute the number of periodic payments.

Parameters
rate : array_like
Rate of interest (per period)
pmt : array_like
Payment
pv : array_like
Present value
fv : array_like, optional
Future value
when : {{‘begin’, 1}, {‘end’, 0}}, {string, int}, optional
When payments are due (‘begin’ (1) or ‘end’ (0))

Notes
The number of periods nper is computed by solving the equation:
fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate*{(1+rate)**nper-1} = 0

but if rate = 0 then:
fv + pv + pmt*nper = 0
Examples

If you only had $150/month to pay towards the loan, how long would it take to pay-off a loan of $8,000 at 7% annual interest?

```python
>>> print round(np.nper(0.07/12, -150, 8000), 5)
64.07335
```

So, over 64 months would be required to pay off the loan.

The same analysis could be done with several different interest rates and/or payments and/or total amounts to produce an entire table.

```python
>>> np.nper(*(np.ogrid[0.07/12: 0.08/12: 0.01/12,
... -150 : -99 : 50 ,
... 8000 : 9001 : 1000]))
array([[ 64.07334877, 74.06368256],
       [ 108.07548412, 127.99022654],
       [ 66.12443902, 76.87897353],
       [ 114.70165583, 137.90124779]]))
```

`numpy.rate(nper, pmt, pv, fv, when='end', guess=0.1, tol=1e-06, maxiter=100)`

Compute the rate of interest per period.

**Parameters**

- `nper`: array_like
  - Number of compounding periods
- `pmt`: array_like
  - Payment
- `pv`: array_like
  - Present value
- `fv`: array_like
  - Future value
- `when`: {'begin', 1}, {'end', 0}, {string, int}, optional
  - When payments are due ('begin' (1) or 'end' (0))
- `guess`: float, optional
  - Starting guess for solving the rate of interest
- `tol`: float, optional
  - Required tolerance for the solution
- `maxiter`: int, optional
  - Maximum iterations in finding the solution

**Notes**

The rate of interest is computed by iteratively solving the (non-linear) equation:

\[ fv + pv \times (1+rate)^{nper} + pmt \times (1+rate^{when}) / rate \times ((1+rate)^{nper} - 1) = 0 \]

for `rate`. 

3.12. Financial functions
3.13 Functional programming

```
np.apply_along_axis(func1d, axis, arr, *args)
``` Apply a function to 1-D slices along the given axis.

Execute `func1d(a, *args)` where `func1d` operates on 1-D arrays and `a` is a 1-D slice of `arr` along `axis`.

**Parameters**

- **func1d**: function
  
  This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis.

- **axis**: integer
  
  Axis along which `arr` is sliced.

- **arr**: ndarray
  
  Input array.

- **args**: any
  
  Additional arguments to `func1d`.

**Returns**

```
apply_along_axis : ndarray
```

The output array. The shape of `outarr` is identical to the shape of `arr`, except along the `axis` dimension, where the length of `outarr` is equal to the size of the return value of `func1d`. If `func1d` returns a scalar `outarr` will have one fewer dimensions than `arr`.

**See Also:**

- `apply_over_axes`

  Apply a function repeatedly over multiple axes.

**Examples**

```python
>>> def my_func(a):
...     return (a[0] + a[-1]) * 0.5
... >>> b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> np.apply_along_axis(my_func, 0, b)
```
array([ 4.,  5.,  6.])
>>> np.apply_along_axis(my_func, 1, b)
array([ 2.,  5.,  8.])

For a function that doesn’t return a scalar, the number of dimensions in `outarr` is the same as `arr`.

```python
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

`numpy.apply_over_axes(func, a, axes)`

Apply a function repeatedly over multiple axes. `func` is called as `res = func(a, axis)`, where `axis` is the first element of `axes`. The result `res` of the function call must have either the same dimensions as `a` or one less dimension. If `res` has one less dimension than `a`, a dimension is inserted before `axis`. The call to `func` is then repeated for each axis in `axes`, with `res` as the first argument.

**Parameters**
- `func` : function
  - This function must take two arguments, `func(a, axis)`.
- `a` : array_like
  - Input array.
- `axes` : array_like
  - Axes over which `func` is applied; the elements must be integers.

**Returns**
- `apply_over_axis` : ndarray
  - The output array. The number of dimensions is the same as `a`, but the shape can be different. This depends on whether `func` changes the shape of its output with respect to its input.

See Also:
- `apply_along_axis`

**Examples**

```python
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
        [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
```

Sum over axes 0 and 2. The result has same number of dimensions as the original array:

```python
>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[ 60],
        [ 92],
        [124]]])
```
class numpy.vectorize(pyfunc, otypes='', doc=None, excluded=None, cache=False)

Generalized function class.

Define a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns a numpy array as output. The vectorized function evaluates pyfunc over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The data type of the output of vectorized is determined by calling the function with the first element of the input. This can be avoided by specifying the otypes argument.

Parameters

pyfunc : callable
A python function or method.

otypes : str or list of dtypes, optional
The output data type. It must be specified as either a string of typecode characters or a list of data type specifiers. There should be one data type specifier for each output.

doc : str, optional
The docstring for the function. If None, the docstring will be the pyfunc.__doc__.

excluded : set, optional
Set of strings or integers representing the positional or keyword arguments for which the function will not be vectorized. These will be passed directly to pyfunc unmodified.

New in version 1.7.0.

cache : bool, optional
If True, then cache the first function call that determines the number of outputs if otypes is not provided.

New in version 1.7.0.

Returns

vectorized : callable
Vectorized function.

Notes

The vectorize function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.

If otypes is not specified, then a call to the function with the first argument will be used to determine the number of outputs. The results of this call will be cached if cache is True to prevent calling the function twice. However, to implement the cache, the original function must be wrapped which will slow down subsequent calls, so only do this if your function is expensive.

The new keyword argument interface and excluded argument support further degrades performance.

Examples

```python
>>> def myfunc(a, b):
...     "Return a-b if a>b, otherwise return a+b"
...     if a > b:
...         return a - b
...     else:
...         return a + b
```
```python
>>> vfunc = np.vectorize(myfunc)
>>> vfunc([1, 2, 3, 4], 2)
array([3, 4, 1, 2])
```

The docstring is taken from the input function to `vectorize` unless it is specified
```python
>>> vfunc.__doc__
'Return a-b if a>b, otherwise return a+b'
>>> vfunc = np.vectorize(myfunc, doc='Vectorized `myfunc`')
>>> vfunc.__doc__
'Vectorized `myfunc`'
```

The output type is determined by evaluating the first element of the input, unless it is specified
```python
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<type 'numpy.int32'>
>>> vfunc = np.vectorize(myfunc, outtypes=[np.float])
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<type 'numpy.float64'>
```

The `excluded` argument can be used to prevent vectorizing over certain arguments. This can be useful for array-like arguments of a fixed length such as the coefficients for a polynomial as in `polyval`:
```python
>>> def mypolyval(p, x):
...     _p = list(p)
...     res = _p.pop(0)
...     while _p:
...         res = res*x + _p.pop(0)
...     return res
>>> vpolyval = np.vectorize(mypolyval, excluded=['p'])
>>> vpolyval(p=[1, 2, 3], x=[0, 1])
array([3, 6])
```

Positional arguments may also be excluded by specifying their position:
```python
>>> vpolyval.excluded.add(0)
>>> vpolyval([1, 2, 3], x=[0, 1])
array([3, 6])
```

**Methods**

```python
__call__(*args, **kwargs) Return arrays with the results of `pyfunc` broadcast (vectorized) over
```

```python
vectorize.__call__(*args, **kwargs)
Return arrays with the results of `pyfunc` broadcast (vectorized) over `args` and `kwargs` not in `excluded`.
```

```python
numpy.frompyfunc(func, nin, nout)
Takes an arbitrary Python function and returns a Numpy ufunc.
Can be used, for example, to add broadcasting to a built-in Python function (see Examples section).
```

**Parameters**

- **func**: Python function object
  An arbitrary Python function.
- **nin**: int

3.13. Functional programming
The number of input arguments.

\textbf{nout} : int

The number of objects returned by \texttt{func}.

\textbf{Returns}

\textbf{out} : ufunc

Returns a Numpy universal function (ufunc) object.

\textbf{Notes}

The returned ufunc always returns PyObject arrays.

\textbf{Examples}

Use frompyfunc to add broadcasting to the Python function \texttt{oct}:

\begin{verbatim}
>>> oct_array = np.frompyfunc(oct, 1, 1)
>>> oct_array(np.array((10, 30, 100)))
array([012, 036, 0144], dtype=object)
>>> np.array((oct(10), oct(30), oct(100))) # for comparison
array(['012', '036', '0144'], dtype='|S4')
\end{verbatim}

\textbf{numpy.piecewise} \texttt{(x, condlist, funclist, *args, **kw)}

Evaluate a piecewise-defined function.

Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

\textbf{Parameters}

\textbf{x} : ndarray

The input domain.

\textbf{condlist} : list of bool arrays

Each boolean array corresponds to a function in \texttt{funclist}. Wherever \texttt{condlist[i]} is True, \texttt{funclist[i]}(\texttt{x}) is used as the output value.

Each boolean array in \texttt{condlist} selects a piece of \texttt{x}, and should therefore be of the same shape as \texttt{x}.

The length of \texttt{condlist} must correspond to that of \texttt{funclist}. If one extra function is given, i.e. if \texttt{len(funclist) - len(condlist) == 1}, then that extra function is the default value, used wherever all conditions are false.

\textbf{funclist} : list of callables, \texttt{f(x,*args,**kw)}, or scalars

Each function is evaluated over \texttt{x} wherever its corresponding condition is True. It should take an array as input and give an array or a scalar value as output. If, instead of a callable, a scalar is provided then a constant function (\texttt{lambda x: scalar}) is assumed.

\textbf{args} : tuple, optional

Any further arguments given to \textbf{piecewise} are passed to the functions upon execution, i.e., if called \textbf{piecewise}(..., ..., 1, 'a'), then each function is called as \texttt{f(x, 1, 'a')}.

\textbf{kw} : dict, optional
Keyword arguments used in calling `piecewise` are passed to the functions upon execution, i.e., if called `piecewise(..., ..., lambda=1)`, then each function is called as \( f(x, \ lambda=1) \).

**Returns**

`out : ndarray`

The output is the same shape and type as `x` and is found by calling the functions in `funclist` on the appropriate portions of `x`, as defined by the boolean arrays in `condlist`. Portions not covered by any condition have undefined values.

**See Also:**

`choose`, `select`, `where`

**Notes**

This is similar to choose or select, except that functions are evaluated on elements of `x` that satisfy the corresponding condition from `condlist`.

The result is:

\[
\begin{align*}
| & \quad \text{funclist}[0](x[\text{condlist}[0]]) \\
\text{out} & \quad | \text{funclist}[1](x[\text{condlist}[1]]) \\
& \quad | \text{...} \\
& \quad | \text{funclist}[n](x[\text{condlist}[n]]) \\
| & \quad \text{...}
\end{align*}
\]

**Examples**

Define the sigma function, which is -1 for \( x < 0 \) and +1 for \( x \geq 0 \).

```python
>>> x = np.linspace(-2.5, 2.5, 6)
>>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
array([-1., -1., -1., 1., 1., 1.])
```

Define the absolute value, which is \(-x\) for \( x < 0 \) and \( x \) for \( x \geq 0 \).

```python
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([ 2.5,  1.5,  0.5,  0.5,  1.5,  2.5])
```

## 3.14 Numpy-specific help functions

### 3.14.1 Finding help

```python
numpy.lookfor(what[, module, import_modules, ...])     Do a keyword search on docstrings.
```

`numpy.lookfor` *(what, module=None, import_modules=True, regenerate=False, output=None)*

Do a keyword search on docstrings.

A list of objects that matched the search is displayed, sorted by relevance. All given keywords need to be found in the docstring for it to be returned as a result, but the order does not matter.

**Parameters**

- `what` : str
String containing words to look for.

**module** : str or list, optional

Name of module(s) whose docstrings to go through.

**import_modules** : bool, optional

Whether to import sub-modules in packages. Default is True.

**regenerate** : bool, optional

Whether to re-generate the docstring cache. Default is False.

**output** : file-like, optional

File-like object to write the output to. If omitted, use a pager.

**See Also:**

`source`, `info`

**Notes**

Relevance is determined only roughly, by checking if the keywords occur in the function name, at the start of a docstring, etc.

**Examples**

```python
>>> np.lookfor('binary representation')
Search results for 'binary representation'
------------------------------------------
numpy.binary_repr
  Return the binary representation of the input number as a string.
numpy.core.setup_common.long_double_representation
  Given a binary dump as given by GNU od -b, look for long double
numpy.base_repr
  Return a string representation of a number in the given base system.
...```

### 3.14.2 Reading help

| `info(object, maxwidth, output, toplevel)` | Get help information for a function, class, or module. |
| `source(object[, output])` | Print or write to a file the source code for a Numpy object. |

```python
numpy.info (object=None, maxwidth=76, output=<open file '<stdout>', mode 'w' at 0x2ac351aa0150>,
toplevel='numpy')
```

Get help information for a function, class, or module.

**Parameters**

- **object** : object or str, optional

  Input object or name to get information about. If `object` is a numpy object, its docstring is given. If it is a string, available modules are searched for matching objects. If None, information about `info` itself is returned.

- **maxwidth** : int, optional

  Printing width.

- **output** : file like object, optional
File like object that the output is written to, default is stdout. The object has to be opened in 'w' or 'a' mode.

toplevel : str, optional
    Start search at this level.

See Also:

source, lookfor

Notes

When used interactively with an object, np.info(obj) is equivalent to help(obj) on the Python prompt or "obj? on the IPython prompt.

Examples

>>> np.info(np.polyval)
polyval(p, x)
    Evaluate the polynomial p at x.
...

When using a string for object it is possible to get multiple results.

>>> np.info('fft')
*** Found in numpy ***
Core FFT routines
...
*** Found in numpy.fft ***
fft(a, n=None, axis=-1)
...
*** Repeat reference found in numpy.fft.fftpack ***
*** Total of 3 references found. ***

numpy.source (object, output=<open file '<stdout>' mode 'w' at 0x2ac351aa0150>)
Print or write to a file the source code for a Numpy object.

The source code is only returned for objects written in Python. Many functions and classes are defined in C and will therefore not return useful information.

Parameters

object : numpy object
    Input object. This can be any object (function, class, module, ...).

output : file object, optional
    If output not supplied then source code is printed to screen (sys.stdout). File object must be created with either write 'w' or append 'a' modes.

See Also:

lookfor, info

Examples

>>> np.source(np.interp)
In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
def interp(x, xp, fp, left=None, right=None):
    """
    .... (full docstring printed)"
    if isinstance(x, (float, int, number)):
        return compiled_interp([x], xp, fp, left, right).item()
else:
    return compiled_interp(x, xp, fp, left, right)

The source code is only returned for objects written in Python.

>>> np.source(np.array)
Not available for this object.

3.15 Indexing routines

See Also:

Indexing

3.15.1 Generating index arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_</td>
<td>Translates slice objects to concatenation along the second axis.</td>
</tr>
<tr>
<td>r_</td>
<td>Translates slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td>s_</td>
<td>A nicer way to build up index tuples for arrays.</td>
</tr>
<tr>
<td>nonzero(a)</td>
<td>Return the indices of the elements that are non-zero.</td>
</tr>
<tr>
<td>where(condition, [x, y])</td>
<td>Return elements, either from x or y, depending on condition.</td>
</tr>
<tr>
<td>indices(dimensions[, dtype])</td>
<td>Return an array representing the indices of a grid.</td>
</tr>
<tr>
<td>ix_(*args)</td>
<td>Construct an open mesh from multiple sequences.</td>
</tr>
<tr>
<td>ogrid</td>
<td>nd_grid instance which returns an open multi-dimensional “meshgrid”.</td>
</tr>
<tr>
<td>ravel_multi_index(multi_index, dims[, mode, ...])</td>
<td>Converts a tuple of index arrays into an array of flat indices, applying boundary conditions.</td>
</tr>
<tr>
<td>unravel_index(indices, dims[, order])</td>
<td>Converts a flat index or array of flat indices into a tuple of coordinate arrays.</td>
</tr>
<tr>
<td>diag_indices(n[, ndim])</td>
<td>Return the indices to access the main diagonal of an array.</td>
</tr>
<tr>
<td>diag_indices_from(arr)</td>
<td>Return the indices to access the main diagonal of an n-dimensional array.</td>
</tr>
<tr>
<td>mask_indices(n, mask_func[, k])</td>
<td>Return the indices to access (n, n) arrays, given a masking function.</td>
</tr>
<tr>
<td>tril_indices(n[, k])</td>
<td>Return the indices for the lower-triangle of an (n, n) array.</td>
</tr>
<tr>
<td>tril_indices_from(arr[, k])</td>
<td>Return the indices for the lower-triangle of arr.</td>
</tr>
<tr>
<td>triu_indices(n[, k])</td>
<td>Return the indices for the upper-triangle of an (n, n) array.</td>
</tr>
<tr>
<td>triu_indices_from(arr[, k])</td>
<td>Return the indices for the upper-triangle of a (N, N) array.</td>
</tr>
</tbody>
</table>

numpy.c_ = <numpy.lib.index_tricks.CClass object at 0x25df410>

Translates slice objects to concatenation along the second axis.

This is short-hand for np.r_['-1,2,0', index expression], which is useful because of its common occurrence. In particular, arrays will be stacked along their last axis after being upgraded to at least 2-D with 1’s post-pended to the shape (column vectors made out of 1-D arrays).

For detailed documentation, see r_.

Examples

>>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
array([[1, 2, 3, 0, 0, 4, 5, 6]])

numpy.r_ = <numpy.lib.index_tricks.RClass object at 0x25df350>

Translates slice objects to concatenation along the first axis.

This is a simple way to build up arrays quickly. There are two use cases.

648 Chapter 3. Routines
1. If the index expression contains comma separated arrays, then stack them along their first axis.

2. If the index expression contains slice notation or scalars then create a 1-D array with a range indicated by the slice notation.

If slice notation is used, the syntax `start:stop:step` is equivalent to `np.arange(start, stop, step)` inside of the brackets. However, if `step` is an imaginary number (i.e., 100j) then its integer portion is interpreted as a number-of-points desired and the start and stop are inclusive. In other words `start:stop:stepj` is interpreted as `np.linspace(start, stop, step, endpoint=1)` inside of the brackets. After expansion of slice notation, all comma separated sequences are concatenated together.

Optional character strings placed as the first element of the index expression can be used to change the output. The strings ‘r’ or ‘c’ result in matrix output. If the result is 1-D and ‘r’ is specified a 1 x N (row) matrix is produced. If the result is 1-D and ‘c’ is specified, then a N x 1 (column) matrix is produced. If the result is 2-D then both provide the same matrix result.

A string integer specifies which axis to stack multiple comma separated arrays along. A string of two comma-separated integers allows indication of the minimum number of dimensions to force each entry into as the second integer (the axis to concatenate along is still the first integer).

A string with three comma-separated integers allows specification of the axis to concatenate along, the minimum number of dimensions to force the entries to, and which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the 1’s should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of ‘0’ would place the 1’s at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is ‘-1’.

Parameters

Not a function, so takes no parameters

Returns

A concatenated ndarray or matrix.

See Also:

concatenate

Join a sequence of arrays together.

c_

Translates slice objects to concatenation along the second axis.

Examples

```python
c>>> np.r_[np.array([[1, 2, 3]], 0, 0, np.array([4, 5, 6])]
array([[1, 2, 3, 0, 0, 4, 5, 6]])
c>>> np.r_[-1:1:6j, [0]*3, 5, 6]
array([-1., -0.6, -0.2, 0.2, 0.6, 1., 0., 0., 0., 5., 6.])
cString integers specify the axis to concatenate along or the minimum number of dimensions to force entries into.
```
Using ‘r’ or ‘c’ as a first string argument creates a matrix.

```python
>>> np.r_['r', [1, 2, 3], [4, 5, 6]]
matrix([[1, 2, 3, 4, 5, 6]])
```

`numpy.s_ = <numpy.lib.index_tricks.IndexExpression object at 0x25df550>`
A nicer way to build up index tuples for arrays.

**Note:** Use one of the two predefined instances `index_exp` or `s_` rather than directly using `IndexExpression`.

For any index combination, including slicing and axis insertion, `a[indices]` is the same as `a[np.index_exp[indices]]` for any array `a`. However, `np.index_exp[indices]` can be used anywhere in Python code and returns a tuple of slice objects that can be used in the construction of complex index expressions.

**Parameters**

- `maketuple` : bool
  - If True, always returns a tuple.

**See Also**

- `index_exp`
  - Predefined instance that always returns a tuple: `index_exp = IndexExpression(maketuple=True)`.

- `s_`
  - Predefined instance without tuple conversion: `s_ = IndexExpression(maketuple=False)`.

**Notes**

You can do all this with `slice()` plus a few special objects, but there’s a lot to remember and this version is simpler because it uses the standard array indexing syntax.

**Examples**

```python
>>> np.s_[2::2]
slice(2, None, 2)
>>> np.index_exp[2::2]
(slice(2, None, 2),)
```

```python
>>> np.array([0, 1, 2, 3, 4])[np.s_[2::2]]
array([2, 4])
```

`numpy.nonzero(a)`
- Return the indices of the elements that are non-zero.
Returns a tuple of arrays, one for each dimension of \( a \), containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

\[
a[\text{nonzero}(a)]
\]

To group the indices by element, rather than dimension, use:

\[
\text{transpose}(\text{nonzero}(a))
\]

The result of this is always a 2-D array, with a row for each non-zero element.

**Parameters**

- **a** : array_like
  - Input array.

**Returns**

- **tuple_of_arrays** : tuple
  - Indices of elements that are non-zero.

**See Also:**

- **flatnonzero**  
  Return indices that are non-zero in the flattened version of the input array.

- **ndarray.nonzero**  
  Equivalent ndarray method.

- **count_nonzero**  
  Counts the number of non-zero elements in the input array.

**Examples**

```python
>>> x = np.eye(3)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))

>>> x[np.nonzero(x)]
array([ 1.,  1.,  1.])

>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

A common use for **nonzero** is to find the indices of an array, where a condition is True. Given an array \( a \), the condition \( a > 3 \) is a boolean array and since False is interpreted as 0, np.nonzero(a > 3) yields the indices of the \( a \) where the condition is true.

```python
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
array([[False, False, False],
       [ True, True, True],
       [ True, True, True]], dtype=bool)

>>> np.nonzero(a > 3)
(array([1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```
The `nonzero` method of the boolean array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`numpy.where(condition[, x, y])`

Return elements, either from x or y, depending on `condition`.

If only `condition` is given, return `condition.nonzero()`.

**Parameters**

- `condition` : array_like, bool
  When True, yield x, otherwise yield y.

- `x, y` : array_like, optional
  Values from which to choose. x and y need to have the same shape as `condition`.

**Returns**

- `out` : ndarray or tuple of ndarrays
  If both x and y are specified, the output array contains elements of x where `condition` is True, and elements from y elsewhere.
  
  If only `condition` is given, return the tuple `condition.nonzero()`, the indices where `condition` is True.

**See Also:**

`nonzero`, `choose`

**Notes**

If `x` and `y` are given and input arrays are 1-D, `where` is equivalent to:

```python
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

**Examples**

```python
>>> np.where([[True, False], [True, True]],
            [[1, 2], [3, 4]],
            [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])

>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))

>>> x = np.arange(9.).reshape(3, 3)
>>> np.where( x > 5 )
(array([2, 2, 2]), array([0, 1, 2]))

>>> x[np.where( x > 3.0 )]   # Note: result is 1D.
array([4., 5., 6., 7., 8.])

>>> np.where(x < 5, x, -1)  # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```

Find the indices of elements of `x` that are in `goodvalues`.
```python
>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
       [ True,  True, False],
       [False,  True, False]], dtype=bool)
```

```python
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

numpy.indices(dimensions, dtype=<type 'int'>)

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

Parameters

dimensions : sequence of ints
The shape of the grid.

dtype : dtype, optional
Data type of the result.

Returns
grid : ndarray
The array of grid indices, grid.shape = (len(dimensions),) +
tuple(dimensions).

See Also:
mgrid, meshgrid

Notes

The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if
dimensions is a tuple (r0, ..., rN-1) of length N, the output shape is (N,r0,...,rN-1).

The subarrays grid[k] contains the N-D array of indices along the k-th axis. Explicitly:

grid[k,i0,i1,...,iN-1] = ik

Examples

```python
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]       # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]       # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```python
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```
Note that it would be more straightforward in the above example to extract the required elements directly with:

```python
x[:, 2, :3]
```

`numpy.ix_(*args)`

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using `ix_` one can quickly construct index arrays that will index the cross product. `a[np.ix_([1, 3], [2, 5])]` returns the array `[[a[1, 2] a[1, 5]], [a[3, 2] a[3, 5]]]`.

**Parameters**
- `args` : 1-D sequences

**Returns**
- `out` : tuple of ndarrays
  - N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

**See Also:**
- `ogrid`, `mgrid`, `meshgrid`

**Examples**

```python
>>> a = np.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> ixgrid = np.ix_([0, 1], [2, 4])
>>> ixgrid
(array([[0],
        [1]]), array([[2, 4]]))
>>> ixgrid[0].shape, ixgrid[1].shape
((2, 1), (1, 2))
>>> a[ixgrid]
array([[2, 4],
       [7, 9]])
```

`numpy.ogrid = <numpy.lib.index_tricks.nd_grid object at 0x25df2d0>`

`nd_grid` instance which returns an open multi-dimensional “meshgrid”.

An instance of `numpy.lib.index_tricks.nd_grid` which returns an open (i.e. not fleshed out) meshgrid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

**Returns**
- mesh-grid `ndarrays` with only one dimension ≠ 1

**See Also:**
- `np.lib.index_tricks.nd_grid`
  - class of `ogrid` and `mgrid` objects
- `mgrid`
  - like `ogrid` but returns dense (or fleshed out) mesh grids
array concatenator

Examples

```python
>>> from numpy import ogrid
>>> ogrid[-1:1:5j]
array([-1. , -0.5, 0. , 0.5, 1. ])
>>> ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

numpy.ravel_multi_index

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

Parameters

- `multi_index`: tuple of array_like
  A tuple of integer arrays, one array for each dimension.
- `dims`: tuple of ints
  The shape of array into which the indices from `multi_index` apply.
- `mode`: {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices are handled. Can specify either one mode or a tuple of modes, one mode per index.
  - 'raise' – raise an error (default)
  - 'wrap' – wrap around
  - 'clip' – clip to the range
  In 'clip' mode, a negative index which would normally wrap will clip to 0 instead.
- `order`: {'C', 'F'}, optional
  Determines whether the multi-index should be viewed as indexing in C (row-major) order or FORTRAN (column-major) order.

Returns

- `raveled_indices`: ndarray
  An array of indices into the flattened version of an array of dimensions `dims`.

See Also:

unravel_index

Notes

New in version 1.6.0.

Examples

```python
>>> arr = np.array([[3,6,6],[4,5,1]])
>>> np.ravel_multi_index(arr, (7,6))
array([22, 41, 37])
>>> np.ravel_multi_index(arr, (7,6), order='F')
```

3.15. Indexing routines
>>> np.ravel_multi_index(arr, (4, 6), mode='clip')
array([22, 23, 19])

>>> np.ravel_multi_index(arr, (4,4), mode=('clip','wrap'))
array([12, 13, 13])

numpy.unravel_index(indices, dims, order='C')

Converts a flat index or array of flat indices into a tuple of coordinate arrays.

Parameters:
- `indices`: array_like
  An integer array whose elements are indices into the flattened version of an array of dimensions `dims`. Before version 1.6.0, this function accepted just one index value.
- `dims`: tuple of ints
  The shape of the array to use for unraveling `indices`.
- `order`: {'C', 'F'}, optional
  New in version 1.6.0. Determines whether the indices should be viewed as indexing in C (row-major) order or FORTRAN (column-major) order.

Returns:
- `unraveled_coords`: tuple of ndarray
  Each array in the tuple has the same shape as the `indices` array.

See Also:
- `ravel_multi_index`

Examples:

```python
>>> np.unravel_index([22, 41, 37], (7,6))
(array([3, 6, 6]), array([4, 5, 1]))

>>> np.unravel_index([31, 41, 13], (4,4), order='F')
(array([3, 6, 6]), array([4, 5, 1]))

>>> np.unravel_index((3,1,4,1), (6,7,8,9))
1621
```

numpy.diag_indices(n, ndim=2)

Return the indices to access the main diagonal of an array.

This returns a tuple of indices that can be used to access the main diagonal of an array `a` with `a.ndim` >= 2 dimensions and shape `(n, n, ..., n)`. For `a.ndim = 2` this is the usual diagonal, for `a.ndim > 2` this is the set of indices to access `a[i, i, ..., i]` for `i = [0..n-1]`.

Parameters:
- `n`: int
  The size, along each dimension, of the arrays for which the returned indices can be used.
- `ndim`: int, optional
  The number of dimensions.
See Also:

diag_indices_from

Notes
New in version 1.4.0.

Examples
Create a set of indices to access the diagonal of a (4, 4) array:

```python
>>> di = np.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
```

```python
>>> a = np.arange(16).reshape(4, 4)
```

```python
>>> a
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
       [12, 13, 14, 15]])
```

```python
>>> a[di] = 100
>>> a
array([[100, 1, 2, 3],
       [ 4, 100, 6, 7],
       [ 8, 9, 100, 11],
       [12, 13, 14, 100]])
```

Now, we create indices to manipulate a 3-D array:

```python
>>> d3 = np.diag_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))
```

And use it to set the diagonal of an array of zeros to 1:

```python
>>> a = np.zeros((2, 2, 2), dtype=np.int)
>>> a[d3] = 1
>>> a
array([[[ 1,  0],
        [ 0,  0]],
       [[ 0,  0],
        [ 0,  1]]])
```

```
 numpy.diag_indices_from(arr)
Return the indices to access the main diagonal of an n-dimensional array.

See diag_indices for full details.

Parameters
arr : array, at least 2-D

See Also:

diag_indices

Notes
New in version 1.4.0.
```

```
 numpy.mask_indices(n, mask_func, k=0)
Return the indices to access (n, n) arrays, given a masking function.
```

3.15. Indexing routines
Assume \texttt{mask\_func} is a function that, for a square array \texttt{a} of size \((n, n)\) with a possible offset argument \(k\), when called as \texttt{mask\_func(a, k)} returns a new array with zeros in certain locations (functions like \texttt{triu} or \texttt{tril} do precisely this). Then this function returns the indices where the non-zero values would be located.

**Parameters**

- \texttt{n : int}
  
The returned indices will be valid to access arrays of shape \((n, n)\).

- \texttt{mask\_func : callable}
  
  A function whose call signature is similar to that of \texttt{triu, tril}. That is, \texttt{mask\_func(x, k)} returns a boolean array, shaped like \texttt{x}. \(k\) is an optional argument to the function.

- \texttt{k : scalar}
  
  An optional argument which is passed through to \texttt{mask\_func}. Functions like \texttt{triu, tril} take a second argument that is interpreted as an offset.

**Returns**

- \texttt{indices : tuple of arrays.}
  
  The \(n\) arrays of indices corresponding to the locations where \(\texttt{mask\_func(np.ones((n, n)), k)}\) is True.

**See Also:**

\texttt{triu, tril, triu\_indices, tril\_indices}

**Notes**

New in version 1.4.0.

**Examples**

These are the indices that would allow you to access the upper triangular part of any 3x3 array:

```python
>>> iu = np.mask_indices(3, np.triu)
```

For example, if \(a\) is a 3x3 array:

```python
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a[iu]
array([0, 1, 2, 4, 5, 8])
```

An offset can be passed also to the masking function. This gets us the indices starting on the first diagonal right of the main one:

```python
>>> iul = np.mask_indices(3, np.triu, 1)
```

with which we now extract only three elements:

```python
>>> a[iul]
array([1, 2, 5])
```

\texttt{numpy.tril\_indices(n, k=0)}

Return the indices for the lower-triangle of an \((n, n)\) array.
Parameters

n : int

The row dimension of the square arrays for which the returned indices will be valid.

k : int, optional

Diagonal offset (see tril for details).

Returns

inds : tuple of arrays

The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

See Also:

triu_indices
similar function, for upper-triangular.

mask_indices
generic function accepting an arbitrary mask function.

tril, triu

Notes

New in version 1.4.0.

Examples

Compute two different sets of indices to access 4x4 arrays, one for the lower triangular part starting at the main diagonal, and one starting two diagonals further right:

```python
>>> i11 = np.tril_indices(4)
>>> i12 = np.tril_indices(4, 2)
```

Here is how they can be used with a sample array:

```python
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
```

Both for indexing:

```python
>>> a[i11]
array([ 0,  4,  5,  8,  9, 10, 12, 13, 14, 15])
```

And for assigning values:

```python
>>> a[i11] = -1
>>> a
array([[-1,  1,  2,  3],
        [-1, -1,  6,  7],
        [-1, -1, -1, 11],
        [-1, -1, -1, -1]])
```

These cover almost the whole array (two diagonals right of the main one):
```python
>>> a[112] = -10
>>> a
array([[-10, -10, -10,  3],
       [-10, -10, -10, -10],
       [-10, -10, -10, -10],
       [-10, -10, -10, -10]])
```

```python
numpy.tril_indices_from(arr, k=0)
```

Return the indices for the lower-triangle of `arr`.

See `tril_indices` for full details.

Parameters

- `arr` : array_like
  The indices will be valid for square arrays whose dimensions are the same as `arr`.

- `k` : int, optional
  Diagonal offset (see `tril` for details).

See Also:

- `tril_indices`, `tril`

Notes

New in version 1.4.0.

```python
numpy.triu_indices(n, k=0)
```

Return the indices for the upper-triangle of an `(n, n)` array.

Parameters

- `n` : int
  The size of the arrays for which the returned indices will be valid.

- `k` : int, optional
  Diagonal offset (see `triu` for details).

Returns

- `inds` : tuple, shape(2) of ndarrays, shape(`n`)
  The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array. Can be used to slice a ndarray of shape(`n, n`).

See Also:

- `tril_indices`
  similar function, for lower-triangular.

- `mask_indices`
  generic function accepting an arbitrary mask function.

- `triu, tril`

Notes

New in version 1.4.0.
Examples

Compute two different sets of indices to access 4x4 arrays, one for the upper triangular part starting at the main diagonal, and one starting two diagonals further right:

```python
>>> iu1 = np.triu_indices(4)
>>> iu2 = np.triu_indices(4, 2)
```

Here is how they can be used with a sample array:

```python
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Both for indexing:

```python
>>> a[iu1]
array([ 0,  1,  2,  3,  5,  6,  7, 10, 11, 15])
```

And for assigning values:

```python
>>> a[iu1] = -1
>>> a
array([[ -1, -1, -10, -10],
       [ 4, -1, -1, -10],
       [ 8,  9, -1, -1],
       [12, 13, 14, -1]])
```

These cover only a small part of the whole array (two diagonals right of the main one):

```python
>>> a[iu2] = -10
>>> a
array([[ -1, -1, -10, -10],
       [ 4, -1, -1,  -10],
       [ 8,  9, -1,  -1],
       [12, 13, 14,  -1]])
```

numpy.triu_indices_from(arr, k=0)
Return the indices for the upper-triangle of a (N, N) array.

See `triu_indices` for full details.

Parameters

- **arr**: ndarray, shape(N, N)
  The indices will be valid for square arrays.
- **k**: int, optional
  Diagonal offset (see `triu` for details).

Returns

- **triu_indices_from**: tuple, shape(2) of ndarray, shape(N)
  Indices for the upper-triangle of `arr`.

See Also:

- `triu_indices`
Notes

New in version 1.4.0.

3.15.2 Indexing-like operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>take(a, indices[, axis, out, mode])</code></td>
<td>Take elements from an array along an axis.</td>
</tr>
<tr>
<td><code>choose(a, choices[, out, mode])</code></td>
<td>Construct an array from an index array and a set of arrays to choose from.</td>
</tr>
<tr>
<td><code>compress(condition, a[, axis, out])</code></td>
<td>Return selected slices of an array along given axis.</td>
</tr>
<tr>
<td><code>diag(v[, k])</code></td>
<td>Extract a diagonal or construct a diagonal array.</td>
</tr>
<tr>
<td><code>diagonal(a[, offset, axis1, axis2])</code></td>
<td>Return specified diagonals.</td>
</tr>
<tr>
<td><code>select(condlist, choicelist[, default])</code></td>
<td>Return an array drawn from elements in choicelist, depending on conditions.</td>
</tr>
</tbody>
</table>

```python
numpy.take(a, indices, axis=None, out=None, mode='raise')
```

Take elements from an array along an axis.

This function does the same thing as “fancy” indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

Parameters

- `a`: array_like
  The source array.
- `indices`: array_like
  The indices of the values to extract. New in version 1.8.0. Also allow scalars for indices.
- `axis`: int, optional
  The axis over which to select values. By default, the flattened input array is used.
- `out`: ndarray, optional
  If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.
- `mode`: {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices will behave.
  - 'raise' – raise an error (default)
  - 'wrap' – wrap around
  - 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns

- `subarray`: ndarray
  The returned array has the same type as `a`.

See Also:

- `ndarray.take`
  equivalent method
Examples

```python
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if `a` is an ndarray, “fancy” indexing can be used.

```python
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

If `indices` is not one dimensional, the output also has these dimensions.

```python
>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
       [5, 7]])
```

```
numpy.choose(a, choices, out=None, mode='raise')
```

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = numpy.lib.index_tricks):

```
np.choose(a,c) == np.array([c[a[I]] for I in ndi.ndindex(a.shape)])
```

But this omits some subtleties. Here is a fully general summary:

Given an “index” array `a` of integers and a sequence of `n` arrays (`choices`), `a` and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these `Ba` and `Bchoices[i]`, `i = 0,...,n-1` we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each `i`. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode=raise` (the default), then, first of all, each element of `a` (and thus `Ba`) must be in the range `[0, n-1]`; now, suppose that `i` (in that range) is the value at the `(j0, j1, ..., jm)` position in `Ba` - then the value at the same position in the new array is the value in `Bchoices[i]` at that same position;
- if `mode=wrap`, values in `a` (and thus `Ba`) may be any (signed) integer; modular arithmetic is used to map integers outside the range `[0, n-1]` back into that range; and then the new array is constructed as above;
- if `mode=clip`, values in `a` (and thus `Ba`) may be any (signed) integer; negative integers are mapped to 0; values greater than `n-1` are mapped to `n-1`; and then the new array is constructed as above.

```
Parameters
```

- `a` : int array
  This array must contain integers in `[0, n-1]`, where `n` is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.

- `choices` : sequence of arrays
  Choice arrays. `a` and all of the choices must be broadcastable to the same shape. If `choices` is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the “sequence”.

- `out` : array, optional
  If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

- `mode` : {'raise' (default), 'wrap', 'clip'}, optional

3.15. Indexing routines 663
Specifies how indices outside \([0, n-1]\) will be treated:

- 'raise': an exception is raised
- 'wrap': value becomes value mod \(n\)
- 'clip': values < 0 are mapped to 0, values > n-1 are mapped to n-1

**Returns**

merged_array : array

The merged result.

**Raises**

ValueError: shape mismatch

If \(a\) and each choice array are not all broadcastable to the same shape.

**See Also:**

```
ndarray.choose
equivalent method
```

**Notes**

To reduce the chance of misinterpretation, even though the following “abuse” is nominally supported, choices should neither be, nor be thought of as, a single array, i.e., the outermost sequence-like container should be either a list or a tuple.

**Examples**

```python
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
... [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices)
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... array([20, 31, 12, 3])
```

```python
>>> np.choose([2, 4, 1, 0], choices, mode='clip')  # 4 goes to 3 (4-1)
array([20, 31, 12, 3])
```

```python
>>> np.choose([2, 4, 1, 0], choices, mode='wrap')  # 4 goes to (4 mod 4)
array([20, 1, 12, 3])
```

```python
# because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap')  # 4 goes to (4 mod 4)
array([20, 1, 12, 3])
>>> # i.e., 0
```

A couple examples illustrating how choose broadcasts:

```python
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
```

```python
>>> choices = [-10, 10]
```

```python
>>> np.choose(a, choices)
array([[-10, 10, -10],
[-10, 10, -10],
[ 10, -10, 10]])
```

```python
>>> # With thanks to Anne Archibald
>>> a = np.array([0, 1]).reshape((2, 1))
```

```python
>>> c1 = np.array([1, 2, 3]).reshape((1, 3, 1))
```

```python
>>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1, 5))
```

```python
>>> np.choose(a, (c1, c2))  # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
```
numpy.compress(condition, a, axis=None, out=None)

Return selected slices of an array along given axis.

When working along a given axis, a slice along that axis is returned in output for each index where condition evaluates to True. When working on a 1-D array, compress is equivalent to extract.

Parameters

- condition : 1-D array of bools
  Array that selects which entries to return. If len(condition) is less than the size of a along the given axis, then output is truncated to the length of the condition array.

- a : array_like
  Array from which to extract a part.

- axis : int, optional
  Axis along which to take slices. If None (default), work on the flattened array.

- out : ndarray, optional
  Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns

- compressed_array : ndarray
  A copy of a without the slices along axis for which condition is false.

See Also:

take, choose, diag, diagonal, select

ndarray.compress
Equivalent method in ndarray

np.extract
Equivalent method when working on 1-D arrays

numpy.doc.ufuncs
Section “Output arguments”

Examples

```python
>>> a = np.array([[[1, 2], [3, 4], [5, 6]]])
>>> a
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]]])
>>> np.compress([[0, 1], [1, 2], [0, 2]], a, axis=0)
array([[3, 4],
        [5, 6]])
>>> np.compress([[False, True, True], [True, False, False]], a, axis=0)
array([[3, 4],
        [5, 6]])
>>> np.compress([[False, True], [False, True]], a, axis=1)
array([[2],
        [5]])
```
Working on the flattened array does not return slices along an axis but selects elements.

```python
>>> np.compress([False, True], a)
array([2])
```

**numpy.diag(v, k=0)**

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for `numpy.diagonal` if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

**Parameters**

- `v`: array_like
  If `v` is a 2-D array, return a copy of its `k`-th diagonal. If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th diagonal.
- `k`: int, optional
  Diagonal in question. The default is 0. Use `k>0` for diagonals above the main diagonal, and `k<0` for diagonals below the main diagonal.

**Returns**

- `out`: ndarray
  The extracted diagonal or constructed diagonal array.

**See Also:**

- `diagonal`: Return specified diagonals.
- `diagflat`: Create a 2-D array with the flattened input as a diagonal.
- `trace`: Sum along diagonals.
- `triu`: Upper triangle of an array.
- `tril`: Lower triangle of an array.

**Examples**

```python
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])

>>> np.diag(x, k=1)
array([1, 5])

>>> np.diag(x, k=-1)
array([3, 7])
```
```python
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

`numpy.diag(a, offset=0, axis1=0, axis2=1)`

Return specified diagonals.

If `a` is 2-D, returns the diagonal of `a` with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If `a` has more than two dimensions, then the axes specified by `axis1` and `axis2` are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing `axis1` and `axis2` and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning will be issued.

In NumPy 1.9, it will switch to returning a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In NumPy 1.10, it will still return a view, but this view will no longer be marked read-only. Writing to the returned array will alter your original array as well.

If you don’t write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diag(a).copy()` instead of just `np.diag(a)`. This will work with both past and future versions of NumPy.

**Parameters**

- `a` : array_like
  Array from which the diagonals are taken.

- `offset` : int, optional
  Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).

- `axis1` : int, optional
  Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).

- `axis2` : int, optional
  Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

**Returns**

- `array_of_diagonals` : ndarray
  If `a` is 2-D, a 1-D array containing the diagonal is returned. If the dimension of `a` is larger, then an array of diagonals is returned, “packed” from left-most dimension to right-most (e.g., if `a` is 3-D, then the diagonals are “packed” along rows).

**Raises**

- `ValueError`
  If the dimension of `a` is less than 2.

**See Also:**

3.15. Indexing routines
**diag**
MATLAB work-a-like for 1-D and 2-D arrays.

**diagflat**
Create diagonal arrays.

**trace**
Sum along diagonals.

**Examples**

```python
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
        [2, 3]])

>>> a.diagonal()
array([0, 3])

>>> a.diagonal(1)
array([1])
```

A 3-D example:

```python
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
        [[4, 5],
        [6, 7]]])

>>> a.diagonal(0, # Main diagonals of two arrays created by skipping ...
... 0, # across the outer(left)-most axis last and ...
... 1) # the "middle" (row) axis first.
array([[0, 6],
        [1, 7]])
```

The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are “packed” in rows.

```python
>>> a[:,:,0] # main diagonal is [0 6]
array([[0, 2],
        [4, 6]])

>>> a[:,:,1] # main diagonal is [1 7]
array([[1, 3],
        [5, 7]])
```

**numpy.select** *(condlist, choicelist, default=0)*

Return an array drawn from elements in choicelist, depending on conditions.

**Parameters**

- **condlist** : list of bool ndarrays
  
The list of conditions which determine from which array in choicelist the output elements are taken. When multiple conditions are satisfied, the first one encountered in condlist is used.

- **choicelist** : list of ndarrays
  
The list of arrays from which the output elements are taken. It has to be of the same length as condlist.

- **default** : scalar, optional
  
The element inserted in output when all conditions evaluate to False.
Returns

output : ndarray

The output at position m is the m-th element of the array in choicelist where the m-th element of the corresponding array in condlist is True.

See Also:

where

Return elements from one of two arrays depending on condition.

take, choose, compress, diag, diagonal

Examples

>>> x = np.arange(10)
>>> condlist = [x<3, x>5]
>>> choicelist = [x, x**2]
>>> np.select(condlist, choicelist)
array([ 0, 1, 2, 0, 0, 0, 36, 49, 64, 81])

3.15.3 Inserting data into arrays

<table>
<thead>
<tr>
<th>numpy.place(arr, mask, vals)</th>
<th>Change elements of an array based on conditional and input values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(a, ind, v[, mode])</td>
<td>Replaces specified elements of an array with given values.</td>
</tr>
<tr>
<td>putmask(a, mask, values)</td>
<td>Changes elements of an array based on conditional and input values.</td>
</tr>
<tr>
<td>fill_diagonal(a, val[, wrap])</td>
<td>Fill the main diagonal of the given array of any dimensionality.</td>
</tr>
</tbody>
</table>

numpy.place(arr, mask, vals)

Change elements of an array based on conditional and input values.

Similar to np.copyto(arr, vals, where=mask), the difference is that place uses the first N elements of vals, where N is the number of True values in mask, while copyto uses the elements where mask is True.

Note that extract does the exact opposite of place.

Parameters

arr : array_like
    Array to put data into.

mask : array_like
    Boolean mask array. Must have the same size as a.

vals : 1-D sequence
    Values to put into a. Only the first N elements are used, where N is the number of True values in mask. If vals is smaller than N it will be repeated.

See Also:

copyto, put, take, extract
Examples

```python
>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[ 0,  1,  2],
       [44, 55, 44]])
```

numpy.put(a, ind, v, mode='raise')

Replaces specified elements of an array with given values.

The indexing works on the flattened target array. put is roughly equivalent to:

```
a.flat[ind] = v
```

Parameters

- **a**: ndarray
  Target array.
- **ind**: array_like
  Target indices, interpreted as integers.
- **v**: array_like
  Values to place in `a` at target indices. If `v` is shorter than `ind` it will be repeated as necessary.
- **mode**: {'raise', 'wrap', 'clip'}, optional
  Specifies how out-of-bounds indices will behave.
  - 'raise' – raise an error (default)
  - 'wrap' – wrap around
  - 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

See Also:

putmask, place

Examples

```python
>>> a = np.arange(5)
>>> np.put(a, [0, 2], [-44, -55])
>>> a
array([-44,  1, -55,  3,  4])
```

```python
>>> a = np.arange(5)
>>> np.put(a, 22, -5, mode='clip')
>>> a
array([ 0,  1,  2, -5,  4])
```

numpy.putmask(a, mask, values)

Changes elements of an array based on conditional and input values.

Sets `a.flat[n] = values[n]` for each `n` where `mask.flat[n]` is True.
If `values` is not the same size as `a` and `mask` then it will repeat. This gives behavior different from `a[mask] = values`.

**Note:** The `putmask` functionality is also provided by `copyto`, which can be significantly faster and in addition is NA-aware (`preservena` keyword). Replacing `putmask` with `np.copyto(a, values, where=mask)` is recommended.

### Parameters
- **a**: array_like
  - Target array.
- **mask**: array_like
  - Boolean mask array. It has to be the same shape as `a`.
- **values**: array_like
  - Values to put into `a` where `mask` is True. If `values` is smaller than `a` it will be repeated.

### See Also:
- `place`, `put`, `take`, `copyto`

### Examples
```python
>>> x = np.arange(6).reshape(2, 3)
>>> np.putmask(x, x>2, x**2)
>>> x
array([[ 0,  1,  2],
       [ 9, 16, 25]])
```

If `values` is smaller than `a` it is repeated:
```python
>>> x = np.arange(5)
>>> np.putmask(x, x>1, [-33, -44])
>>> x
array([ 0,  1, -33, -44, -33])
```

### numpy.fill_diagonal
Fill the main diagonal of the given array of any dimensionality.

For an array `a` with `a.ndim > 2`, the diagonal is the list of locations with indices `a[i, i, ..., i]` all identical. This function modifies the input array in-place, it does not return a value.

### Parameters
- **a**: array, at least 2-D.
  - Array whose diagonal is to be filled, it gets modified in-place.
- **val**: scalar
  - Value to be written on the diagonal, its type must be compatible with that of the array `a`.
- **wrap**: bool
  - For tall matrices in NumPy version up to 1.6.2, the diagonal “wrapped” after N columns. You can have this behavior with this option. This affect only tall matrices.

### See Also:
- `diag_indices`, `diag_indices_from`
Notes

New in version 1.4.0. This functionality can be obtained via `diag_indices`, but internally this version uses a much faster implementation that never constructs the indices and uses simple slicing.

Examples

```python
>>> a = np.zeros((3, 3), int)
>>> np.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

The same function can operate on a 4-D array:

```python
>>> a = np.zeros((3, 3, 3, 3), int)
>>> np.fill_diagonal(a, 4)
```

We only show a few blocks for clarity:

```python
>>> a[0, 0]
array([[4, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> a[1, 1]
array([[0, 0, 0],
       [4, 0, 0],
       [0, 0, 0]])
>>> a[2, 2]
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 4]])
```

# tall matrices no wrap >>> a = np.zeros((5, 3), int) >>> fill_diagonal(a, 4) array([[4, 0, 0],
[0, 4, 0], [0, 0, 4], [0, 0, 0], [0, 0, 0]])

# tall matrices wrap >>> a = np.zeros((5, 3), int) >>> fill_diagonal(a, 4) array([[4, 0, 0],
[0, 4, 0], [0, 0, 4], [0, 0, 0], [4, 0, 0]])

# wide matrices >>> a = np.zeros((3, 5), int) >>> fill_diagonal(a, 4) array([[4, 0, 0, 0, 0],
[0, 4, 0, 0, 0], [0, 0, 4, 0, 0]])

3.15.4 Iterating over arrays

<table>
<thead>
<tr>
<th>Class</th>
<th><code>nditer</code></th>
<th>Efficient multi-dimensional iterator object to iterate over arrays.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>ndenumerate(arr)</code></td>
<td>Multidimensional index iterator.</td>
</tr>
<tr>
<td>Class</td>
<td><code>ndindex(*shape)</code></td>
<td>An N-dimensional iterator object to index arrays.</td>
</tr>
<tr>
<td>Class</td>
<td><code>flatiter</code></td>
<td>Flat iterator object to iterate over arrays.</td>
</tr>
</tbody>
</table>

class `numpy.nditer`

Efficient multi-dimensional iterator object to iterate over arrays. To get started using this object, see the introductory guide to array iteration.

Parameters
**op**: ndarray or sequence of array_like

The array(s) to iterate over.

**flags**: sequence of str, optional

Flags to control the behavior of the iterator.

- “buffered” enables buffering when required.
- “c_index” causes a C-order index to be tracked.
- “f_index” causes a Fortran-order index to be tracked.
- “multi_index” causes a multi-index, or a tuple of indices with one per iteration dimension, to be tracked.
- “common_dtype” causes all the operands to be converted to a common data type, with copying or buffering as necessary.
- “delay_bufalloc” delays allocation of the buffers until a reset() call is made. Allows “allocate” operands to be initialized before their values are copied into the buffers.
- “external_loop” causes the values given to be one-dimensional arrays with multiple values instead of zero-dimensional arrays.
- “grow_inner” allows the value array sizes to be made larger than the buffer size when both “buffered” and “external_loop” is used.
- “ranged” allows the iterator to be restricted to a sub-range of the iterindex values.
- “refs_ok” enables iteration of reference types, such as object arrays.
- “reduce_ok” enables iteration of “readwrite” operands which are broadcasted, also known as reduction operands.
- “zerosize_ok” allows itersize to be zero.

**op_flags**: list of list of str, optional

This is a list of flags for each operand. At minimum, one of “readonly”, “readwrite”, or “writeonly” must be specified.

- “readonly” indicates the operand will only be read from.
- “readwrite” indicates the operand will be read from and written to.
- “writeonly” indicates the operand will only be written to.
- “no_broadcast” prevents the operand from being broadcasted.
- “contig” forces the operand data to be contiguous.
- “aligned” forces the operand data to be aligned.
- “nbo” forces the operand data to be in native byte order.
- “copy” allows a temporary read-only copy if required.
- “updateifcopy” allows a temporary read-write copy if required.
- “allocate” causes the array to be allocated if it is None in the op parameter.
- “no_subtype” prevents an “allocate” operand from using a subtype.
- “arraymask” indicates that this operand is the mask to use for selecting elements when writing to operands with the ‘writemasked’ flag set. The iterator does not enforce
this, but when writing from a buffer back to the array, it only copies those elements indicated by this mask.

- ‘writemasked’ indicates that only elements where the chosen ‘arraymask’ operand is True will be written to.

**op_dtypes** : dtype or tuple of dtype(s), optional

The required data type(s) of the operands. If copying or buffering is enabled, the data will be converted to/from their original types.

**order** : {'C', 'F', 'A', 'K'}, optional

Controls the iteration order. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. This also affects the element memory order of “allocate” operands, as they are allocated to be compatible with iteration order. Default is ‘K’.

**casting** : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional

Controls what kind of data casting may occur when making a copy or buffering. Setting this to ‘unsafe’ is not recommended, as it can adversely affect accumulations.

- ‘no’ means the data types should not be cast at all.
- ‘equiv’ means only byte-order changes are allowed.
- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

**op_axes** : list of list of ints, optional

If provided, is a list of ints or None for each operands. The list of axes for an operand is a mapping from the dimensions of the iterator to the dimensions of the operand. A value of -1 can be placed for entries, causing that dimension to be treated as “newaxis”.

**itershape** : tuple of ints, optional

The desired shape of the iterator. This allows “allocate” operands with a dimension mapped by op_axes not corresponding to a dimension of a different operand to get a value not equal to 1 for that dimension.

**buffersize** : int, optional

When buffering is enabled, controls the size of the temporary buffers. Set to 0 for the default value.

**Notes**

`nditer` supersedes `flatiter`. The iterator implementation behind `nditer` is also exposed by the Numpy C API.

The Python exposure supplies two iteration interfaces, one which follows the Python iterator protocol, and another which mirrors the C-style do-while pattern. The native Python approach is better in most cases, but if you need the iterator’s coordinates or index, use the C-style pattern.
Examples

Here is how we might write an `iter_add` function, using the Python iterator protocol:

```python
def iter_add_py(x, y, out=None):
    addop = np.add
    it = np.nditer([x, y, out], [],
        [['readonly'], ['readonly'], ['writeonly', 'allocate']])
    for (a, b, c) in it:
        addop(a, b, out=c)
    return it.operands[2]
```

Here is the same function, but following the C-style pattern:

```python
def iter_add(x, y, out=None):
    addop = np.add
    it = np.nditer([x, y, out], [],
        [['readonly'], ['readonly'], ['writeonly', 'allocate']])

    while not it.finished:
        addop(it[0], it[1], out=it[2])
        it.iternext()
    return it.operands[2]
```

Here is an example outer product function:

```python
def outer_it(x, y, out=None):
    mulop = np.multiply

    it = np.nditer([x, y, out], ['external_loop'],
        [['readonly'], ['readonly'], ['writeonly', 'allocate']],
        op_axes=[range(x.ndim)+[-1]*y.ndim,
            [-1]*x.ndim+range(y.ndim), None])
    for (a, b, c) in it:
        mulop(a, b, out=c)
    return it.operands[2]
```

```python
>>> a = np.arange(2)+1
>>> b = np.arange(3)+1
>>> outer_it(a,b)
array([[1, 2, 3],
       [2, 4, 6]])
```

Here is an example function which operates like a “lambda” ufunc:

```python
def luf(lamdaexpr, *args, **kwargs):
    "luf(lambdaexpr, op1, ..., opn, out=None, order='K', casting='safe', buffersize=0)"
    nargs = len(args)
    op = (kwargs.get('out', None),) + args
    it = np.nditer(op, ['buffered', 'external_loop'],
        [['writeonly', 'allocate', 'no_broadcast']] +
        [['readonly', 'nbo', 'aligned']]*nargs,
        order=kwargs.get('order', 'K'),
        casting=kwargs.get('casting', 'safe'),
        buffersize=kwargs.get('buffersize', 0))
```

3.15. Indexing routines
while not it.finished:
    it[0] = lambdaexpr(*it[1:]),
    it.iternext()
return it.operands[0]

>>> a = np.arange(5)
>>> b = np.ones(5)
>>> luf(lambda i,j:i*i + j/2, a, b)
array([ 0.5, 1.5, 4.5, 9.5, 16.5])

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtypes</td>
<td>(tuple of dtype(s)) The data types of the values provided in value. This may be different from the operand data types if buffering is enabled.</td>
</tr>
<tr>
<td>finished</td>
<td>(bool) Whether the iteration over the operands is finished or not.</td>
</tr>
<tr>
<td>has_delayed_bufalloc</td>
<td>If True, the iterator was created with the “delay_bufalloc” flag, and no reset() function was called on it yet.</td>
</tr>
<tr>
<td>has_index</td>
<td>(bool) If True, the iterator was created with either the “c_index” or the “f_index” flag, and the property index can be used to retrieve it.</td>
</tr>
<tr>
<td>has_multi_index</td>
<td>(bool) If True, the iterator was created with the “multi_index” flag, and the property multi_index can be used to retrieve it.</td>
</tr>
<tr>
<td>index</td>
<td>When the “c_index” or “f_index” flag was used, this property provides access to the index. Raises a ValueError if accessed and has_index is False.</td>
</tr>
<tr>
<td>iterationneedsapi</td>
<td>(bool) Whether iteration requires access to the Python API, for example if one of the operands is an object array.</td>
</tr>
<tr>
<td>iterindex</td>
<td>(int) An index which matches the order of iteration.</td>
</tr>
<tr>
<td>itsize</td>
<td>(int) Size of the iterator.</td>
</tr>
<tr>
<td>itviews</td>
<td>Structured view(s) of operands in memory, matching the reordered and optimized iterator access pattern.</td>
</tr>
<tr>
<td>multi_index</td>
<td>When the “multi_index” flag was used, this property provides access to the index. Raises a ValueError if accessed and has_multi_index is False.</td>
</tr>
<tr>
<td>ndim</td>
<td>(int) The iterator's dimension.</td>
</tr>
<tr>
<td>nop</td>
<td>(int) The number of iterator operands.</td>
</tr>
<tr>
<td>operands</td>
<td>(tuple of operand(s)) The array(s) to be iterated over.</td>
</tr>
<tr>
<td>shape</td>
<td>(tuple of ints) Shape tuple, the shape of the iterator.</td>
</tr>
<tr>
<td>value</td>
<td>Value of operands at current iteration. Normally, this is a tuple of array scalars, but if the flag “external_loop” is used, it is a tuple of one dimensional arrays.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy()</td>
<td>Get a copy of the iterator in its current state.</td>
</tr>
<tr>
<td>debug_print()</td>
<td>Print the current state of the nditer instance and debug info to stdout.</td>
</tr>
<tr>
<td>enable_external_loop()</td>
<td>When the “external_loop” was not used during construction, but</td>
</tr>
<tr>
<td>iternext()</td>
<td>Check whether iterations are left, and perform a single internal iteration without returning the result</td>
</tr>
<tr>
<td>next</td>
<td>x.next() -&gt; the next value, or raise StopIteration</td>
</tr>
<tr>
<td>remove_axis(i)</td>
<td>Removes axis i from the iterator.</td>
</tr>
<tr>
<td>remove_multi_index()</td>
<td>When the “multi_index” flag was specified, this removes it, allowing</td>
</tr>
<tr>
<td>reset()</td>
<td>Reset the iterator to its initial state.</td>
</tr>
</tbody>
</table>

```
nditer.copy()
```

Get a copy of the iterator in its current state.
Examples

```python
def debug_print():
    """Print the current state of the nditer instance and debug info to stdout."""

def enable_external_loop():
    """When the "external_loop" was not used during construction, but is desired, this modifies the iterator to behave as if the flag was specified."""

def iternext():
    """Check whether iterations are left, and perform a single internal iteration without returning the result. Used in the C-style pattern do-while pattern. For an example, see nditer."

    Returns
    --------
    iternext: bool
        Whether or not there are iterations left.

def next():
    """x.next() -> the next value, or raise StopIteration"""

def remove_axis(i):
    """Removes axis i from the iterator. Requires that the flag "multi_index" be enabled."""

def remove_multi_index():
    """When the "multi_index" flag was specified, this removes it, allowing the internal iteration structure to be optimized further."""

def reset():
    """Reset the iterator to its initial state."""

class ndenumerate(arr):
    """Multidimensional index iterator."

    Return an iterator yielding pairs of array coordinates and values.

    Parameters
    ----------
    a : ndarray
        Input array.

    See Also:
    ---------
    ndindex, flatiter

Examples

```
Methods

`next()`  Standard iterator method, returns the index tuple and array value.

`ndenumerate.next()`  
Standard iterator method, returns the index tuple and array value.

Returns

`coords`: tuple of ints

The indices of the current iteration.

`val`: scalar

The array element of the current iteration.

class numpy.ndindex(*shape)

An N-dimensional iterator object to index arrays.

Given the shape of an array, an ndindex instance iterates over the N-dimensional index of the array. At each iteration a tuple of indices is returned, the last dimension is iterated over first.

Parameters

`*args`: ints

The size of each dimension of the array.

See Also:

ndenumerate, flatiter

Examples

```python
>>> for index in np.ndindex(3, 2, 1):
...     print(index)
(0, 0, 0)
(0, 1, 0)
(1, 0, 0)
(1, 1, 0)
(2, 0, 0)
(2, 1, 0)
```
class `numpy.flatiter`  
Flat iterator object to iterate over arrays.

A `flatiter` iterator is returned by `x.flat` for any array `x`. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its `next` method.

Iteration is done in C-contiguous style, with the last index varying the fastest. The iterator can also be indexed using basic slicing or advanced indexing.

See Also:

`ndarray.flat`  
Return a flat iterator over an array.

`ndarray.flatten`  
Returns a flattened copy of an array.

Notes

A `flatiter` iterator can not be constructed directly from Python code by calling the `flatiter` constructor.

Examples

```python
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> type(fl)
<type 'numpy.flatiter'>
>>> for item in fl:
...   print item
... 0
1
2
3
4
5

>>> fl[2:4]
array([2, 3])
```

Attributes

- `coords`  
  An N-dimensional tuple of current coordinates.

```
flatiter.coords  
An N-dimensional tuple of current coordinates.

Examples

```python
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> fl.coords
(0, 0)
>>> fl.next()
0
>>> fl.coords
(0, 1)
```
Methods

- `copy()` - Get a copy of the iterator as a 1-D array.
- `next` - x.next() -> the next value, or raise StopIteration

`flatiter.copy()`
Get a copy of the iterator as a 1-D array.

Examples

```python
>>> x = np.arange(6).reshape(2, 3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> fl = x.flat
>>> fl.copy()
array([0, 1, 2, 3, 4, 5])
```

`flatiter.next`

x.next() -> the next value, or raise StopIteration

3.16 Input and output

3.16.1 NPZ files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load(file[, mmap_mode])</code></td>
<td>Load an array(s) or pickled objects from .npy, .npz, or pickled files.</td>
</tr>
<tr>
<td><code>save(file, arr)</code></td>
<td>Save an array to a binary file in NumPy .npy format.</td>
</tr>
<tr>
<td><code>savez(file, *args, **kwds)</code></td>
<td>Save several arrays into a single file in uncompressed .npz format.</td>
</tr>
<tr>
<td><code>savez_compressed(file, *args, **kwds)</code></td>
<td>Save several arrays into a single file in compressed .npz format.</td>
</tr>
</tbody>
</table>

`numpy.load(file, mmap_mode=None)`
Load an array(s) or pickled objects from .npy, .npz, or pickled files.

Parameters

- **file** : file-like object or string
  The file to read. Compressed files with the filename extension .gz are acceptable. File-like objects must support the `seek()` and `read()` methods. Pickled files require that the file-like object support the `readline()` method as well.

- **mmap_mode** : {None, ‘r+’, ‘r’, ‘w+’, ‘c’}, optional
  If not None, then memory-map the file, using the given mode (see `numpy.memmap` for a detailed description of the modes). A memory-mapped array is kept on disk. However, it can be accessed and sliced like any ndarray. Memory mapping is especially useful for accessing small fragments of large files without reading the entire file into memory.

Returns

- **result** : array, tuple, dict, etc.
  Data stored in the file. For .npz files, the returned instance of NpzFile class must be closed to avoid leaking file descriptors.
 Raises

 IOError

 If the input file does not exist or cannot be read.

 See Also:

 `save`, `savez`, `savez_compressed`, `loadtxt`

 `memmap`

 Create a memory-map to an array stored in a file on disk.

 Notes

 • If the file contains pickle data, then whatever object is stored in the pickle is returned.
 • If the file is a `.npy` file, then a single array is returned.
 • If the file is a `.npz` file, then a dictionary-like object is returned, containing `{filename: array}`
   key-value pairs, one for each file in the archive.
 • If the file is a `.npz` file, the returned value supports the context manager protocol in a similar fashion to
   the open function:

   ```python
   with load('foo.npz') as data:
       a = data['a']
   ```

   The underlying file descriptor is closed when exiting the ‘with’ block.

 Examples

 Store data to disk, and load it again:

 ```python
 >>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
 >>> np.load('/tmp/123.npy')
 array([[1, 2, 3],
        [4, 5, 6]])
 ```

 Store compressed data to disk, and load it again:

 ```python
 >>> a=np.array([[1, 2, 3], [4, 5, 6]])
 >>> b=np.array([[-1, 2]])
 >>> np.savez('/tmp/123.npz', a=a, b=b)
 >>> data = np.load('/tmp/123.npz')
 >>> data['a']
 array([[1, 2, 3],
        [4, 5, 6]])
 >>> data['b']
 array([[-1, 2]])
 >>> data.close()
 ```

 Mem-map the stored array, and then access the second row directly from disk:

 ```python
 >>> X = np.load('/tmp/123.npy', mmap_mode='r')
 >>> X[1, :]
 memmap([4, 5, 6])
 ```

 `numpy.save(file, arr)`

 Save an array to a binary file in NumPy `.npy` format.

 Parameters

 `file` : file or str
File or filename to which the data is saved. If file is a file-object, then the filename is unchanged. If file is a string, a `.npy` extension will be appended to the file name if it does not already have one.

```python
arr : array_like
    Array data to be saved.
```

See Also:

```python
savez
    Save several arrays into a `.npz` archive
```

```python
savetxt, load
```

Notes

For a description of the `.npy` format, see `format`.

Examples

```python
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()

>>> x = np.arange(10)
>>> np.save(outfile, x)

>>> outfile.seek(0)  # Only needed here to simulate closing & reopening file
>>> np.load(outfile)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```python
numpy.savez(file, *args, **kwds)
    Save several arrays into a single file in uncompressed `.npz` format.
```

If arguments are passed in with no keywords, the corresponding variable names, in the `.npz` file, are ‘arr_0’, ‘arr_1’, etc. If keyword arguments are given, the corresponding variable names, in the `.npz` file will match the keyword names.

```python
Parameters
file : str or file
    Either the file name (string) or an open file (file-like object) where the data will be saved.
If file is a string, the `.npz` extension will be appended to the file name if it is not already there.

args : Arguments, optional
    Arrays to save to the file. Since it is not possible for Python to know the names of the arrays outside `savez`, thearrays will be saved with names “arr_0”, “arr_1”, and so on. These arguments can be any expression.

kwds : Keyword arguments, optional
    Arrays to save to the file. Arrays will be saved in the file with the keyword names.
```

```python
Returns
None
```

See Also:

```python
save
    Save a single array to a binary file in NumPy format.
```
**savetxt**  
Save an array to a file as plain text.

**savez_compressed**  
Save several arrays into a compressed .npz file format

**Notes**  
The .npz file format is a zipped archive of files named after the variables they contain. The archive is not compressed and each file in the archive contains one variable in .npy format. For a description of the .npy format, see `format`.

When opening the saved .npz file with `load` a `NpzFile` object is returned. This is a dictionary-like object which can be queried for its list of arrays (with the `.files` attribute), and for the arrays themselves.

**Examples**

```python
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = np.arange(10)
>>> y = np.sin(x)

Using savez with *args, the arrays are saved with default names.

```python
>>> np.savez(outfile, x, y)
```  
```python
>>> outfile.seek(0)  # Only needed here to simulate closing & reopening file
>>> npzfile = np.load(outfile)
>>> npzfile.files
['arr_1', 'arr_0']
>>> npzfile['arr_0']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Using savez with **kwds, the arrays are saved with the keyword names.

```python
>>> outfile = TemporaryFile()
>>> np.savez(outfile, x=x, y=y)
>>> outfile.seek(0)
>>> npzfile = np.load(outfile)
>>> npzfile.files
['y', 'x']
>>> npzfile['x']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

numpy.savez_compressed(file, *args, **kwds)  
Save several arrays into a single file in compressed .npz format.

If keyword arguments are given, then filenames are taken from the keywords. If arguments are passed in with no keywords, then stored file names are arr_0, arr_1, etc.

**Parameters**

- **file**: str
  File name of .npz file.

- **args**: Arguments
  Function arguments.

- **kwds**: Keyword arguments
  Keywords.

**See Also:**

3.16. Input and output 683
**numpy.savez**
Save several arrays into an uncompressed .npz file format.

**numpy.load**
Load the files created by savez_compressed.

### 3.16.2 Text files

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.loadtxt</td>
<td>Load data from a text file.</td>
</tr>
<tr>
<td>numpy.savetxt</td>
<td>Save an array to a text file.</td>
</tr>
<tr>
<td>genfromtxt</td>
<td>Load data from a text file, with missing values handled as specified.</td>
</tr>
<tr>
<td>fromregex</td>
<td>Construct an array from a text file, using regular expression parsing.</td>
</tr>
<tr>
<td>fromstring</td>
<td>A new 1-D array initialized from raw binary or text data in a string.</td>
</tr>
<tr>
<td>ndarray.tofile</td>
<td>Write array to a file as text or binary (default).</td>
</tr>
<tr>
<td>ndarray.tolist</td>
<td>Return the array as a (possibly nested) list.</td>
</tr>
</tbody>
</table>

**numpy.loadtxt**

Load data from a text file.

Each row in the text file must have the same number of values.

**Parameters**

- `fname`: file or str
  
  File, filename, or generator to read. If the filename extension is .gz or .bz2, the file is first decompressed. Note that generators should return byte strings for Python 3k.

- `dtype`: data-type, optional
  
  Data-type of the resulting array; default: float. If this is a record data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.

- `comments`: str, optional
  
  The character used to indicate the start of a comment; default: ‘#’.

- `delimiter`: str, optional
  
  The string used to separate values. By default, this is any whitespace.

- `converters`: dict, optional
  
  A dictionary mapping column number to a function that will convert that column to a float. E.g., if column 0 is a date string: converters = {0: datestr2num}. Converters can also be used to provide a default value for missing data (but see also genfromtxt): converters = {3: lambda s: float(s.strip() or 0)}. Default: None.

- `skiprows`: int, optional
  
  Skip the first skiprows lines; default: 0.

- `usecols`: sequence, optional
  
  Which columns to read, with 0 being the first. For example, usecols = (1, 4, 5) will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.
unpack : bool, optional
    If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`. When used with a record data-type, arrays are returned for each field. Default is False.

ndmin : int, optional
    The returned array will have at least `ndmin` dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2. New in version 1.6.0.

Returns
out : ndarray
    Data read from the text file.

See Also:
load, fromstring, fromregex

genfromtxt
    Load data with missing values handled as specified.

scipy.io.loadmat
    reads MATLAB data files

Notes
This function aims to be a fast reader for simply formatted files. The `genfromtxt` function provides more sophisticated handling of, e.g., lines with missing values.

Examples
>>> from StringIO import StringIO
    # StringIO behaves like a file object
>>> c = StringIO("0 1
2 3")
>>> np.loadtxt(c)
array([[ 0., 1.],
       [ 2., 3.]])

>>> d = StringIO("M 21 72
F 35 58")
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
                      'formats': ('S1', 'i4', 'f4')})
array([('M', 21, 72.0), ('F', 35, 58.0)],
      dtype=[('gender', '|S1'), ('age', '<i4'), ('weight', '<f4')])

>>> c = StringIO("1,0,2
3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([ 1., 3.])
>>> y
array([ 2., 4.])

numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='
', header='', footer='', comments='# ')
    Save an array to a text file.

Parameters
fname : filename or file handle
    If the filename ends in `.gz`, the file is automatically saved in compressed gzip format.
loadtxt understands gzipped files transparently.

X : array_like
Data to be saved to a text file.

**fmt**: str or sequence of strs, optional

A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. '%d – %10.5f', in which case delimiter is ignored. For complex \( X \), the legal options for \( \text{fmt} \) are:

1. a single specifier, \( \text{fmt} = \%4.4e \), resulting in numbers formatted like ‘(%s+%sj)’ \( \% \text{(fmt, fmt)} \)
2. a full string specifying every real and imaginary part, e.g. ‘%4e %+.4j %4e %+.4j %4e %+.4j’ for 3 columns
3. a list of specifiers, one per column - in this case, the real and imaginary part must have separate specifiers, e.g. [‘%3e + %3ej’, ‘(%15e%+.15ej)’] for 2 columns

**delimiter**: str, optional

Character separating columns.

**newline**: str, optional

New in version 1.5.0.

**header**: str, optional

String that will be written at the beginning of the file. New in version 1.7.0.

**footer**: str, optional

String that will be written at the end of the file. New in version 1.7.0.

**comments**: str, optional

String that will be prepended to the header and footer strings, to mark them as comments. Default: ‘#’, as expected by e.g. `numpy.loadtxt`. New in version 1.7.0.

Character separating lines.

See Also:

**save**

Save an array to a binary file in NumPy `.npy` format

**savetz**

Save several arrays into a `.npz` compressed archive

Notes

Further explanation of the \( \text{fmt} \) parameter (%[flag]width[.precision]specifier):

**flags:**

- : left justify
+ : Forces to preceed result with + or -.
0 : Left pad the number with zeros instead of space (see width).

**width:**

Minimum number of characters to be printed. The value is not truncated if it has more characters.

**precision:**

- For integer specifiers (eg. \( d, i, o, x \)), the minimum number of digits.
• For e, E and f specifiers, the number of digits to print after the decimal point.
• For g and G, the maximum number of significant digits.
• For s, the maximum number of characters.

specifiers:
c : character
d or i : signed decimal integer
e or E : scientific notation with e or E.
f : decimal floating point
g, G : use the shorter of e, E or f
o : signed octal
s : string of characters
u : unsigned decimal integer
x, X : unsigned hexadecimal integer

This explanation of fmt is not complete, for an exhaustive specification see [R242].

References
[R242]

Examples

```python
>>> x = y = z = np.arange(0.0, 5.0, 1.0)
>>> np.savetxt('test.out', x, delimiter=',')  # X is an array
>>> np.savetxt('test.out', (x, y, z))  # x, y, z equal sized 1D arrays
>>> np.savetxt('test.out', x, fmt='%.4e')  # use exponential notation
```

numpy.**genfromtxt**(fname, dtype=<type 'float'>, comments='#', delimiter=None, skiprows=0, skip_header=0, skip_footer=0, converters=None, missing='', missing_values=None, filling_values=None, usecols=None, names=None, excludelist=None, deletechars=None, replace_space='_', autostrip=False, case_sensitive=True, defaultfmt='f%i', unpack=None, usemask=False, loose=True, invalid_raise=True)

Load data from a text file, with missing values handled as specified.

Each line past the first `skip_header` lines is split at the `delimiter` character, and characters following the `comments` character are discarded.

Parameters
---
fname : file or str

File, filename, or generator to read. If the filename extension is gz or bz2, the file is first decompressed. Note that generators must return byte strings in Python 3k.

dtype : dtype, optional

Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.

comments : str, optional

The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded.

delimiter : str, int, or sequence, optional
The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.

**skip_header**: int, optional

The numbers of lines to skip at the beginning of the file.

**skip_footer**: int, optional

The numbers of lines to skip at the end of the file.

**converters**: variable, optional

The set of functions that convert the data of a column to a value. The converters can also be used to provide a default value for missing data: `converters = {3: lambda s: float(s or 0)}`.

**missing_values**: variable, optional

The set of strings corresponding to missing data.

**filling_values**: variable, optional

The set of values to be used as default when the data are missing.

**usecols**: sequence, optional

Which columns to read, with 0 being the first. For example, `usecols = (1, 4, 5)` will extract the 2nd, 5th and 6th columns.

**names**: {None, True, str, sequence}, optional

If `names` is True, the field names are read from the first valid line after the first `skip_header` lines. If `names` is a sequence or a single-string of comma-separated names, the names will be used to define the field names in a structured dtype. If `names` is None, the names of the dtype fields will be used, if any.

**excludelist**: sequence, optional

A list of names to exclude. This list is appended to the default list ['return', 'file', 'print']. Excluded names are appended an underscore: for example, `file` would become `file_`.

**deletechars**: str, optional

A string combining invalid characters that must be deleted from the names.

**defaultfmt**: str, optional

A format used to define default field names, such as “f%i” or “f_%02i”.

**autostrip**: bool, optional

Whether to automatically strip white spaces from the variables.

**replace_space**: char, optional

Character(s) used in replacement of white spaces in the variables names. By default, use a ‘_’.

**case_sensitive**: {True, False, ‘upper’, ‘lower’}, optional

If True, field names are case sensitive. If False or ‘upper’, field names are converted to upper case. If ‘lower’, field names are converted to lower case.

**unpack**: bool, optional
If True, the returned array is transposed, so that arguments may be unpacked using $x, y, z = \text{loadtxt}(\ldots)$

usemask : bool, optional

If True, return a masked array. If False, return a regular array.

invalid_raise : bool, optional

If True, an exception is raised if an inconsistency is detected in the number of columns. If False, a warning is emitted and the offending lines are skipped.

Returns

out : ndarray

Data read from the text file. If usemask is True, this is a masked array.

See Also:

numpy.loadtxt

equivalent function when no data is missing.

Notes

• When spaces are used as delimiters, or when no delimiter has been given as input, there should not be any missing data between two fields.

• When the variables are named (either by a flexible dtype or with names, there must not be any header in the file (else a ValueError exception is raised).

• Individual values are not stripped of spaces by default. When using a custom converter, make sure the function does remove spaces.

References

[R20]

Examples

```python
>>> from StringIO import StringIO
>>> import numpy as np

Comma delimited file with mixed dtype

>>> s = StringIO("1,1.3,abcde")
>>> data = np.genfromtxt(s, dtype=[('myint','i8'), ('myfloat','f8'),
... ('mystring','S5')], delimiter="")
>>> data
array((1, 1.3, 'abcde'),
     dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', '|S5')])

Using dtype = None

>>> s.seek(0) # needed for StringIO example only
>>> data = np.genfromtxt(s, dtype=None,
... names = ['myint','myfloat','mystring'], delimiter="")
>>> data
array((1, 1.3, 'abcde'),
     dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', '|S5')])
```

Specifying dtype and names

3.16. Input and output
An example with fixed-width columns

```python
>>> s = StringIO("11.3abcde")
>>> data = np.genfromtxt(s, dtype=None, names=['intvar','fltvar','strvar'],
...                       delimiter=[1,3,5])
>>> data
array((1, 1.3, 'abcde'),
      dtype=[('intvar', '<i8'), ('fltvar', '<f8'), ('strvar', '|S5')])
```

numpy.fromregex (file, regexp, dtype)

Construct an array from a text file, using regular expression parsing.

The returned array is always a structured array, and is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields of the structured array.

Parameters

- **file**: str or file
  File name or file object to read.

- **regexp**: str or regexp
  Regular expression used to parse the file. Groups in the regular expression correspond to fields in the dtype.

- **dtype**: dtype or list of dtypes
  Dtype for the structured array.

Returns

- **output**: ndarray
  The output array, containing the part of the content of file that was matched by regexp. output is always a structured array.

Raises

- **TypeError**
  When dtype is not a valid dtype for a structured array.

See Also:

fromstring, loadtxt

Notes

Dtypes for structured arrays can be specified in several forms, but all forms specify at least the data type and field name. For details see doc.structured_arrays.

Examples

```python
>>> f = open('test.dat', 'w')
>>> f.write("1312 foo
1534 bar
444 qux")
>>> f.close()
```
```python
>>> regexp = r"(\d+) \S+(...)
# match \[digits, whitespace, anything\]
>>> output = np.fromregex('test.dat', regexp, ...
[('num', np.int64), ('key', 'S3')])
>>> output
array([(1312L, 'foo'), (1534L, 'bar'), (444L, 'qux')],
dtype=[('num', '<i8'), ('key', '|S3')])
>>> output['num']
array([1312, 1534, 444], dtype=int64)
```

```python
np.array(['num'])
array([1312, 1534, 444], dtype=int64)
```

data. 

```python
numpy.fromstring(string, dtype=float, count=-1, sep='')
```
A new 1-D array initialized from raw binary or text data in a string.

**Parameters**

- **string**: str
  A string containing the data.

- **dtype**: data-type, optional
  The data type of the array; default: float. For binary input data, the data must be in exactly this format.

- **count**: int, optional
  Read this number of dtype elements from the data. If this is negative (the default), the count will be determined from the length of the data.

- **sep**: str, optional
  If not provided or, equivalently, the empty string, the data will be interpreted as binary data; otherwise, as ASCII text with decimal numbers. Also in this latter case, this argument is interpreted as the string separating numbers in the data; extra whitespace between elements is also ignored.

**Returns**

- **arr**: ndarray
  The constructed array.

**Raises**

- **ValueError**
  If the string is not the correct size to satisfy the requested dtype and count.

**See Also:**

frombuffer, fromfile, fromiter

**Examples**

```python
>>> np.fromstring('0102', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
>>> np.fromstring('0102030405', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

```python
ndarray.tofile(fid, sep='', format='%s')
```
Write array to a file as text or binary (default).
Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can be recovered using the function fromfile().

**Parameters**

- **fid**: file or str
  
  An open file object, or a string containing a filename.

- **sep**: str
  
  Separator between array items for text output. If “” (empty), a binary file is written, equivalent to file.write(a.tostring()).

- **format**: str
  
  Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

**Notes**

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

### ndarray.tolist()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

**Parameters**

- none

**Returns**

- y: list
  
  The possibly nested list of array elements.

**Notes**

The array may be recreated, a = np.array(a.tolist()).

**Examples**

```python
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

### 3.16.3 Raw binary files

- `fromfile(file[, dtype, count, sep])`  
  Construct an array from data in a text or binary file.

- `ndarray.tofile(fid[, sep, format])`  
  Write array to a file as text or binary (default).
### 3.16.4 String formatting

#### `array_repr`  

Return the string representation of an array.

**Parameters**

- **arr**: ndarray  
  Input array.

- **max_line_width**: int, optional  
  The maximum number of columns the string should span. Newline characters split the string appropriately after array elements.

- **precision**: int, optional  
  Floating point precision. Default is the current printing precision (usually 8), which can be altered using `set_printoptions`.

- **suppress_small**: bool, optional  
  Represent very small numbers as zero, default is False. Very small is defined by `precision`, if the precision is 8 then numbers smaller than 5e-9 are represented as zero.

**Returns**

- **string**: str  
  The string representation of an array.

**See Also:**

- `array_str`, `array2string`, `set_printoptions`

**Examples**

```python
>>> np.array_repr(np.array([1, 2]))
'array([1, 2])'
>>> np.array_repr(np.ma.array([0.]))
'MaskedArray([0.])'
>>> np.array_repr(np.array([[], np.int32]))
'array([], dtype=int32)'

>>> x = np.array([1e-6, 4e-7, 2, 3])
>>> np.array_repr(x, precision=6, suppress_small=True)
'array([0.000001, 0. , 2. , 3. ])
```

#### `array_str`  

Return a string representation of the data in an array.

The data in the array is returned as a single string. This function is similar to `array_repr`, the difference being that `array_repr` also returns information on the kind of array and its data type.

**Parameters**

- **a**: ndarray  
  Input array.

- **max_line_width**: int, optional
Inserts newlines if text is longer than \texttt{max_line_width}. The default is, indirectly, 75.

\texttt{precision} : int, optional

Floating point precision. Default is the current printing precision (usually 8), which can be altered using \texttt{set_printoptions}.

\texttt{suppress_small} : bool, optional

Represent numbers “very close” to zero as zero; default is False. Very close is defined by precision: if the precision is 8, e.g., numbers smaller (in absolute value) than $5\times10^{-9}$ are represented as zero.

\textbf{See Also:}

array2string, array\_repr, set\_printoptions

\textbf{Examples}

```python
>>> np.array_str(np.arange(3))
'[[0 1 2]]'
```

\section*{3.16.5 Memory mapping files}

\texttt{memmap} Create a memory-map to an array stored in a \textit{binary} file on disk.

\textbf{class numpy.memmap}

Create a memory-map to an array stored in a \textit{binary} file on disk.

Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. Numpy’s memmap’s are array-like objects. This differs from Python’s \texttt{mmap} module, which uses file-like objects.

This subclass of ndarray has some unpleasant interactions with some operations, because it doesn’t quite fit properly as a subclass. An alternative to using this subclass is to create the \texttt{mmap} object yourself, then create an ndarray with \texttt{ndarray.__new__} directly, passing the object created in its ‘buffer=’ parameter.

This class may at some point be turned into a factory function which returns a view into an mmap buffer.

\textbf{Parameters}

\texttt{filename} : str or file-like object

The file name or file object to be used as the array data buffer.

\texttt{dtype} : data-type, optional

The data-type used to interpret the file contents. Default is \texttt{uint8}.

\texttt{mode} : {'r+', 'r', 'w+', 'c'}, optional

The file is opened in this mode:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘r’</td>
<td>Open existing file for reading only.</td>
</tr>
<tr>
<td>‘r+’</td>
<td>Open existing file for reading and writing.</td>
</tr>
<tr>
<td>‘w+’</td>
<td>Create or overwrite existing file for reading and writing.</td>
</tr>
<tr>
<td>‘c’</td>
<td>Copy-on-write: assignments affect data in memory, but changes are not saved to disk. The file on disk is read-only.</td>
</tr>
</tbody>
</table>

Default is ‘r+’.

\texttt{offset} : int, optional
In the file, array data starts at this offset. Since offset is measured in bytes, it should normally be a multiple of the byte-size of dtype. When mode != ‘r’, even positive offsets beyond end of file are valid; The file will be extended to accommodate the additional data. The default offset is 0.

**shape**: tuple, optional

The desired shape of the array. If mode == ‘r’ and the number of remaining bytes after offset is not a multiple of the byte-size of dtype, you must specify shape. By default, the returned array will be 1-D with the number of elements determined by file size and data-type.

**order**: {’C’, ’F’}, optional

Specify the order of the ndarray memory layout: C (row-major) or Fortran (column-major). This only has an effect if the shape is greater than 1-D. The default order is ‘C’.

**Notes**

The memmap object can be used anywhere an ndarray is accepted. Given a memmap fp, isinstance(fp, numpy.ndarray) returns True.

Memory-mapped arrays use the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always < 2GB even on 64-bit systems.

**Examples**

```python
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))
```

This example uses a temporary file so that doctest doesn’t write files to your directory. You would use a ‘normal’ filename.

```python
>>> from tempfile import mkdtemp
>>> import os.path as path
```

Create a memmap with dtype and shape that matches our data:

```python
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
```

Write data to memmap array:

```python
>>> fp[:] = data[:]
```

Deletion flushes memory changes to disk before removing the object:

```python
>>> del fp
```
Load the memmap and verify data was stored:

```python
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
```

Read-only memmap:

```python
>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> fpr.flags.writeable
False
```

Copy-on-write memmap:

```python
>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))
>>> fpc.flags.writeable
True
```

It’s possible to assign to copy-on-write array, but values are only written into the memory copy of the array, and not written to disk:

```python
>>> fpc
memmap([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
>>> fpc[0,:] = 0
>>> fpc
memmap([[ 0., 0., 0., 0.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
```

File on disk is unchanged:

```python
>>> fpr
memmap([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]], dtype=float32)
```

Offset into a memmap:

```python
>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([ 4., 5., 6., 7., 8., 9., 10., 11.], dtype=float32)
```

### Attributes

<table>
<thead>
<tr>
<th>filename</th>
<th>(str) Path to the mapped file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>(int) Offset position in the file.</td>
</tr>
<tr>
<td>mode</td>
<td>(str) File mode.</td>
</tr>
</tbody>
</table>

### Methods

- **flush()** Write any changes in the array to the file on disk.

```python
memmap.flush()
```

Write any changes in the array to the file on disk.
For further information, see `memmap`.

**Parameters**

None

**See Also:**

`memmap`

`close` Close the memmap file.

### 3.16.6 Text formatting options

- **set_printoptions**
  - Syntax: `set_printoptions([precision, threshold, ...])`
  - Set printing options.

- **get_printoptions**
  - Syntax: `get_printoptions()`
  - Return the current print options.

- **set_string_function**
  - Syntax: `set_string_function(f[, repr])`
  - Set a Python function to be used when pretty printing arrays.

#### numpy.set_printoptions

Set printing options.

These options determine the way floating point numbers, arrays and other NumPy objects are displayed.

**Parameters**

- **precision** : int, optional
  - Number of digits of precision for floating point output (default 8).

- **threshold** : int, optional
  - Total number of array elements which trigger summarization rather than full repr (default 1000).

- **edgeitems** : int, optional
  - Number of array items in summary at beginning and end of each dimension (default 3).

- **linewidth** : int, optional
  - The number of characters per line for the purpose of inserting line breaks (default 75).

- **suppress** : bool, optional
  - Whether or not suppress printing of small floating point values using scientific notation (default False).

- **nanstr** : str, optional
  - String representation of floating point not-a-number (default nan).

- **infstr** : str, optional
  - String representation of floating point infinity (default inf).

- **formatter** : dict of callables, optional
  - If not None, the keys should indicate the type(s) that the respective formatting function applies to. Callables should return a string. Types that are not specified (by their corresponding keys) are handled by the default formatters. Individual types for which a formatter can be set are:
- 'bool'
- 'int'
- 'timedelta' : a `numpy.timedelta64`
- 'datetime' : a `numpy.datetime64`
- 'float'
- 'longfloat' : 128-bit floats
- 'complexfloat'
- 'longcomplexfloat' : composed of two 128-bit floats
- 'numpy_str' : types 'numpy.string_' and 'numpy.unicode_'
- 'str' : all other strings

Other keys that can be used to set a group of types at once are:

- 'all' : sets all types
- 'int_kind' : sets 'int'
- 'float_kind' : sets 'float' and 'longfloat'
- 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
- 'str_kind' : sets 'str' and 'numpystr'

See Also:

`get_printoptions`, `set_string_function`, `array2string`

Notes

`formatter` is always reset with a call to `set_printoptions`.

Examples

Floating point precision can be set:

```python
>>> np.set_printoptions(precision=4)
>>> print np.array([1.123456789])
[ 1.1235]
```

Long arrays can be summarised:

```python
>>> np.set_printoptions(threshold=5)
>>> print np.arange(10)
[0 1 2 ..., 7 8 9]
```

Small results can be suppressed:

```python
>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
array([-4.9304e-32, -4.4409e-16, 0.0000e+00, 0.0000e+00])
>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0., 0., 0.])
```

A custom formatter can be used to display array elements as desired:

```python
>>> np.set_printoptions(formatter={'all': lambda x: 'int: ' + str(-x)})
>>> x = np.arange(3)
>>> x
array([int: 0, int: -1, int: -2])
>>> np.set_printoptions()  # formatter gets reset
>>> x
array([0, 1, 2])
```
To put back the default options, you can use:

```python
>>> np.set_printoptions(edgeitems=3, infstr='inf',
... linewidth=75, nanstr='nan', precision=8,
... suppress=False, threshold=1000, formatter=None)
```

`numpy.get_printoptions()`
Return the current print options.

**Returns**

`print_opts` : dict  
Dictionary of current print options with keys

- precision : int  
- threshold : int  
- edgeitems : int  
- linewidth : int  
- suppress : bool  
- nanstr : str  
- infstr : str  
- formatter : dict of callables

For a full description of these options, see `set_printoptions`.

**See Also:**

`set_printoptions`, `set_string_function`

`numpy.set_string_function(f, repr=True)`  
Set a Python function to be used when pretty printing arrays.

**Parameters**

- `f` : function or None
  Function to be used to pretty print arrays. The function should expect a single array argument and return a string of the representation of the array. If None, the function is reset to the default NumPy function to print arrays.

- `repr` : bool, optional
  If True (default), the function for pretty printing (`__repr__`) is set, if False the function that returns the default string representation (`__str__`) is set.

**See Also:**

`set_printoptions`, `get_printoptions`

**Examples**

```python
>>> def pprint(arr):
...     return 'HA! - What are you going to do now?'
... 
... #
>>> np.set_string_function(pprint)
>>> a = np.arange(10)
>>> a
HA! - What are you going to do now?
>>> print a
[0 1 2 3 4 5 6 7 8 9]
```
We can reset the function to the default:

```python
>>> np.set_string_function(None)
a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`repr` affects either pretty printing or normal string representation. Note that `__repr__` is still affected by setting `__str__` because the width of each array element in the returned string becomes equal to the length of the result of `__str__()`.

```python
>>> x = np.arange(4)
>>> np.set_string_function(lambda x: 'random', repr=False)
>>> x.__str__()
'random'
>>> x.__repr__()
'array([ 0, 1, 2, 3])'
```

### 3.16.7 Base-n representations

**`binary_repr(num[, width])`**
Return the binary representation of the input number as a string.

**`base_repr(number[, base, padding])`**
Return a string representation of a number in the given base system.

#### NumPy

**`numpy.binary_repr(num, width=None)`**
Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two's complement of the number is returned, with respect to that width.

In a two's-complement system negative numbers are represented by the two’s complement of the absolute value. This is the most common method of representing signed integers on computers [R16]. A N-bit two’s-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1} - 1$.

**Parameters**

- `num` : int
  Only an integer decimal number can be used.

- `width` : int, optional
  The length of the returned string if `num` is positive, the length of the two’s complement if `num` is negative.

**Returns**

- `bin` : str
  Binary representation of `num` or two’s complement of `num`.

**See Also:**

- `base_repr`
  Return a string representation of a number in the given base system.

**Notes**

`binary_repr` is equivalent to using `base_repr` with base 2, but about 25x faster.

**References**

[R16]
Examples

```python
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

The two’s complement is returned when the input number is negative and width is specified:

```python
>>> np.binary_repr(-3, width=4)
'1101'
```

**numpy.base_repr** *(number, base=2, padding=0)*

Return a string representation of a number in the given base system.

**Parameters**

- `number`: int
  The value to convert. Only positive values are handled.

- `base`: int, optional
  Convert `number` to the `base` number system. The valid range is 2-36, the default value is 2.

- `padding`: int, optional
  Number of zeros padded on the left. Default is 0 (no padding).

**Returns**

- `out`: str
  String representation of `number` in `base` system.

**See Also:**

- `binary_repr`
  Faster version of `base_repr` for base 2.

Examples

```python
>>> np.base_repr(5)
'101'
>>> np.base_repr(6, 5)
'11'
>>> np.base_repr(7, base=5, padding=3)
'00012'
>>> np.base_repr(10, base=16)
'A'
>>> np.base_repr(32, base=16)
'20'
```

### 3.16.8 Data sources

| **DataSource** ([destpath]) | A generic data source file (file, http, ftp, ...) |
class `numpy.DataSource` *(destpath=’.’)*  
A generic data source file (file, http, ftp, ...).  

DataSources can be local files or remote files/URLs. The files may also be compressed or uncompressed. DataSource hides some of the low-level details of downloading the file, allowing you to simply pass in a valid file path (or URL) and obtain a file object.

**Parameters**  

- `destpath` : str or None, optional  
  Path to the directory where the source file gets downloaded to for use. If `destpath` is None, a temporary directory will be created. The default path is the current directory.

**Notes**  

URLs require a scheme string (`http://`) to be used, without it they will fail:

```python  
>>> repos = DataSource()  
>>> repos.exists('www.google.com/index.html')  
False  
>>> repos.exists('http://www.google.com/index.html')  
True  
```

Temporary directories are deleted when the DataSource is deleted.

**Examples**  

```python  
>>> ds = DataSource('/home/guido')  
>>> urlname = 'http://www.google.com/index.html'  
>>> gfile = ds.open('http://www.google.com/index.html')  
# remote file  
>>> ds.abspath(urlname)  
'/home/guido/www.google.com/site/index.html'  
>>> ds = DataSource(None)  
# use with temporary file  
>>> (open file '/home/guido/foobar.txt', mode 'r' at 0x91d4430)  
>>> ds.abspath('/home/guido/foobar.txt')  
'/tmp/tmpy4pgsP/home/guido/foobar.txt'  
```

**Methods**

- `abspath(path)`  
  Return absolute path of file in the DataSource directory.  

- `exists(path)`  
  Test if path exists.

- `open(path[, mode])`  
  Open and return file-like object.

**DataSource.abspath**(path)  
Return absolute path of file in the DataSource directory.  

If `path` is an URL, then `abspath` will return either the location the file exists locally or the location it would exist when opened using the `open` method.

**Parameters**

- `path` : str  
  Can be a local file or a remote URL.

**Returns**

- `out` : str  
  Complete path, including the DataSource destination directory.
Notes

The functionality is based on `os.path.abspath`.

`DataSource.exists(path)`

Test if path exists.

Test if `path` exists as (and in this order):

- a local file.
- a remote URL that has been downloaded and stored locally in the `DataSource` directory.
- a remote URL that has not been downloaded, but is valid and accessible.

Parameters

path : str

Can be a local file or a remote URL.

Returns

out : bool

True if `path` exists.

Notes

When `path` is an URL, `exists` will return True if it’s either stored locally in the `DataSource` directory, or is a valid remote URL. `DataSource` does not discriminate between the two, the file is accessible if it exists in either location.

`DataSource.open(path, mode='r')`

Open and return file-like object.

If `path` is an URL, it will be downloaded, stored in the `DataSource` directory and opened from there.

Parameters

path : str

Local file path or URL to open.

mode : {'r', 'w', 'a'}, optional

Mode to open `path`. Mode ‘r’ for reading, ‘w’ for writing, ‘a’ to append. Available modes depend on the type of object specified by `path`. Default is ‘r’.

Returns

out : file object

File object.

3.17 Linear algebra (numpy.linalg)

3.17.1 Matrix and vector products

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot(a, b[, out])</td>
<td>Dot product of two arrays.</td>
</tr>
<tr>
<td>vdot(a, b)</td>
<td>Return the dot product of two vectors.</td>
</tr>
<tr>
<td>inner(a, b)</td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td>outer(a, b)</td>
<td>Compute the outer product of two vectors.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.72 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tensordot(a, b[, axes])</code></td>
<td>Compute tensor dot product along specified axes for arrays &gt;= 1-D.</td>
</tr>
<tr>
<td><code>einsum(subscripts, *operands[, out, dtype, ...])</code></td>
<td>Evaluates the Einstein summation convention on the operands.</td>
</tr>
<tr>
<td><code>linalg.matrix_power(M, n)</code></td>
<td>Raise a square matrix to the (integer) power n.</td>
</tr>
<tr>
<td><code>kron(a, b)</code></td>
<td>Kronecker product of two arrays.</td>
</tr>
</tbody>
</table>

```python
numpy.dot(a, b, out=None)
```

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of `a` and the second-to-last of `b`:

\[
dot(a, b)[i,j,k,m] = \sum(a[i,j,:]*b[k,:,m])
\]

**Parameters**

- `a`: array_like
  - First argument.
- `b`: array_like
  - Second argument.
- `out`: ndarray, optional
  - Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for `dot(a, b)`. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

**Returns**

- `output`: ndarray
  - Returns the dot product of `a` and `b`. If `a` and `b` are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If `out` is given, then it is returned.

**Raises**

- `ValueError`
  - If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.

**See Also:**

- `vdot`
  - Complex-conjugating dot product.
- `tensordot`
  - Sum products over arbitrary axes.
- `einsum`
  - Einstein summation convention.

**Examples**

```python
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:
For 2-D arrays it’s the matrix product:

```python
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```python
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:]*b[1,2,:,2])
499128
```

```python
numpy.vdot(a, b)
```

Return the dot product of two vectors.

The `vdot(a, b)` function handles complex numbers differently than `dot(a, b)`. If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

Note that `vdot` handles multidimensional arrays differently than `dot`: it does not perform a matrix product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

**Parameters**

- `a`: array_like
  - If `a` is complex the complex conjugate is taken before calculation of the dot product.

- `b`: array_like
  - Second argument to the dot product.

**Returns**

- `output`: ndarray
  - Dot product of `a` and `b`. Can be an int, float, or complex depending on the types of `a` and `b`.

**See Also:**

- `dot`
  - Return the dot product without using the complex conjugate of the first argument.

**Examples**

```python
>>> a = np.array([[1+2j, 3+4j]])
>>> b = np.array([[5+6j, 7+8j]])
>>> np.vdot(a, b)
(70-8j)
>>> np.vdot(b, a)
(70+8j)
```

Note that higher-dimensional arrays are flattened!

```python
>>> a = np.array([[[1, 4], [5, 6]]])
>>> b = np.array([[[4, 1], [2, 2]]])
>>> np.vdot(a, b)
```

3.17. Linear algebra (numpy.linalg)
numpy.inner(a, b)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum
product over the last axes.

Parameters

a, b : array_like

If a and b are nonscalar, their last dimensions of must match.

Returns

out : ndarray

out.shape = a.shape[:-1] + b.shape[:-1]

Raises

ValueError

If the last dimension of a and b has different size.

See Also:

tensordot

Sum products over arbitrary axes.

dot

Generalised matrix product, using second last dimension of b.

einsum

Einstein summation convention.

Notes

For vectors (1-D arrays) it computes the ordinary inner-product:

\[
\text{np.inner}(a, b) = \sum(a[:]*b[:])
\]

More generally, if ndim(a) = r > 0 and ndim(b) = s > 0:

\[
\text{np.inner}(a, b) = \text{np.tensordot}(a, b, \text{axes}=(-1,-1))
\]

or explicitly:

\[
\text{np.inner}(a, b)[i0,...,ir-1,j0,...,js-1]
= \sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
\]

In addition a or b may be scalars, in which case:

\[
\text{np.inner}(a,b) = a*b
\]

Examples

Ordinary inner product for vectors:
A multidimensional example:

```python
g = np.arange(24).reshape((2,3,4))
>> g
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

An example where \( b \) is a scalar:

```python
g = np.eye(2)
>> np.inner(a, b)
array([[ 0.5,  1.5],
       [ 2.,  3.]])
```

**numpy.outer(a, b)**

Compute the outer product of two vectors.

Given two vectors, \( a = [a0, a1, ..., aM] \) and \( b = [b0, b1, ..., bN] \), the outer product [R53] is:

\[
\begin{array}{ccc}
  a0*b0 & a0*b1 & \ldots & a0*bN \\
  a1*b0 & \ldots & \ldots & \ldots & a1*bN \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  aM*b0 & \ldots & \ldots & \ldots & aM*bN \\
\end{array}
\]

**Parameters**

- **a**: (M,) array_like
  
  First input vector. Input is flattened if not already 1-dimensional.

- **b**: (N,) array_like
  
  Second input vector. Input is flattened if not already 1-dimensional.

**Returns**

- **out**: (M, N) ndarray
  
  \( out[i, j] = a[i] \times b[j] \)

**See Also**

- **inner**, **einsum**

**References**

[R53]

**Examples**

Make a (very coarse) grid for computing a Mandelbrot set:

```python
g = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>> g
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])
```
numpy.tensordot(a, b, axes=2)
Compute tensor dot product along specified axes for arrays >= 1-D.

Given two tensors (arrays of dimension greater than or equal to one), a and b, and an array_like object containing two array_like objects, (a_axes, b_axes), sum the products of a’s and b’s elements (components) over the axes specified by a_axes and b_axes. The third argument can be a single non-negative integer_like scalar, N; if it is such, then the last N dimensions of a and the first N dimensions of b are summed over.

Parameters
a, b : array_like, len(shape) >= 1
   Tensors to “dot”.
axes : variable type
   • integer_like scalar Number of axes to sum over (applies to both arrays); or
   • (2,) array_like, both elements array_like of the same length List of axes to be summed over, first sequence applying to a, second to b.

See Also:
dot, einsum

Notes
When there is more than one axis to sum over - and they are not the last (first) axes of a (b) - the argument axes should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

Examples
A “traditional” example:

```python
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> c = np.tensordot(a, b, axes=([1,0],[0,1]))
>>> c.shape
```


An extended example taking advantage of the overloading of + and *:

```python
>>> a = np.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = np.array(('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a; A
array([[1, 2],
       [3, 4]])
array([[a, b],
       [c, d]], dtype=object)

>>> np.tensordot(a, A) # third argument default is 2
array([abbcccdddd, aaaaabbbbbcddddddddd], dtype=object)

>>> np.tensordot(a, A, 1)
array([[[acc, bdd],
        [aaaccccc, bbbddddd]],
       [[aaaaccccccc, bbbbbddddddd],
        [aaaaacccccccc, bbbbbbbddddddd]], dtype=object)

>>> np.tensordot(a, A, 0) # "Left for reader" (result too long to incl.)
array([[[[a, b],
        [c, d]],
       ...]
```

```python
>>> np.tensordot(a, A, (0, 1))
array([[abbcccddd, aaaaabbbbbcddddddddd],
       [ahhbb, cccddd]], dtype=object)

>>> np.tensordot(a, A, (2, 1))
array([[[abb, cdd],
        [aabbb, cccddd]],
```
```python
>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array([abbbcccccddddddd, aabbbccccccdddddddd], dtype=object)
>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array([acccbbdddd, aaaaacccccccbbbbbbdddddddd], dtype=object)
```

**numpy.einsum()**

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way compute such summations. The best way to understand this function is to try the examples below, which show how many common NumPy functions can be implemented as calls to `einsum`.

**Parameters**

- **subscripts** : str
  Specifies the subscripts for summation.

- **operands** : list of array_like
  These are the arrays for the operation.

- **out** : ndarray, optional
  If provided, the calculation is done into this array.

- **dtype** : data-type, optional
  If provided, forces the calculation to use the data type specified. Note that you may have to also give a more liberal `casting` parameter to allow the conversions.

- **order** : {'C', 'F', 'A', 'K'}, optional
  Controls the memory layout of the output. ‘C’ means it should be C contiguous. ‘F’ means it should be Fortran contiguous, ‘A’ means it should be ‘F’ if the inputs are all ‘F’, ‘C’ otherwise. ‘K’ means it should be as close to the layout as the inputs as is possible, including arbitrarily permuted axes. Default is ‘K’.

- **casting** : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
  Controls what kind of data casting may occur. Setting this to ‘unsafe’ is not recommended, as it can adversely affect accumulations.
  - ‘no’ means the data types should not be cast at all.
  - ‘equiv’ means only byte-order changes are allowed.
  - ‘safe’ means only casts which can preserve values are allowed.
  - ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
  - ‘unsafe’ means any data conversions may be done.

**Returns**

- **output** : ndarray
  The calculation based on the Einstein summation convention.
See Also:

dot, inner, outer, tensordot

Notes

New in version 1.6.0. The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand. Repeated subscripts labels in one operand take the diagonal. For example, np.einsum('ii', a) is equivalent to np.trace(a).

Whenever a label is repeated, it is summed, so np.einsum('i,i', a, b) is equivalent to np.inner(a, b). If a label appears only once, it is not summed, so np.einsum('i', a) produces a view of a with no changes.

The order of labels in the output is by default alphabetical. This means that np.einsum('ij', a) doesn’t affect a 2D array, while np.einsum('ji', a) takes its transpose.

The output can be controlled by specifying output subscript labels as well. This specifies the label order, and allows summing to be disallowed or forced when desired. The call np.einsum('i->', a) is like np.sum(a, axis=-1), and np.einsum('ii->i', a) is like np.diag(a). The difference is that einsum does not allow broadcasting by default.

To enable and control broadcasting, use an ellipsis. Default NumPy-style broadcasting is done by adding an ellipsis to the left of each term, like np.einsum('...ii->...i', a). To take the trace along the first and last axes, you can do np.einsum('...i', a), or to do a matrix-matrix product with the left-most indices instead of rightmost, you can do np.einsum('ij...,jk...->ik...', a, b).

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new array. Thus, taking the diagonal as np.einsum('ii->i', a) produces a view.

An alternative way to provide the subscripts and operands is as einsum(op0, sublist0, op1, sublist1, ..., [sublistout]). The examples below have corresponding einsum calls with the two parameter methods.

Examples

```python
>>> a = np.arange(25).reshape(5,5)
>>> b = np.arange(5)
>>> c = np.arange(6).reshape(2,3)

>>> np.einsum('ii', a)
60
>>> np.einsum(a, [0, 0])
60
>>> np.trace(a)
60

>>> np.einsum('ii->i', a)
array([ 0,  6, 12, 18, 24])
>>> np.einsum(a, [0, 0], [0])
array([ 0,  6, 12, 18, 24])
>>> np.diag(a)
array([ 0,  6, 12, 18, 24])

>>> np.einsum('ij,j', a, b)
array([[ 30,  80, 130, 180, 230]])
>>> np.einsum(a, [0, 1], b, [1])
array([[ 30,  80, 130, 180, 230]])
```
```python
>>> np.dot(a, b)
numpy.array([ 30, 80, 130, 180, 230])

>>> np.einsum('ji', c)
numpy.array([[0, 3],
             [1, 4],
             [2, 5]])

>>> np.einsum(c, [1,0])
numpy.array([[0, 3],
             [1, 4],
             [2, 5]])

>>> c.T
numpy.array([[0, 3],
             [1, 4],
             [2, 5]])

>>> np.einsum('..., ...', 3, c)
numpy.array([[ 0, 3, 6],
             [ 9, 12, 15]])

>>> np.einsum(3, [Ellipsis], c, [Ellipsis])
numpy.array([[ 0, 3, 6],
             [ 9, 12, 15]])

>>> np.multiply(3, c)
numpy.array([[ 0, 3, 6],
             [ 9, 12, 15]])

>>> np.einsum('i,i', b, b)
30

>>> np.einsum(b, [0], b, [0])
30

>>> np.inner(b,b)
30

>>> np.einsum('i,j', np.arange(2)+1, b)
numpy.array([[0, 1, 2, 3, 4],
             [0, 2, 4, 6, 8]])

>>> np.einsum(np.arange(2)+1, [0], b, [1])
numpy.array([[0, 1, 2, 3, 4],
             [0, 2, 4, 6, 8]])

>>> np.outer(np.arange(2)+1, b)
numpy.array([[0, 1, 2, 3, 4],
             [0, 2, 4, 6, 8]])

>>> np.einsum('i...->...', a)
numpy.array([50, 55, 60, 65, 70])

>>> np.einsum(a, [0,Ellipsis], [Ellipsis])
numpy.array([50, 55, 60, 65, 70])

>>> np.sum(a, axis=0)
numpy.array([50, 55, 60, 65, 70])

>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> np.einsum('ijk,jil->kl', a, b)
numpy.array([[ 4400., 4730.],
             [ 4532., 4874.],
             [ 4664., 5018.],
             [ 4796., 5162.],
             [ 4928., 5306.]]
```
```python
>>> np.einsum(a, [0,1,2], b, [1,0,3], [2,3])
array([[ 4400., 4730.],
       [ 4532., 4874.],
       [ 4664., 5018.],
       [ 4796., 5162.],
       [ 4928., 5306.]]))
```

```python
>>> np.tensordot(a,b, axes=([1,0],[0,1]))
array([[ 4400., 4730.],
       [ 4532., 4874.],
       [ 4664., 5018.],
       [ 4796., 5162.],
       [ 4928., 5306.]]))
```

```python
numpy.linalg.matrix_power(M, n)
```

Raise a square matrix to the (integer) power \( n \).

For positive integers \( n \), the power is computed by repeated matrix squarings and matrix multiplications. If \( n = 0 \), the identity matrix of the same shape as \( M \) is returned. If \( n < 0 \), the inverse is computed and then raised to the \( \text{abs}(n) \).

**Parameters**

- M : ndarray or matrix object
  
  Matrix to be "powered." Must be square, i.e. \( M.\text{shape} == (m, m) \), with \( m \) a positive integer.

- n : int
  
  The exponent can be any integer or long integer, positive, negative, or zero.

**Returns**

- M**n : ndarray or matrix object
  
  The return value is the same shape and type as \( M \); if the exponent is positive or zero then the type of the elements is the same as those of \( M \). If the exponent is negative the elements are floating-point.

**Raises**

- LinAlgError
  
  If the matrix is not numerically invertible.

**See Also:**

- matrix
  
  Provides an equivalent function as the exponentiation operator (**, not ^).

**Examples**

```python
>>> from numpy import linalg as LA
>>> i = np.array([[0, 1], [-1, 0]]) # matrix equiv. of the imaginary unit
>>> LA.matrix_power(i, 3) # should = -i
array([[ 0, -1],
       [ 1,  0]])
>>> LA.matrix_power(np.matrix(i), 3) # matrix arg returns matrix
matrix([[ 0, -1],
       [ 1,  0]])
>>> LA.matrix_power(i, 0)
array([[1, 0],
       [0, 1]])
>>> LA.matrix_power(i, -3) # should = 1/(-i) = i, but w/ f.p. elements
```

3.17. Linear algebra (numpy.linalg)
```python
array([[ 0.,  1.],
       [-1.,  0.]]

Somewhat more sophisticated example

```n
```python
>>> q = np.zeros((4, 4))
>>> q[0:2, 0:2] = -i
>>> q[2:4, 2:4] = i
>>> q # one of the three quaternion units not equal to 1
array([[ 0., -1., 0., 0.],
       [ 1.,  0., 0., 0.],
       [ 0.,  0., 0., 1.],
       [ 0.,  0., -1., 0.]]

>>> LA.matrix_power(q, 2) # = -np.eye(4)
array([[-1., 0., 0., 0.],
       [ 0., -1., 0., 0.],
       [ 0.,  0., -1., 0.],
       [ 0.,  0.,  0., -1.]]
```

numpy.kron(a, b)
Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

Parameters

- a, b : array_like

Returns

- out : ndarray

See Also:

- outer
  The outer product

Notes

The function assumes that the number of dimensions of a and b are the same, if necessary prepending the smallest with ones. If a.shape = (r0,r1,...,rN) and b.shape = (s0,s1,...,sN), the Kronecker product has shape (r0*s0, r1*s1, ..., rN*sN). The elements are products of elements from a and b, organized explicitly by:

kron(a,b)[k0,k1,...,kN] = a[i0,i1,...,iN] * b[j0,j1,...,jN]

where:

kt = it * st + jt, t = 0,...,N

In the common 2-D case (N=1), the block structure can be visualized:

```
[[ a[0,0]*b, a[0,1]*b, ... , a[0,-1]*b ],
  [ ...               ...               ],
  [ a[-1,0]*b, a[-1,1]*b, ... , a[-1,-1]*b ]]```

Examples

```python
>>> np.kron([1,10,100], [5,6,7])
array([ 5,  6,  7,  50,  60,  70,  500,  600,  700])
>>> np.kron([5,6,7], [1,10,100])
array([ 5,  6,  7,  50,  60,  70,  500,  600,  700])
```
```python
>>> np.kron(np.eye(2), np.ones((2,2)))
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  1.,  1.]])

>>> a = np.arange(100).reshape((2,5,2,5))
>>> b = np.arange(24).reshape((2,3,4))
>>> c = np.kron(a,b)
>>> c.shape
(2, 10, 6, 20)
>>> I = (1,3,0,2)
>>> J = (0,2,1)
>>> J1 = (0,) + J  # extend to ndim=4
>>> S1 = (1,) + b.shape
>>> K = tuple(np.array(I) * np.array(S1) + np.array(J1))
>>> c[K] == a[I]*b[J]
True
```

3.17.2 Decompositions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linalg.cholesky(a)</code></td>
<td>Cholesky decomposition.</td>
</tr>
<tr>
<td><code>linalg.qr(a[, mode])</code></td>
<td>Compute the qr factorization of a matrix.</td>
</tr>
<tr>
<td><code>linalg.svd(a[, full_matrices, compute_uv])</code></td>
<td>Singular Value Decomposition.</td>
</tr>
</tbody>
</table>

**numpy.linalg.cholesky(a)**

Cholesky decomposition.

Return the Cholesky decomposition, \( L \cdot L^\top \), of the square matrix \( a \), where \( L \) is lower-triangular and \( .^\top \) is the conjugate transpose operator (which is the ordinary transpose if \( a \) is real-valued). \( a \) must be Hermitian (symmetric if real-valued) and positive-definite. Only \( L \) is actually returned.

**Parameters**

- \( a : (...) \times M \times M \) array_like
  
  Hermitian (symmetric if all elements are real), positive-definite input matrix.

**Returns**

- \( L : (...) \times M \times M \) array_like
  
  Upper or lower-triangular Cholesky factor of \( a \). Returns a matrix object if \( a \) is a matrix object.

**Raises**

- LinAlgError
  
  If the decomposition fails, for example, if \( a \) is not positive-definite.

**Notes**

Broadcasting rules apply, see the numpy.linalg documentation for details.

The Cholesky decomposition is often used as a fast way of solving

\[
Ax = b
\]
(when $A$ is both Hermitian/symmetric and positive-definite).

First, we solve for $y$ in

$$Ly = b,$$

and then for $x$ in

$$LHx = y.$$

### Examples

```python
>>> A = np.array([[1,-2j],[2j,5]])
>>> A
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> L = np.linalg.cholesky(A)
>>> L
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> np.dot(L, L.T.conj()) # verify that L * L.H = A
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> A = [[1,-2j],[2j,5]] # what happens if A is only array_like?
>>> np.linalg.cholesky(A) # an ndarray object is returned
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> # But a matrix object is returned if A is a matrix object
>>> LA.cholesky(np.matrix(A))
matrix([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
```

`numpy.linalg.qr (a, mode='reduced')`

Compute the qr factorization of a matrix.

Factor the matrix $a$ as $qr$, where $q$ is orthonormal and $r$ is upper-triangular.

#### Parameters

- **a** : array_like, shape (M, N)
  
  Matrix to be factored.

- **mode** : {'reduced', 'complete', 'r', 'raw', 'full', 'economic'}, optional
  
  If $K = \min(M, N)$, then

  - ‘reduced’ : returns $q, r$ with dimensions $(M, K), (K, N)$ (default) ‘complete’ : returns $q, r$ with dimensions $(M, M), (M, N)$ ‘r’ : returns $r$ only with dimensions $(K, N)$ ‘raw’ : returns $h, \tau$ with dimensions $(N, M), (K, K)$ ‘full’ : alias of ‘reduced’, deprecated

  The options ‘reduced’, ‘complete’, and ‘raw’ are new in numpy 1.8, see the notes for more information. The default is ‘reduced’ and to maintain backward compatibility with earlier versions of numpy both it and the old default ‘full’ can be omitted. Note that array $h$ returned in ‘raw’ mode is transposed for calling Fortran. The ‘economic’ mode is deprecated. The modes ‘full’ and ‘economic’ may be passed using only the first letter for backwards compatibility, but all others must be spelled out. See the Notes for more explanation.
Returns

- **q**: ndarray of float or complex, optional
  A matrix with orthonormal columns. When mode = ‘complete’ the result is an orthogonal/unitary matrix depending on whether or not a is real/complex. The determinant may be either +/- 1 in that case.

- **r**: ndarray of float or complex, optional
  The upper-triangular matrix.

- **(h, tau)**: ndarrays of np.double or np.cdouble, optional
  The array h contains the Householder reflectors that generate q along with r. The tau array contains scaling factors for the reflectors. In the deprecated ‘economic’ mode only h is returned.

Raises

- **LinAlgError**
  If factoring fails.

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, and zungqr.

For more information on the qr factorization, see for example: [http://en.wikipedia.org/wiki/QR_factorization](http://en.wikipedia.org/wiki/QR_factorization)

Subclasses of `ndarray` are preserved except for the ‘raw’ mode. So if a is of type `matrix`, all the return values will be matrices too.

New ‘reduced’, ‘complete’, and ‘raw’ options for mode were added in NumPy 1.8 and the old option ‘full’ was made an alias of ‘reduced’. In addition the options ‘full’ and ‘economic’ were deprecated. Because ‘full’ was the previous default and ‘reduced’ is the new default, backward compatibility can be maintained by letting mode default. The ‘raw’ option was added so that LAPACK routines that can multiply arrays by q using the Householder reflectors can be used. Note that in this case the returned arrays are of type np.double or np.cdouble and the h array is transposed to be FORTRAN compatible. No routines using the ‘raw’ return are currently exposed by numpy, but some are available in lapack_lite and just await the necessary work.

Examples

```python
>>> a = np.random.randn(9, 6)
>>> q, r = np.linalg.qr(a)
>>> np.allclose(a, np.dot(q, r))  # a does equal qr
True
>>> r2 = np.linalg.qr(a, mode='r')
>>> r3 = np.linalg.qr(a, mode='economic')
>>> np.allclose(r, r2)  # mode='r' returns the same r as mode='full'
True
>>> # But only triu parts are guaranteed equal when mode='economic'
>>> np.allclose(r, np.triu(r3[:6,:6], k=0))
True
```

Example illustrating a common use of `qr`: solving of least squares problems

What are the least-squares-best \( m \) and \( y_0 \) in \( y = y_0 + mx \) for the following data: \{(0,1), (1,0), (1,2), (2,1)\}.

(Graph the points and you’ll see that it should be \( y_0 = 0, m = 1 \).) The answer is provided by solving the over-determined matrix equation \( Ax = b \), where:

\[
A = \begin{bmatrix}
[0, 1], [1, 1], [1, 1], [2, 1]
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
y_0
\end{bmatrix}
\]

\[
b = \begin{bmatrix}
[1], [0], [2], [1]
\end{bmatrix}
\]
If $A = qr$ such that $q$ is orthonormal (which is always possible via Gram-Schmidt), then $x = \text{inv}(r) \times (q.T) \times b$. (In numpy practice, however, we simply use \texttt{lstsq}.)

```python
>>> A = np.array([[0, 1], [1, 1], [1, 1], [2, 1]])
>>> A
array([[0, 1],
       [1, 1],
       [1, 1],
       [2, 1]])
>>> b = np.array([1, 0, 2, 1])
>>> q, r = LA.qr(A)
>>> p = np.dot(q.T, b)
>>> np.dot(LA.inv(r), p)
array([ 1.1e-16, 1.0e+00])
```

\texttt{numpy.linalg.svd}(a, full_matrices=1, compute_uv=1)

Singular Value Decomposition.

Factors the matrix $a$ as $u \times \text{np.diag}(s) \times v$, where $u$ and $v$ are unitary and $s$ is a 1-d array of $a$'s singular values.

**Parameters**

- **a**: (..., M, N) array_like
  - A real or complex matrix of shape $(M, N)$.
- **full_matrices** : bool, optional
  - If True (default), $u$ and $v$ have the shapes $(M, M)$ and $(N, N)$, respectively. Otherwise, the shapes are $(M, K)$ and $(K, N)$, respectively, where $K = \min(M, N)$.
- **compute_uv** : bool, optional
  - Whether or not to compute $u$ and $v$ in addition to $s$. True by default.

**Returns**

- **u** : (..., M, M), (..., M, K) array
  - Unitary matrices. The actual shape depends on the value of full_matrices. Only returned when compute_uv is True.
- **s** : (..., K) array
  - The singular values for every matrix, sorted in descending order.
- **v** : (..., N, N), (..., K, N) array
  - Unitary matrices. The actual shape depends on the value of full_matrices. Only returned when compute_uv is True.

**Raises**

- **LinAlgError**
  - If SVD computation does not converge.

**Notes**

Broadcasting rules apply, see the \texttt{numpy.linalg} documentation for details.

The decomposition is performed using LAPACK routine _gesdd

The SVD is commonly written as $a = U S V.H$. The $v$ returned by this function is $V.H$ and $u = U$.

If $U$ is a unitary matrix, it means that it satisfies $U.H = \text{inv}(U)$. 

718 Chapter 3. Routines
The rows of \( v \) are the eigenvectors of \( \mathbf{a} \mathbf{a}^\text{T} \). The columns of \( u \) are the eigenvectors of \( \mathbf{a}^\text{T} \mathbf{a} \). For row \( i \) in \( v \) and column \( i \) in \( u \), the corresponding eigenvalue is \( s[i]^2 \).

If \( \mathbf{a} \) is a matrix object (as opposed to an ndarray), then so are all the return values.

**Examples**

```python
>>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
```

Reconstruction based on full SVD:

```python
>>> U, s, V = np.linalg.svd(a, full_matrices=True)
>>> U.shape, V.shape, s.shape
((9, 9), (6, 6), (6,))
>>> S = np.zeros((9, 6), dtype=complex)
>>> S[:6, :6] = np.diag(s)
>>> np.allclose(a, np.dot(U, np.dot(S, V)))
True
```

Reconstruction based on reduced SVD:

```python
>>> U, s, V = np.linalg.svd(a, full_matrices=False)
>>> U.shape, V.shape, s.shape
((9, 6), (6, 6), (6,))
>>> S = np.diag(s)
>>> np.allclose(a, np.dot(U, np.dot(S, V)))
True
```

### 3.17.3 Matrix eigenvalues

<table>
<thead>
<tr>
<th>NumPy linalg</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eig(a)</code></td>
<td>Compute the eigenvalues and right eigenvectors of a square array.</td>
</tr>
<tr>
<td><code>eigh(a[, UPLO])</code></td>
<td>Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.</td>
</tr>
<tr>
<td><code>eigvals(a)</code></td>
<td>Compute the eigenvalues of a general matrix.</td>
</tr>
<tr>
<td><code>eigvalsh(a[, UPLO])</code></td>
<td>Compute the eigenvalues of a Hermitian or real symmetric matrix.</td>
</tr>
</tbody>
</table>

**NumPy linalg.eig(a)**

Compute the eigenvalues and right eigenvectors of a square array.

**Parameters**

\( a : (\ldots, M, M) \) array

Matrices for which the eigenvalues and right eigenvectors will be computed

**Returns**

\( w : (\ldots, M) \) array

The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered. The resulting array will be always be of complex type. When \( a \) is real the resulting eigenvalues will be real (0 imaginary part) or occur in conjugate pairs

\( v : (\ldots, M, M) \) array

The normalized (unit “length”) eigenvectors, such that the column \( v[:,i] \) is the eigenvector corresponding to the eigenvalue \( w[i] \).

**Raises**

LinAlgError

If the eigenvalue computation does not converge.
See Also:

**eigvalsh**

eigenvalues of a symmetric or Hermitian (conjugate symmetric) array.

**eigvals**
eigenvalues of a non-symmetric array.

Notes

Broadcasting rules apply, see the numpy.linalg documentation for details.

This is implemented using the \_geev LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

The number \( w \) is an eigenvalue of \( a \) if there exists a vector \( v \) such that \( \text{dot}(a, v) = w \times v \). Thus, the arrays \( a, w, \) and \( v \) satisfy the equations \( \text{dot}(a[:, :], v[:, i]) = w[i] \times v[:, i] \) for \( i \in \{0, ..., M - 1\} \).

The array \( v \) of eigenvectors may not be of maximum rank, that is, some of the columns may be linearly dependent, although round-off error may obscure that fact. If the eigenvalues are all different, then theoretically the eigenvectors are linearly independent. Likewise, the (complex-valued) matrix of eigenvectors \( v \) is unitary if the matrix \( a \) is normal, i.e., if \( \text{dot}(a, a.H) = \text{dot}(a.H, a) \), where \( a.H \) denotes the conjugate transpose of \( a \).

Finally, it is emphasized that \( v \) consists of the right (as in right-hand side) eigenvectors of \( a \). A vector \( y \) satisfying \( \text{dot}(y, a) = z \times y \) for some number \( z \) is called a left eigenvector of \( a \), and, in general, the left and right eigenvectors of a matrix are not necessarily the (perhaps conjugate) transposes of each other.

References


Examples

```python
>>> from numpy import linalg as LA

(Almost) trivial example with real e-values and e-vectors.

```python
>>> w, v = LA.eig(np.diag((1, 2, 3)))
```python
>>> w; v
```
array([ 1., 2., 3.])
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

Real matrix possessing complex e-values and e-vectors; note that the e-values are complex conjugates of each other.

```python
>>> w, v = LA.eig(np.array([[1, -1], [-1j, 1]]))
```python
>>> w; v
```
array([[ 1. + 1.j , 1. - 1.j ]],
      [ 0.70710678+0.j , 0.70710678+0.j ])
array([0.00000000-0.70710678j, 0.00000000+0.70710678j])
```

Complex-valued matrix with real e-values (but complex-valued e-vectors); note that a.conj().T = a, i.e., a is Hermitian.

```python
>>> a = np.array([[1, 1j], [-1j, 1]])
```python
>>> w, v = LA.eig(a)
```python
>>> w; v
```
array([ 2.00000000e+00+0.j , 5.98651912e-36+0.j ]) # i.e., {2, 0}
```
Be careful about round-off error!

```python
>>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
>>> # Theor. e-values are 1 +/- 1e-9
>>> w, v = LA.eig(a)
>>> w; v
array([ 1., 1.])
array([[ 1., 0.],
       [ 0., 1.]])
```

```python
numpy.linalg.eigh(a, UPLO='L')
```

Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of `a`, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

**Parameters**

- `A`: (..., M, M) array
  - Hermitian/Symmetric matrices whose eigenvalues and eigenvectors are to be computed.

- `UPLO`: {'L', 'U'}, optional
  - Specifies whether the calculation is done with the lower triangular part of `a` (‘L’, default) or the upper triangular part (‘U’).

**Returns**

- `w`: (..., M) ndarray
  - The eigenvalues, not necessarily ordered.

- `v`: {(..., M, M) ndarray, (..., M, M) matrix}
  - The column `v[:, i]` is the normalized eigenvector corresponding to the eigenvalue `w[i]`. Will return a matrix object if `a` is a matrix object.

**Raises**

- `LinAlgError`
  - If the eigenvalue computation does not converge.

**See Also:**

- `eigvalsh`
  - Eigenvalues of symmetric or Hermitian arrays.

- `eig`
  - Eigenvalues and right eigenvectors for non-symmetric arrays.

- `eigvals`
  - Eigenvalues of non-symmetric arrays.

**Notes**

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

The eigenvalues/eigenvectors are computed using LAPACK routines `ssyevd`, `heevd`

The eigenvalues of real symmetric or complex Hermitian matrices are always real. [R38] The array `v` of (column) eigenvectors is unitary and `a`, `w`, and `v` satisfy the equations `dot(a, v[:, i]) = w[i] * v[:, i]`.
References

[R38]

Examples

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> a
array([[ 1.+0.j, 0.-2.j],
       [ 0.+2.j, 5.+0.j]])
>>> w, v = LA.eigh(a)
>>> w; v
array([ 0.17157288, 5.82842712]),
       array([[ -0.92387953+0.j , -0.38268343+0.j ],
               [ 0.00000000+0.38268343j, 0.00000000-0.92387953j]])

>>> np.dot(a, v[:, 0]) - w[0] * v[:, 0] # verify 1st e-val/vec pair
array([ 2.77555756e-17 + 0.j, 0. + 1.38777878e-16j])
>>> np.dot(a, v[:, 1]) - w[1] * v[:, 1] # verify 2nd e-val/vec pair
array([ 0.+0.j, 0.+0.j])

>>> A = np.matrix(a) # what happens if input is a matrix object
>>> A
matrix([[ 1.+0.j, 0.-2.j],
       [ 0.+2.j, 5.+0.j]])
>>> w, v = LA.eigh(A)
>>> w; v
array([ 0.17157288, 5.82842712]),
       matrix([[ -0.92387953+0.j , -0.38268343+0.j ],
               [ 0.00000000+0.38268343j, 0.00000000-0.92387953j]])
```

numpy.linalg.eigvals(a)

Compute the eigenvalues of a general matrix.

Main difference between eigvals and eig: the eigenvectors aren’t returned.

Parameters

- `a` : (..., M, M) array_like
  A complex- or real-valued matrix whose eigenvalues will be computed.

Returns

- `w` : (..., M) ndarray
  The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

Raises

- LinAlgError
  If the eigenvalue computation does not converge.

See Also:

- eig
  eigenvalues and right eigenvectors of general arrays

- eigvalsh
  eigenvalues of symmetric or Hermitian arrays.
**eigh**

eigenvalues and eigenvectors of symmetric/Hermitian arrays.

**Notes**

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

This is implemented using the `_geev` LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

**Examples**

Illustration, using the fact that the eigenvalues of a diagonal matrix are its diagonal elements, that multiplying a matrix on the left by an orthogonal matrix, $Q$, and on the right by $Q^T$ (the transpose of $Q$), preserves the eigenvalues of the “middle” matrix. In other words, if $Q$ is orthogonal, then $Q \cdot A \cdot Q^T$ has the same eigenvalues as $A$:

```python
>>> from numpy import linalg as LA
>>> x = np.random.random()
>>> Q = np.array([[np.cos(x), -np.sin(x)], [np.sin(x), np.cos(x)]])
>>> LA.norm(Q[0, :]), LA.norm(Q[1, :]), np.dot(Q[0, :],Q[1, :])
(1.0, 1.0, 0.0)

Now multiply a diagonal matrix by $Q$ on one side and by $Q^T$ on the other:

```python
>>> D = np.diag((-1,1))
>>> LA.eigvals(D)
array([-1., 1.])
>>> A = np.dot(Q, D)
>>> A = np.dot(A, Q.T)
>>> LA.eigvals(A)
array([ 1., -1.])
```

**numpy.linalg.eigvalsh**($a$, `UPLO`='L')

Compute the eigenvalues of a Hermitian or real symmetric matrix.

Main difference from `eigh`: the eigenvectors are not computed.

**Parameters**

- $a$ : (..., M, M) array_like
  A complex- or real-valued matrix whose eigenvalues are to be computed.

- `UPLO` : {'L', 'U'}, optional
  Same as `lower`, with 'L' for lower and 'U' for upper triangular. Deprecated.

**Returns**

- $w$ : (..., M,) ndarray
  The eigenvalues, not necessarily ordered, each repeated according to its multiplicity.

**Raises**

- `LinAlgError`
  If the eigenvalue computation does not converge.

**See Also:**

- `eigh`
  eigenvalues and eigenvectors of symmetric/Hermitian arrays.
**eigvals**

eigenvalues of general real or complex arrays.

**eig**

eigenvalues and right eigenvectors of general real or complex arrays.

**Notes**

Broadcasting rules apply, see the numpy.linalg documentation for details.

The eigenvalues are computed using LAPACK routines _ssyevd, _heevd

**Examples**

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> LA.eigvalsh(a)
array([ 0.17157288+0.j, 5.82842712+0.j])
```

### 3.17.4 Norms and other numbers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linalg.norm(x, ord, axis)</td>
<td>Matrix or vector norm.</td>
</tr>
<tr>
<td>linalg.cond(x, p)</td>
<td>Compute the condition number of a matrix.</td>
</tr>
<tr>
<td>linalg.det(a)</td>
<td>Compute the determinant of an array.</td>
</tr>
<tr>
<td>linalg.slogdet(a)</td>
<td>Compute the sign and (natural) logarithm of the determinant of an array.</td>
</tr>
<tr>
<td>trace(a[, offset, axis1, axis2, dtype, out])</td>
<td>Return the sum along diagonals of the array.</td>
</tr>
</tbody>
</table>

**numpy.linalg.norm(x, ord=None, axis=None)**

Matrix or vector norm.

This function is able to return one of seven different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the ord parameter.

**Parameters**

- **x** : array_like
  
  Input array. If axis is None, x must be 1-D or 2-D.

- **ord** : {non-zero int, inf, -inf, ‘fro’}, optional
  
  Order of the norm (see table under Notes). inf means numpy’s inf object.

- **axis** : {int, 2-tuple of ints, None}, optional
  
  If axis is an integer, it specifies the axis of x along which to compute the vector norms. If axis is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If axis is None then either a vector norm (when x is 1-D) or a matrix norm (when x is 2-D) is returned.

**Returns**

- **n** : float or ndarray
  
  Norm of the matrix or vector(s).

**Notes**

For values of ord <= 0, the result is, strictly speaking, not a mathematical ‘norm’, but it may still be useful for various numerical purposes.
The following norms can be calculated:

<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Frobenius norm</td>
<td>2-norm</td>
</tr>
<tr>
<td>‘fro’</td>
<td>Frobenius norm</td>
<td>–</td>
</tr>
<tr>
<td>inf</td>
<td>max(sum(abs(x), axis=1))</td>
<td>max(abs(x))</td>
</tr>
<tr>
<td>-inf</td>
<td>min(sum(abs(x), axis=1))</td>
<td>min(abs(x))</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>sum(x != 0)</td>
</tr>
<tr>
<td>1</td>
<td>max(sum(abs(x), axis=0))</td>
<td>as below</td>
</tr>
<tr>
<td>-1</td>
<td>min(sum(abs(x), axis=0))</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>2-norm (largest sing. value)</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other</td>
<td>–</td>
<td>sum(abs(x)<strong>ord)</strong>(1./ord)</td>
</tr>
</tbody>
</table>

The Frobenius norm is given by [R39]:

\[ \|A\|_F = \left[ \sum_{i,j} |a_{i,j}|^2 \right]^{1/2} \]

**References**

[R39]

**Examples**

```python
>>> from numpy import linalg as LA
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> b = a.reshape((3, 3))
>>> b
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])

>>> LA.norm(a)
7.745966692414834
>>> LA.norm(b)
7.745966692414834
>>> LA.norm(b, ‘fro’)
7.745966692414834
>>> LA.norm(a, np.inf)
4
>>> LA.norm(b, np.inf)
9
>>> LA.norm(a, -np.inf)
0
>>> LA.norm(b, -np.inf)
2
>>> LA.norm(a, 1)
20
>>> LA.norm(b, 1)
7
>>> LA.norm(a, -1)
-4.656128774142013e-010
>>> LA.norm(b, -1)
6
>>> LA.norm(a, 2)
7.745966692414834
```
>>> LA.norm(b, 2)
7.3484692283495345

>>> LA.norm(a, -2)
nan

>>> LA.norm(b, -2)
1.8570331885190563e-016

>>> LA.norm(a, 3)
5.8480354764257312

>>> LA.norm(a, -3)
nan

Using the \texttt{axis} argument to compute vector norms:

```python
>>> c = np.array(
    [[ 1, 2, 3],
     [-1, 1, 4]])
```

```python
>>> LA.norm(c, axis=0)
array([ 1.41421356,  2.23606798,  5.  ])

>>> LA.norm(c, axis=1)
array([ 3.74165739,  4.24264069])

>>> LA.norm(c, ord=1, axis=1)
array([6, 6])
```

Using the \texttt{axis} argument to compute matrix norms:

```python
>>> m = np.arange(8).reshape(2,2,2)
>>> LA.norm(m, axis=(1,2))
array([ 3.74165739, 11.22497216])
```

```python
>>> LA.norm(m[0, :, :], LA.norm(m[1, :, :])
(3.7416573867739413, 11.224972160321824)
```

\texttt{numpy.linalg.cond}(x, p=None)

\textbf{Compute the condition number of a matrix.}

This function is capable of returning the condition number using one of seven different norms, depending on the value of \textit{p} (see Parameters below).

\textbf{Parameters}

\begin{itemize}
    \item \textbf{x} : (M, N) array_like
        The matrix whose condition number is sought.
    \item \textbf{p} : {None, 1, -1, 2, -2, \textit{inf}, -\textit{inf}, \textquote{fro'}}, optional
        Order of the norm:

        \begin{tabular}{|c|c|}
        \hline
        \textbf{p} & \textbf{norm for matrices} \\
        \hline
        None & 2-norm, computed directly using the SVD \\
        \textquote{fro'} & Frobenius norm \\
        \textit{inf} & max(sum(abs(x), axis=1)) \\
        -\textit{inf} & min(sum(abs(x), axis=1)) \\
        1 & max(sum(abs(x), axis=0)) \\
        -1 & min(sum(abs(x), axis=0)) \\
        2 & 2-norm (largest sing. value) \\
        -2 & smallest singular value \\
        \hline
        \end{tabular}

\end{itemize}

\textit{inf} means the \texttt{numpy.inf} object, and the Frobenius norm is the root-of-sum-of-squares norm.

\textbf{Returns}

\begin{itemize}
    \item \textbf{c} : \{float, inf\}
\end{itemize}
The condition number of the matrix. May be infinite.

See Also:

numpy.linalg.norm

Notes

The condition number of \( x \) is defined as the norm of \( x \) times the norm of the inverse of \( x \) [R37]; the norm can be the usual L2-norm (root-of-sum-of-squares) or one of a number of other matrix norms.

References

[R37]

Examples

```python
>>> from numpy import linalg as LA
>>> a = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])
>>> a
array([[ 1, 0, -1],
       [ 0, 1, 0],
       [ 1, 0, 1]])
>>> LA.cond(a)
1.4142135623730951
>>> LA.cond(a, 'fro')
3.1622776601683795
>>> LA.cond(a, np.inf)
2.0
>>> LA.cond(a, -np.inf)
1.0
>>> LA.cond(a, 1)
2.0
>>> LA.cond(a, -1)
1.0
>>> LA.cond(a, 2)
1.4142135623730951
>>> LA.cond(a, -2)
0.70710678118654746
>>> min(LA.svd(a, compute_uv=0))*min(LA.svd(LA.inv(a), compute_uv=0))
0.70710678118654746
```

numpy.linalg.det(a)

Compute the determinant of an array.

Parameters

- **a**: (..., M, M) array_like
  
  Input array to compute determinants for.

Returns

- **det**: (...) array_like
  
  Determinant of \( a \).

See Also:

slogdet

Another way to representing the determinant, more suitable for large matrices where underflow/overflow may occur.
Notes

Broadcasting rules apply, see the numpy.linalg documentation for details.

The determinant is computed via LU factorization using the LAPACK routine z/dgetrf.

Examples

The determinant of a 2-D array [[a, b], [c, d]] is ad - bc:

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0
```

Computing determinants for a stack of matrices:

```python
>>> a = np.array([[[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]]])
>>> a.shape
(2, 2, 2)
>>> np.linalg.det(a)
array([-2., -3., -8.])
```

**numpy.linalg.slogdet(a)**

Compute the sign and (natural) logarithm of the determinant of an array.

If an array has a very small or very large determinant, than a call to det may overflow or underflow. This routine is more robust against such issues, because it computes the logarithm of the determinant rather than the determinant itself.

**Parameters**

- `a`: (..., M, M) array_like
  
  Input array, has to be a square 2-D array.

**Returns**

- `sign`: (...) array_like

  A number representing the sign of the determinant. For a real matrix, this is 1, 0, or -1. For a complex matrix, this is a complex number with absolute value 1 (i.e., it is on the unit circle), or else 0.

- `logdet`: (...) array_like

  The natural log of the absolute value of the determinant.

  If the determinant is zero, then `sign` will be 0 and `logdet` will be -Inf. In all cases, the determinant is equal to `sign * np.exp(logdet)`.

**See Also:**

- `det`

**Notes**

Broadcasting rules apply, see the numpy.linalg documentation for details.

The determinant is computed via LU factorization using the LAPACK routine z/dgetrf. New in version 1.6.0.

**Examples**

The determinant of a 2-D array [[a, b], [c, d]] is ad - bc:
```python
code:language=python
>>> a = np.array([[1, 2], [3, 4]])
>>> (sign, logdet) = np.linalg.slogdet(a)
>>> (sign, logdet)
(-1, 0.69314718055994529)
```
Returns

\textbf{sum\_along\_diagonals} : ndarray

If \( a \) is 2-D, the sum along the diagonal is returned. If \( a \) has larger dimensions, then an array of sums along diagonals is returned.

\textbf{See Also:}

diag, diagonal, diagflat

\textbf{Examples}

>>> np.trace(np.eye(3))
3.0

array([0, 1, 2])

>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)

3.17.5 Solving equations and inverting matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{linalg.solve}(a, b)</td>
<td>Solve a linear matrix equation, or system of linear scalar equations.</td>
</tr>
<tr>
<td>\texttt{linalg.tensorsolve}(a, b[, axes])</td>
<td>Solve the tensor equation ( a x = b ) for ( x ).</td>
</tr>
<tr>
<td>\texttt{linalg.lstsq}(a, b[, rcond])</td>
<td>Return the least-squares solution to a linear matrix equation.</td>
</tr>
<tr>
<td>\texttt{linalg.inv}(a)</td>
<td>Compute the (multiplicative) inverse of a matrix.</td>
</tr>
<tr>
<td>\texttt{linalg.pinv}(a[, rcond])</td>
<td>Compute the (Moore-Penrose) pseudo-inverse of a matrix.</td>
</tr>
<tr>
<td>\texttt{linalg.tensorinv}(a[, ind])</td>
<td>Compute the ‘inverse’ of an N-dimensional array.</td>
</tr>
</tbody>
</table>

\texttt{numpy.linalg.solve}(a, b)

Solve a linear matrix equation, or system of linear scalar equations.

Computes the “exact” solution, \( x \), of the well-determined, i.e., full rank, linear matrix equation \( ax = b \).

\textbf{Parameters}

\begin{itemize}
  \item \texttt{a} : (..., M, M) array_like
  Coefficient matrix.
  \item \texttt{b} : (..., M,), (..., M, K), array_like
  Ordinate or “dependent variable” values.
\end{itemize}

\textbf{Returns}

\begin{itemize}
  \item \texttt{x} : (..., M,), (..., M, K) ndarray
  Solution to the system \( a x = b \). Returned shape is identical to \( b \).
\end{itemize}

\textbf{Notes}

Broadcasting rules apply, see the \texttt{numpy.linalg} documentation for details.
The solutions are computed using LAPACK routine _gesv

\(a\) must be square and of full-rank, i.e., all rows (or, equivalently, columns) must be linearly independent; if either is not true, use \texttt{linalg} for the least-squares best “solution” of the system/equation.

**References**

[R41]

**Examples**

Solve the system of equations \(3 \times x_0 + x_1 = 9\) and \(x_0 + 2 \times x_1 = 8\):

```python
>>> a = np.array([[3, 1], [1, 2]]
>>> b = np.array([9, 8])
>>> x = np.linalg.solve(a, b)
>>> x
array([ 2.,  3.])
```

Check that the solution is correct:

```python
>>> np.allclose(np.dot(a, x), b)
True
```

**numpy.linalg.tensorsolve**(*a*, *b*, *axes=None*)

Solve the tensor equation \(a \times = b\) for \(x\).

It is assumed that all indices of \(x\) are summed over in the product, together with the rightmost indices of \(a\), as is done in, for example, \texttt{tensordot}(a, x, axes=len(b.shape)).

**Parameters**

- **a**: array_like
  
  Coefficient tensor, of shape \(b.shape + Q\). \(Q\), a tuple, equals the shape of that sub-tensor of \(a\) consisting of the appropriate number of its rightmost indices, and must be such that
  
  \(\text{prod}(Q) = \text{prod}(b.shape)\) (in which sense \(a\) is said to be ‘square’).

- **b**: array_like
  
  Right-hand tensor, which can be of any shape.

- **axes**: tuple of ints, optional
  
  Axes in \(a\) to reorder to the right, before inversion. If None (default), no reordering is done.

**Returns**

- **x**: ndarray, shape \(Q\)

**Raises**

- **LinAlgError**
  
  If \(a\) is singular or not ‘square’ (in the above sense).

**See Also:**

- tensordot, tensorinv, einsum

**Examples**
```python
>>> a = np.eye(2*3*4)
>>> a.shape = (2*3, 4, 2, 3, 4)
>>> b = np.random.randn(2*3, 4)
>>> x = np.linalg.tensorsolve(a, b)
>>> x.shape
(2, 3, 4)
>>> np.allclose(np.tensordot(a, x, axes=3), b)
True
```

```python
numpy.linalg.lstsq(a, b, rcond=-1)

Return the least-squares solution to a linear matrix equation.

Solves the equation \( a x = b \) by computing a vector \( x \) that minimizes the Euclidean 2-norm \( \| b - a x \|_2^2 \). The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of \( a \) can be less than, equal to, or greater than its number of linearly independent columns). If \( a \) is square and of full rank, then \( x \) (but for round-off error) is the “exact” solution of the equation.

**Parameters**

- \( a \) : \((M, N)\) array_like
  - “Coefficient” matrix.
- \( b \) : \((M,)\), \((M, K)\) array_like
  - Ordinate or “dependent variable” values. If \( b \) is two-dimensional, the least-squares solution is calculated for each of the \( K \) columns of \( b \).
- \( rcond \) : float, optional
  - Cut-off ratio for small singular values of \( a \). Singular values are set to zero if they are smaller than \( rcond \) times the largest singular value of \( a \).

**Returns**

- \( x \) : \((N,)\), \((N, K)\) ndarray
  - Least-squares solution. If \( b \) is two-dimensional, the solutions are in the \( K \) columns of \( x \).
- residuals : \((0,)\), \((1,)\), \((K,)\) ndarray
  - Sums of residuals; squared Euclidean 2-norm for each column in \( b - a x \). If the rank of \( a \) is \( < N \) or \( > M \), this is an empty array. If \( b \) is 1-dimensional, this is a \((1,)\) shape array. Otherwise the shape is \((K,)\).
- rank : int
  - Rank of matrix \( a \).
- s : \((\min(M, N),)\) ndarray
  - Singular values of \( a \).

**Raises**

- LinAlgError
  - If computation does not converge.

**Notes**

If \( b \) is a matrix, then all array results are returned as matrices.

**Examples**

Fit a line, \( y = mx + c \), through some noisy data-points:
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])

By examining the coefficients, we see that the line should have a gradient of roughly 1 and cut the y-axis at, more or less, -1.

We can rewrite the line equation as \( y = Ap \), where \( A = [[x, 1]] \) and \( p = [[m], [c]] \). Now use \texttt{linalg.lstsq} to solve for \( p \):

```python
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0., 1.],
       [ 1., 1.],
       [ 2., 1.],
       [ 3., 1.]]

>>> m, c = np.linalg.lstsq(A, y)[0]
>>> print m, c
1.0 -0.95
```

Plot the data along with the fitted line:

```python
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', label='Original data', markersize=10)
>>> plt.plot(x, m*x + c, 'r', label='Fitted line')
>>> plt.legend()
>>> plt.show()
```

**numpy.linalg.inv(a)**

Compute the (multiplicative) inverse of a matrix.

Given a square matrix \( a \), return the matrix \( a^{-1} \) satisfying \( \text{dot}(a, a^{-1}) = \text{dot}(a^{-1}, a) = \text{eye}(a.shape[0]) \).

**Parameters**
- \( a \): (..., M, M) array_like
  Matrix to be inverted.

**Returns**
- \( a^{-1} \): (..., M, M) ndarray or matrix
  (Multiplicative) inverse of the matrix \( a \).

**Raises**
- LinAlgError
  If \( a \) is not square or inversion fails.

**Notes**

Broadcasting rules apply, see the numpy.linalg documentation for details.

**Examples**

```python
>>> from numpy.linalg import inv
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> ainv = inv(a)
>>> np.allclose(np.dot(a, ainv), np.eye(2))
True
>>> np.allclose(np.dot(ainv, a), np.eye(2))
True
```
If `a` is a matrix object, then the return value is a matrix as well:

```python
>>> ainv = inv(np.matrix(a))
>>> ainv
matrix([[[-2. , 1. ],
         [ 1.5, -0.5]]])
```

Inverses of several matrices can be computed at once:

```python
>>> a = np.array([[1., 2.], [3., 4.]], [[1, 3], [3, 5]])
>>> inv(a)
array([[[-2. , 1. ],
        [ 1.5, -0.5]],
       [[-5. , 2. ],
        [ 3. , -1. ]]])
```

```python
numpy.linalg.pinv(a, rcond=1e-15)
```

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all large singular values.

Parameters

- `a`: (M, N) array_like
  Matrix to be pseudo-inverted.

- `rcond`: float
  Cutoff for small singular values. Singular values smaller (in modulus) than `rcond` * largest_singular_value (again, in modulus) are set to zero.

Returns

- `B`: (N, M) ndarray
  The pseudo-inverse of `a`. If `a` is a `matrix` instance, then so is `B`.

Raises

- `LinAlgError`
  If the SVD computation does not converge.

Notes

The pseudo-inverse of a matrix `A`, denoted `A^+`, is defined as: “the matrix that ‘solves’ [the least-squares problem] `Ax = b`,” i.e., if `x` is said solution, then `A^+` is that matrix such that `A^+ x = A^+ b`.

It can be shown that if `Q_1Sigma Q_2^T = A` is the singular value decomposition of `A`, then `A^+ = Q_2Sigma^+ Q_1^T`, where `Q_{1,2}` are orthogonal matrices, `Sigma` is a diagonal matrix consisting of `A`’s so-called singular values, (followed, typically, by zeros), and then `Sigma^+` is simply the diagonal matrix consisting of the reciprocals of `A`’s singular values (again, followed by zeros). \[R40\]

References

\[R40\]

Examples

The following example checks that `a * a+ * a == a` and `a+ * a * a+ == a+`:

```python
>>> a = np.random.randn(3, 6)
>>> B = np.linalg.pinv(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
```
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True

numpy.linalg.tensorinv(a, ind=2)
Compute the ‘inverse’ of an N-dimensional array.

The result is an inverse for $a$ relative to the tensordot operation $\text{tensordot}(a, b, \text{ind})$, i.e., up to floating-point accuracy, $\text{tensordot}(\text{tensorinv}(a), a, \text{ind})$ is the “identity” tensor for the tensordot operation.

Parameters
a : array_like
    Tensor to ‘invert’. Its shape must be ‘square’, i.e., $\text{prod}(a.shape[:\text{ind}]) = \text{prod}(a.shape[\text{ind}:])$.

ind : int, optional
    Number of first indices that are involved in the inverse sum. Must be a positive integer, default is 2.

Returns
b : ndarray
    $a$’s tensordot inverse, shape $a.shape[:\text{ind}] + a.shape[\text{ind}:]$.

Raises
LinAlgError
    If $a$ is singular or not ‘square’ (in the above sense).

See Also:
tensordot, tensorsolve

Examples
>>> a = np.eye(4*6)
>>> a.shape = (4, 6, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=2)
>>> ainv.shape
(8, 3, 4, 6)
>>> b = np.random.randn(4, 6)
>>> np.allclose(np.tensordot(ainv, b), np.linalg.tensorsolve(a, b))
True

>>> a = np.eye(4*6)
>>> a.shape = (24, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=1)
>>> ainv.shape
(8, 3, 24)
>>> b = np.random.randn(24)
>>> np.allclose(np.tensordot(ainv, b, 1), np.linalg.tensorsolve(a, b))
True

3.17.6 Exceptions

linalg.LinAlgError  Generic Python-exception-derived object raised by linalg functions.
exception numpy.linalg.LinAlgError

Generic Python-exception-derived object raised by linalg functions.

General purpose exception class, derived from Python's exception.Exception class, programmatically raised in linalg functions when a Linear Algebra-related condition would prevent further correct execution of the function.

Parameters
    None

Examples

```python
>>> from numpy import linalg as LA
>>> LA.inv(np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...linalg.py", line 350, in inv
    return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
  File "...linalg.py", line 249, in solve
    raise LinAlgError('Singular matrix')
numpy.linalg.LinAlgError: Singular matrix
```

3.17.7 Linear algebra on several matrices at once

Several of the linear algebra routines listed above are able to compute results for several matrices at once, if they are stacked into the same array.

This is indicated in the documentation via input parameter specifications such as `a : (..., M, M)` array_like. This means that if for instance given an input array `a.shape == (N, M, M)`, it is interpreted as a “stack” of `N` matrices, each of size `M`-by-`M`. Similar specification applies to return values, for instance the determinant has `det : (...)` and will in this case return an array of shape `det(a).shape == (N,)`. This generalizes to linear algebra operations on higher-dimensional arrays: the last 1 or 2 dimensions of a multidimensional array are interpreted as vectors or matrices, as appropriate for each operation.

3.18 Logic functions

3.18.1 Truth value testing

```python
all(a[, axis, out, keepdims])  Test whether all array elements along a given axis evaluate to True.
any(a[, axis, out, keepdims])  Test whether any array element along a given axis evaluates to True.
```

numpy.all(a, axis=None, out=None, keepdims=False)

Test whether all array elements along a given axis evaluate to True.

Parameters
    a : array_like
        Input array or object that can be converted to an array.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a logical AND reduction is performed. The default (axis = None) is perform a logical OR over all the dimensions of the input array. axis may be negative, in which case it counts from the last to the first axis. New in version 1.7.0. If
this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

**out**: ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if dtype(out) is float, the result will consist of 0.0’s and 1.0’s). See `doc.ufuncs` (Section “Output arguments”) for more details.

**keepdims**: bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.

**Returns**

**all**: ndarray, bool

A new boolean or array is returned unless `out` is specified, in which case a reference to `out` is returned.

**See Also:**

`ndarray.all`

equivalent method

`any`

Test whether any element along a given axis evaluates to True.

**Notes**

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

**Examples**

```python
>>> np.all([[True,False],[True,True]])
False

>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False], dtype=bool)

>>> np.all([-1, 4, 5])
True

>>> np.all([1.0, np.nan])
True

>>> o=np.array([False])
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array([ True], dtype=bool))
```

`numpy.any (a, axis=None, out=None, keepdims=False)`

Test whether any array element along a given axis evaluates to True.

**Returns**

single boolean unless *axis* is not *None*

**Parameters**

`a`: array_like

Input array or object that can be converted to an array.
**axis**: None or int or tuple of ints, optional

Axis or axes along which a logical OR reduction is performed. The default \((axis = \text{None})\) is perform a logical OR over all the dimensions of the input array. \(axis\) may be negative, in which case it counts from the last to the first axis. New in version 1.7.0. If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

**out**: ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if it is of type float, then it will remain so, returning 1.0 for True and 0.0 for False, regardless of the type of \(a\)). See \texttt{doc.ufuncs} (Section “Output arguments”) for details.

**keepdims**: bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \(arr\).

**Returns**

\texttt{any} : bool or ndarray

A new boolean or \texttt{ndarray} is returned unless \texttt{out} is specified, in which case a reference to \texttt{out} is returned.

**See Also:**

\texttt{ndarray.any}

equivalent method

\texttt{all}

Test whether all elements along a given axis evaluate to True.

**Notes**

Not a Number (NaN), positive infinity and negative infinity evaluate to \textit{True} because these are not equal to zero.

**Examples**

```python
>>> np.any([[True, False], [True, True]])
True

>>> np.any([[True, False], [False, False]], axis=0)
array([True, False], dtype=bool)

>>> np.any([-1, 0, 5])
True

>>> np.any(np.nan)
True

>>> o=np.array([False])
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array([True]), array([True]))

>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o)  # identity of z and o
(191614240, 191614240)
```
### 3.18.2 Array contents

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isfinite(x[, out])</code></td>
<td>Test element-wise for finiteness (not infinity or not Not a Number).</td>
</tr>
<tr>
<td><code>isinf(x[, out])</code></td>
<td>Test element-wise for positive or negative infinity.</td>
</tr>
<tr>
<td><code>isnan(x[, out])</code></td>
<td>Test element-wise for NaN and return result as a boolean array.</td>
</tr>
<tr>
<td><code>isneginf(x[, y])</code></td>
<td>Test element-wise for negative infinity, return result as bool array.</td>
</tr>
<tr>
<td><code>isposinf(x[, y])</code></td>
<td>Test element-wise for positive infinity, return result as bool array.</td>
</tr>
</tbody>
</table>

```python
numpy.isfinite(x[, out]) = <ufunc 'isfinite'>
```

Test element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean array.

**Parameters**

- **x**: array_like
  - Input values.

- **out**: ndarray, optional
  - Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See `doc.ufuncs`.

**Returns**

- **y**: ndarray, bool
  - For scalar input, the result is a new boolean with value True if the input is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).
  - For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is finite; otherwise the values are False (element is either positive infinity, negative infinity or Not a Number).

**See Also:**

`isinf`, `isneginf`, `isposinf`, `isnan`

**Notes**

Not a Number, positive infinity and negative infinity are considered to be non-finite.

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when `x` is a scalar input, or if first and second arguments have different shapes.

**Examples**

```python
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
```
NumPy Reference, Release 1.8.1

```python
>>> np.isfinite([np.log(-1.), 1., np.log(0)])
array([False, True, False], dtype=bool)
```

```python
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isfinite(x, y)
array([0, 1, 0])
```

```python
numpy.isinf(x[, out]) = <ufunc 'isinf'>
```

Test element-wise for positive or negative infinity.

Returns a boolean array of the same shape as `x`, True where `x` == +/-\(\infty\), otherwise False.

Parameters

- `x`: array_like
  - Input values
- `out`: array_like, optional
  - An array with the same shape as `x` to store the result.

Returns

- `y`: bool (scalar) or boolean ndarray
  - For scalar input, the result is a new boolean with value True if the input is positive or negative infinity; otherwise the value is False.
  - For array input, the result is a boolean array with the same shape as the input and the values are True where the corresponding element of the input is positive or negative infinity; elsewhere the values are False. If a second argument was supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True, respectively. The return value `y` is then a reference to that array.

See Also:
isneginf, isposinf, isnan, isfinite

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.

Examples

```python
>>> np.isinf(np.inf)
True
>>> np.isinf(np.nan)
False
>>> np.isinf(np.NINF)
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])
array([ True,  True, False, False], dtype=bool)
```

```python
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isinf(x, y)
```
array([[1, 0, 1]])

>>> y
array([[1, 0, 1]])

np.isnan(x[, out]) = <ufunc 'isnan'>
Test element-wise for NaN and return result as a boolean array.

Parameters
   x : array_like
       Input array.

Returns
   y : {ndarray, bool}
       For scalar input, the result is a new boolean with value True if the input is NaN; otherwise the value is False.

       For array input, the result is a boolean array of the same dimensions as the input and the values are True if the corresponding element of the input is NaN; otherwise the values are False.

See Also:
   isinf, isneginf, isposinf, isfinite

Notes
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

>>> np.isnan(np.nan)
True

>>> np.isnan(np.inf)
False

>>> np.isnan([np.log(-1.),1.,np.log(0)])
array([ True, False, False], dtype=bool)

np.isneginf(x, y=None)
Test element-wise for negative infinity, return result as bool array.

Parameters
   x : array_like
       The input array.

   y : array_like, optional
       A boolean array with the same shape and type as x to store the result.

Returns
   y : ndarray
       A boolean array with the same dimensions as the input. If second argument is not supplied then a numpy boolean array is returned with values True where the corresponding element of the input is negative infinity and values False where the element of the input is not negative infinity.

       If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value y is then a reference to that array.
See Also:

isinf, isneginf, isnan, isfinite

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when x is a scalar input, or if first and second arguments have different shapes.

Examples

```python
>>> np.isneginf(np.NINF)
array(True, dtype=bool)
>>> np.isneginf(np.inf)
array(False, dtype=bool)
>>> np.isneginf(np.PINF)
array(False, dtype=bool)
>>> np.isneginf([-np.inf, 0., np.inf])
array([ True, False, False], dtype=bool)
```

```python
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isneginf(x, y)
array([1, 0, 0])
```

```python
numpy.isposinf(x, y=None)
Test element-wise for positive infinity, return result as bool array.

Parameters

- **x**: array_like
  The input array.
- **y**: array_like, optional
  A boolean array with the same shape as x to store the result.

Returns

- **y**: ndarray
  A boolean array with the same dimensions as the input. If second argument is not supplied then a boolean array is returned with values True where the corresponding element of the input is positive infinity and values False where the element of the input is not positive infinity.

  If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value y is then a reference to that array.

See Also:

isinf, isneginf, isfinite, isnan

Notes

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when x is a scalar input, or if first and second arguments have different shapes.
Examples

```python
>>> np.isposinf(np.PINF)
array(True, dtype=bool)
>>> np.isposinf(np.inf)
array(True, dtype=bool)
>>> np.isposinf(np.NINF)
array(False, dtype=bool)
>>> np.isposinf([-np.inf, 0., np.inf])
array([False, False, True], dtype=bool)

>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isposinf(x, y)
array([0, 0, 1])
>>> y
array([0, 0, 1])
```

3.18.3 Array type testing

```latex
\begin{verbatim}

numpy.iscomplex(x)
Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

Parameters
  x : array_like
     Input array.

Returns
  out : ndarray of bools
     Output array.

See Also:

isreal

iscomplexobj

Return True if x is a complex type or an array of complex numbers.

Examples

>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([ True, False, False, False, False, True], dtype=bool)

numpy.iscomplexobj(x)
Check for a complex type or an array of complex numbers.

\end{verbatim}
```

3.18. Logic functions
The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, iscomplexobj evaluates to True.

Parameters

- **x**: any
  
The input can be of any type and shape.

Returns

- **iscomplexobj**: bool
  
The return value, True if x is of a complex type or has at least one complex element.

See Also:

isrealobj, iscomplex

Examples

>>> np.iscomplexobj(1)
False
>>> np.iscomplexobj(1+0j)
True
>>> np.iscomplexobj([3, 1+0j, True])
True

numpy.isfortran(a)

Returns True if array is arranged in Fortran-order in memory and not C-order.

Parameters

- **a**: ndarray
  
Input array.

Examples

np.array allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False

>>> b = np.array([[1, 2, 3], [4, 5, 6]], order='FORTRAN')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(b)
True

The transpose of a C-ordered array is a FORTRAN-ordered array.

>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False
>>> b = a.T

>>>
>>> b
array([[1, 4],
       [2, 5],
       [3, 6]])

>>> np.isfortran(b)
True

C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.

>>> np.isfortran(np.array([1, 2], order='FORTRAN'))
False

numpy.isreal(x)
Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

Parameters
  x : array_like
     Input array.

Returns
  out : ndarray, bool
     Boolean array of same shape as x.

See Also:
  iscomplex
  isrealobj
  Return True if x is not a complex type.

Examples

>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([False, True, True, True, True, False], dtype=bool)

numpy.isrealobj(x)
Return True if x is a not complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, isrealobj evaluates to False if the data type is complex.

Parameters
  x : any

The input can be of any type and shape.

Returns
  y : bool
     The return value, False if x is of a complex type.

See Also:
  iscomplexobj, isreal

Examples

>>> np.isrealobj(1)
True
>>> np.isrealobj(1+0j)
False
>>> np.isrealobj([3, 1+0j, True])
False

call
numpy.isrealobj(num)
Returns True if the type of num is a scalar type.

Parameters
num : any
Input argument, can be of any type and shape.

Returns
val : bool
True if num is a scalar type, False if it is not.

Examples
>>> np.isrealobj(3.1)
True
>>> np.isrealobj([3.1])
False
>>> np.isrealobj(False)
True

3.18.4 Logical operations

numpy.logical_and(x1, x2[, out])  Compute the truth value of x1 AND x2 element-wise.

Parameters
x1, x2 : array_like
Input arrays. x1 and x2 must be of the same shape.

Returns
y : {ndarray, bool}
Boolean result with the same shape as x1 and x2 of the logical AND operation on corresponding elements of x1 and x2.

See Also:
logical_or, logical_not, logical_xor, bitwise_and

Examples
>>> np.logical_and(True, False)
False
>>> np.logical_and([True, False], [False, False])
array([False, False], dtype=bool)
>>> x = np.arange(5)
>>> np.logical_and(x>1, x<4)
array([False, False,  True,  True, False], dtype=bool)

numpy.logical_or(x1, x2[, out]) = <ufunc ‘logical_or’>
Compute the truth value of x1 OR x2 element-wise.

Parameters
x1, x2 : array_like
Logical OR is applied to the elements of x1 and x2. They have to be of the same shape.

Returns
y : {ndarray, bool}
Boolean result with the same shape as x1 and x2 of the logical OR operation on elements of x1 and x2.

See Also:
logical_and, logical_not, logical_xor, bitwise_or

Examples
>>> np.logical_or(True, False)
True
>>> np.logical_or([[True, False], [False, False]])
array([[ True, False],
       [False, False]], dtype=bool)

>>> x = np.arange(5)
>>> np.logical_or(x < 1, x > 3)
array([False, False, False, False,  True], dtype=bool)

numpy.logical_not(x[, out]) = <ufunc ‘logical_not’>
Compute the truth value of NOT x element-wise.

Parameters
x : array_like
Logical NOT is applied to the elements of x.

Returns
y : bool or ndarray of bool
Boolean result with the same shape as x of the NOT operation on elements of x.

See Also:
logical_and, logical_or, logical_xor

Examples
>>> np.logical_not(3)
False
>>> np.logical_not([True, False, 0, 1])
array([False, True, False, False], dtype=bool)

>>> x = np.arange(5)
>>> np.logical_not(x<3)
array([False, False, False,  True,  True], dtype=bool)

numpy.logical_xor(x1, x2[, out]) = <ufunc ‘logical_xor’>
Compute the truth value of x1 XOR x2, element-wise.
Parameters

x1, x2 : array_like

Logical XOR is applied to the elements of x1 and x2. They must be broadcastable to
the same shape.

Returns

y : bool or ndarray of bool

Boolean result of the logical XOR operation applied to the elements of x1 and x2; the
shape is determined by whether or not broadcasting of one or both arrays was required.

See Also:

logical_and, logical_or, logical_not, bitwise_xor

Examples

>>> np.logical_xor(True, False)
True

>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False, True, True, False], dtype=bool)

>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([ True, False, False, False, True], dtype=bool)

Simple example showing support of broadcasting

>>> np.logical_xor(0, np.eye(2))
array([[ True, False],
       [False, True]], dtype=bool)

3.18.5 Comparison

allclose(a, b[, rtol, atol]) Returns True if two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (rtol * abs(b)) and the
absolute difference atol are added together to compare against the absolute difference between
a and b.

If either array contains one or more NaNs, False is returned. Infs are treated as equal if they are in the same
place and of the same sign in both arrays.

Parameters

a, b : array_like

Input arrays to compare.

rtol : float

The relative tolerance parameter (see Notes).

atol : float

numpy.allclose (a, b, rtol=1e-05, atol=1e-08)

Returns True if two arrays are element-wise equal within a tolerance.

array_equal(a1, a2) True if two arrays have the same shape and elements, False otherwise.

array_equiv(a1, a2) Returns True if input arrays are shape consistent and all elements equal.
The absolute tolerance parameter (see Notes).

Returns

\texttt{allclose} : bool

Returns True if the two arrays are equal within the given tolerance; False otherwise.

See Also:

\texttt{isclose}, \texttt{all}, \texttt{any}

Notes

If the following equation is element-wise True, then \texttt{allclose} returns True.

\[
\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} \times \text{absolute}(b))
\]

The above equation is not symmetric in \(a\) and \(b\), so that \texttt{allclose}(a, b) might be different from \texttt{allclose}(b, a) in some rare cases.

Examples

```python
>>> np.allclose([1e10, 1e-7], [1.00001e10, 1e-8])
False
>>> np.allclose([1e10, 1e-8], [1.0001e10, 1e-9])
True
>>> np.allclose([1e10, 1e-8], [1.00001e10, 1e-9])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan])
False
```

\texttt{numpy.isclose}(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)

Returns a boolean array where two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference \((\text{rtol} \times \text{abs}(b))\) and the absolute difference \(\text{atol}\) are added together to compare against the absolute difference between \(a\) and \(b\).

Parameters

\(a, b\) : array_like

Input arrays to compare.

\texttt{rtol} : float

The relative tolerance parameter (see Notes).

\texttt{atol} : float

The absolute tolerance parameter (see Notes).

\texttt{equal_nan} : bool

Whether to compare NaN’s as equal. If True, NaN’s in \(a\) will be considered equal to NaN’s in \(b\) in the output array.

Returns

\(y\) : array_like

Returns a boolean array of where \(a\) and \(b\) are equal within the given tolerance. If both \(a\) and \(b\) are scalars, returns a single boolean value.

See Also:

\texttt{allclose}
Notes

New in version 1.7.0. For finite values, isclose uses the following equation to test whether two floating point values are equivalent.

$$\text{absolute}(a - b) <= (\text{atol} + \text{rtol} \times \text{absolute}(b))$$

The above equation is not symmetric in $a$ and $b$, so that $\text{isclose}(a, b)$ might be different from $\text{isclose}(b, a)$ in some rare cases.

Examples

```python
>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])
array([True, False])
>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])
array([True, True])
>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])
array([False, True])
>>> np.isclose([1.0, np.nan], [1.0, np.nan])
array([True, False])
>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
array([True, True])
```

numpy.array_equal($a1, a2$)

True if two arrays have the same shape and elements, False otherwise.

Parameters

- $a1, a2$: array_like
  
  Input arrays.

Returns

- $b$: bool
  
  Returns True if the arrays are equal.

See Also:

allclose

Returns True if two arrays are element-wise equal within a tolerance.

array_equiv

Returns True if input arrays are shape consistent and all elements equal.

Examples

```python
>>> np.array_equal([[1, 2], [1, 2]])
True
>>> np.array_equal(np.array([[1, 2]]), np.array([[1, 2]]))
True
>>> np.array_equal([[1, 2], [1, 2, 3]])
False
>>> np.array_equal([[1, 2], [1, 4]])
False
```

numpy.array_equiv($a1, a2$)

Returns True if input arrays are shape consistent and all elements equal.

Shape consistent means they are either the same shape, or one input array can be broadcasted to create the same shape as the other one.
Parameters

- `a1, a2`: array_like
  - Input arrays.

Returns

- `out`: bool
  - True if equivalent, False otherwise.

Examples

```python
>>> np.array_equiv([1, 2], [1, 2])
True
>>> np.array_equiv([1, 2], [1, 3])
False

Showing the shape equivalence:

```python
>>> np.array_equiv([1, 2], [[1, 2], [1, 2]])
True
>>> np.array_equiv([1, 2], [[1, 2, 1, 2], [1, 2, 1, 2]])
False
>>> np.array_equiv([1, 2], [[1, 2], [1, 3]])
False
```

```
greater(x1, x2[, out])

Return the truth value of (x1 > x2) element-wise.
```

```
greater_equal(x1, x2[, out])

Return the truth value of (x1 >= x2) element-wise.
```

```
less(x1, x2[, out])

Return the truth value of (x1 < x2) element-wise.
```

```
less_equal(x1, x2[, out])

Return the truth value of (x1 <= x2) element-wise.
```

```
equal(x1, x2[, out])

Return (x1 == x2) element-wise.
```

```
not_equal(x1, x2[, out])

Return (x1 != x2) element-wise.
```

numpy.greater(x1, x2[, out]) = <ufunc ‘greater’>

Return the truth value of (x1 > x2) element-wise.

Parameters

- `x1, x2`: array_like
  - Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

Returns

- `out`: bool or ndarray of bool
  - Array of bools, or a single bool if `x1` and `x2` are scalars.

See Also:

- `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`

Examples

```python
>>> np.greater([4, 2], [2, 2])
array([ True, False], dtype=bool)
```

If the inputs are ndarrays, then np.greater is equivalent to `>`.  

3.18. Logic functions
```python
>>> a = np.array([4, 2])
>>> b = np.array([2, 2])
>>> a > b
array([ True, False], dtype=bool)
```

The `greater_equal` function in NumPy returns the truth value of \((x1 \geq x2)\) element-wise.

**Parameters**
- **x1, x2**: array_like
  - Input arrays. If \(x1.shape \neq x2.shape\), they must be broadcastable to a common shape (which may be the shape of one or the other).

**Returns**
- **out**: bool or ndarray of bool
  - Array of bools, or a single bool if \(x1\) and \(x2\) are scalars.

**Examples**
```python
>>> np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False], dtype=bool)
```

The `less` function in NumPy returns the truth value of \((x1 < x2)\) element-wise.

**Parameters**
- **x1, x2**: array_like
  - Input arrays. If \(x1.shape \neq x2.shape\), they must be broadcastable to a common shape (which may be the shape of one or the other).

**Returns**
- **out**: bool or ndarray of bool
  - Array of bools, or a single bool if \(x1\) and \(x2\) are scalars.

**Examples**
```python
>>> np.less([1, 2], [2, 2])
array([ True, False], dtype=bool)
```

The `less_equal` function in NumPy returns the truth value of \((x1 \leq x2)\) element-wise.

**Parameters**
- **x1, x2**: array_like
  - Input arrays. If \(x1.shape \neq x2.shape\), they must be broadcastable to a common shape (which may be the shape of one or the other).

**Returns**
- **out**: bool or ndarray of bool
  - Array of bools, or a single bool if \(x1\) and \(x2\) are scalars.
See Also:
greater, less, greater_equal, equal, not_equal

Examples
>>> np.less_equal([4, 2, 1], [2, 2, 2])
array([False, True, True], dtype=bool)

numpy.equal(x1, x2[, out]) = <ufunc 'equal'>
Return (x1 == x2) element-wise.

Parameters
- x1, x2 : array_like
  Input arrays of the same shape.

Returns
- out : {ndarray, bool}
  Output array of bools, or a single bool if x1 and x2 are scalars.

See Also:
not_equal, greater_equal, less_equal, greater, less

Examples
>>> np.equal([0, 1, 3], np.arange(3))
array([ True, True, False], dtype=bool)

What is compared are values, not types. So an int (1) and an array of length one can evaluate as True:

>>> np.equal(1, np.ones(1))
array([ True], dtype=bool)

numpy.not_equal(x1, x2[, out]) = <ufunc 'not_equal'>
Return (x1 != x2) element-wise.

Parameters
- x1, x2 : array_like
  Input arrays.

out : ndarray, optional
  A placeholder the same shape as x1 to store the result. See doc.ufuncs (Section “Output arguments”) for more details.

Returns
- not_equal : ndarray bool, scalar bool
  For each element in x1, x2, return True if x1 is not equal to x2 and False otherwise.

See Also:
equal, greater, greater_equal, less, less_equal

Examples
>>> np.not_equal([1.,2.], [1., 3.])
array([False, True], dtype=bool)

>>> np.not_equal([1, 2], [[1, 3],[1, 4]])
array([[False, True],
        [False, True]], dtype=bool)
3.19 Masked array operations

3.19.1 Constants

```latex
\texttt{ma.MaskType} \quad \text{Numpy’s Boolean type. Character code: \texttt{?}. Alias: \texttt{bool8}}
```

```python
numpy.ma.MaskType
    alias of \texttt{bool8}
```

3.19.2 Creation

From existing data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.masked_array</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.array(data[, dtype, copy, order, mask, ...])</code></td>
<td>An array class with possibly masked values.</td>
</tr>
<tr>
<td><code>ma.copy</code></td>
<td>copy</td>
</tr>
<tr>
<td><code>ma.frombuffer(buffer[, dtype, count, offset])</code></td>
<td>Interpret a buffer as a 1-dimensional array.</td>
</tr>
<tr>
<td><code>ma.fromfunction(function, shape, **kwargs)</code></td>
<td>Construct an array by executing a function over each coordinate.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.copy([order])</code></td>
<td>Return a copy of the array.</td>
</tr>
</tbody>
</table>

```python
numpy.ma.masked_array
    alias of \texttt{MaskedArray}
```

```python
numpy.ma.array (data, dtype=None, copy=False, order=False, mask=False, fill_value=None, keep_mask=True, hard_mask=None, shrink=True, subok=True, ndmin=0)
    An array class with possibly masked values.
```

Masked values of True exclude the corresponding element from any computation.

Construction:

```python
x = MaskedArray(data, mask=nomask, dtype=None,
                copy=False, subok=True, ndmin=0, fill_value=None,
                keep_mask=True, hard_mask=None, shrink=True)
```

**Parameters**

- `data` : array_like
  Input data.
- `mask` : sequence, optional
  Mask. Must be convertible to an array of booleans with the same shape as `data`. True indicates a masked (i.e. invalid) data.
- `dtype` : dtype, optional
  Data type of the output. If `dtype` is None, the type of the data argument (`data.dtype`) is used. If `dtype` is not None and different from `data.dtype`, a copy is performed.
- `copy` : bool, optional
Whether to copy the input data (True), or to use a reference instead. Default is False.

**subok** : bool, optional

Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.

**ndmin** : int, optional

Minimum number of dimensions. Default is 0.

**fill_value** : scalar, optional

Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

**keep_mask** : bool, optional

Whether to combine `mask` with the mask of the input data, if any (True), or to use only `mask` for the output (False). Default is True.

**hard_mask** : bool, optional

Whether to use a hard mask or not. With a hard mask, masked values cannot be un-masked. Default is False.

**shrink** : bool, optional

Whether to force compression of an empty mask. Default is True.

```python
numpy.ma.copy = <numpy.ma.core._frommethod instance at 0x2820290>
copy a.copy(order='C')
```

Return a copy of the array.

**Parameters**

**order** : {'C', 'F', 'A', 'K'}, optional

Controls the memory layout of the copy. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if `a` is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of `a` as closely as possible. (Note that this function and :func:`numpy.copy` are very similar, but have different default values for their order= arguments.)

**Examples**

```python
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```
**NumPy Reference, Release 1.8.1**

```python
numpy.ma.frombuffer (buffer, dtype=float, count=-1, offset=0) = <numpy.ma.core._convert2ma instance at 0x2820dd0>
```

Interpret a buffer as a 1-dimensional array.

**Parameters**

- `buffer`: buffer_like
  - An object that exposes the buffer interface.
- `dtype`: data-type, optional
  - Data-type of the returned array; default: float.
- `count`: int, optional
  - Number of items to read. -1 means all data in the buffer.
- `offset`: int, optional
  - Start reading the buffer from this offset; default: 0.

**Notes**

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```python
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

**Examples**

```python
>>> s = 'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array(['w', 'o', 'r', 'l', 'd'],
      dtype='|S1')
```

```python
numpy.ma.fromfunction (function, shape, **kwargs) = <numpy.ma.core._convert2ma instance at 0x2820e18>
```

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value \( f(x, y, z) \) at coordinate \((x, y, z)\).

**Parameters**

- `function`: callable
  - The function is called with N parameters, where N is the rank of `shape`. Each parameter represents the coordinates of the array varying along a specific axis. For example, if `shape` were \((2, 2)\), then the parameters in turn be \((0, 0), (0, 1), (1, 0), (1, 1)\).
- `shape`: (N,) tuple of ints
  - Shape of the output array, which also determines the shape of the coordinate arrays passed to `function`.
- `dtype`: data-type, optional
  - Data-type of the coordinate arrays passed to `function`. By default, `dtype` is float.

**Returns**

- `fromfunction`: any
The result of the call to function is passed back directly. Therefore the shape of fromfunction is completely determined by function. If function returns a scalar value, the shape of fromfunction would match the shape parameter.

See Also:
indices, meshgrid

Notes
Keywords other than dtype are passed to function.

Examples
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]], dtype=bool)

>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

MaskedArray.copy(order='C')
Return a copy of the array.

Parameters
order : {'C', 'F', 'A', 'K'}, optional
    Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A'
    means 'F' if a is Fortran contiguous, 'C' otherwise. 'K' means match the layout of a as
closely as possible. (Note that this function and :func:numpy.copy are very similar, but
have different default values for their order= arguments.)

See Also:
numpy.copy, numpy.copyto

Examples
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy()
>>> x.fill(0)
>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True

Ones and zeros
numpy.ma.empty(shape, dtype=float, order='C') = <numpy.ma.core._convert2ma instance at 0x2820cf8>
Return a new array of given shape and type, without initializing entries.

Parameters

shape : int or tuple of int
Shape of the empty array
dtype : data-type, optional
   Desired output data-type.
order : {'C', 'F'}, optional
   Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See Also:
empty_like, zeros, ones

Notes
empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

>>> np.empty([2, 2])
array([[ -9.74999359e+001, 6.69583040e-309],
       [ 2.13182611e-314, 3.06959433e-309]])
#random

>>> np.empty([2, 2], dtype=int)
array([[ -1073741821, -1067949133],
       [ 496041986, 19249760]])
#random

numpy.ma.empty_like(a, dtype=None, order='K', subok=True) = <numpy.ma.core._convert2ma instance at 0x2820d88>
Return a new array with the same shape and type as a given array.

Parameters

a : array_like
   The shape and data-type of a define these same attributes of the returned array.
dtype : data-type, optional
   New in version 1.6.0. Overrides the data type of the result.
order : {'C', 'F', 'A', or 'K'}, optional
   New in version 1.6.0. Overrides the memory layout of the result. ‘C’ means C-order, ‘F’ means F-order, ‘A’ means ‘F’ if a is Fortran contiguous, ‘C’ otherwise. ‘K’ means match the layout of a as closely as possible.
subok : bool, optional.

If True, then the newly created array will use the sub-class type of ‘a’, otherwise it will be a base-class array. Defaults to True.

Returns

out : ndarray

Array of uninitialized (arbitrary) data with the same shape and type as a.

See Also:

ones_like
Return an array of ones with shape and type of input.

zeros_like
Return an array of zeros with shape and type of input.

empty
Return a new uninitialized array.

ones
Return a new array setting values to one.

zeros
Return a new array setting values to zero.

Notes

This function does not initialize the returned array; to do that use zeros_like or ones_like instead. It may be marginally faster than the functions that do set the array values.

Examples

```python
>>> a = ([1,2,3], [4,5,6])  # a is array-like
>>> np.empty_like(a)
array([[ 0, 0, 0], #random
       [-1073741821, -1073741821, 3]],
       [-1073741821, 0, -1073741821])
```

.numpy.ma.masked_all(shape, dtype=<type 'float'>)
Empty masked array with all elements masked.

Return an empty masked array of the given shape and dtype, where all the data are masked.

Parameters

shape : tuple
Shape of the required MaskedArray.

dtype : dtype, optional
Data type of the output.

Returns

a : MaskedArray
A masked array with all data masked.

See Also:
**masked_all_like**

Empty masked array modelled on an existing array.

**Examples**

```python
>>> import numpy.ma as ma
>>> ma.masked_all((3, 3))
masked_array(data =
[[- -- --]  
[ -- -- --]
[ -- -- --],
mask =  
[[ True  True]  
[ True  True]  
[ True  True]],
fill_value=1e+20)
```

The `dtype` parameter defines the underlying data type.

```python
>>> a = ma.masked_all((3, 3))
>>> a.dtype
dtype('float64')
>>> a = ma.masked_all((3, 3), dtype=np.int32)
>>> a.dtype
dtype('int32')
```

```python
numpy.ma.masked_all_like(arr)
```

Empty masked array with the properties of an existing array.

Return an empty masked array of the same shape and dtype as the array `arr`, where all the data are masked.

**Parameters**

- `arr` : ndarray
  
  An array describing the shape and dtype of the required MaskedArray.

**Returns**

- `a` : MaskedArray
  
  A masked array with all data masked.

**Raises**

- AttributeError
  
  If `arr` doesn’t have a shape attribute (i.e. not an ndarray)

**See Also**:

**masked_all**

Empty masked array with all elements masked.

**Examples**

```python
>>> import numpy.ma as ma
>>> arr = np.zeros((2, 3), dtype=np.float32)
>>> arr
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
>>> ma.masked_all_like(arr)
masked_array(data =
[[- -- -- --]],
mask =
[[ True  True  True]],
fill_value=1e+20)
```
```python
mask = [[ True  True  True],
        [ True  True  True]],
        fill_value=1e+20)

The dtype of the masked array matches the dtype of `arr`.

```python
>>> arr.dtype
dtype('float32')
```  
```python
>>> ma.masked_all_like(arr).dtype
dtype('float32')
```

```python
numpy.ma.ones(shape, dtype=None, order='C') = <numpy.ma.core._convert2ma instance at 0x2820ef0>
```

Return a new array of given shape and type, filled with ones.

**Parameters**

- `shape` : int or sequence of ints  
  Shape of the new array, e.g., `(2, 3)` or `2`.

- `dtype` : data-type, optional  
  The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

- `order` : {'C', 'F'}, optional  
  Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

- `out` : ndarray  
  Array of ones with the given shape, dtype, and order.

**See Also:**

- `zeros`, `ones_like`

**Examples**

```python
>>> np.ones(5)
array([ 1., 1., 1., 1., 1.])
```  
```python
>>> np.ones((5,), dtype=np.int)
array([1, 1, 1, 1, 1])
```  
```python
>>> np.ones((2, 1))
array([[ 1.],
       [ 1.]])
```  
```python
>>> s = (2,2)
>>> np.ones(s)
array([[ 1., 1.],
       [ 1., 1.]])
```

```python
numpy.ma.zeros(shape, dtype=float, order='C') = <numpy.ma.core._convert2ma instance at 0x2820f80>
```

Return a new array of given shape and type, filled with zeros.

**Parameters**

- `shape` : int or sequence of ints
Shape of the new array, e.g., (2, 3) or 2.

**dtype** : data-type, optional

The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order** : {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

- **out** : ndarray

Array of zeros with the given shape, dtype, and order.

**See Also:**

- `zeros_like`
  Return an array of zeros with shape and type of input.

- `ones_like`
  Return an array of ones with shape and type of input.

- `empty_like`
  Return an empty array with shape and type of input.

- `ones`
  Return a new array setting values to one.

- `empty`
  Return a new uninitialized array.

**Examples**

```python
>>> np.zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> np.zeros((5,), dtype=numpy.int)
array([0, 0, 0, 0, 0])

>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])

>>> s = (2,2)
>>> np.zeros(s)
array([[ 0., 0.],
       [ 0., 0.]])

>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
       dtype=[('x', '<i4'), ('y', '<i4')])
```

### 3.19.3 Inspecting the array

- `ma.all(self[, axis, out])` Check if all of the elements of `a` are true.

Continued on next page
Table 3.87 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.any(self[, axis, out])</code></td>
<td>Check if any of the elements of ( a ) are true.</td>
</tr>
<tr>
<td><code>ma.count(a[, axis])</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.count_masked(arr[, axis])</code></td>
<td>Count the number of masked elements along the given axis.</td>
</tr>
<tr>
<td><code>ma.getmask(a)</code></td>
<td>Return the mask of a masked array, or nomask.</td>
</tr>
<tr>
<td><code>ma.getmaskarray(arr)</code></td>
<td>Return the mask of a masked array, or full boolean array of False.</td>
</tr>
<tr>
<td><code>ma.getdata(a[, subok])</code></td>
<td>Return the data of a masked array as an ndarray.</td>
</tr>
<tr>
<td><code>ma.nonzero(self)</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.size(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.data</code></td>
<td>Return the current data, as a view of the original</td>
</tr>
<tr>
<td><code>ma.MaskedArray.mask</code></td>
<td>Mask</td>
</tr>
<tr>
<td><code>ma.MaskedArray.recordmask</code></td>
<td>Return the mask of the records.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.all((axis, out))</code></td>
<td>Check if all of the elements of ( a ) are true.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.any((axis, out))</code></td>
<td>Check if any of the elements of ( a ) are true.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.count((axis))</code></td>
<td>Count the non-masked elements of the array along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.nonzero()</code></td>
<td>Return the indices of unmasked elements that are not zero.</td>
</tr>
<tr>
<td><code>ma.shape(obj)</code></td>
<td>Return the shape of an array.</td>
</tr>
<tr>
<td><code>ma.size(obj[, axis])</code></td>
<td>Return the number of elements along a given axis.</td>
</tr>
</tbody>
</table>

**numpy.ma.all (self, axis=None, out=None)** = `<numpy.ma.core._frommethod instance at 0x2819e18>

Check if all of the elements of \( a \) are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

**Parameters**

- **axis**: {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array.

- **out**: {None, array}, optional
  
  Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**See Also:**

- `all`
  
  equivalent function

**Examples**

```python
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

**numpy.ma.any (self, axis=None, out=None)** = `<numpy.ma.core._frommethod instance at 0x2819fc8>

Check if any of the elements of \( a \) are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

**Parameters**

- **axis**: {None, integer}
  
  Axis to perform the operation over. If None, perform over flattened array.
Axis to perform the operation over. If None, perform over flattened array and return a scalar.

def count(a, axis=None):
    
    out : {None, array}, optional
    
    Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

any
equivalent function

numpy.ma.count(arr, axis=None)

Count the non-masked elements of the array along the given axis.

Parameters

axis : int, optional
    Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

Returns

result : int or ndarray
    If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

See Also:

count_masked

Count masked elements in array or along a given axis.

Examples

>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
     [[0 1 2]
      [-- -- --]],
    mask =
     [[False False False]
      [ True  True True]],
    fill_value = 999999)
>>> a.count()
3

When the axis keyword is specified an array of appropriate size is returned.

>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])

numpy.ma.count_masked(arr, axis=None)

Count the number of masked elements along the given axis.

Parameters

arr : array_like
An array with (possibly) masked elements.

**axis** : int, optional

Axis along which to count. If None (default), a flattened version of the array is used.

**Returns**

**count** : int, ndarray

The total number of masked elements (axis=None) or the number of masked elements along each slice of the given axis.

**See Also:**

**MaskedArray.count**

Count non-masked elements.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.arange(9).reshape((3,3))
>>> a = ma.array(a)
>>> a[1, 0] = ma.masked
>>> a[1, 2] = ma.masked
>>> a[2, 1] = ma.masked
>>> a
masked_array(data =
 [[0 1 2]
  [4 -- --]
  [6 -- 8]],
 mask =
 [[False False False]
  [ True False True]
  [False True False]],
 fill_value=999999)
>>> ma.count_masked(a)
3
```

When the **axis** keyword is used an array is returned.

```python
>>> ma.count_masked(a, axis=0)
array([1, 1, 1])
>>> ma.count_masked(a, axis=1)
array([0, 2, 1])
```

**numpy.ma.getmask(a)**

Return the mask of a masked array, or nomask.

Return the mask of a as an ndarray if a is a **MaskedArray** and the mask is not **nomask**, else return **nomask**. To guarantee a full array of booleans of the same shape as a, use **getmaskarray**.

**Parameters**

**a** : array_like

Input **MaskedArray** for which the mask is required.

**See Also:**

**getdata**

Return the data of a masked array as an ndarray.
getmaskarray

Return the mask of a masked array, or full array of False.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1, 2], [3, 4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False  True]
[False False]],
fill_value=999999)
>>> ma.getmask(a)
array([[False,  True],
       [False, False]], dtype=bool)
```

Equivalently use the MaskedArray mask attribute.

```python
>>> a.mask
array([[False,  True],
       [False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1, 2], [3, 4]])
>>> b
masked_array(data =
[[1 2]
[3 4]],
mask =
False,
fill_value=999999)
>>> ma.nomask
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True
```

g{}tempy.ma.getmaskarray(arr)

Return the mask of a masked array, or full boolean array of False.

Return the mask of `arr` as an ndarray if `arr` is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as `arr`.

**Parameters**

- `arr`: array_like

  Input MaskedArray for which the mask is required.

**See Also:**

getmask

Return the mask of a masked array, or nomask.

g{}tempy.ma.getdata

Return the data of a masked array as an ndarray.
Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
```  
```python
>>> a
masked_array(data =
[[1  --]
[3 4 ]],
mask =
[[False  True]
[False False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
[False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
[3 4 ]],
mask = False,
fill_value=999999)
>>> >ma.getmaskarray(b)
array([[False, False],
[False, False]], dtype=bool)
```

numpy.ma.getdata(a, subok=True)
Return the data of a masked array as an ndarray.

Return the data of a (if any) as an ndarray if a is a MaskedArray, else return a as a ndarray or subclass (depending on subok) if not.

**Parameters**

- `a`: array_like
  Input MaskedArray, alternatively a ndarray or a subclass thereof.

- `subok`: bool
  Whether to force the output to be a pure ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

**See Also:**

ggetitem
Return the mask of a masked array, or nomask.

ggetmaskarray
Return the mask of a masked array, or full array of False.

Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
```  
```python
>>> a
masked_array(data =
[[1  --]
[3 4 ]],
mask =
[[False  True]
[False False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
[False, False]], dtype=bool)
```
```
[3 4],
mask =
[[False  True]
 [False False]],
    fill_value=999999)
>>> ma.getdata(a)
array([[1, 2],
      [3, 4]])
```

Equivalently use the MaskedArray data attribute.

```
>>> a.data
array([[1, 2],
      [3, 4]])
```

```
numpy.ma.nonzero(self) = <numpy.ma.core._frommethod instance at 0x28204d0>
```

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

None

**Returns**

```
tuple_of_arrays : tuple
```

Indices of elements that are non-zero.

**See Also:**

```
numpy.nonzero
```

Function operating on ndarrays.

```
flatnonzero
```

Return indices that are non-zero in the flattened version of the input array.

```
ndarray.nonzero
```

Equivalent ndarray method.

```
count_nonzero
```

Counts the number of non-zero elements in the input array.

**Examples**

```
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]],
    mask =
```

...
False,
    fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))

Masked elements are ignored.

>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[1.0 0.0 0.0]
[0.0 -- 0.0]
[0.0 0.0 1.0]],
    mask =
[[False False False]
[False True False]
[False False False]],
    fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))

Indices can also be grouped by element.

>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])

A common use for nonzero is to find the indices of an array, where a condition is True. Given an array a, the condition a > 3 is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the a where the condition is true.

>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
[[False False False]
[ True True True]
[ True True True]],
    mask =
False,
    fill_value=999999)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

The nonzero method of the condition array can also be called.

>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

numpy.ma.shape(obj)
Return the shape of an array.

    Parameters
    a : array_like
        Input array.

    Returns
    shape : tuple of ints
        The elements of the shape tuple give the lengths of the corresponding array dimensions.

3.19. Masked array operations
See Also:
alen

```
ndarray.shape
   Equivalent array method.
```

Examples

```python
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
numpy.ma.size(obj, axis=None)
   Return the number of elements along a given axis.
```

Parameters

```
a : array_like
   Input data.
axis : int, optional
   Axis along which the elements are counted. By default, give the total number of ele-
   ments.
```

Returns

```
element_count : int
   Number of elements along the specified axis.
```

See Also:

```
shape
dimensions of array
```

```
ndarray.shape
dimensions of array
```

```
ndarray.size
   number of elements in array
```

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.size(a)
(2,)
>>> a.shape
(2,)
```
MaskedArray.data
Return the current data, as a view of the original underlying data.

MaskedArray.mask
Mask

MaskedArray.recordmask
Return the mask of the records. A record is masked when all the fields are masked.

MaskedArray.all(axis=None, out=None)
Check if all of the elements of a are true.

Performs a logical_and over the given axis and returns the result. Masked values are considered as True during computation. For convenience, the output array is masked where ALL the values along the current axis are masked: if the output would have been a scalar and that all the values are masked, then the output is masked.

Parameters
axis : {None, integer}
Axis to perform the operation over. If None, perform over flattened array.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

all
equivalent function

Examples

>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True

MaskedArray.any(axis=None, out=None)
Check if any of the elements of a are true.

Performs a logical_or over the given axis and returns the result. Masked values are considered as False during computation.

Parameters
axis : {None, integer}
Axis to perform the operation over. If None, perform over flattened array and return a scalar.

out : {None, array}, optional
Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

See Also:

any
equivalent function
MaskedArray.count (axis=None)

Count the non-masked elements of the array along the given axis.

**Parameters**

axis : int, optional

Axis along which to count the non-masked elements. If axis is None, all non-masked elements are counted.

**Returns**

result : int or ndarray

If axis is None, an integer count is returned. When axis is not None, an array with shape determined by the lengths of the remaining axes, is returned.

**See Also:**

count_masked

Count masked elements in array or along a given axis.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(data =
[[0 1 2]
[-- -- --]],
mask =
[[False False False]
[ True True True]],
fill_value = 999999)
>>> a.count()
3
```

When the axis keyword is specified an array of appropriate size is returned.

```python
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

MaskedArray.nonzero()

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```python
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```python
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

None

**Returns**

tuple_of_arrays : tuple
Indices of elements that are non-zero.

**See Also:**

* `numpy.nonzero`  
  Function operating on ndarrays.

* `flatnonzero`  
  Return indices that are non-zero in the flattened version of the input array.

* `ndarray.nonzero`  
  Equivalent ndarray method.

* `count_nonzero`  
  Counts the number of non-zero elements in the input array.

**Examples**

```
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(data =
[ [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]],
mask =
False,
fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))
```

Masked elements are ignored.

```
>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[ [1.0 0.0 0.0]
 [0.0 -- 0.0]
 [0.0 0.0 1.0]],
mask =
[[False False False]
 [False  True False]
 [False False False]],
fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
```

Indices can also be grouped by element.

```
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `ma.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(data =
...
The `nonzero` method of the condition array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

**numpy.ma.shape (obj)**

Return the shape of an array.

**Parameters**

- **a**: array_like
  
  Input array.

**Returns**

- **shape**: tuple of ints
  
  The elements of the shape tuple give the lengths of the corresponding array dimensions.

**See Also**:

- `alen`

**ndarray.shape**

Equivalent array method.

**Examples**

```python
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```python
a = np.array([[1, 2], (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

**numpy.ma.size (obj, axis=None)**

Return the number of elements along a given axis.

**Parameters**

- **a**: array_like
  
  Input data.

- **axis**: int, optional
Axis along which the elements are counted. By default, give the total number of elements.

Returns

element_count : int

Number of elements along the specified axis.

See Also:

shape
dimensions of array

ndarray.shape
dimensions of array

ndarray.size
number of elements in array

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.size(a)
6
>>> np.size(a, 1)
3
>>> np.size(a, 0)
2
```

3.19.4 Manipulating a MaskedArray

Changing the shape

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.ravel(self)</code></td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td><code>ma.reshape(a, new_shape[, order])</code></td>
<td>Returns an array containing the same data with a new shape.</td>
</tr>
<tr>
<td><code>ma.resize(x, new_shape)</code></td>
<td>Return a new masked array with the specified size and shape.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.flatten([order])</code></td>
<td>Return a copy of the array collapsed into one dimension.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ravel()</code></td>
<td>Returns a 1D version of self, as a view.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.reshape(*s, **kwargs)</code></td>
<td>Give a new shape to the array without changing its data.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.resize(newshape[, refcheck, ...])</code></td>
<td></td>
</tr>
</tbody>
</table>

```python
numpy.ma.ravel(self) = <numpy.ma.core._frommethod instance at 0x2820680>
```

Returns a 1D version of self, as a view.

Returns

MaskedArray

Output view is of shape (self.size,) (or (np.ma.product(self.shape),)).

Examples

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
>>> print x
```
NumPy Reference, Release 1.8.1

>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]

def x.ravel():
    return [1, 2, 3, 4, 5, 6, 7, 8, 9]

def y.ravel():
    return [1, 2, 3, 4, 5, 6, 7, 8, 9]

def z.ravel():
    return [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> c = [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> np.array(a).ravel()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> np.array(a, dtype='float').ravel()
array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])

>>> np.array(a, dtype='float32').ravel()
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])

>>> np.array(a, dtype='float64').ravel()
array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0])

>>> np.array(a, dtype='complex').ravel()
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j, 6.+0.j, 7.+0.j, 8.+0.j, 9.+0.j])

>>> np.array(a, dtype='complex64').ravel()
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j, 6.+0.j, 7.+0.j, 8.+0.j, 9.+0.j])

>>> np.array(a, dtype='complex128').ravel()
array([1.0+0.0j, 2.0+0.0j, 3.0+0.0j, 4.0+0.0j, 5.0+0.0j, 6.0+0.0j, 7.0+0.0j, 8.0+0.0j, 9.0+0.0j])

A MaskedArray is always returned, regardless of the input type.
```python
>>> a = np.array([[1, 2], [3, 4]])
>>> ma.resize(a, (3, 3))
masked_array(data =
[[1 2 3]
 [4 1 2]
 [3 4 1]],
mask = False,
fill_value = 999999)
```

MaskedArray\.\texttt{flatten}(\texttt{order}='C')

Return a copy of the array collapsed into one dimension.

**Parameters**

- **order**: {'C', 'F', 'A'}, optional
  
  Whether to flatten in C (row-major), Fortran (column-major) order, or preserve the C/Fortran ordering from \(a\). The default is 'C'.

**Returns**

- **y**: ndarray
  
  A copy of the input array, flattened to one dimension.

**See Also:**

- \texttt{ravel}
  
  Return a flattened array.

- \texttt{flat}
  
  A 1-D flat iterator over the array.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
daarray([1, 2, 3, 4])
>>> a.flatten('F')
daarray([1, 3, 2, 4])
```

MaskedArray\.\texttt{ravel}()

Returns a 1D version of self, as a view.

**Returns**

MaskedArray

Output view is of shape (self.size,) (or (np.ma.product(self.shape),)).

**Examples**

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1, 0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.ravel()
[1 -- 3 -- 5 -- 7 -- 9]
```

MaskedArray\.\texttt{reshape}(*\texttt{a}, **\texttt{kwargs})

Give a new shape to the array without changing its data.
Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

Parameters

- **shape**: int or tuple of ints
  
The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

- **order**: {'C', 'F'}, optional
  
  Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

Returns

- **reshaped_array**: array
  
  A new view on the array.

See Also:

- reshape
  
  Equivalent function in the masked array module.

- numpy.ndarray.reshape
  
  Equivalent method on ndarray object.

- numpy.reshape
  
  Equivalent function in the NumPy module.

Notes

The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`

Examples

```python
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> print x
[[-- 2]
 [3 --]]
>>> x = x.reshape((4,1))
>>> print x
[[--]
 [2]
 [3]
 [--]]
```

**Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the `numpy.ma.resize` function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

Modifying axes

```python
ma.swapaxes  swapaxes
```

Continued on next page
Table 3.89 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.transpose(a[, axes])</code></td>
<td>Permute the dimensions of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.swapaxes(axis1, axis2)</code></td>
<td>Return a view of the array with axis1 and axis2 interchanged.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

```
numpy.ma.swapaxes = <numpy.ma.core._frommethod instance at 0x2820998>

swapaxes a.swapaxes(axis1, axis2)

Return a view of the array with axis1 and axis2 interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also:

`numpy.swapaxes`

equivalent function
```

```
numpy.ma.transpose(a, axes=None)

Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

See Also:

`numpy.transpose`

Equivalent function in top-level NumPy module.

Examples

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked

>>> x
masked_array(data =
[[0 1]
 [2 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)

>>> ma.transpose(x)
masked_array(data =
[[0 2]
 [1 --]],
mask =
[[False False]
 [False  True]],
fill_value = 999999)
```

```
MaskedArray.swapaxes(axis1, axis2)

Return a view of the array with axis1 and axis2 interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also:

`numpy.swapaxes`

equivalent function
```

3.19. Masked array operations 779
MaskedArray.transpose(*axes)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape = (i[0], i[1], ... i[n-2], i[n-1]), then a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0]).

Parameters

axes : None, tuple of ints, or n ints

• None or no argument: reverses the order of the axes.
• tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
• n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

Returns

out : ndarray

View of a, with axes suitably permuted.

See Also:

ndarray.T
Array property returning the array transposed.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])

Changing the number of dimensions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.atleast_1d(*arys)</code></td>
<td>Convert inputs to arrays with at least one dimension.</td>
</tr>
<tr>
<td><code>ma.atleast_2d(*arys)</code></td>
<td>View inputs as arrays with at least two dimensions.</td>
</tr>
<tr>
<td><code>ma.atleast_3d(*arys)</code></td>
<td>View inputs as arrays with at least three dimensions.</td>
</tr>
<tr>
<td><code>ma.expand_dims(x, axis)</code></td>
<td>Expand the shape of an array.</td>
</tr>
<tr>
<td><code>ma.squeeze(a[, axis])</code></td>
<td>Remove single-dimensional entries from the shape of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.squeeze([axis])</code></td>
<td>Remove single-dimensional entries from the shape of a.</td>
</tr>
<tr>
<td><code>ma.column_stack(tup)</code></td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td><code>ma.concatenate(arrays[, axis])</code></td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
</tbody>
</table>
Table 3.90 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.dstack(tup)</td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td>ma.hstack(tup)</td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td>ma.hsplit(ary, indices_or_sections)</td>
<td>Split an array into multiple sub-arrays horizontally (column-wise).</td>
</tr>
<tr>
<td>ma.mr_</td>
<td>Translate slice objects to concatenation along the first axis.</td>
</tr>
<tr>
<td>ma.row_stack(tup)</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
<tr>
<td>ma.vstack(tup)</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

```
numpy.ma.atleast_1d(*arys) = <numpy.ma.extras._fromnxfunction instance at 0x28216c8>
```

Convert inputs to arrays with at least one dimension.
Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

arys1, arys2, ... : array_like
One or more input arrays.

**Returns**

ret : ndarray
An array, or sequence of arrays, each with a.ndim >= 1. Copies are made only if necessary.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> np.atleast_1d(1.0)
array([ 1.])

>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])

>>> np.atleast_1d(x) is x
True

>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

```
numpy.ma.atleast_2d(*arys) = <numpy.ma.extras._fromnxfunction instance at 0x28217e8>
```

View inputs as arrays with at least two dimensions.

**Parameters**

arys1, arys2, ... : array_like
One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**

res, res2, ... : ndarray

3.19. Masked array operations
An array, or tuple of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> np.atleast_2d(3.0)
array([[ 3.]])
```

```python
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[ 0.,  1.,  2.]])
```

```python
>>> np.atleast_2d(x).base is x
True
```

```python
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

**numpy.ma.atleast_3d(*arys)** = `<numpy.ma.extras._fromnxfunction instance at 0x2821878>`

View inputs as arrays with at least three dimensions.

**Parameters**

- `arys1, arys2, ... : array_like`

  One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

**Returns**

- `res1, res2, ... : ndarray`

  An array, or tuple of arrays, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

**Notes**

The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> np.atleast_3d(3.0)
array([[[ 3.]]])
```

```python
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
```

```python
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
```

```python
>>> np.atleast_3d(x).base is x
True
```
>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print arr, arr.shape
...
[[[1]
 [2]]] (1, 2, 1)
[[[1]
 [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)

**numpy.ma.expand_dims** *(x, axis)*

Expand the shape of an array.

Expands the shape of the array by including a new axis before the one specified by the *axis* parameter. This function behaves the same as **numpy.expand_dims** but preserves masked elements.

**See Also:**

**numpy.expand_dims**

Equivalent function in top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.array([1, 2, 4])
>>> x[1] = ma.masked
>>> x
masked_array(data = [1 -- 4],
             mask = [False True False],
             fill_value = 999999)
>>> np.expand_dims(x, axis=0)
array([[1, 2, 4]])
```

The same result can be achieved using slicing syntax with **np.newaxis**.

```python
>>> x[np.newaxis, :]
masked_array(data =
             [[1 -- 4]],
             mask =
             [[False True False]],
             fill_value = 999999)
```

**numpy.ma.squeeze** *(a, axis=None)*

Remove single-dimensional entries from the shape of an array.

**Parameters**

- **a**: array_like
  
  Input data.

- **axis**: None or int or tuple of ints, optional

  New in version 1.7.0. Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.
Returns

`squeezed` : ndarray

The input array, but with with all or a subset of the dimensions of length 1 removed. This is always `a` itself or a view into `a`.

Examples

```python
generate examples code here
```

MaskedArray.

`squeeze`  
(axis=None)

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See Also:

`numpy.squeeze`  
equivalent function

`numpy.ma.column_stack`  
(tup)

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

Parameters

`tup` : sequence of 1-D or 2-D arrays.

Arrays to stack. All of them must have the same first dimension.

Returns

`stacked` : 2-D array

The array formed by stacking the given arrays.

Notes

The function is applied to both the `_data` and the `_mask`, if any.

Examples

```python
generate examples code here
```
Parameters
arrays : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to axis (the first, by default).

axis : int, optional
The axis along which the arrays will be joined. Default is 0.

Returns
result : MaskedArray

The concatenated array with any masked entries preserved.

See Also:

numpy.concatenate
Equivalent function in the top-level NumPy module.

Examples
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
mask = [False True False],
fill_value = 999999)
>>> b
masked_array(data = [2 3 4],
mask = False,
fill_value = 999999)
>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
mask = [False True False False False False],
fill_value = 999999)

numpy.ma.dstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821ab8>

Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by dsplit. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

Parameters
tup : sequence of arrays

Arrays to stack. All of them must have the same shape along all but the third axis.

Returns
stacked : ndarray

The array formed by stacking the given arrays.

See Also:
vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.

dsplit
Split array along third axis.

Notes
The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[1, 2],
 [2, 3],
 [3, 4]])

>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[1, 2],
 [2, 3],
 [3, 4]])
```

numpy.ma.hstack (tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821998>

Stack arrays in sequence horizontally (column wise).
Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

Parameters

- tup : sequence of ndarrays
  All arrays must have the same shape along all but the second axis.

Returns

- stacked : ndarray
  The array formed by stacking the given arrays.

See Also:

vstack
Stack arrays in sequence vertically (row wise).

dstack
Stack arrays in sequence depth wise (along third axis).

concatenate
Join a sequence of arrays together.
**hstack**
Split array along second axis.

**Notes**
The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> a = np.array((1, 2, 3))
>>> b = np.array((2, 3, 4))
>>> np.hstack((a, b))
array([1, 2, 3, 2, 3, 4])
```

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a, b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

numpy.ma.hsplit (ary, indices_or_sections) = <numpy.ma.extras_fromnxfunction instance at 0x2821b00>

Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the split documentation. hsplit is equivalent to split with axis=1, the array is always split along the second axis regardless of the array dimension.

**See Also:**

**split**
Split an array into multiple sub-arrays of equal size.

**Notes**
The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.])
```

```python
>>> np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
       array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]]))
```

```python
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
       array([[ 3.],
        [ 7.],
        [11.],
        [15.]])
```

3.19. Masked array operations
With a higher dimensional array the split is still along the second axis.

```python
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
```

```python
cpy.np.hsplit(x, 2)
[array([[ 0.,  1.],
        [ 4.,  5.]]),
 array([[ 2.,  3.],
        [ 6.,  7.]])]
```

```
numpy.ma._ = <numpy.ma.extras.mr_class object at 0x281edd0>
  Translate slice objects to concatenation along the first axis.
  This is the masked array version of lib.index_tricks.RClass.

Examples

```python
>>> np.ma.mr_[np.ma.array([1,2,3]), 0, 0, np.ma.array([4,5,6])]
array([1, 2, 3, 0, 0, 4, 5, 6])
```

```
numpy.ma.row_stack (tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821908>
  Stack arrays in sequence vertically (row wise).
  Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided
  by vsplit.

Parameters
  tup : sequence of ndarrays
      Tuple containing arrays to be stacked. The arrays must have the same shape along all
      but the first axis.

Returns
  stacked : ndarray
      The array formed by stacking the given arrays.

See Also:

  hstack
  Stack arrays in sequence horizontally (column wise).

  dstack
  Stack arrays in sequence depth wise (along third dimension).
```
concatenate

Join a sequence of arrays together.

vsplit

Split array into a list of multiple sub-arrays vertically.

Notes

The function is applied to both _data and the _mask, if any.

Examples

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a, b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

numpy.ma.vstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821908>

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

Parameters

tup : sequence of ndarrays

Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

hstack
Stack arrays in sequence horizontally (column wise).

dstack
Stack arrays in sequence depth wise (along third dimension).

concatenate
Join a sequence of arrays together.

vsplit
Split array into a list of multiple sub-arrays vertically.
Notes

The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])

>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1, 2, 3,
        [2, 3, 4]])
```

Joining arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.column_stack(tup)</td>
<td>Stack 1-D arrays as columns into a 2-D array.</td>
</tr>
<tr>
<td>ma.concatenate(arrays[, axis])</td>
<td>Concatenate a sequence of arrays along the given axis.</td>
</tr>
<tr>
<td>ma.dstack(tup)</td>
<td>Stack arrays in sequence depth wise (along third axis).</td>
</tr>
<tr>
<td>ma.hstack(tup)</td>
<td>Stack arrays in sequence horizontally (column wise).</td>
</tr>
<tr>
<td>ma.vstack(tup)</td>
<td>Stack arrays in sequence vertically (row wise).</td>
</tr>
</tbody>
</table>

```
numpy.ma.column_stack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821a28>
```

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are
stacked as-is, just like with hstack. 1-D arrays are turned into 2-D columns first.

Parameters

- **tup**: sequence of 1-D or 2-D arrays.

  Arrays to stack. All of them must have the same first dimension.

Returns

- **stacked**: 2-D array

  The array formed by stacking the given arrays.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

```python
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
```
```
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.ma.concatenate(arrays, axis=0)`  
Concateenate a sequence of arrays along the given axis.

**Parameters**
- `arrays` : sequence of array_like  
  The arrays must have the same shape, except in the dimension corresponding to `axis` (the first, by default).
- `axis` : int, optional  
  The axis along which the arrays will be joined. Default is 0.

**Returns**
- `result` : MaskedArray  
  The concatenated array with any masked entries preserved.

**See Also:**
- `numpy.concatenate`  
  Equivalent function in the top-level NumPy module.

**Examples**
```python
>>> import numpy.ma as ma

>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data = [0 -- 2],
             mask = [False True False],
             fill_value = 999999)

>>> b
masked_array(data = [2 3 4],
             mask = False,
             fill_value = 999999)

>>> ma.concatenate([a, b])
masked_array(data = [0 -- 2 2 3 4],
             mask = [False True False False False False],
             fill_value = 999999)
```

`numpy.ma.dstack(tup)`  
Stack arrays in sequence depth wise (along third axis).

Takes a sequence of arrays and stack them along the third axis to make a single array. Rebuilds arrays divided by `dsplit`. This is a simple way to stack 2D arrays (images) into a single 3D array for processing.

**Parameters**
- `tup` : sequence of arrays  
  Arrays to stack. All of them must have the same shape along all but the third axis.
Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

vstack
Stack along first axis.

hstack
Stack along second axis.

concatenate
Join arrays.

dsplits
Split array along third axis.

Notes

The function is applied to both the _data and the _mask, if any.

Examples

>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[[1, 2],
       [2, 3],
       [3, 4]]])

>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
       [[2, 3]],
       [[3, 4]]])

numpy.ma.hstack (tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821998>

Stack arrays in sequence horizontally (column wise).

Take a sequence of arrays and stack them horizontally to make a single array. Rebuild arrays divided by hsplit.

Parameters

tup : sequence of ndarrays

All arrays must have the same shape along all but the second axis.

Returns

stacked : ndarray

The array formed by stacking the given arrays.

See Also:

vstack
Stack arrays in sequence vertically (row wise).
**dstack**  
Stack arrays in sequence depth wise (along third axis).

**concatenate**  
Join a sequence of arrays together.

**hsplit**  
Split array along second axis.

**Notes**  
The function is applied to both the _data and the _mask, if any.

**Examples**

```python
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
```

```python
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

def numpy.ma.vstack(tup) = <numpy.ma.extras._fromnxfunction instance at 0x2821908>

Stack arrays in sequence vertically (row wise).

Take a sequence of arrays and stack them vertically to make a single array. Rebuild arrays divided by vsplit.

**Parameters**

- **tup** : sequence of ndarrays
  
  Tuple containing arrays to be stacked. The arrays must have the same shape along all but the first axis.

- **Returns**
  
  stacked : ndarray
  
  The array formed by stacking the given arrays.

**See Also:**

**hstack**  
Stack arrays in sequence horizontally (column wise).

**dstack**  
Stack arrays in sequence depth wise (along third dimension).

**concatenate**  
Join a sequence of arrays together.

**vsplit**  
Split array into a list of multiple sub-arrays vertically.
Notes
The function is applied to both the _data and the _mask, if any.

Examples
```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])
```
```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a, b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

3.19.5 Operations on masks

Creating a mask

```python
numpy.ma.make_mask(m[, copy, shrink, dtype])
```
Create a boolean mask from an array.

Return m as a boolean mask from an array.

Parameters
- **m**: array_like
  Potential mask.
- **copy**: bool, optional
  Whether to return a copy of m (True) or m itself (False).
- **shrink**: bool, optional
  Whether to shrink m to nomask if all its values are False.
- **dtype**: dtype, optional
  Data-type of the output mask. By default, the output mask has a dtype of MaskType (bool). If the dtype is flexible, each field has a boolean dtype.
Returns

result : ndarray

A boolean mask derived from m.

Examples

```python
>>> import numpy.ma as ma
>>> m = [True, False, True, True]
>>> ma.make_mask(m)
array([True, False, True, True], dtype=bool)
>>> m = [1, 0, 1, 1]
>>> ma.make_mask(m)
array([True, False, True, True], dtype=bool)
>>> m = [1, 0, 2, -3]
>>> ma.make_mask(m)
array([True, False, True, True], dtype=bool)
```

Effect of the `shrink` parameter.

```python
>>> m = np.zeros(4)
>>> m
array([ 0., 0., 0., 0.])
>>> ma.make_mask(m)
False
>>> ma.make_mask(m, shrink=False)
array([False, False, False, False], dtype=bool)
```

Using a flexible `dtype`.

```python
>>> m = [1, 0, 1, 1]
>>> n = [0, 1, 0, 0]
>>> arr = []
>>> for man, mouse in zip(m, n):
...    arr.append((man, mouse))
>>> arr
[(1, 0), (0, 1), (1, 0), (1, 0)]
```

```python
dtype = np.dtype([('names':[\'man\', \'mouse\'],
                   \'formats':[np.int, np.int]])
>>> arr = np.array(arr, dtype=dtype)
>>> arr
array([(1, 0), (0, 1), (1, 0), (1, 0)],
      dtype=[\'man\', \'\<i4\'], \'mouse\', \'\<i4\'])
>>> ma.make_mask(arr, dtype=dtype)
array([(True, False), (False, True), (True, False), (True, False)],
      dtype=[\'man\', \'\|b1\'], (\'mouse\', \'\|b1\'])
```

numpy.ma.make_mask_none(newshape, dtype=None)

Return a boolean mask of the given shape, filled with False. This function returns a boolean ndarray with all entries False, that can be used in common mask manipulations. If a complex dtype is specified, the type of each field is converted to a boolean type.

Parameters

    newshape : tuple

A tuple indicating the shape of the mask.

    dtype : {None, dtype}, optional
If None, use a MaskType instance. Otherwise, use a new datatype with the same fields as `dtype`, converted to boolean types.

**Returns**

result : ndarray

An ndarray of appropriate shape and dtype, filled with False.

**See Also:**

`make_mask`
Create a boolean mask from an array.

`make_mask_descr`
Construct a dtype description list from a given dtype.

**Examples**

```python
>>> import numpy.ma as ma
>>> ma.make_mask_none((3,))
array([False, False, False], dtype=bool)
```

Defining a more complex dtype.

```python
>>> dtype = np.dtype([('names': ['foo', 'bar'],
                   'formats': [np.float32, np.int])])
>>> dtype
dtype([('names': ['foo', 'bar'], 'formats': [np.float32, np.int])])
>>> ma.make_mask_none((3,), dtype=dtype)
array([(False, False), (False, False), (False, False)],
      dtype=[('foo', '|b1'), ('bar', '|b1')])
```

`numpy.ma.mask_or(m1, m2, copy=False, shrink=True)`

Combine two masks with the `logical_or` operator.

The result may be a view on `m1` or `m2` if the other is `nomask` (i.e. False).

**Parameters**

- `m1, m2` : array_like
  Input masks.

- `copy` : bool, optional
  If copy is False and one of the inputs is `nomask`, return a view of the other input mask. Defaults to False.

- `shrink` : bool, optional
  Whether to shrink the output to `nomask` if all its values are False. Defaults to True.

**Returns**

- `mask` : output mask
  The result masks values that are masked in either `m1` or `m2`.

**Raises**

- `ValueError`
  If `m1` and `m2` have different flexible dtypes.
Examples

```python
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True,  True,  True, False], dtype=bool)
```

```python
numpy.ma.make_mask_descr(ndtype)
Construct a dtype description list from a given dtype.

Returns a new dtype object, with the type of all fields in ndtype to a boolean type. Field names are not altered.

Parameters
    ndtype : dtype
        The dtype to convert.

Returns
    result : dtype
        A dtype that looks like ndtype, the type of all fields is boolean.
```

Examples

```python
>>> import numpy.ma as ma
>>> dtype = np.dtype({'names': ['foo', 'bar'],
                     'formats': [np.float32, np.int]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i4')])
>>> ma.make_mask_descr(dtype)
dtype([('foo', '|b1'), ('bar', '|b1')])
>>> ma.make_mask_descr(np.float32)
<type 'numpy.bool_'>
```

Accessing a mask

```python
ma.getmask(a)  # Return the mask of a masked array, or nomask.
ma.getmaskarray(arr)  # Return the mask of a masked array, or full boolean array of False.
ma.masked_array.mask  # Mask
```

```python
numpy.ma.getmask(a)
Return the mask of a masked array, or nomask.

Return the mask of a as an ndarray if a is a MaskedArray and the mask is not nomask, else return nomask.
To guarantee a full array of booleans of the same shape as a, use getmaskarray.

Parameters
    a : array_like
        Input MaskedArray for which the mask is required.

See Also:
    getdata
        Return the data of a masked array as an ndarray.
    getmaskarray
        Return the mask of a masked array, or full array of False.
```

3.19. Masked array operations
Examples
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 2]
 [3 4]],
mask =
[[False  True]
 [False False]],
fill_value=999999)
>>> ma.getmask(a)
array([[False,  True],
       [False, False]], dtype=boolean)

Equivalently use the MaskedArray mask attribute.
>>> a.mask
array([[False,  True],
       [False, False]], dtype=boolean)

Result when mask == nomask
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
 [3 4]],
mask = False,
fill_value=999999)
>>> ma.nomask
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True

numpy.ma.getmaskarray(arr)
Return the mask of a masked array, or full boolean array of False.

Return the mask of arr as an ndarray if arr is a MaskedArray and the mask is not nomask, else return a full boolean array of False of the same shape as arr.

Parameters
arr : array_like
Input MaskedArray for which the mask is required.

See Also:
getmask
Return the mask of a masked array, or nomask.
getdata
Return the data of a masked array as an ndarray.
Examples

```python
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(data =
[[1 --]
[3 4]],
mask =
[[False True]
[False False]],
fill_value=999999)
>>> ma.getmaskarray(a)
array([[False, True],
       [False, False]], dtype=bool)
```

Result when mask == nomask

```python
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(data =
[[1 2]
[3 4]],
mask =
False,
fill_value=999999)
>>> ma.getmaskarray(b)
array([[False, False],
       [False, False]], dtype=bool)
```

masked_array
Mask

Finding masked data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.flatnotmasked_contiguous(a)</code></td>
<td>Find contiguous unmasked data in a masked array along the given axis.</td>
</tr>
<tr>
<td><code>ma.flatnotmasked_edges(a)</code></td>
<td>Find the indices of the first and last unmasked values.</td>
</tr>
<tr>
<td><code>ma.notmasked_contiguous(a[, axis])</code></td>
<td>Find contiguous unmasked data in a masked array along the given axis.</td>
</tr>
<tr>
<td><code>ma.notmasked_edges(a[, axis])</code></td>
<td>Find the indices of the first and last unmasked values along an axis.</td>
</tr>
</tbody>
</table>

`numpy.ma.flatnotmasked_contiguous(a)`
Find contiguous unmasked data in a masked array along the given axis.

Parameters
- **a**: array
  The input array.

Returns
- **slice_list**: list
  A sorted sequence of slices (start index, end index).

See Also:
- `flatnotmasked_edges`, `notmasked_contiguous`, `notmasked_edges`, `clump_masked`, `clump_unmasked`
Notes

Only accepts 2-D arrays at most.

Examples

```python
>>> a = np.ma.arange(10)
>>> np.ma.extras.flatnotmasked_contiguous(a)
slice(0, 10, None)

>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> a[mask] = np.ma.masked

>>> np.array(a[~a.mask])
array([3, 4, 6, 7, 8])
```

```python
>>> np.ma.extras.flatnotmasked_contiguous(a)
[slice(3, 5, None), slice(6, 9, None)]
>>> a[:] = np.ma.masked

>>> print np.ma.extras.flatnotmasked_edges(a)
None
```

```python
def flatnotmasked_edges(arr):
    """Find the indices of the first and last unmasked values."

    Expects a 1-D MaskedArray, returns None if all values are masked.

    Parameters
    ----------
    arr : array_like
        Input 1-D MaskedArray

    Returns
    -------
    edges : ndarray or None
        The indices of first and last non-masked value in the array. Returns None if all values are masked.

    See Also:
    --------
    flatnotmasked_contiguous, notmasked_contiguous, notmasked_edges, clump_masked, clump_unmasked

    Notes
    -----,

    Only accepts 1-D arrays.

    Examples
    --------

    ```python
    >>> a = np.ma.arange(10)
    >>> flatnotmasked_edges(a)
    [0, -1]
    ```
    ```python
    >>> mask = (a < 3) | (a > 8) | (a == 5)
    >>> a[mask] = np.ma.masked

    >>> np.array(a[~a.mask])
    array([3, 4, 6, 7, 8])
    ```
    ```python
    >>> flatnotmasked_edges(a)
    array([3, 8])
    ```
```python
>>> a[:] = np.ma.masked
>>> print flatnotmasked_edges(ma)
None
```

```python
numpy.ma.notmasked_contiguous(a, axis=None)
Find contiguous unmasked data in a masked array along the given axis.

Parameters
  a : array_like
      The input array.
  axis : int, optional
      Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

Returns
  endpoints : list
      A list of slices (start and end indexes) of unmasked indexes in the array.

See Also:
  flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges, clump_masked, clump_unmasked

Notes
Only accepts 2-D arrays at most.

Examples
```python
>>> a = np.arange(9).reshape((3, 3))
>>> mask = np.zeros_like(a)
>>> mask[1:, 1:] = 1

>>> ma = np.ma.array(a, mask=mask)
>>> np.array(ma[~ma.mask])
array([0, 1, 2, 3, 6])

```python
>>> np.ma.extras.notmasked_contiguous(ma)
[slice(0, 4, None), slice(6, 7, None)]
```

```python
numpy.ma.notmasked_edges(a, axis=None)
Find the indices of the first and last unmasked values along an axis.

If all values are masked, return None. Otherwise, return a list of two tuples, corresponding to the indices of the first and last unmasked values respectively.

Parameters
  a : array_like
      The input array.
  axis : int, optional
      Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

Returns
  edges : ndarray or list
      A list of two tuples, corresponding to the indices of the first and last unmasked values respectively.
```
An array of start and end indexes if there are any masked data in the array. If there are no masked data in the array, edges is a list of the first and last index.

See Also:

- flatnotmasked_contiguous
- flatnotmasked_edges
- notmasked_contiguous
- clump_masked
- clump_unmasked

Examples

```python
>>> a = np.arange(9).reshape((3, 3))
>>> m = np.zeros_like(a)
>>> m[1:, 1:] = 1

>>> am = np.ma.array(a, mask=m)
>>> np.array(am[~am.mask])
array([0, 1, 2, 3, 6])

>>> np.ma.extras.notmasked_edges(ma)
array([0, 6])
```

Modifying a mask

```
ma.mask_cols(a[, axis]) Mask columns of a 2D array that contain masked values.
machine mask_or(m1, m2[, copy, shrink]) Combine two masks with the logical_or operator.
machine mask_rowcols(a[, axis]) Mask rows and/or columns of a 2D array that contain masked values.
machine mask_rows(a[, axis]) Mask rows of a 2D array that contain masked values.
machine harden_mask(self) Force the mask to hard.
machine soften_mask(self) Force the mask to soft.
machine MaskedArray.harden_mask() Force the mask to hard.
machine MaskedArray.soften_mask() Force the mask to soft.
machine MaskedArray.shrink_mask() Reduce a mask to nomask when possible.
machine MaskedArray.unshare_mask() Copy the mask and set the sharedmask flag to False.
```

numpy.ma.mask_cols(a, axis=None) Mask columns of a 2D array that contain masked values.

This function is a shortcut to mask_rowcols with axis equal to 1.

See Also:

- mask_rowcols
  Mask rows and/or columns of a 2D array.

- masked_where
  Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma

>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1

>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
```
```python
global a
a = ma.masked_equal(a, 1)
a
masked_array(data =
    [[0 0 0]
    [0 -- 0]
    [0 0 0]],
    mask =
    [[False False False]
    [False True False]
    [False False False]],
    fill_value=999999)
```

```python
ma.mask_cols(a)
masked_array(data =
    [[0 -- 0]
    [0 -- 0]
    [0 -- 0]],
    mask =
    [[False True False]
    [False True False]
    [False True False]],
    fill_value=999999)
```

```
numpy.ma.**mask_or**(m1, m2, copy=False, shrink=True)
```

Combine two masks with the `logical_or` operator.

The result may be a view on `m1` or `m2` if the other is `nomask` (i.e. False).

**Parameters**

- `m1, m2` : array_like
  Input masks.

- `copy` : bool, optional
  If copy is False and one of the inputs is `nomask`, return a view of the other input mask. Defaults to False.

- `shrink` : bool, optional
  Whether to shrink the output to `nomask` if all its values are False. Defaults to True.

**Returns**

- `mask` : output mask
  The result masks values that are masked in either `m1` or `m2`.

**Raises**

- `ValueError`
  If `m1` and `m2` have different flexible dtypes.

**Examples**

```python
m1 = np.ma.make_mask([0, 1, 1, 0])
m2 = np.ma.make_mask([1, 0, 0, 0])
np.ma.mask_or(m1, m2)
array([ True,  True,  True, False], dtype=bool)
```

```
numpy.ma.**mask_rowcols** *(a, axis=None)*
```

Mask rows and/or columns of a 2D array that contain masked values.
Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected using the \textit{axis} parameter.

- If \textit{axis} is None, rows \textit{and} columns are masked.
- If \textit{axis} is 0, only rows are masked.
- If \textit{axis} is 1 or -1, only columns are masked.

**Parameters**

- \textit{a} : array_like, MaskedArray
  
  The array to mask. If not a MaskedArray instance (or if no array elements are masked). The result is a MaskedArray with \textit{mask} set to \texttt{nomask} (False). Must be a 2D array.

- \textit{axis} : int, optional
  
  Axis along which to perform the operation. If None, applies to a flattened version of the array.

**Returns**

- \textit{a} : MaskedArray
  
  A modified version of the input array, masked depending on the value of the \textit{axis} parameter.

**Raises**

- \texttt{NotImplementedError}
  
  If input array \textit{a} is not 2D.

**See Also:**

- \texttt{mask_rows}
  
  Mask rows of a 2D array that contain masked values.

- \texttt{mask_cols}
  
  Mask cols of a 2D array that contain masked values.

- \texttt{masked_where}
  
  Mask where a condition is met.

**Notes**

The input array’s mask is modified by this function.

**Examples**

```python
got
```
```python
[[False False False]
[False True False]
[False False False]],

fill_value=999999)

>>> ma.mask_rowcols(a)
masked_array(data =

[[0 -- 0]
[-- -- --]
[0 -- 0]],

mask =

[[False True False]
[ True True True]
[False True False]],

fill_value=999999)

numpy.ma.mask_rows(a, axis=None)

Mask rows of a 2D array that contain masked values.

This function is a shortcut to `mask_rowcols` with `axis` equal to 0.

See Also:

`mask_rowcols`  
Mask rows and/or columns of a 2D array.

`masked_where`  
Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=np.int)
>>> a[1, 1] = 1

array([[0, 0, 0],

[0, 1, 0],

[0, 0, 0]])

>>> a = ma.masked_equal(a, 1)

masked_array(data =

[[0 0 0]
[-- -- --]
[0 -- 0]],

mask =

[[False False False]
[False True False]
[False False False]],

fill_value=999999)

>>> ma.mask_rows(a)
masked_array(data =

[[0 0 0]
[-- -- --]
[0 0 0]],

mask =

[[False False False]
[ True True True]
[False False False]],

fill_value=999999)
```
NumPy Reference, Release 1.8.1

```
numpy.ma.harden_mask(self) = <numpy.ma.core._frommethod instance at 0x2820368>
    Force the mask to hard.
    Whether the mask of a masked array is hard or soft is determined by its hardmask property. harden_mask sets hardmask to True.

See Also:
    hardmask

numpy.ma.soften_mask(self) = <numpy.ma.core._frommethod instance at 0x28207a0>
    Force the mask to soft.
    Whether the mask of a masked array is hard or soft is determined by its hardmask property. soften_mask sets hardmask to False.

See Also:
    hardmask

MaskedArray.harden_mask()
    Force the mask to hard.
    Whether the mask of a masked array is hard or soft is determined by its hardmask property. harden_mask sets hardmask to True.

See Also:
    hardmask

MaskedArray.soften_mask()
    Force the mask to soft.
    Whether the mask of a masked array is hard or soft is determined by its hardmask property. soften_mask sets hardmask to False.

See Also:
    hardmask

MaskedArray.shrink_mask()
    Reduce a mask to nomask when possible.

Parameters
    None

Returns
    None

Examples
    >>> x = np.ma.array([[1,2 ], [3, 4]], mask=[0]*4)
    >>> x.mask
    array([[False, False],
           [False, False]], dtype=bool)
    >>> x.shrink_mask()
    >>> x.mask
    False

MaskedArray.unshare_mask()
    Copy the mask and set the sharedmask flag to False.
    Whether the mask is shared between masked arrays can be seen from the sharedmask property. unshare_mask ensures the mask is not shared. A copy of the mask is only made if it was shared.
```
3.19.6 Conversion operations

> to a masked array

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.asarray(a[, dtype, order])</code></td>
<td>Convert the input to a masked array of the given data-type.</td>
</tr>
<tr>
<td><code>ma.asanyarray(a[, dtype])</code></td>
<td>Convert the input to a masked array, conserving subclasses.</td>
</tr>
<tr>
<td><code>ma.fix_invalid(a[, mask, copy, fill_value])</code></td>
<td>Return input with invalid data masked and replaced by a fill value.</td>
</tr>
<tr>
<td><code>mamasked_equal(x, value[, copy])</code></td>
<td>Mask an array where equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater(x, value[, copy])</code></td>
<td>Mask an array where greater than a given value.</td>
</tr>
<tr>
<td><code>ma.masked_greater_equal(x, value[, copy])</code></td>
<td>Mask an array where greater than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_inside(x, v1, v2[, copy])</code></td>
<td>Mask an array inside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_invalid(a[, copy])</code></td>
<td>Mask an array where invalid values occur (NaNs or infs).</td>
</tr>
<tr>
<td><code>ma.masked_less_equal(x, value[, copy])</code></td>
<td>Mask an array where less than or equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_not_equal(x, value[, copy])</code></td>
<td>Mask an array where not equal to a given value.</td>
</tr>
<tr>
<td><code>ma.masked_object(x, value[, copy, shrink])</code></td>
<td>Mask the array where the data are exactly equal to value.</td>
</tr>
<tr>
<td><code>ma.masked_outside(x, v1, v2[, copy])</code></td>
<td>Mask an array outside a given interval.</td>
</tr>
<tr>
<td><code>ma.masked_values(x, value[, rtol, atol, ...])</code></td>
<td>Mask using floating point equality.</td>
</tr>
<tr>
<td><code>ma.masked_where(condition, a[, copy])</code></td>
<td>Mask an array where a condition is met.</td>
</tr>
</tbody>
</table>

**numpy.ma.asarray (a, dtype=None, order=None)**

Convert the input to a masked array of the given data-type.

No copy is performed if the input is already an ndarray. If `a` is a subclass of `MaskedArray`, a base class `MaskedArray` is returned.

**Parameters**

- `a` : array_like
  Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples of tuples, tuples of lists, ndarrays and masked arrays.

- `dtype` : dtype, optional
  By default, the data-type is inferred from the input data.

- `order` : {‘C’, ‘F’}, optional
  Whether to use row-major (‘C’) or column-major (‘FORTRAN’) memory representation. Default is ‘C’.

**Returns**

- `out` : MaskedArray
  Masked array interpretation of `a`.

**See Also:**

- `asanyarray`
  Similar to `asarray`, but conserves subclasses.
Examples

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asarray(x)
masked_array(data =
              [[ 0.  1.  2.  3.  4.]
              [ 5.  6.  7.  8.  9.]],
     mask = False,
      fill_value = 1e+20)
>>> type(np.ma.asarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

`numpy.ma.asanyarray(a, dtype=None)`

Convert the input to a masked array, conserving subclasses.

If `a` is a subclass of `MaskedArray`, its class is conserved. No copy is performed if the input is already an `ndarray`.

Parameters

- `a`: array_like
  - Input data, in any form that can be converted to an array.

- `dtype`: dtype, optional
  - By default, the data-type is inferred from the input data.

- `order`: {'C', 'F'}, optional
  - Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

Returns

- `out`: MaskedArray
  - MaskedArray interpretation of `a`.

See Also:

- `asarray`
  - Similar to `asanyarray`, but does not conserve subclass.

Examples

```python
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> np.ma.asanyarray(x)
masked_array(data =
              [[ 0.  1.  2.  3.  4.]
              [ 5.  6.  7.  8.  9.]],
     mask = False,
      fill_value = 1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
```
numpy.ma.fix_invalid(a, mask=False, copy=True, fill_value=None)

Return input with invalid data masked and replaced by a fill value.

Invalid data means values of nan, inf, etc.

Parameters

a : array_like
    Input array, a (subclass of) ndarray.

copy : bool, optional
    Whether to use a copy of a (True) or to fix a in place (False). Default is True.

fill_value : scalar, optional
    Value used for fixing invalid data. Default is None, in which case the a.fill_value
    is used.

Returns

b : MaskedArray
    The input array with invalid entries fixed.

Notes

A copy is performed by default.

Examples

>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data = [-- -1.0 nan inf],
        mask = [ True False False False],
        fill_value = 1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data = [-- -1.0 -- --],
        mask = [ True False True True],
        fill_value = 1e+20)

>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.00000000e+00, -1.00000000e+00, 1.00000000e+20, 1.00000000e+20])
>>> x.data
array([ 1., -1., NaN, Inf])

numpy.ma.masked_equal(x, value, copy=True)

Mask an array where equal to a given value.

This function is a shortcut to masked_where, with condition = (x == value). For floating point arrays, consider
using masked_values(x, value).

See Also:

masked_where
    Mask where a condition is met.

masked_values
    Mask using floating point equality.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data = [0 1 -- 3],
             mask = [False False True False],
             fill_value=999999)
```

**numpy.ma.masked_greater** \((x, \text{value}, \text{copy=\text{True}})\)

Mask an array where greater than a given value.

This function is a shortcut to **masked_where**, with \(condition = (x > \text{value})\).

**See Also:**

**masked_where**

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data = [0 1 2 --],
             mask = [False False False True],
             fill_value=999999)
```

**numpy.ma.masked_greater_equal** \((x, \text{value}, \text{copy=\text{True}})\)

Mask an array where greater than or equal to a given value.

This function is a shortcut to **masked_where**, with \(condition = (x \geq \text{value})\).

**See Also:**

**masked_where**

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data = [0 1 -- --],
             mask = [False False True True],
             fill_value=999999)
```

**numpy.ma.masked_inside** \((x, \text{vl}, \text{v2}, \text{copy=\text{True}})\)

Mask an array inside a given interval.

Shortcut to **masked_where**, where \(condition\) is True for \(x\) inside the interval \([\text{vl}, \text{v2}]\) \((\text{vl} \leq x \leq \text{v2})\). The boundaries \(\text{vl}\) and \(\text{v2}\) can be given in either order.

**See Also:**
masked_where

Mask where a condition is met.

Notes

The array x is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
            fill_value=1e+20)
```

The order of v1 and v2 doesn’t matter.

```python
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data = [0.31 1.2 -- -- -0.4 -1.1],
             mask = [False False True True False False],
            fill_value=1e+20)
```

```
numpy.ma.masked_invalid(a, copy=True)

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to masked_where, with condition = ~(np.isfinite(a)). Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

See Also:

masked_where

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=np.float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0., 1., NaN, Inf, 4.])
>>> ma.masked_invalid(a)
masked_array(data = [0.0 1.0 -- -- 4.0],
             mask = [False False True True False],
            fill_value=1e+20)
```

```
numpy.ma.masked_less(x, value, copy=True)

Mask an array where less than a given value.

This function is a shortcut to masked_where, with condition = (x < value).

See Also:

masked_where

Mask where a condition is met.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data = [-- -- 2 3],
              mask = [ True  True False False],
              fill_value=999999)
```

`numpy.ma.masked_less_equal(x, value, copy=True)`

Mask an array where less than or equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x <= value)`.

See Also:

`masked_where`

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data = [-- -- -- 3],
              mask = [ True  True  True False],
              fill_value=999999)
```

`numpy.ma.masked_not_equal(x, value, copy=True)`

Mask an array where not equal to a given value.

This function is a shortcut to `masked_where`, with `condition = (x != value)`.

See Also:

`masked_where`

Mask where a condition is met.

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data = [-- -- 2 --],
              mask = [ True  True False True],
              fill_value=999999)
```

`numpy.ma.masked_object(x, value, copy=True, shrink=True)`

Mask the array `x` where the data are exactly equal to value.

This function is similar to `masked_values`, but only suitable for object arrays: for floating point, use `masked_values` instead.

Parameters

- `x`: array_like
Array to mask

value : object

Comparison value

copy : {True, False}, optional

Whether to return a copy of x.

shrink : {True, False}, optional

Whether to collapse a mask full of False to nomask

Returns

result : MaskedArray

The result of masking x where equal to value.

See Also:

masked_where

Mask where a condition is met.

masked_equal

Mask where equal to a given value (integers).

masked_values

Mask using floating point equality.

Examples

```python
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> print eat
[-- ham]
>>> # plain ol' ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> print eat
[cheese ham pineapple]
```

Note that mask is set to nomask if possible.

```python
>>> eat
masked_array(data = [cheese ham pineapple],
             mask = False,
             fill_value=?)
```

**numpy.ma**.masked_outside(x, v1, v2, copy=True)

Mask an array outside a given interval.

Shortcut to masked_where, where condition is True for x outside the interval [v1,v2] (x < v1)(x > v2). The boundaries v1 and v2 can be given in either order.

See Also:

masked_where

Mask where a condition is met.
Notes

The array \( x \) is prefilled with its filling value.

Examples

```python
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
mixed_array(data = [- - 0.01 0.2 -- --],
            mask = [ True True False False True True],
            fill_value=1e+20)
```

The order of \( v1 \) and \( v2 \) doesn’t matter.

```python
>>> ma.masked_outside(x, 0.3, -0.3)
mixed_array(data = [- - 0.01 0.2 -- --],
            mask = [ True True False False True True],
            fill_value=1e+20)
```

```python
numpy.ma.masked_values(x, value, rtol=1e-05, atol=1e-08, copy=True, shrink=True)
```

Mask using floating point equality.

Return a MaskedArray, masked where the data in array \( x \) are approximately equal to \( value \), i.e. where the following condition is True:

\[
\text{abs}(x - value) \leq \text{atol} + \text{rtol} \times \text{abs}(value)
\]

The fill_value is set to \( value \) and the mask is set to nomask if possible. For integers, consider using masked_equal.

Parameters

- \( x \) : array_like
  - Array to mask.
- \( value \) : float
  - Masking value.
- \( rtol \) : float, optional
  - Tolerance parameter.
- \( atol \) : float, optional
  - Tolerance parameter (1e-8).
- \( copy \) : bool, optional
  - Whether to return a copy of \( x \).
- \( shrink \) : bool, optional
  - Whether to collapse a mask full of False to nomask.

Returns

- \( result \) : MaskedArray
  - The result of masking \( x \) where approximately equal to \( value \).

See Also:

- masked_where
  - Mask where a condition is met.
masked_equal

Mask where equal to a given value (integers).

Examples

```python
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data = [1.0 -- 2.0 -- 3.0],
              mask = [False True False True False],
              fill_value=1.1)
```

Note that `mask` is set to `nomask` if possible.

```python
>>> ma.masked_values(x, 1.5)
masked_array(data = [ 1. 1.1 2. 1.1 3.],
              mask = False,
              fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```python
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data = [0 1 -- 3 4],
              mask = [False False True False False],
              fill_value=2)
```

```python
>>> ma.masked_equal(x, 2)
masked_array(data = [0 1 -- 3 4],
              mask = [False False True False False],
              fill_value=999999)
```

`numpy.ma.masked_where(condition, a, copy=True)`

Mask an array where a condition is met.

Return `a` as an array masked where `condition` is True. Any masked values of `a` or `condition` are also masked in the output.

**Parameters**

- `condition` : array_like
  Masking condition. When `condition` tests floating point values for equality, consider using `masked_values` instead.

- `a` : array_like
  Array to mask.

- `copy` : bool (default: True)
  If True (default) make a copy of `a` in the result. If False modify `a` in place and return a view.

**Returns**

- `result` : MaskedArray
  The result of masking `a` where `condition` is True.
masked_values
Mask using floating point equality.

masked_equal
Mask where equal to a given value.

masked_not_equal
Mask where not equal to a given value.

masked_less_equal
Mask where less than or equal to a given value.

masked_greater_equal
Mask where greater than or equal to a given value.

masked_less
Mask where less than a given value.

masked_greater
Mask where greater than a given value.

masked_inside
Mask inside a given interval.

masked_outside
Mask outside a given interval.

masked_invalid
Mask invalid values (NaNs or infs).

Examples

```python
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data=[-- -- -- 3],
             mask=[ True True True False],
             fill_value=999999)
Mask array b conditional on a.

>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data=['a' 'b' -- 'd'],
             mask=[False False True False],
             fill_value=N/A)
```

Effect of the copy argument.

```python
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data=[-- -- -- 3],
             mask=[ True True True False],
             fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data=[99 -- -- 3],
             mask=[False True True False],
             fill_value=999999)
```
array([0, 1, 2, 3])
>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c[0] = 99
>>> c
masked_array(data = [99 -- -- 3],
             mask = [False  True  True False],
             fill_value=999999)

When `condition` or `a` contain masked values.

>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data = [0 1 -- 3],
             mask = [False False  True False],
             fill_value=999999)

>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data = [-- 1 2 3],
             mask = [ True False False False],
             fill_value=999999)

>>> ma.masked_where(a == 3, b)
masked_array(data = [-- 1 -- --],
             mask = [ True  False  True  True],
             fill_value=999999)

> to a ndarray

```
ma.compress_cols(a) Suppress whole columns of a 2-D array that contain masked values.
ma.compress_rowcols(x[, axis]) Suppress the rows and/or columns of a 2-D array that contain
    masked values.
ma.compress_rows(a) Suppress whole rows of a 2-D array that contain masked values.
ma.compresseds(x) Return all the non-masked data as a 1-D array.
ma.filled(a[, fill_value]) Return input as an array with masked data replaced by a fill value.
ma.MaskedArray.compressed() Return all the non-masked data as a 1-D array.
ma.MaskedArray.filled([fill_value]) Return a copy of self, with masked values filled with a given value.
```

```
numpy.ma.compress_cols(a) Suppress whole columns of a 2-D array that contain masked values.
    This is equivalent to numpy.maextras.compress_rowcols(a, 1), see
    extras.compress_rowcols for details.

    See Also:
    extras.compress_rowcols
```

```
numpy.ma.compress_rowcols(x, axis=None) Suppress the rows and/or columns of a 2-D array that contain masked values.
    The suppression behavior is selected with the `axis` parameter.
    • If axis is None, both rows and columns are suppressed.
    • If axis is 0, only rows are suppressed.
```
• If axis is 1 or -1, only columns are suppressed.

Parameters
axis : int, optional
Axis along which to perform the operation. Default is None.

Returns
compressed_array : ndarray
The compressed array.

Examples
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
... [1, 0, 0],
... [0, 0, 0]])
>>> x
masked_array(data =
[[-- 1 2]
[-- 4 5]
[6 7 8]],
mask =
[[ True False False]
[ True False False]
[False False False]],
fill_value = 999999)

>>> np.ma.extras.compress_rowcols(x)
array([[7, 8]])
>>> np.ma.extras.compress_rowcols(x, 0)
array([[6, 7, 8]])
>>> np.ma.extras.compress_rowcols(x, 1)
array([[1, 2],
[4, 5],
[7, 8]])

numpy.ma.compressed(x)
Return all the non-masked data as a 1-D array.
This function is equivalent to calling the “compressed” method of a MaskedArray, see MaskedArray.compressed for details.

See Also:
MaskedArray.compressed

numpy.ma.filled(a, fill_value=None)
Return input as an array with masked data replaced by a fill value.
If $a$ is not a `MaskedArray`, $a$ itself is returned. If $a$ is a `MaskedArray` and `fill_value` is None, `fill_value` is set to $a.fill_value$.

**Parameters**

- $a$: MaskedArray or array_like
  
  An input object.

- `fill_value`: scalar, optional
  
  Filling value. Default is None.

**Returns**

- `a`: ndarray
  
  The filled array.

**See Also:**

`compressed`

**Examples**

```python
def x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0], ... [0, 0, 0]])
>>> x.filled()
array([[999999, 1, 2],
       [999999, 4, 5],
       [ 6, 7, 8]])
```

**Notes**

The result is not a MaskedArray!

**Examples**

```python
def x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<type 'numpy.ndarray'>
```

**Methods**

- `.compressed()`
  
  Return all the non-masked data as a 1-D array.

  **Returns**

  - `data`: ndarray
    
    A new `ndarray` holding the non-masked data is returned.

  **Notes**

  The result is not a MaskedArray!

  **Examples**

  ```python
def x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<type 'numpy.ndarray'>
```

**Methods**

- `.filled(fill_value=None)`
  
  Return a copy of self, with masked values filled with a given value.

  **Parameters**

  - `fill_value`: scalar, optional
    
    The value to use for invalid entries (None by default). If None, the `fill_value` attribute of the array is used instead.

  **Returns**

  - `filled_array`: ndarray
    
    A copy of self, with masked values filled with a given value.

**Notes**

The result is not a MaskedArray!
A copy of `self` with invalid entries replaced by `fill_value` (be it the function argument or the attribute of `self`.

**Notes**

The result is **not** a MaskedArray!

**Examples**

```python
>>> x = np.ma.array([1, 2, 3, 4, 5], mask=[0, 0, 1, 0, 1], fill_value=-999)
>>> x.filled()
array([1, 2, -999, 4, -999])
```

Subclassing is preserved. This means that if the data part of the masked array is a matrix, `filled` returns a matrix:

```python
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.filled()
matrix([[ 1, 999999],
        [999999, 4]])
```

> to another object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.MaskedArray.tofile(fid[, sep, format])</code></td>
<td>Save a masked array to a file in binary format.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.tolist([fill_value])</code></td>
<td>Return the data portion of the masked array as a hierarchical Python list.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.torecords()</code></td>
<td>Transforms a masked array into a flexible-type array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.tostring([fill_value, order])</code></td>
<td>Return the array data as a string containing the raw bytes in the array.</td>
</tr>
</tbody>
</table>

**Warning:** This function is not implemented yet.

Raises

- `NotImplementedError`        
  When `tofile` is called.

**MaskedArray.tolist** (**fill_value=** `None`)        
Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to `fill_value`. If `fill_value` is None, the corresponding entries in the output list will be `None`.

**Parameters**

- `fill_value` : scalar, optional
  The value to use for invalid entries. Default is None.

**Returns**

- `result` : list
  The Python list representation of the masked array.
Examples

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]
```  
```python
>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]
```

**MaskedArray.torecords()**

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

- the _data_ field stores the _data_ part of the array.
- the _mask_ field stores the _mask_ part of the array.

**Parameters**

None

**Returns**

record : ndarray

A new flexible-type `ndarray` with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

**Notes**

A side-effect of transforming a masked array into a flexible `ndarray` is that meta information (``fill_value``, ...) will be lost.

**Examples**

```python
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
```  
```python
>>> print x.toflex()
[[[1, False) (2, True) (3, False)]
[[4, True) (5, False) (6, True)]
[[7, False) (8, True) (9, False)]]
```

**MaskedArray.tostring**

Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

**Parameters**

- `fill_value` : scalar, optional
  Value used to fill in the masked values. Default is None, in which case `MaskedArray.fill_value` is used.

  Order of the data item in the copy. Default is ‘C’.
  - ‘C’ – C order (row major).
  - ‘F’ – Fortran order (column major).
• ‘A’ – Any, current order of array.
• None – Same as ‘A’.

See Also:
ndarray.tostring, tolist, tofile

Notes
As for ndarray.tostring, information about the shape, dtype, etc., but also about fill_value, will be lost.

Examples
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tostring()
'\x01\x00\x00\x00?B\x0f\x00?B\x0f\x00\x04\x00\x00\x00'

Pickling and unpickling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.dump(a, F)</td>
<td>Pickle a masked array to a file.</td>
</tr>
<tr>
<td>ma.dumps(a)</td>
<td>Return a string corresponding to the pickling of a masked array.</td>
</tr>
<tr>
<td>ma.load(F)</td>
<td>Wrapper around cPickle.load which accepts either a file-like object</td>
</tr>
<tr>
<td>ma.loads(strg)</td>
<td>Load a pickle from the current string.</td>
</tr>
</tbody>
</table>

numpy.ma.dump(a, F)

Pickle a masked array to a file.

This is a wrapper around cPickle.dump.

Parameters
a : MaskedArray
  The array to be pickled.
F : str or file-like object
  The file to pickle a to. If a string, the full path to the file.

numpy.ma.dumps(a)

Return a string corresponding to the pickling of a masked array.

This is a wrapper around cPickle.dumps.

Parameters
a : MaskedArray
  The array for which the string representation of the pickle is returned.

numpy.ma.load(F)

Wrapper around cPickle.load which accepts either a file-like object or a filename.

Parameters
F : str or file
  The file or file name to load.

See Also:
dump
  Pickle an array
Notes

This is different from `numpy.load`, which does not use cPickle but loads the NumPy binary .npy format.

`numpy.ma.loads(strg)`

Load a pickle from the current string.

The result of `cPickle.loads(strg)` is returned.

**Parameters**

strg : str

The string to load.

**See Also:**

`dumps`

Return a string corresponding to the pickling of a masked array.

Filling a masked array

```
ma.common_fill_value(a, b)  # Return the common filling value of two masked arrays, if any.
ma.default_fill_value(obj)  # Return the default fill value for the argument object.
ma.maximum_fill_value(obj)  # Return the minimum value that can be represented by the dtype of an object.
ma.maximum_fill_value(obj)  # Return the minimum value that can be represented by the dtype of an object.
ma.set_fill_value(a, fill_value)  # Set the filling value of a, if a is a masked array.
ma.MaskedArray.get_fill_value()  # Return the filling value of the masked array.
ma.MaskedArray.set_fill_value([value])  # Set the filling value of the masked array.
ma.MaskedArray.fill_value  # Filling value.
```

```
numpy.ma.common_fill_value(a, b)
# Return the common filling value of two masked arrays, if any.

If a.fill_value == b.fill_value, return the fill value, otherwise return None.

**Parameters**

a, b : MaskedArray

The masked arrays for which to compare fill values.

**Returns**

fill_value : scalar or None

The common fill value, or None.

**Examples**

```python
>>> x = np.ma.array([0, 1.], fill_value=3)
>>> y = np.ma.array([0, 1.], fill_value=3)
>>> np.ma.common_fill_value(x, y)
3.0
```

```
numpy.ma.default_fill_value(obj)
# Return the default fill value for the argument object.

The default filling value depends on the datatype of the input array or the type of the input scalar:

3.19. Masked array operations  823
### Parameters

**obj**: ndarray, dtype or scalar

The array data-type or scalar for which the default fill value is returned.

### Returns

**fill_value**: scalar

The default fill value.

### Examples

```python
>>> np.ma.default_fill_value(1)
999999
>>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
1e+20
>>> np.ma.default_fill_value(np.dtype(complex))
(1e+20+0j)
```

### numpy.ma.maximum_fill_value(obj)

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

#### Parameters

**obj**: {ndarray, dtype}

An object that can be queried for it’s numeric type.

#### Returns

**val**: scalar

The minimum representable value.

#### Raises

**TypeError**

If `obj` isn’t a suitable numeric type.

### See Also:

- **minimum_fill_value**
  The inverse function.

- **set_fill_value**
  Set the filling value of a masked array.

- **MaskedArray.fill_value**
  Return current fill value.
Examples

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.

```python
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

`numpy.ma.maximum_fill_value(obj)`

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

**Parameters**

- `obj`: {ndarray, dtype}
  
  An object that can be queried for it’s numeric type.

**Returns**

- `val`: scalar
  
  The minimum representable value.

**Raises**

- `TypeError`
  
  If `obj` isn’t a suitable numeric type.

**See Also:**

- `minimum_fill_value`
  
  The inverse function.

- `set_fill_value`
  
  Set the filling value of a masked array.

- `MaskedArray.fill_value`
  
  Return current fill value.

**Examples**

```python
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf

```
import numpy.ma as ma
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

**numpy.ma.set_fill_value** *(a, fill_value)*

Set the filling value of `a`, if `a` is a masked array.

This function changes the fill value of the masked array `a` in place. If `a` is not a masked array, the function returns silently, without doing anything.

**Parameters**

- `a` : array_like
  Input array.
- `fill_value` : dtype
  Filling value. A consistency test is performed to make sure the value is compatible with the dtype of `a`.

**Returns**

- None
  Nothing returned by this function.

**See Also:**

- `maximum_fill_value`
  Return the default fill value for a dtype.
- `MaskedArray.fill_value`
  Return current fill value.
- `MaskedArray.set_fill_value`
  Equivalent method.

**Examples**

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True  True  True False False],
             fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data = [-- -- -- 3 4],
             mask = [ True  True  True False False],
             fill_value=-999)
```

Nothing happens if `a` is not a masked array.

```
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
```
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])

MaskedArray.get_fill_value()
Return the filling value of the masked array.

Returns
fill_value : scalar
The filling value.

Examples

```python
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...     np.ma.array([0, 1], dtype=dt).get_fill_value()
... 999999
999999
le+20
(le+20+0j)
```

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.get_fill_value()
-inf
```

MaskedArray.set_fill_value(value=None)
Set the filling value of the masked array.

Parameters
value : scalar, optional
The new filling value. Default is None, in which case a default based on the data type is used.

See Also:

ma.set_fill_value
Equivalent function.

Examples

```python
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.set_fill_value(np.pi)
>>> x.fill_value
3.1415926535897931
```

Reset to default:

```python
>>> x.set_fill_value()
>>> x.fill_value
le+20
```
MaskedArray.fill_value
Filling value.

3.19.7 Masked arrays arithmetics

Arithmetics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.anom</td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td>ma.anomalies</td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td>ma.average</td>
<td>Return the weighted average of array over the given axis.</td>
</tr>
<tr>
<td>ma.conjugate</td>
<td>Return the complex conjugate, element-wise.</td>
</tr>
<tr>
<td>ma.corrcoef</td>
<td>Return correlation coefficients of the input array.</td>
</tr>
<tr>
<td>ma.cov</td>
<td>Estimate the covariance matrix.</td>
</tr>
<tr>
<td>ma.cumsum</td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td>ma.cumprod</td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td>ma.median</td>
<td>Compute the median along the specified axis.</td>
</tr>
<tr>
<td>ma.power</td>
<td>Returns element-wise base array raised to power from second array.</td>
</tr>
<tr>
<td>ma.prod</td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td>ma.std</td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td>ma.sum</td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td>ma.var</td>
<td>Compute the variance along the specified axis.</td>
</tr>
<tr>
<td>MaskedArray.anom</td>
<td>Compute the anomalies (deviations from the arithmetic mean) along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.cumsum</td>
<td>Return the cumulative sum of the elements along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.cumprod</td>
<td>Return the cumulative product of the elements along the given axis.</td>
</tr>
<tr>
<td>MaskedArray.mean</td>
<td>Returns the average of the array elements.</td>
</tr>
<tr>
<td>MaskedArray.prod</td>
<td>Return the product of the array elements over the given axis.</td>
</tr>
<tr>
<td>MaskedArray.std</td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td>MaskedArray.sum</td>
<td>Return the sum of the array elements over the given axis.</td>
</tr>
<tr>
<td>MaskedArray.var</td>
<td>Compute the variance along the specified axis.</td>
</tr>
</tbody>
</table>

numpy.ma.anom(self, axis=None, dtype=None) = <numpy.ma.core._frommethod instance at 0x2819f38>

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters

axis : int, optional

Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional

Type to use in computing the variance. For arrays of integer type

the default is float32; for arrays of float types it is the same as the array type.

See Also:

mean
Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1.  0.  1.],
             mask = False,
             fill_value = 1e+20)
```

```
numpy.ma.anomalies(self, axis=None, dtype=None) = <numpy.ma.core._frommethod instance at 0x2819f38>
```

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters

- **axis**: int, optional
  - Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

- **dtype**: dtype, optional
  - Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

- **mean**
  - Compute the mean of the array.

Examples

```python
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1.  0.  1.],
             mask = False,
             fill_value = 1e+20)
```

```
numpy.ma.average(a, axis=None, weights=None, returned=False)
```

Return the weighted average of array over the given axis.

Parameters

- **a**: array_like
  - Data to be averaged. Masked entries are not taken into account in the computation.

- **axis**: int, optional
  - Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- **weights**: array_like, optional
  - The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of `a` along the given axis) or of the same shape as `a`. If `weights=None`, then all data in `a` are assumed to have a weight equal to one. If `weights` is complex, the imaginary parts are ignored.

- **returned**: bool, optional
Flag indicating whether a tuple \((\text{result, sum of weights})\) should be returned as output (True), or just the result (False). Default is False.

**Returns**

\(\text{average, [sum of weights]}\) : (tuple of) scalar or MaskedArray

The average along the specified axis. When returned is \(\text{True}\), return a tuple with the average as the first element and the sum of the weights as the second element. The return type is \(\text{np.float64}\) if \(a\) is of integer type, otherwise it is of the same type as \(a\). If returned, \(\text{sum of weights}\) is of the same type as \(\text{average}\).

**Examples**

```python
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25

>>> x = np.ma.arange(6.).reshape(3, 2)
>>> print x
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]]
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
          returned=True)
>>> print avg
[2.66666666667 3.66666666667]
```

**numpy.ma.conjugate** \((x[, out])\) = <numpy.ma.core._MaskedUnaryOperation instance at 0x272bfc8>

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

**Parameters**

\(x\) : array_like

Input value.

**Returns**

\(y\) : ndarray

The complex conjugate of \(x\), with same dtype as \(y\).

**Examples**

```python
>>> np.conjugate(1+2j)
(1-2j)

>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j, 0.-0.j],
       [ 0.-0.j, 1.-1.j]])
```

**numpy.ma.corrcoef** \((x, y=None[, rowvar=True, bias=False, allow_masked=True, ddof=None])\)

Return correlation coefficients of the input array.

Except for the handling of missing data this function does the same as \(\text{numpy.corrcoef}\). For more details and examples, see \(\text{numpy.corrcoef}\).

**Parameters**

\(x\) : array_like
A 1-D or 2-D array containing multiple variables and observations. Each row of \( x \) represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

\( y \): array_like, optional

An additional set of variables and observations. \( y \) has the same shape as \( x \).

`rowvar` : bool, optional

If `rowvar` is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

`bias` : bool, optional

Default normalization (False) is by \((N-1)\), where \( N \) is the number of observations given (unbiased estimate). If `bias` is 1, then normalization is by \( N \). This keyword can be overridden by the keyword `ddof` in numpy versions >= 1.5.

`allow_masked` : bool, optional

If True, masked values are propagated pair-wise: if a value is masked in \( x \), the corresponding value is masked in \( y \). If False, raises an exception.

`ddof` : {None, int}, optional

New in version 1.5. If not None normalization is by \((N \ - \ ddof)\), where \( N \) is the number of observations; this overrides the value implied by `bias`. The default value is None.

See Also:

- `numpy.corrcoef`
  Equivalent function in top-level NumPy module.

- `cov`
  Estimate the covariance matrix.

- `numpy.ma.cov`(x, y=None, rowvar=True, bias=False, allow_masked=True, ddof=None)
  Estimate the covariance matrix.

Except for the handling of missing data this function does the same as `numpy.cov`. For more details and examples, see `numpy.cov`.

By default, masked values are recognized as such. If \( x \) and \( y \) have the same shape, a common mask is allocated: if \( x[i, j] \) is masked, then \( y[i, j] \) will also be masked. Setting `allow_masked` to False will raise an exception if values are missing in either of the input arrays.

**Parameters**

- \( x \) : array_like
  A 1-D or 2-D array containing multiple variables and observations. Each row of \( x \) represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.

- \( y \) : array_like, optional
  An additional set of variables and observations. \( y \) has the same form as \( x \).

- `rowvar` : bool, optional
If `rowvar` is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

**bias**: bool, optional

Default normalization (False) is by \((N-1)\), where \(N\) is the number of observations given (unbiased estimate). If `bias` is True, then normalization is by \(N\). This keyword can be overridden by the keyword `ddof` in numpy versions >= 1.5.

**allow_masked**: bool, optional

If True, masked values are propagated pair-wise: if a value is masked in \(x\), the corresponding value is masked in \(y\). If False, raises a `ValueError` exception when some values are missing.

**ddof**: {None, int}, optional

New in version 1.5. If not None normalization is by \((N - ddof)\), where \(N\) is the number of observations; this overrides the value implied by `bias`. The default value is None.

**Raises**

`ValueError`

Raised if some values are missing and `allow_masked` is False.

**See Also:**

`numpy.cov`  
`numpy.ma.cumsum`  

```
832 Chapter 3. Routines
```
Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

```
numpy.ma.cumprod(self, axis=None, dtype=None, out=None)
```

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

- **axis** : {None, -1, int}, optional
  
  Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

- **dtype** : {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out** : ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns

- **cumprod** : ndarray
  
  A new array holding the result is returned unless out is specified, in which case a reference to out is returned.

Notes

The mask is lost if out is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

```
numpy.ma.mean(self, axis=None, dtype=None, out=None)
```

Returns the average of the array elements.

Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to numpy.mean for the full documentation.

Parameters

- **a** : array_like

  Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.

- **axis** : int, optional
Axis along which the means are computed. The default is to compute the mean of the flattened array.

**dtype** : dtype, optional

Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dtype.

**out** : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **mean** : ndarray, see dtype parameter above
  
  If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

- `numpy.ma.mean`
  Equivalent function.

- `numpy.mean`
  Equivalent function on non-masked arrays.

- `numpy.ma.average`
  Weighted average.

**Examples**

```python
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
mask = [False False True],
fill_value = 999999)
>>> a.mean()
1.5
```

**numpy.ma.median** (`a, axis=None, out=None, overwrite_input=False`)

Compute the median along the specified axis.

**Returns**

Returns the median of the array elements.

**Parameters**

- **a** : array_like
  
  Input array or object that can be converted to an array.

- **axis** : int, optional
  
  Axis along which the medians are computed. The default (None) is to compute the median along a flattened version of the array.

- **out** : ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

- **overwrite_input** : bool, optional
  
  If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to
preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if overwrite_input is True, and the input is not already an ndarray, an error will be raised.

Returns
median : ndarray
A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return data-type is float64 for integers and floats smaller than float64, or the input data-type, otherwise.

See Also:
mean

Notes
Given a vector V with N non masked values, the median of V is the middle value of a sorted copy of V (Vs) - i.e. Vs[(N-1)/2], when N is odd, or {Vs[N/2 - 1] + Vs[N/2]}/2 when N is even.

Examples
>>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)
>>> np.ma.extras.median(x)
1.5

>>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
>>> np.ma.extras.median(x)
2.5
>>> np.ma.extras.median(x, axis=-1, overwrite_input=True)
mixed_array(data = [ 2. 5.],
             mask = False,
             fill_value = 1e+20)

numpy.ma.power (a, b, third=None)
Returns element-wise base array raised to power from second array.

This is the masked array version of numpy.power. For details see numpy.power.

See Also:
numpy.power

Notes
The out argument to numpy.power is not supported, third has to be None.

numpy.ma.prod (self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820560>
Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

Parameters
axis : {None, int}, optional
    Axis over which the product is taken. If None is used, then the product is over all the array elements.

dtype : {None, dtype}, optional
    Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision
less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

**out**: {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **product_along_axis**: {array, scalar}, see dtype parameter above.

Returns an array whose shape is the same as a with the specified axis removed. Returns a 0d array when a is 1d or axis=None. Returns a reference to the specified output array if specified.

**See Also:**

- **prod**
  equivalent function

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> np.prod([1.,2.])
2.0
>>> np.prod([1.,2.], dtype=np.int32)
2
>>> np.prod([[1.,2.],[3.,4.]])
24.0
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
```

numpy.ma.std(self, axis=None, dtype=None, out=None, ddof=0) = <numpy.ma.core._frommethod instance at 0x2820878>

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

- **a**: array_like
  Calculate the standard deviation of these values.

- **axis**: int, optional
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- **dtype**: dtype, optional
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- **ddof**: int, optional
Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof = 0 \).

**keepdims**: bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \( arr \).

**Returns**

- **standard_deviation**: ndarray, see dtype parameter above.

  If \( out \) is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

**See Also**:

- var, mean, nanmean, nanstd, nanvar

- numpy.doc.ufuncs
  - Section “Output arguments”

**Notes**

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( std = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean()}**2))} \).

The average squared deviation is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of the infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( ddof=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( std \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( std \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the \( dtype \) keyword can alleviate this issue.

**Examples**

```python
c = np.array([[1, 2], [3, 4]])
c
1.1180339887498949
>>> np.std(c, axis=0)
array([ 1., 1.])
```

In single precision, \( \text{std}() \) can be inaccurate:

```python
c = np.zeros((2,512*512), dtype=np.float32)
c[0, :] = 1.0
c[1, :] = 0.1
c
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:

```python
c = np.zeros((2,512*512), dtype=np.float64)
c
0.4499999992552653
```
numpy.ma.sum(self, axis=None, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x2820908>

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

Parameters
axis : {None, -1, int}, optional
Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.
dtype : {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.
out : {None, ndarray}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

Returns
sum_along_axis : MaskedArray or scalar
An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

Examples
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0] + [1,0]*4)
>>> print x
[[1 -- 3]
 [-- 5 --]
 [7 -- 9]]
>>> print x.sum()
25
>>> print x.sum(axis=1)
[4 5 16]
>>> print x.sum(axis=0)
[8 5 12]
>>> print type(x.sum(axis=0, dtype=np.int64)[0])
<type 'numpy.int64'>

numpy.ma.var(self, axis=None, dtype=None, out=None, ddof=0) = <numpy.ma.core._frommethod instance at 0x2820a28>

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters
a : array_like
Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.
axis : int, optional
Axis along which the variance is computed. The default is to compute the variance of the flattened array.
**NumPy Reference, Release 1.8.1**

**dtype**: data-type, optional

Type to use in computing the variance. For arrays of integer type the default is `float32`; for arrays of float types it is the same as the array type.

**out**: ndarray, optional

Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof**: int, optional

“Delta Degrees of Freedom”: the divisor used in the calculation is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default ddof is zero.

**keepdims**: bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

**Returns**

**variance**: ndarray, see dtype parameter above

If out=None, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**See Also:**

std, mean, nanmean, nanstd, nanvar

**numpy.doc.ufuncs**

Section “Output arguments”

**Notes**

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\lvert x - x.\text{mean()}\rvert^2) \).

The mean is normally calculated as \( x.\text{sum()}/N \), where \( N = \text{len}(x) \). If, however, ddof is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, ddof=1 provides an unbiased estimator of the variance of a hypothetical infinite population. ddof=0 provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1. , 1.])
>>> np.var(a, axis=1)
array([0.25, 0.25])
```

In single precision, var() can be inaccurate:
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0,:] = 1.0
>>> a[1,:] = 0.1
>>> np.var(a)
0.20405951142311096

Computing the variance in float64 is more accurate:

>>> np.var(a, dtype=np.float64)
0.20249999932997387
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001

MaskedArray.anom(axis=None, dtype=None)

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

Parameters

axis : int, optional
    Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

dtype : dtype, optional
    Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

See Also:

mean
    Compute the mean of the array.

Examples

>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data = [-1. 0. 1.],
    mask = False,
    fill_value = 1e+20)

MaskedArray.cumprod(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis. The cumulative product is taken over the flattened array by default, otherwise over the specified axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Parameters

axis : {None, -1, int}, optional
    Axis along which the product is computed. The default (axis = None) is to compute over the flattened array.

dtype : {None, dtype}, optional
    Determines the type of the returned array and of the accumulator where the elements are multiplied. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.
**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

**cumprod**: ndarray

A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> print marr.cumsum()
[0 1 3 -- -- -- 9 16 24 33]
```

**cumsum**

Returns the cumulative sum of the elements along the given axis. The cumulative sum is calculated over the flattened array by default, otherwise over the specified axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

**Parameters**

**axis**: {None, -1, int}, optional

Axis along which the sum is computed. The default (`axis = None`) is to compute over the flattened array. `axis` may be negative, in which case it counts from the last to the first axis.

**dtype**: {None, dtype}, optional

Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the dtype of `a`, unless `a` has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

**cumsum**: ndarray.

A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned.

**Notes**

The mask is lost if `out` is not a valid MaskedArray!

Arithmetic is modular when using integer types, and no error is raised on overflow.
Masked entries are ignored. The average is taken over the flattened array by default, otherwise over the specified axis. Refer to `numpy.mean` for the full documentation.

**Parameters**

- **a**: array_like
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- **dtype**: dtype, optional
  Type to use in computing the mean. For integer inputs, the default is float64; for floating point, inputs it is the same as the input dttype.

- **out**: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- **mean**: ndarray, see dtype parameter above
  If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See Also:**

- `numpy.ma.mean`
  Equivalent function.

- `numpy.mean`
  Equivalent function on non-masked arrays.

- `numpy.ma.average`
  Weighted average.

**Examples**

```python
>>> a = np.ma.array([1, 2, 3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False True],
            fill_value = 999999)
>>> a.mean()
1.5
```

**MaskedArray.prod** *(axis=None, dtype=None, out=None)*

Return the product of the array elements over the given axis. Masked elements are set to 1 internally for computation.

**Parameters**

- **axis**: {None, int}, optional
  Axis over which the product is taken. If None is used, then the product is over all the array elements.

- **dtype**: {None, dtype}, optional
Determines the type of the returned array and of the accumulator where the elements are multiplied. If `dtype` has the value `None` and the type of `a` is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the `dtype` is the same as that of `a`.

**out** : {None, array}, optional

Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

- `product_along_axis` : {array, scalar}, see `dtype` parameter above.

Returns an array whose shape is the same as `a` with the specified axis removed. Returns a 0d array when `a` is 1d or `axis=None`. Returns a reference to the specified output array if specified.

**See Also:**

- `prod`

  equivalent function

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> np.prod([1., 2.])
2.0
>>> np.prod([1., 2.], dtype=np.int32)
2
>>> np.prod([[1., 2.], [3., 4.]])
24.0
>>> np.prod([[1., 2.], [3., 4.]], axis=1)
array([ 2., 12.])
```

```
MaskedArray.std(axis=None, dtype=None, out=None, ddof=0)
```

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

- `a` : array_like

  Calculate the standard deviation of these values.

- `axis` : int, optional

  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- `dtype` : dtype, optional

  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- `out` : ndarray, optional

  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.
ddof : int, optional

Means Delta Degrees of Freedom. The divisor used in calculations is \( N - \text{ddof} \), where \( N \) represents the number of elements. By default ddof is zero.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

Returns

standard_deviation : ndarray, see dtype parameter above.

If out is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

See Also:

var, mean, nanmean, nanstd, nanvar

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - \text{mean}(x))^2)} \).

The average squared deviation is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, ddof is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, ddof=1 provides an unbiased estimator of the variance of the infinite population. ddof=0 provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with ddof=1, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, std takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the std is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949

>>> np.std(a, axis=0)
array([ 1., 1.])

>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, std() can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1

>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in float64 is more accurate:
NumPy Reference, Release 1.8.1

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

**MaskedArray.sum(axis=None, dtype=None, out=None)**

Return the sum of the array elements over the given axis. Masked elements are set to 0 internally.

**Parameters**

- **axis**: {None, -1, int}, optional
  
  Axis along which the sum is computed. The default (axis = None) is to compute over the flattened array.

- **dtype**: {None, dtype}, optional
  
  Determines the type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and the type of a is an integer type of precision less than the default platform integer, then the default platform integer precision is used. Otherwise, the dtype is the same as that of a.

- **out**: {None, ndarray}, optional
  
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**Returns**

- **sum_along_axis**: MaskedArray or scalar
  
  An array with the same shape as self, with the specified axis removed. If self is a 0-d array, or if axis is None, a scalar is returned. If an output array is specified, a reference to out is returned.

**Examples**

```python
>>> x = np.ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mask=[0, 1, 0]*4)
```

```python
>>> print(x)
[[1 -- 3]
[-- 5 --]
[7 -- 9]]
```

```python
>>> print(x.sum())
25
```

```python
>>> print(x.sum(axis=1))
[4 5 16]
```

```python
>>> print(x.sum(axis=0))
[8 5 12]
```

```python
>>> print(type(x.sum(axis=0, dtype=np.int64)[0]))
type('numpy.int64')
```

**MaskedArray.var(axis=None, dtype=None, out=None, ddof=0)**

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

- **a**: array_like
  
  Array containing numbers whose variance is desired. If a is not an array, a conversion is attempted.

- **axis**: int, optional
  
  Axis along which the variance is computed. The default (axis = None) is to compute over the flattened array.
Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional
Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

out : ndarray, optional
Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof : int, optional
“Delta Degrees of Freedom”: the divisor used in the calculation is $N - ddof$, where $N$ represents the number of elements. By default $ddof$ is zero.

keepdims : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

Returns

variance : ndarray, see dtype parameter above
If out=None, returns a new array containing the variance; otherwise, a reference to the output array is returned.

See Also:

std, mean, nanmean, nanstd, nanvar

numpy.doc.ufuncs
Section “Output arguments”

Notes
The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(|x - x.\text{mean()}|^2) \).

The mean is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( ddof \) is specified, the divisor \( N - ddof \) is used instead. In standard statistical practice, \( ddof=1 \) provides an unbiased estimator of the variance of a hypothetical infinite population. \( ddof=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1,2],[3,4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([ 1., 1.])
>>> np.var(a, axis=1)
array([ 0.25, 0.25])
```
In single precision, `var()` can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
```

```python
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

### Minimum/maximum

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argmax(a[, axis, fill_value])</code></td>
<td>Function version of the eponymous method.</td>
</tr>
<tr>
<td><code>ma.argmin(a[, axis, fill_value])</code></td>
<td>Returns array of indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.max(obj[, axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.min(obj[, axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.ptp(obj[, axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. range of values)</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmax([axis, fill_value, out])</code></td>
<td>Returns array of indices of the maximum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argmin([axis, fill_value, out])</code></td>
<td>Return array of indices to the minimum values along the given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.max([axis, out, fill_value])</code></td>
<td>Return the maximum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.min([axis, out, fill_value])</code></td>
<td>Return the minimum along a given axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.ptp([axis, out, fill_value])</code></td>
<td>Return (maximum - minimum) along the the given dimension (i.e. range of values).</td>
</tr>
</tbody>
</table>

numpy.ma.**argmax**(a, axis=None, fill_value=None)

Function version of the eponymous method.

Example:

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
```
```python
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

numpy.ma.max(obj=None, axis=None, out=None, fill_value=None)

Return the maximum along a given axis.

Parameters

- **axis**: {None, int}, optional
  
  Axis along which to operate. By default, *axis* is None and the flattened input is used.

- **out**: array_like, optional
  
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- **fill_value**: {var}, optional
  
  Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns

- **amax**: array_like
  
  New array holding the result. If *out* was specified, *out* is returned.

See Also:

- maximum_fill_value

numpy.ma.min(obj=None, axis=None, out=None, fill_value=None)

Return the minimum along a given axis.

Parameters

- **axis**: {None, int}, optional
  
  Axis along which to operate. By default, *axis* is None and the flattened input is used.

- **out**: array_like, optional
  
  Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

- **fill_value**: {var}, optional
  
  Value used to fill in the masked values. If None, use the output of minimum_fill_value().

Returns

- **amin**: array_like
  
  New array holding the result. If *out* was specified, *out* is returned.

See Also:

- minimum_fill_value

numpy.ma.ptp(obj=None, axis=None, out=None, fill_value=None)

Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).
Parameters

axis : {None, int}, optional

Axis along which to find the peaks. If None (default) the flattened array is used.

out : {None, array_like}, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

fill_value : {var}, optional

Value used to fill in the masked values.

Returns

ptp : ndarray

A new array holding the result, unless out was specified, in which case a reference to out is returned.

Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

Parameters

axis : {None, integer}

If None, the index is into the flattened array, otherwise along the specified axis

fill_value : {var}, optional

Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

index_array : {integer_array}

Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a.argmin()
0
>>> a.argmin(0)
array([0, 0, 0])
>>> a.argmin(1)
array([0, 0])
```

3.19. Masked array operations
out : {None, array}, optional

Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

Returns

{ndarray, scalar}

If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

Examples

>>> x = np.ma.array(arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> print x
[[-- --]
 [2 3]]
>>> print x.argmin(axis=0, fill_value=-1)
[0 0]
>>> print x.argmin(axis=0, fill_value=9)
[1 1]

MaskedArray.max (axis=None, out=None, fill_value=None)

Return the maximum along a given axis.

Parameters

axis : {None, int}, optional

Axis along which to operate. By default, axis is None and the flattened input is used.

out : array_like, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

fill_value : {var}, optional

Value used to fill in the masked values. If None, use the output of maximum_fill_value().

Returns

amax : array_like

New array holding the result. If out was specified, out is returned.

See Also:

maximum_fill_value

Returns the maximum filling value for a given datatype.

MaskedArray.min (axis=None, out=None, fill_value=None)

Return the minimum along a given axis.

Parameters

axis : {None, int}, optional

Axis along which to operate. By default, axis is None and the flattened input is used.

out : array_like, optional

850 Chapter 3. Routines
Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value** : {var}, optional
Value used to fill in the masked values. If None, use the output of `minimum_fill_value`.

**Returns**

**amin**: array_like
New array holding the result. If `out` was specified, `out` is returned.

**See Also:**

`minimum_fill_value`
Returns the minimum filling value for a given datatype.

**MaskedArray.ptp** (axis=None, **out**=None, fill_value=None)
Return (maximum - minimum) along the the given dimension (i.e. peak-to-peak value).

**Parameters**

**axis** : {None, int}, optional
Axis along which to find the peaks. If None (default) the flattened array is used.

**out** : {None, array_like}, optional
Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**fill_value** : {var}, optional
Value used to fill in the masked values.

**Returns**

**ptp** : ndarray.
A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

**Sorting**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.argsort(a[, axis, kind, order, fill_value])</code></td>
<td>Return an ndarray of indices that sort the array along the specified axis.</td>
</tr>
<tr>
<td><code>ma.sort(a[, axis, kind, endwith, ...])</code></td>
<td>Sort the array, in-place</td>
</tr>
<tr>
<td><code>ma.MaskedArray.argsort([axis, kind, order, ...])</code></td>
<td>Return an ndarray of indices that sort the array along the specified axis.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.sort([axis, kind, order, ...])</code></td>
<td>Sort the array, in-place</td>
</tr>
</tbody>
</table>

**numpy.ma.argsort** (a, axis=None, kind='quicksort', order=None, fill_value=None)
Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to `fill_value`.

**Parameters**

**axis** : int, optional
Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.

**fill_value** : var, optional
Value used to fill the array before sorting. The default is the fill_value attribute of the input array.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm.

order : list, optional
    When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

index_array : ndarray, int
    Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:

sort
    Describes sorting algorithms used.
lexsort
    Indirect stable sort with multiple keys.
ndarray.sort
    Inplace sort.

Notes

See sort for notes on the different sorting algorithms.

Examples

>>> a = np.ma.array([3,2,1], mask=[False, False, True])

>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
            fill_value = 999999)

>>> a.argsort()
array([1, 0, 2])

numpy.ma.sort (a, axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)
Sort the array, in-place

Parameters

a : array_like
    Array to be sorted.

axis : int, optional
    Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm. Default is 'quicksort'.

order : list, optional
    When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.
endwith : {True, False}, optional

Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

fill_value : {var}, optional

Value used internally for the masked values. If fill_value is not None, it supersedes endwith.

Returns

sorted_array : ndarray

Array of the same type and shape as a.

See Also:

ndarray.sort

Method to sort an array in-place.

argsort

Indirect sort.

lexsort

Indirect stable sort on multiple keys.

searchsorted

Find elements in a sorted array.

Notes

See sort for notes on the different sorting algorithms.

Examples

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Default
>>> a.sort()
>>> print a
[1 3 5 -- --]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # Put missing values in the front
>>> a.sort(endwith=False)
>>> print a
[-- -- 1 3 5]

>>> a = ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
>>> # fill_value takes over endwith
>>> a.sort(endwith=False, fill_value=3)
>>> print a
[1 -- -- 3 5]

MaskedArray[argsort(axis=None, kind='quicksort', order=None, fill_value=None)]

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to fill_value.

Parameters

axis : int, optional

Axis along which to sort. The default is -1 (last axis). If None, the flattened array is used.
fill_value : var, optional
    Value used to fill the array before sorting. The default is the fill_value attribute of the input array.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm.

order : list, optional
    When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns
index_array : ndarray, int
    Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:
sort
    Describes sorting algorithms used.

lexsort
    Indirect stable sort with multiple keys.

ndarray.sort
    Inplace sort.

Notes
See sort for notes on the different sorting algorithms.

Examples
>>> a = np.ma.array([3, 2, 1], mask=[False, False, True])
>>> a
masked_array(data = [3 2 --],
             mask = [False False True],
             fill_value = 999999)
>>> a.argsort()
aarray([1, 0, 2])

MaskedArray.sort (axis=-1, kind='quicksort', order=None, endwith=True, fill_value=None)
Sort the array, in-place

Parameters
a : array_like
    Array to be sorted.
axis : int, optional
    Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm. Default is 'quicksort'.

order : list, optional
When a is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

**endwith**: {True, False}, optional

Whether missing values (if any) should be forced in the upper indices (at the end of the array) (True) or lower indices (at the beginning).

**fill_value**: {var}, optional

Value used internally for the masked values. If fill_value is not None, it supersedes endwith.

**Returns**

**sorted_array**: ndarray

Array of the same type and shape as a.

**See Also:**

* *ndarray.sort* Method to sort an array in-place.

* *argsort* Indirect sort.

* *lexsort* Indirect stable sort on multiple keys.

* *searchsorted* Find elements in a sorted array.

**Notes**

See sort for notes on the different sorting algorithms.

**Examples**

```python
generate code here
```

**Algebra**

* *ma.diag(v[, k])* Extract a diagonal or construct a diagonal array.
### Table 3.104 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ma.dot(a, b[, strict])</code></td>
<td>Return the dot product of two arrays.</td>
</tr>
<tr>
<td><code>ma.identity(n[, dtype])</code></td>
<td>Return the identity array.</td>
</tr>
<tr>
<td><code>ma.inner(a, b)</code></td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td><code>ma.identity(n[, dtype])</code></td>
<td>Return the identity array.</td>
</tr>
<tr>
<td><code>ma.inner(a, b)</code></td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td><code>ma.innerproduct(a, b)</code></td>
<td>Inner product of two arrays.</td>
</tr>
<tr>
<td><code>ma.outer(a, b)</code></td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td><code>ma.outerproduct(a, b)</code></td>
<td>Compute the outer product of two vectors.</td>
</tr>
<tr>
<td><code>ma.trace(self[, offset, axis1, axis2, ...])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>ma.transpose(a[, axes])</code></td>
<td>Permute the dimensions of an array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.trace([offset, axis1, axis2, ...])</code></td>
<td>Return the sum along diagonals of the array.</td>
</tr>
<tr>
<td><code>ma.MaskedArray.transpose(*axes)</code></td>
<td>Returns a view of the array with axes transposed.</td>
</tr>
</tbody>
</table>

### `numpy.ma.diag(v, k=0)`

Extract a diagonal or construct a diagonal array.

This function is the equivalent of `numpy.diag` that takes masked values into account, see `numpy.diag` for details.

**See Also:**

- `numpy.diag`
  
  Equivalent function for ndarrays.

### `numpy.ma.dot(a, b, strict=False)`

Return the dot product of two arrays.

**Note:** Works only with 2-D arrays at the moment.

This function is the equivalent of `numpy.dot` that takes masked values into account, see `numpy.dot` for details.

**Parameters**

- `a, b` : ndarray
  
  Inputs arrays.

- `strict` : bool, optional
  
  Whether masked data are propagated (True) or set to 0 (False) for the computation. Default is False. Propagating the mask means that if a masked value appears in a row or column, the whole row or column is considered masked.

**See Also:**

- `numpy.dot`
  
  Equivalent function for ndarrays.

**Examples**

```python
def main():
    a = ma.array([[1, 2, 3], [4, 5, 6]], mask=[[True, False, False], [False, True, False]])
    b = ma.array([[1, 2], [3, 4], [5, 6]], mask=[[True, False], [False, True]])
    np.ma.dot(a, b)
    masked_array(data =
                 [[21 26]    
                  [45 64]],
                  mask =
                  [[False False]
                   [False False]])
```
>>> np.ma.dot(a, b, strict=True)
masked_array(data =
[[- --]  
[ -- 64]],
mask =
[[ True True]  
[ True False]],
fill_value = 999999)

numpy.ma.identity (n, dtype=None) = <numpy.ma.core._convert2ma instance at 0x2820d40>
Return the identity array.

The identity array is a square array with ones on the main diagonal.

Parameters
    n : int
        Number of rows (and columns) in n x n output.
    dtype : data-type, optional
        Data-type of the output. Defaults to float.

Returns
    out : ndarray
        n x n array with its main diagonal set to one, and all other elements 0.

Examples

>>> np.identity(3)
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])

numpy.ma.inner (a, b)
Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum
product over the last axes.

Parameters
    a, b : array_like
        If a and b are nonscalar, their last dimensions of must match.

Returns
    out : ndarray
        out.shape = a.shape[:-1] + b.shape[:-1]

Raises
    ValueError
        If the last dimension of a and b has different size.

See Also:

tensordot
    Sum products over arbitrary axes.
dot

Generalised matrix product, using second last dimension of \( b \).

einsum

Einstein summation convention.

Notes

Masked values are replaced by 0.

Examples

Ordinary inner product for vectors:

```python
>>> a = np.array([1, 2, 3])
>>> b = np.array([0, 1, 0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2, 3, 4))
>>> b = np.arange(4)
>>> np.inner(a, b)
a
array([[ 14,  38,  62],
        [ 86, 110, 134]])
```

An example where \( b \) is a scalar:

```python
>>> np.inner(np.eye(2), 7)
a
array([[ 7.,  0.],
        [ 0.,  7.]])
```

numpy.ma.innerproduct(\( a, b \))

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

Parameters

\( a, b \) : array_like

If \( a \) and \( b \) are nonscalar, their last dimensions of must match.

Returns

\( out \) : ndarray

\( out.shape = a.shape[\cdot-1] + b.shape[\cdot-1] \)

Raises

ValueError

If the last dimension of \( a \) and \( b \) has different size.

See Also:

tensordot

Sum products over arbitrary axes.

dot

Generalised matrix product, using second last dimension of \( b \).
**einsum**

Einstein summation convention.

**Notes**

Masked values are replaced by 0.

**Examples**

Ordinary inner product for vectors:

```python
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```python
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [135, 110, 134]])
```

An example where $b$ is a scalar:

```python
>>> np.inner(np.eye(2), 7)
array([[ 7.,  0.],
       [ 0.,  7.]])
```

`numpy.ma.outer(a, b)`

Compute the outer product of two vectors.

Given two vectors, $a = [a_0, a_1, \ldots, a_M]$ and $b = [b_0, b_1, \ldots, b_N]$, the outer product $[R48]$ is:

$$
\begin{align*}
[a_0*b_0 & a_0*b_1 & \ldots & a_0*b_N \\
[a_1*b_0 & \ldots & \ldots & \ldots \\
[a_M*b_0 & \ldots & a_M*b_N ]
\end{align*}
$$

**Parameters**

- **a**: ($M$,) array_like
  First input vector. Input is flattened if not already 1-dimensional.

- **b**: ($N$,) array_like
  Second input vector. Input is flattened if not already 1-dimensional.

**Returns**

- **out**: ($M$, $N$) ndarray
  out[i, j] = a[i] * b[j]

**See Also:**

- `inner`
- `einsum`

**Notes**

Masked values are replaced by 0.
Examples

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.],
        [-2., -1., 0., 1., 2.]])

>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
        [ 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
        [ 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [ 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
        [ 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])

>>> grid = rl + im
>>> grid
array([[-2.+2.j, -1.+2.j, 0.+2.j, 1.+2.j, 2.+2.j],
        [-2.+1.j, -1.+1.j, 0.+1.j, 1.+1.j, 2.+1.j],
        [-2.+0.j, -1.+0.j, 0.+0.j, 1.+0.j, 2.+0.j],
        [-2.-1.j, -1.-1.j, 0.-1.j, 1.-1.j, 2.-1.j],
        [-2.-2.j, -1.-2.j, 0.-2.j, 1.-2.j, 2.-2.j]])
```

An example using a “vector” of letters:

```python
>>> x = np.array([‘a’, ‘b’, ‘c’], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([[a, aa, aaa],
        [b, bb, bbb],
        [c, cc, ccc]], dtype=object)
```

**numpy.ma.outerproduct** *(a, b)*

Compute the outer product of two vectors.

Given two vectors, \(a = [a_0, a_1, \ldots, a_M]\) and \(b = [b_0, b_1, \ldots, b_N]\), the outer product is:

\[
\begin{bmatrix}
a_0 \times b_0 & a_0 \times b_1 & \ldots & a_0 \times b_N \\
a_1 \times b_0 & \ldots \\
a_M \times b_0 & \ldots \\
\end{bmatrix}
\]

**Parameters**

- \(a\) : (M,) array_like
  
  First input vector. Input is flattened if not already 1-dimensional.

- \(b\) : (N,) array_like
  
  Second input vector. Input is flattened if not already 1-dimensional.

**Returns**

- \(out\) : (M, N) ndarray
  
  \(out[i, j] = a[i] \times b[j]\)
See Also:

inner, einsum

Notes

Masked values are replaced by 0.

References

[R49]

Examples

Make a (very coarse) grid for computing a Mandelbrot set:

```python
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[ 2. ,  1.5,  1. ,  0.5,  0. ,
          -0.5, -1. , -1.5, -2. , -2.5],
         [ 0.5,  1. ,  1.5,  2. ,  2.5],
         [ 2. ,  1.5,  1. ,  0.5,  0. ],
         [ 0. ,  1. ,  1.5,  2. ,  2.5],
         [-2. , -1.5, -1. , -0.5, -2. ]])
```

```python
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[ 2.j,  2.j,  2.j,  2.j,  2.j],
         [ 1.j,  1.j,  1.j,  1.j,  1.j],
         [ 0.j,  0.j,  0.j,  0.j,  0.j],
         [-1.j, -1.j, -1.j, -1.j, -1.j],
         [-2.j, -2.j, -2.j, -2.j, -2.j]])
```

```python
>>> grid = rl + im
>>> grid
array([[-2.+2.j, -1.+2.j, 0.+2.j, 1.+2.j, 2.+2.j],
       [-2.+1.j, -1.+1.j, 0.+1.j, 1.+1.j, 2.+1.j],
       [-2.+0.j, -1.+0.j, 0.+0.j, 1.+0.j, 2.+0.j],
       [-2.-1.j, -1.-1.j, 0.-1.j, 1.-1.j, 2.-1.j],
       [-2.-2.j, -1.-2.j, 0.-2.j, 1.-2.j, 2.-2.j]])
```

An example using a “vector” of letters:

```python
>>> x = np.array([‘a’, ‘b’, ‘c’], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

```
numpy.ma.trace (self, offset=0, axis1=0, axis2=1, dtype=None, out=None) a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None) = <numpy.ma.core._frommethod instance at 0x28209e0>
```

Return the sum along diagonals of the array.

Refer to numpy.trace for full documentation.

See Also:

numpy.trace
equivalent function

```
numpy.ma.transpose (a, axes=None) Permute the dimensions of an array.
```

3.19. Masked array operations
This function is exactly equivalent to `numpy.transpose`.

See Also:

**numpy.transpose**
Equivalent function in top-level NumPy module.

**Examples**

```python
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked
>>> x
masked_array(data =
[[0 1]
[2 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)
>>> ma.transpose(x)
masked_array(data =
[[0 2]
[1 --]],
mask =
[[False False]
[False  True]],
fill_value = 999999)
```

MaskedArray.**trace** (offset=0, axis1=0, axis2=1, dtype=None, out=None)  
Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See Also:

**numpy.trace**  
equivalent function

MaskedArray.**transpose** (*axes*)  
Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

- **axes**: None, tuple of ints, or n ints
  
    - None or no argument: reverses the order of the axes.
  
    - tuple of ints: i in the j-th place in the tuple means a’s i-th axis becomes a.transpose()’s j-th axis.
  
    - n ints: same as an n-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

**Returns**

- **out**: ndarray
View of a, with axes suitably permuted.

See Also:

ndarray.T
Array property returning the array transposed.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

Polynomial fit

```python
ma.vander(x[, n]) Generate a Van der Monde matrix.
ma.polyfit(x, y, deg[, rcond, full, w, cov]) Least squares polynomial fit.
```

```python
numpy.ma.vander(x, n=None) Generate a Van der Monde matrix.
```

The columns of the output matrix are decreasing powers of the input vector. Specifically, the \(i\)-th output column is the input vector raised element-wise to the power of \(N - i - 1\). Such a matrix with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

Parameters

- x : array_like
  1-D input array.
- N : int, optional
  Order of (number of columns in) the output. If \(N\) is not specified, a square array is returned (\(N = \text{len}(x)\)).

Returns

- out : ndarray
  Van der Monde matrix of order \(N\). The first column is \(x^{N-1}\), the second \(x^{N-2}\) and so forth.

Notes

Masked values in the input array result in rows of zeros.
Examples

```python
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])

>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])

>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[ 1,  1,  1,  1],
       [ 8,  4,  2,  1],
       [27,  9,  3,  1],
       [125, 25,  5,  1]])

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```python
>>> np.linalg.det(np.vander(x))
48.000000000000043

```(5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```

**numpy.ma.polyfit** *(x, y, deg, rcond=None, full=False, w=None, cov=False)*

Least squares polynomial fit.

Fit a polynomial \( p(x) = p[0] \times x^{\deg} + \ldots + p[\deg] \) of degree \( \deg \) to points \( (x, y) \). Returns a vector of coefficients \( p \) that minimises the squared error.

**Parameters**

- **x**: array_like, shape (M,)
  - x-coordinates of the M sample points \( (x[i], y[i]) \).
- **y**: array_like, shape (M,) or (M, K)
  - y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.
- **deg**: int
  - Degree of the fitting polynomial
- **rcond**: float, optional
  - Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \( \text{len}(x)^*\text{eps} \), where \( \text{eps} \) is the relative precision of the float type, about 2e-16 in most cases.
- **full**: bool, optional
  - Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.
w : array_like, shape (M,), optional
weights to apply to the y-coordinates of the sample points.

cov : bool, optional
Return the estimate and the covariance matrix of the estimate. If full is True, then cov is not returned.

Returns

- p : ndarray, shape (M,) or (M, K)
  Polynomial coefficients, highest power first. If y was 2-D, the coefficients for k-th data set are in p[:, k].

- residuals, rank, singular_values, rcond : present only if full = True
  Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of rcond. For more details, see linalg.lstsq.

- V : ndarray, shape (M,M) or (M,M,K)
  The covariance matrix of the polynomial coefficient estimates. The diagonal of this matrix are the variance estimates for each coefficient. If y is a 2-d array, then the covariance matrix for the k-th data set are in V[:, :, k].

Warns

- RankWarning
  The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if full = False.

  The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

See Also:

- polyval
  Computes polynomial values.

- linalg.lstsq
  Computes a least-squares fit.

- scipy.interpolate.UnivariateSpline
  Computes spline fits.

Notes

Any masked values in x is propagated in y, and vice-versa.

References

[R50], [R51]

Examples

```python
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
```
>>> z
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])

It is convenient to use `poly1d` objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
-0.80000000000000204
>>> p30(5)
-0.99999999999999445
>>> p30(4.5)
-0.10547061179440398
```

Illustration:

```python
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

![Graph](image)

Clipping and rounding
NumPy Reference, Release 1.8.1

`ma.around`  
Round an array to the given number of decimals.

`ma.clip(a, a_min, a_max[, out])`  
Clip (limit) the values in an array.

`ma.round(a[, decimals, out])`  
Return a copy of a, rounded to ‘decimals’ places.

`ma.MaskedArray.clip(a_min, a_max[, out])`  
Return an array whose values are limited to `[a_min, a_max]`.

`ma.MaskedArray.round([decimals, out])`  
Return a with each element rounded to the given number of decimals.

```python
numpy.ma.around = <numpy.ma.core._MaskedUnaryOperation instance at 0x2816440>
    Round an array to the given number of decimals.
    Refer to around for full documentation.
    See Also:
    around
    equivalent function

numpy.ma.clip(a, a_min, a_max, out=None)
    Clip (limit) the values in an array.
    Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.
    Parameters
    a : array_like
        Array containing elements to clip.
    a_min : scalar or array_like
        Minimum value.
    a_max : scalar or array_like
        Maximum value. If a_min or a_max are array_like, then they will be broadcasted to the shape of a.
    out : ndarray, optional
        The results will be placed in this array. It may be the input array for in-place clipping. out must be of the right shape to hold the output. Its type is preserved.
    Returns
    clipped_array : ndarray
        An array with the elements of a, but where values < a_min are replaced with a_min, and those > a_max with a_max.
    See Also:
    numpy.doc.ufuncs
        Section “Output arguments”

Examples
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
```

3.19. Masked array operations
NumPy Reference, Release 1.8.1

array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3,4,1,1,1,4,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])

numpy.ma.round(a, decimals=0, out=None)
Return a copy of a, rounded to ‘decimals’ places.

When ‘decimals’ is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and ‘decimals’ is greater than or equal to 0.

Parameters

decimals : int
    Number of decimals to round to. May be negative.

out : array_like
    Existing array to use for output. If not given, returns a default copy of a.

Notes

If out is given and does not have a mask attribute, the mask of a is lost!

MaskedArray.clip(a_min, a_max, out=None)
Return an array whose values are limited to [a_min, a_max].

Refer to numpy.clip for full documentation.

See Also:

numpy.clip
    equivalent function

MaskedArray.round(decimals=0, out=None)
Return a with each element rounded to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

numpy.around
    equivalent function

Miscellanea

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ma.allequal(a, b[, fill_value])</td>
<td>Return True if all entries of a and b are equal, using</td>
</tr>
<tr>
<td>ma.aliclose(a, b[, masked_equal, rtol, atol])</td>
<td>Returns True if two arrays are element-wise equal within a tolerance.</td>
</tr>
<tr>
<td>ma.apply_along_axis(func1d, axis, arr, ...)</td>
<td>Apply a function to 1-D slices along the given axis.</td>
</tr>
<tr>
<td>ma.arange([start[, stop[, step[, dtype]]]])</td>
<td>Return evenly spaced values within a given interval.</td>
</tr>
<tr>
<td>ma.choose(indices, choices[, out, mode])</td>
<td>Use an index array to construct a new array from a set of choices.</td>
</tr>
<tr>
<td>ma.ediff1d(arr[, to_begin, to_end])</td>
<td>Compute the differences between consecutive elements of an array.</td>
</tr>
<tr>
<td>ma.indices(dimensions[, dtype])</td>
<td>Return an array representing the indices of a grid.</td>
</tr>
<tr>
<td>ma.where(condition[, x, y])</td>
<td>Return a masked array with elements from x or y, depending on condition.</td>
</tr>
</tbody>
</table>
numpy.ma.allequal(a, b, fill_value=True)

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

Parameters

a, b : array_like
    Input arrays to compare.

fill_value : bool, optional
    Whether masked values in a or b are considered equal (True) or not (False).

Returns

y : bool
    Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

See Also:

all, any, numpy.ma.allclose

Examples

>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
             mask = [False False True],
             fill_value=1e+20)

>>> b = array([1e10, 1e-7, -42.0])
>>> b
array([1.00000000e+10, 1.00000000e-07, -4.20000000e+01])
>>> ma.allequal(a, b, fill_value=False)
False
>>> ma.allequal(a, b)
True

numpy.ma.allclose(a, b, masked_equal=True, rtol=1e-05, atol=1e-08)

Returns True if two arrays are element-wise equal within a tolerance.

This function is equivalent to allclose except that masked values are treated as equal (default) or unequal, depending on the masked_equal argument.

Parameters

a, b : array_like
    Input arrays to compare.

masked_equal : bool, optional
    Whether masked values in a and b are considered equal (True) or not (False). They are considered equal by default.

rtol : float, optional
    Relative tolerance. The relative difference is equal to rtol * b. Default is 1e-5.

atol : float, optional
    Absolute tolerance. The absolute difference is equal to atol. Default is 1e-8.

Returns

y : bool
Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

See Also:
all, any

numpy.allclose
the non-masked allclose.

Notes
If the following equation is element-wise True, then allclose returns True:

\[ \text{absolute}(a - b) \leq (\text{atol} + \text{rtol} \times \text{absolute}(b)) \]

Return True if all elements of a and b are equal subject to given tolerances.

Examples

```python
>>> a = ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data = [10000000000.0 1e-07 --],
              mask = [False False True],
              fill_value = 1e+20)
>>> b = ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
False

>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

Masked values are not compared directly.

```python
>>> a = ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
>>> ma.allclose(a, b)
True
>>> ma.allclose(a, b, masked_equal=False)
False
```

numpy.ma.apply_along_axis(func1d, axis, arr, *args, **kwargs)
Apply a function to 1-D slices along the given axis.

Execute `func1d(a, *args)` where `func1d` operates on 1-D arrays and `a` is a 1-D slice of `arr` along `axis`.

Parameters

func1d : function
This function should accept 1-D arrays. It is applied to 1-D slices of `arr` along the specified axis.

axis : integer
Axis along which `arr` is sliced.

arr : ndarray
Input array.
```python
addrs: any

Additional arguments to func1d.

Returns

apply_along_axis: ndarray

The output array. The shape of outarr is identical to the shape of arr, except along the axis dimension, where the length of outarr is equal to the size of the return value of func1d. If func1d returns a scalar outarr will have one fewer dimensions than arr.

See Also:

apply_over_axes

Apply a function repeatedly over multiple axes.

Examples

>>> def my_func(a):
...     """Average first and last element of a 1-D array""
...     return (a[0] + a[-1]) * 0.5
... >>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
... >>> np.apply_along_axis(my_func, 0, b)
array([ 4., 5., 6.])
... >>> np.apply_along_axis(my_func, 1, b)
array([ 2., 5., 8.])

For a function that doesn’t return a scalar, the number of dimensions in outarr is the same as arr.

>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
... >>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])

numpy.ma.arange([start], stop[, step[, dtype=None]]) = <numpy.ma.core._convert2ma instance at 0x2820bd8>

Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

Parameters

start : number, optional

Start of interval. The interval includes this value. The default start value is 0.

stop : number

End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.

step : number, optional

Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] − out[i]. The default step size is 1. If step is specified, start must also be given.

dtype : dtype
```

The type of the output array. If dtype is not given, infer the data type from the other input arguments.

Returns

arange : ndarray

Array of evenly spaced values.

For floating point arguments, the length of the result is \( \text{ceil}((\text{stop} - \text{start})/\text{step}) \). Because of floating point overflow, this rule may result in the last element of out being greater than stop.

See Also:

linspace

Evenly spaced numbers with careful handling of endpoints.

ogrid

Arrays of evenly spaced numbers in N-dimensions.

mgrid

Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples

```python
>>> np.arange(3)
array([0, 1, 2])

>>> np.arange(3.0)
array([ 0., 1., 2.])

>>> np.arange(3,7)
array([3, 4, 5, 6])

>>> np.arange(3,7,2)
array([3, 5])
```

```python
numpy.ma.choose (indices, choices, out=None, mode='raise')
```

Use an index array to construct a new array from a set of choices.

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of the choice arrays. Where a value in a is i, the new array will have the value that choices[i] contains in the same place.

Parameters

a : ndarray of ints

This array must contain integers in \([0, n-1]\), where n is the number of choices.

choices : sequence of arrays

Choice arrays. The index array and all of the choices should be broadcastable to the same shape.

out : array, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave.

- 'raise' : raise an error
- 'wrap' : wrap around
• ‘clip’ : clip to the range

Returns
merged_array : array

See Also:

choose
equivalent function

Examples

```python
>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data = [3 2 1],
             mask = False,
             fill_value=999999)
```

`numpy.ma.ediff1d(arr, to_end=None, to_begin=None)`

Compute the differences between consecutive elements of an array.

This function is the equivalent of `numpy.ediff1d` that takes masked values into account, see `numpy.ediff1d` for details.

See Also:

`numpy.ediff1d`
Equivalent function for ndarrays.

`numpy.ma.indices(dimensions, dtype=<type 'int'>)`

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0,1,... varying only along the corresponding axis.

Parameters

- `dimensions` : sequence of ints
  The shape of the grid.

- `dtype` : dtype, optional
  Data type of the result.

Returns

grid : ndarray
  The array of grid indices. `grid.shape = (len(dimensions),) +
tuple(dimensions)`.

See Also:

mgrid, meshgrid

Notes

The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if `dimensions` is a tuple `(r0, ..., rN-1)` of length N, the output shape is `(N, r0, ..., rN-1)`.

The subarrays `grid[k]` contains the N-D array of indices along the k-th axis. Explicitly:

```
grid[k,i0,i1,...,iN-1] = ik
```
Examples

```python
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]  # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]  # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```python
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[:2, :3]`.

```
numpy.ma.where(condition, x=None, y=None)
```

Return a masked array with elements from x or y, depending on condition.

Returns a masked array, shaped like condition, where the elements are from x when `condition` is True, and from y otherwise. If neither x nor y are given, the function returns a tuple of indices where `condition` is True (the result of `condition.nonzero()`).

Parameters

- **condition**: array_like, bool
  The condition to meet. For each True element, yield the corresponding element from x, otherwise from y.

- **x, y**: array_like, optional
  Values from which to choose. x and y need to have the same shape as condition, or be broadcast-able to that shape.

Returns

- **out**: MaskedArray or tuple of ndarrays
  The resulting masked array if x and y were given, otherwise the result of `condition.nonzero()`.

See Also:

- `numpy.where`
  Equivalent function in the top-level NumPy module.

Examples

```python
>>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],
    ... [1, 0, 1],
    ... [0, 1, 0]])
>>> print x
[[0.0 -- 2.0]
[-- 4.0 --]
[6.0 -- 8.0]]
```
>>> np.ma.where(x > 5)  # return the indices where x > 5
(array([2, 2]), array([0, 2]))

>>> print np.ma.where(x > 5, x, -3.1416)
[[-3.1416 -- -3.1416]
[-- -3.1416 --]
[6.0 -- 8.0]]

3.20 Mathematical functions

3.20.1 Trigonometric functions

```python
numpy.sin(x, out) = <ufunc 'sin'>
Trigonometric sine, element-wise.
```

Parameters

- `x`: array_like
  Angle, in radians (2π rad equals 360 degrees).

Returns

- `y`: array_like
  The sine of each element of `x`.

See Also:

- `arcsin`, `sinh`, `cos`

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the +x axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray’s intersection with the unit circle is the sine of that angle. It ranges from -1 for x = 3π/2 to +1 for π/2. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.
Examples

Print sine of one angle:

```python
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```python
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180.)
array([ 0. , 0.5 , 0.70710678, 0.8660254 , 1. ])
```

Plot the sine function:

```python
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]

```

numpy.cos(x[, out]) = <ufunc ‘cos’>

Cosine element-wise.

Parameters

- x : array_like
  Input array in radians.

- out : ndarray, optional
  Output array of same shape as x.

Returns

- y : ndarray
  The corresponding cosine values.

Raises

- ValueError: invalid return array shape
NumPy Reference, Release 1.8.1

if out is provided and out.shape != x.shape (See Examples)
Notes
If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)
References
Examples
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([ 1.00000000e+00,
6.12303177e-17, -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched ‘out‘
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid return array shape

numpy.tan(x[, out ]) = <ufunc ‘tan’>
Compute tangent element-wise.
Equivalent to np.sin(x)/np.cos(x) element-wise.
Parameters
x : array_like
Input array.
out : ndarray, optional
Output array of same shape as x.
Returns
y : ndarray
The corresponding tangent values.
Raises
ValueError: invalid return array shape
if out is provided and out.shape != x.shape (See Examples)
Notes
If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)
References
Examples
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([ 1.22460635e-16,
1.63317787e+16,

3.20. Mathematical functions

-1.22460635e-16])

877


numpy.arcsin(x[, out]) = <ufunc ‘arcsin’>
Inverse sine, element-wise.

Parameters
x : array_like
  y-coordinate on the unit circle.

out : ndarray, optional
  Array of the same shape as x, in which to store the results. See doc.ufuncs (Section “Output arguments”) for more details.

Returns
angle : ndarray
  The inverse sine of each element in x, in radians and in the closed interval [-pi/2, pi/2]. If x is a scalar, a scalar is returned, otherwise an array.

See Also:
sin, cos, arccos, tan, arctan, arctan2, emath.arcsin

Notes
arcsin is a multivalued function: for each x there are infinitely many numbers z such that sin(z) = x. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, arcsin always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, arcsin is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as asin or sin^{-1}.

References

Examples
>>> np.arcsin(1)  # pi/2
1.5707963267948966
>>> np.arcsin(-1)  # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
The inverse of \( \cos \) so that, if \( y = \cos(x) \), then \( x = \arccos(y) \).

**Parameters**

- **x**: array_like
  - \( x \)-coordinate on the unit circle. For real arguments, the domain is \([-1, 1]\).
- **out**: ndarray, optional
  - Array of the same shape as \( a \), to store results in. See doc.ufuncs (Section “Output arguments”) for more details.

**Returns**

- **angle**: ndarray
  - The angle of the ray intersecting the unit circle at the given \( x \)-coordinate in radians \([0, \pi]\). If \( x \) is a scalar then a scalar is returned, otherwise an array of the same shape as \( x \) is returned.

**See Also:**

- \( \cos, \arctan, \arcsin, \text{emath.arccos} \)

**Notes**

\( \arccos \) is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \cos(z) = x \). The convention is to return the angle \( z \) whose real part lies in \([0, \pi]\).

For real-valued input data types, \( \arccos \) always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \( \text{nan} \) and sets the \text{invalid} floating point error flag.

For complex-valued input, \( \arccos \) is a complex analytic function that has branch cuts \([-\infty, -1]\) and \([1, \infty]\) and is continuous from above on the former and from below on the latter.

The inverse \( \cos \) is also known as \( \text{acos} \) or \( \cos^{-1} \).

**References**


**Examples**

We expect the arccos of 1 to be 0, and of -1 to be \( \pi \):

```python
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```
numpy.arctan(x[, out]) = <ufunc 'arctan'>

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if y = tan(x) then x = arctan(y).

Parameters
x : array_like
   Input values. arctan is applied to each element of x.

Returns
out : ndarray
   Out has the same shape as x. Its real part is in [-pi/2, pi/2] (arctan(+/-inf) returns +/-pi/2). It is a scalar if x is a scalar.

See Also:

arctan2
   The “four quadrant” arctan of the angle formed by (x, y) and the positive x-axis.

angle
   Argument of complex values.

Notes

arctan is a multi-valued function: for each x there are infinitely many numbers z such that tan(z) = x. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, arctan always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag.

For complex-valued input, arctan is a complex analytic function that has [1j, infj] and [-1j, -infj] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as atan or tan^{-1}.

References

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```python
>>> np.arctan([0, 1])
array([ 0. , 0.78539816])
```

```python
>>> np.pi/4
0.78539816339744828
```

Plot arctan:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```

```
1.0
0.5
0.0
-0.5
-1.0

-10 -5 0 5 10
```

```
3.20. Mathematical functions

numpy.hypot(x1, x2[, out]) = <ufunc 'hypot'>

Given the “legs” of a right triangle, return its hypotenuse.

Equivalent to sqrt(x1**2 + x2**2), element-wise. If x1 or x2 is scalar_like (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

Parameters
- x1, x2 : array_like
  Leg of the triangle(s).
- out : ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns
- z : ndarray
  The hypotenuse of the triangle(s).
```
Examples

```python
>>> np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```python
>>> np.hypot(3*np.ones((3, 3)), [4])
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

`numpy.arctan2(x1, x2[, out]) = <ufunc 'arctan2'>`

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (x2, x1). (Note the role reversal: the "y-coordinate" is the first function parameter, the "x-coordinate" is the second.) By IEEE convention, this function is defined for `x2 = +/-0` and for either or both of `x1` and `x2` = +/-inf (see Notes for specific values).

This function is not defined for complex-valued arguments; for the so-called argument of complex values, use `angle`.

Parameters

- `x1` : array_like, real-valued
  
  y-coordinates.

- `x2` : array_like, real-valued
  
  x-coordinates. `x2` must be broadcastable to match the shape of `x1` or vice versa.

Returns

- `angle` : ndarray
  
  Array of angles in radians, in the range [-pi, pi].

See Also:

arctan, tan, angle

Notes

`arctan2` is identical to the `atan2` function of the underlying C library. The following special values are defined in the C standard: [R6]

<table>
<thead>
<tr>
<th><code>x1</code></th>
<th><code>x2</code></th>
<th><code>arctan2(x1, x2)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-0</td>
<td>+0</td>
<td>+/- 0</td>
</tr>
<tr>
<td>+/-0</td>
<td>-0</td>
<td>+/- pi</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>+/-inf</td>
<td>+0 / +pi</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>+/-inf</td>
<td>-0 / -pi</td>
</tr>
<tr>
<td>+/-inf</td>
<td>+inf</td>
<td>+/- (pi/4)</td>
</tr>
<tr>
<td>+/-inf</td>
<td>-inf</td>
<td>+/- (3*pi/4)</td>
</tr>
</tbody>
</table>

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

References

[R6]
Examples

Consider four points in different quadrants:

```python
>>> x = np.array([-1, +1, +1, -1])
>>> y = np.array([-1, -1, +1, +1])
>>> np.arctan2(y, x) * 180 / np.pi
array([-135., -45.,  45., 135.])
```

Note the order of the parameters. `arctan2` is defined also when $x_2 = 0$ and at several other special points, obtaining values in the range $[-\pi, \pi]$:

```python
>>> np.arctan2([1., -1.], [0., 0.])
array([1.57079633, -1.57079633])
>>> np.arctan2([0., 0., np.inf], [+0., -0., np.inf])
array([0., 3.14159265, 0.78539816])
```

**numpy.**

```
degrees (x[, out]) = <ufunc ‘degrees’>
```

Convert angles from radians to degrees.

**Parameters**

- `x`: array_like
  
  Input array in radians.

- `out`: ndarray, optional
  
  Output array of same shape as `x`.

**Returns**

- `y`: ndarray of floats
  
  The corresponding degree values; if `out` was supplied this is a reference to it.

**See Also:**

- `rad2deg`

  equivalent function

**Examples**

Convert a radian array to degrees

```python
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([ 0., 30., 60., 90., 120., 150., 180., 210., 240.,
       270., 300., 330.])
```

```python
>>> out = np.zeros((rad.shape))
>>> r = degrees(rad, out)
>>> np.all(r == out)
True
```

**numpy.**

```
radians (x[, out]) = <ufunc ‘radians’>
```

Convert angles from degrees to radians.

**Parameters**

- `x`: array_like
  
  Input array in degrees.

- `out`: ndarray, optional
  
  Output array of same shape as `x`.
Output array of same shape as \(x\).

**Returns**

- \(y\): ndarray
  The corresponding radian values.

**See Also:**

- `deg2rad`
  equivalent function

**Examples**

Convert a degree array to radians

```python
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0. , 0.52359878, 1.04719755, 1.57079633, 2.0943951 ,
       2.61799388, 3.14159265, 3.66519143, 4.1887902 , 4.71238898,
       5.23598776, 5.75958653])
```

```python
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

**numpy.unwrap**

Unwrap by changing deltas between values to \(2\pi\) complement.

Unwrap radian phase \(p\) by changing absolute jumps greater than \(\text{discont}\) to their \(2\pi\) complement along the given axis.

**Parameters**

- \(p\): array_like
  Input array.

- \(\text{discont}\): float, optional
  Maximum discontinuity between values, default is \(
\pi\).

- \(\text{axis}\): int, optional
  Axis along which unwrap will operate, default is the last axis.

**Returns**

- \(out\): ndarray
  Output array.

**See Also:**

- `rad2deg`, `deg2rad`

**Notes**

If the discontinuity in \(p\) is smaller than \(\pi\), but larger than \(\text{discont}\), no unwrapping is done because taking the \(2\pi\) complement would only make the discontinuity larger.

**Examples**
```python
>>> phase = np.linspace(0, np.pi, num=5)
>>> phase[3:] += np.pi
>>> phase
array([ 0.        , 0.78539816, 1.57079633, 5.49778714, 6.28318531])
>>> np.unwrap(phase)
array([ 0.        , 0.78539816, 1.57079633, -0.78539816, 0.        ])
```

```python
numpy.deg2rad(x[, out]) = <ufunc 'deg2rad'>
Convert angles from degrees to radians.

Parameters

- x : array_like
  Angles in degrees.

Returns

- y : ndarray
  The corresponding angle in radians.

See Also:

rad2deg
Convert angles from radians to degrees.
unwrap
Remove large jumps in angle by wrapping.

Notes

New in version 1.3.0. deg2rad(x) is x * pi / 180.

Examples

```python
>>> np.deg2rad(180)
3.1415926535897931
```
```
```python
numpy.rad2deg(x[, out]) = <ufunc 'rad2deg'>
Convert angles from radians to degrees.

Parameters

- x : array_like
  Angle in radians.

Returns

- y : ndarray
  The corresponding angle in degrees.

See Also:

deg2rad
Convert angles from degrees to radians.
unwrap
Remove large jumps in angle by wrapping.
```
Notes

New in version 1.3.0. rad2deg(x) is $180 \times x / \pi$.

Examples

```python
>>> np.rad2deg(np.pi/2)
90.0
```

3.20.2 Hyperbolic functions

```python
numpy.sinh(x[, out]) = <ufunc 'sinh'>

Hyperbolic sine, element-wise.

Equivalent to $1/2 \times (np.exp(x) - np.exp(-x))$ or $-1j \times np.sin(1j*x)$.

Parameters

- **x**: array_like
  - Input array.
- **out**: ndarray, optional
  - Output array of same shape as x.

Returns

- **y**: ndarray
  - The corresponding hyperbolic sine values.

Raises

- ValueError: invalid return array shape
  - if out is provided and out.shape != x.shape (See Examples)

Notes

If out is provided, the function writes the result into it, and returns a reference to out. (See Examples)

References


Examples

```python
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j)  # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```
>>> # Example of providing the optional output parameter
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True

>>> # Example of ValueError due to provision of shape mis-matched 'out'
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape

numpy.cosh(x[, out]) = <ufunc 'cosh'>

Hyperbolic cosine, element-wise.

Equivalent to 1/2 * (np.exp(x) + np.exp(-x)) and np.cos(1j*x).

Parameters:
  x : array_like
    Input array.

Returns:
  out : ndarray
    Output array of same shape as x.

Examples:

>>> np.cosh(0)
1.0

The hyperbolic cosine describes the shape of a hanging cable:

>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
Equivalent to \( \frac{\sinh(x)}{\cosh(x)} \) or \(-1j \times \tan(1j \times x)\).

**Parameters**
- \( x \): array_like
  - Input array.
- \( \text{out} \): ndarray, optional
  - Output array of same shape as \( x \).

**Returns**
- \( y \): ndarray
  - The corresponding hyperbolic tangent values.

**Raises**
- ValueError: invalid return array shape
  - if \( \text{out} \) is provided and \( \text{out}.\text{shape} \neq x.\text{shape} \) (See Examples)

**Notes**
If \( \text{out} \) is provided, the function writes the result into it, and returns a reference to \( \text{out} \). (See Examples)

**References**
[R246], [R247]

**Examples**
```python
c>>> np.tanh((0, np.pi*j, np.pi*j/2))
array([ 0. +0.00000000e+00j, 0. -1.22460635e-16j, 0. +1.63317787e+16j])

c>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True

>>> # Example of ValueError due to provision of shape mis-matched 'out'
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

**numpy.arcsinh**

**Inverse hyperbolic sine element-wise.**

**Parameters**
- \( x \): array_like
  - Input array.
- \( \text{out} \): ndarray, optional
  - Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

**Returns**
- \( \text{out} \): ndarray
  - Array of of the same shape as \( x \).
Notes

arcsinh is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \sinh(z) = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi/2, \pi/2]\).

For real-valued input data types, arcsinh always returns real output. For each value that cannot be expressed as a real number or infinity, it returns NaN and sets the invalid floating point error flag.

For complex-valued input, arccosh is a complex analytical function that has branch cuts \([1j, \text{inf}j]\) and \([-1j, -\text{inf}j]\) and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as asinh or \( \sinh^{-1} \).

References

[R4], [R5]

Examples

```python
>>> np.arcsinh(np.array([np.e, 10.0]))
array([1.72538256, 2.99822295])
```

numpy.arccosh(x[, out]) = <ufunc 'arccosh'>

Inverse hyperbolic cosine, element-wise.

Parameters

- x : array_like
  Input array.
- out : ndarray, optional
  Array of the same shape as \( x \), to store results in. See doc.ufuncs (Section “Output arguments”) for details.

Returns

- arccosh : ndarray
  Array of the same shape as \( x \).

See Also:

cosh, arcsinh, sinh, arctanh, tanh

Notes

arccosh is a multivalued function: for each \( x \) there are infinitely many numbers \( z \) such that \( \cosh(z) = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\) and the real part in \([0, \text{inf}]\).

For real-valued input data types, arccosh always returns real output. For each value that cannot be expressed as a real number or infinity, it yields NaN and sets the invalid floating point error flag.

For complex-valued input, arccosh is a complex analytical function that has a branch cut \([-\text{inf}, 1]\) and is continuous from above on it.

References

[R2], [R3]

Examples

```python
>>> np.arccosh([np.e, 10.0])
array([1.65745445, 2.99322285])
```
numpy.arctanh(x[, out]) = <ufunc ‘arctanh’>
Inverse hyperbolic tangent element-wise.

Parameters
x : array_like
   Input array.

Returns
out : ndarray
   Array of the same shape as x.

See Also:
emath.arctanh

Notes
arctanh is a multivalued function: for each x there are infinitely many numbers z
such that \text{tanh}(z) = x. The convention is to return the z whose imaginary
part lies in [-\pi/2, \pi/2].

For real-valued input data types, arctanh always returns real output. For each value
that cannot be expressed as a real number or infinity, it yields \text{nan} and sets the
\text{invalid} floating point error flag.

For complex-valued input, arctanh is a complex analytical function that has branch
cuts [-1, -\text{inf}] and [1, \text{inf}] and is continuous from above on the former
and from below on the latter.

The inverse hyperbolic tangent is also known as \text{atanh} or tanh^{-1}.

References
[R7], [R8]

Examples
>>> np.arctanh([0, -0.5])
array([ 0. , -0.54930614])

3.20.3 Rounding

```python
round(a[, decimals, out])
Round an array to the given number of decimals.

rint(x[, out])
Round elements of the array to the nearest integer.

fix(x[, y])
Round to nearest integer towards zero.
```

numpy.around(a, decimals=0, out=None)
Evenly round to the given number of decimals.

Parameters
a : array_like
   Input data.
decimals : int, optional

   Number of decimal places to round to (default: 0). If decimals is negative, it specifies
   the number of positions to the left of the decimal point.

out : ndarray, optional

   Alternative output array in which to place the result. It must have the same shape as
   the expected output, but the type of the output values will be cast if necessary. See
   doc.ufuncs (Section “Output arguments”) for details.

Returns

   rounded_array : ndarray

   An array of the same type as a, containing the rounded values. Unless out was specified,
   a new array is created. A reference to the result is returned.

   The real and imaginary parts of complex numbers are rounded separately. The result of
   rounding a float is a float.

See Also:

   ndarray.round  
   equivalent method

   ceil, fix, floor, rint, trunc

Notes

   For values exactly halfway between rounded decimal values, Numpy rounds to the nearest even value. Thus
   1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact
   representation of decimal fractions in the IEEE floating point standard [R9] and errors introduced when scaling
   by powers of ten.

References

   [R9], [R10]

Examples

   >>> np.around([0.37, 1.64])
   array([ 0., 2.])
   >>> np.around([0.37, 1.64], decimals=1)
   array([ 0.4, 1.6])
   >>> np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
   array([ 0., 2., 2., 4., 4.])
   >>> np.around([1,2,3,11], decimals=1) # ndarray of ints is returned
   array([ 1, 2, 3, 11])
   >>> np.around([1,2,3,11], decimals=-1)
   array([ 0, 0, 0, 10])

numpy.round(a, decimals=0, out=None)

   Round an array to the given number of decimals.

   Refer to around for full documentation.

See Also:

   around  
   equivalent function
numpy.
rint
(x[, out]) = <ufunc ‘rint’>

Round elements of the array to the nearest integer.

Parameters

x : array_like

Input array.

Returns

out : {ndarray, scalar}

Output array is same shape and type as x.

See Also:

ceil, floor, trunc

Examples

>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0., 0., 2., 2., 2.])

numpy.
fix
(x, y=None)

Round to nearest integer towards zero.

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Parameters

x : array_like

An array of floats to be rounded

y : ndarray, optional

Output array

Returns

out : ndarray of floats

The array of rounded numbers

See Also:

trunc, floor, ceil

around

Round to given number of decimals

Examples

>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2., 2., -2., -2.])

numpy.
floor
(x[, out]) = <ufunc ‘floor’>

Return the floor of the input, element-wise.

The floor of the scalar $x$ is the largest integer $i$, such that $i \leq x$. It is often denoted as $\lfloor x \rfloor$.

Parameters

x : array_like
Input data.

**Returns**

- **y**: {ndarray, scalar}
  
  The floor of each element in x.

**See Also:**

`ceil`, `trunc`, `rint`  

**Notes**

Some spreadsheet programs calculate the “floor-towards-zero”, in other words \( \text{floor}(\text{-2.5}) == -2 \). NumPy instead uses the definition of \( \text{floor} \) where \( \text{floor}(\text{-2.5}) == -3 \).

**Examples**

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
a = np.array([-2., -2., -1., 0., 1., 1., 2.])
```

**numpy.ceil(x[, out]) = <ufunc ‘ceil’>**

Return the ceiling of the input, element-wise.

The ceil of the scalar x is the smallest integer \( i \) such that \( i \geq x \). It is often denoted as \( \lceil x \rceil \).

**Parameters**

- **x**: array_like
  
  Input data.

**Returns**

- **y**: {ndarray, scalar}
  
  The ceiling of each element in x, with float dtype.

**See Also:**

`floor`, `trunc`, `rint`  

**Examples**

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
a = np.array([-1., -1., -0., 1., 2., 2., 2.])
```

**numpy.trunc(x[, out]) = <ufunc ‘trunc’>**

Return the truncated value of the input, element-wise.

The truncated value of the scalar x is the nearest integer \( i \) which is closer to zero than x is. In short, the fractional part of the signed number x is discarded.

**Parameters**

- **x**: array_like
  
  Input data.

**Returns**

- **y**: {ndarray, scalar}
  
  The truncated value of each element in x.
NumPy Reference, Release 1.8.1

See Also:

ceil, floor, rint

Notes

New in version 1.3.0.

Examples

```python
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0., 0., 1., 1., 2.])
```

3.20.4 Sums, products, differences

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod(a, axis, dtype, out, keepdims)</td>
<td>Return the product of array elements over a given axis.</td>
</tr>
<tr>
<td>sum(a, axis, dtype, out, keepdims)</td>
<td>Sum of array elements over a given axis.</td>
</tr>
<tr>
<td>nansum(a, axis, dtype, out, keepdims)</td>
<td>Return the sum of array elements over a given axis treating Not a</td>
</tr>
<tr>
<td>cumprod(a, axis, dtype, out)</td>
<td>Return the cumulative product of elements along a given axis.</td>
</tr>
<tr>
<td>cumsum(a, axis, dtype, out)</td>
<td>Return the cumulative sum of the elements along a given axis.</td>
</tr>
<tr>
<td>diff(a, n, axis)</td>
<td>Calculate the n-th order discrete difference along a given axis.</td>
</tr>
<tr>
<td>ediff1d(ary, to_end, to_begin)</td>
<td>The differences between consecutive elements of an array.</td>
</tr>
<tr>
<td>gradient(f, *varargs)</td>
<td>Return the gradient of an N-dimensional array.</td>
</tr>
<tr>
<td>cross(a, b, axisa, axisb, axisc, axis)</td>
<td>Return the cross product of two (arrays of) vectors.</td>
</tr>
<tr>
<td>trapz(y[, x, dx, axis])</td>
<td>Integrate along the given axis using the composite trapezoidal rule.</td>
</tr>
</tbody>
</table>

```python
numpy.prod(a, axis=None, dtype=None, out=None, keepdims=False)
```

Return the product of array elements over a given axis.

Parameters

- a : array_like
  Input data.

- axis : None or int or tuple of ints, optional
  Axis or axes along which a product is performed. The default (axis = None) is perform a product over all the dimensions of the input array. axis may be negative, in which case it counts from the last to the first axis. New in version 1.7.0. If this is a tuple of ints, a product is performed on multiple axes, instead of a single axis or all the axes as before.

- dtype : data-type, optional
  The data-type of the returned array, as well as of the accumulator in which the elements are multiplied. By default, if a is of integer type, dtype is the default platform integer. (Note: if the type of a is unsigned, then so is dtype.) Otherwise, the dtype is the same as that of a.

- out : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

- keepdims : bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.
Returns

\texttt{product\_along\_axis} : \texttt{ndarray}, see \texttt{dtype} parameter above.

An array shaped as \texttt{a} but with the specified axis removed. Returns a reference to \texttt{out} if specified.

See Also:

\texttt{ndarray.prod}

\texttt{equivalent method}

\texttt{numpy.doc.ufuncs}

Section “Output arguments”

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

\begin{verbatim}
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x)  #random
16
\end{verbatim}

Examples

By default, calculate the product of all elements:

\begin{verbatim}
>>> np.prod([1.,2.])
2.0
\end{verbatim}

Even when the input array is two-dimensional:

\begin{verbatim}
>>> np.prod([[1.,2.],[3.,4.]])
24.0
\end{verbatim}

But we can also specify the axis over which to multiply:

\begin{verbatim}
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])
\end{verbatim}

If the type of \texttt{x} is unsigned, then the output type is the unsigned platform integer:

\begin{verbatim}
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
\end{verbatim}

If \texttt{x} is of a signed integer type, then the output type is the default platform integer:

\begin{verbatim}
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == np.int
True
\end{verbatim}

\texttt{numpy.sum}(\texttt{a, axis=None, dtype=None, out=None, keepdims=False})

Sum of array elements over a given axis.

Parameters

\texttt{a} : \texttt{array\_like}

Elements to sum.

\texttt{axis} : None or int or tuple of ints, optional
Axis or axes along which a sum is performed. The default (*axis = None*) is perform a sum over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis. New in version 1.7.0. If this is a tuple of ints, a sum is performed on multiple axes, instead of a single axis or all the axes as before.

**dtype**: dtype, optional

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead.

**out**: ndarray, optional

Array into which the output is placed. By default, a new array is created. If *out* is given, it must be of the appropriate shape (the shape of *a* with *axis* removed, i.e., *numpy.delete(a.shape, axis)*). Its type is preserved. See *doc.ufuncs* (Section “Output arguments”) for more details.

**keepdims**: bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

**Returns**

**sum_along_axis**: ndarray

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

**See Also**:

*ndarray.sum*

Equivalent method.

*cumsum*

Cumulative sum of array elements.

*trapz*

Integration of array values using the composite trapezoidal rule.

*mean, average*

**Notes**

Arithmetic is modular when using integer types, and no error is raised on overflow.

**Examples**

```python
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```
If the accumulator is too small, overflow occurs:

```python
np.ones(128, dtype=np.int8).sum(dtype=np.int8)
```

```
-128
```

`numpy.nansum(a, axis=None, dtype=None, out=None, keepdims=0)`

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

FutureWarning: In Numpy versions <= 1.8 Nan is returned for slices that are all-NaN or empty. In later versions zero will be returned.

**Parameters**

- `a`: array_like
  
  Array containing numbers whose sum is desired. If `a` is not an array, a conversion is attempted.

- `axis`: int, optional
  
  Axis along which the sum is computed. The default is to compute the sum of the flattened array.

- `dtype`: data-type, optional
  
  The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of `a` is used. An exception is when `a` has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact. New in version 1.8.0.

- `out`: ndarray, optional
  
  Alternate output array in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See `doc.ufuncs` for details. The casting of NaN to integer can yield unexpected results. New in version 1.8.0.

- `keepdims`: bool, optional
  
  If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`. New in version 1.8.0.

**Returns**

- `y`: ndarray or numpy scalar

**See Also:**

- `numpy.sum`
  
  Sum across array propagating NaNs.

- `isnan`
  
  Show which elements are NaN.

- `isfinite`
  
  Show which elements are not NaN or +/-inf.

**Notes**

If both positive and negative infinity are present, the sum will be Not A Number (NaN).

Numpy integer arithmetic is modular. If the size of a sum exceeds the size of an integer accumulator, its value will wrap around and the result will be incorrect. Specifying `dtype=double` can alleviate that problem.
Examples

```python
>>> np.nansum(1)
1
>>> np.nansum([1])
1
>>> np.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> np.nansum(a)
3.0
>>> np.nansum(a, axis=0)
array([ 2.,  1.])
>>> np.nansum([1, np.nan, np.inf])
inf
>>> np.nansum([1, np.nan, np.NINF])
-inf
>>> np.nansum([1, np.nan, np.inf, -np.inf]) # both +/- infinity present
nan
```

`numpy.cumprod(a, axis=None, dtype=None, out=None)`

Return the cumulative product of elements along a given axis.

Parameters

- `a`: array_like
  Input array.

- `axis`: int, optional
  Axis along which the cumulative product is computed. By default the input is flattened.

- `dtype`: dtype, optional
  Type of the returned array, as well as of the accumulator in which the elements are multiplied. If `dtype` is not specified, it defaults to the dtype of `a`, unless `a` has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

- `out`: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

Returns

- `cumprod`: ndarray
  A new array holding the result is returned unless `out` is specified, in which case a reference to out is returned.

See Also:

- `numpy.doc.ufuncs`
  Section “Output arguments”

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.
Examples

```python
>>> a = np.array([1, 2, 3])
>>> np.cumprod(a)  # intermediate results 1, 1*2
...                # total product 1*2*3 = 6
array([1, 2, 6])
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float)  # specify type of output
array([ 1.,  2.,  6., 24., 120., 720.])
```

The cumulative product for each column (i.e., over the rows) of `a`:

```python
>>> np.cumprod(a, axis=0)
array([[ 1.,  2.,  6.],
       [ 4., 10., 18.]])
```

The cumulative product for each row (i.e., over the columns) of `a`:

```python
>>> np.cumprod(a, axis=1)
array([[ 1,  2,  6],
       [ 4, 20, 120]])
```

**numpy.cumsum(a, axis=None, dtype=None, out=None)**

Return the cumulative sum of the elements along a given axis.

**Parameters**

- `a`: array_like
  Input array.
- `axis`: int, optional
  Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- `dtype`: dtype, optional
  Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the dtype of `a`, unless `a` has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.
- `out`: ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See `doc.ufuncs (Section “Output arguments”) for more details.

**Returns**

- `cumsum_along_axis`: ndarray.
  A new array holding the result is returned unless `out` is specified, in which case a reference to `out` is returned. The result has the same size as `a`, and the same shape as `a` if `axis` is not None or `a` is a 1-d array.

**See Also:**

- `sum`  
  Sum array elements.
- `trapz`  
  Integration of array values using the composite trapezoidal rule.

3.20. Mathematical functions
diff

Calculate the n-th order discrete difference along given axis.

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21]) # specifies type of output value(s)
array([ 1.,  3.,  6., 10., 15., 21.])
>>> np.cumsum(a, axis=0)  # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a, axis=1)  # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

numpy.diff(a, n=1, axis=-1)

Calculate the n-th order discrete difference along given axis.

The first order difference is given by \( \text{out}[n] = a[n+1] - a[n] \) along the given axis, higher order differences are calculated by using `diff` recursively.

Parameters

- `a` : array_like
  
  Input array

- `n` : int, optional
  
  The number of times values are differentiated.

- `axis` : int, optional
  
  The axis along which the difference is taken, default is the last axis.

Returns

- `diff` : ndarray
  
  The \( n \) order differences. The shape of the output is the same as \( a \) except along \( axis \) where the dimension is smaller by \( n \).

See Also:

`gradient`, `ediff1d`, `cumsum`

Examples

```python
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.diff(x)
array([1, 1, -7])
>>> np.diff(x, n=2)
array([1, 1, -10])
```
```python
>>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])
>>> np.diff(x)
array([[2, 3, 4],
[5, 1, 2]])
>>> np.diff(x, axis=0)
array([[-1, 2, 0, -2]])
```

**numpy.ediff1d** *(ary, to_end=None, to_begin=None)*

The differences between consecutive elements of an array.

**Parameters**

- *ary*: array_like
  If necessary, will be flattened before the differences are taken.
- *to_end*: array_like, optional
  Number(s) to append at the end of the returned differences.
- *to_begin*: array_like, optional
  Number(s) to prepend at the beginning of the returned differences.

**Returns**

- *ediff1d*: ndarray
  The differences. Loosely, this is *ary*.flat[1:] − *ary*.flat[:-1].

**See Also:**

diff, gradient

**Notes**

When applied to masked arrays, this function drops the mask information if the *to_begin* and/or *to_end* parameters are used.

**Examples**

```python
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.ediff1d(x)
array([ 1, 2, 3, -7])
>>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
array([-99, 1, 2, 3, -7, 88, 99])
```

The returned array is always 1D.

```python
>>> y = [[1, 2, 4], [1, 6, 24]]
>>> np.ediff1d(y)
array([1, 2, -3, 5, 18])
```

**numpy.gradient** *(f, *varargs)*

Return the gradient of an N-dimensional array.

The gradient is computed using central differences in the interior and first differences at the boundaries. The returned gradient hence has the same shape as the input array.

**Parameters**

- *f*: array_like
  An N-dimensional array containing samples of a scalar function.
- *varargs*: scalars
0, 1, or N scalars specifying the sample distances in each direction, that is: \( dx, dy, dz \). ... The default distance is 1.

**Returns**

- **gradient**: ndarray

  N arrays of the same shape as \( f \) giving the derivative of \( f \) with respect to each dimension.

**Examples**

```python
gf = 1, 2, 4, 7, 11, 16], dtype=np.float)
>>> np.gradient(x)
array([ 1. , 1.5, 2.5, 3.5, 4.5, 5. ])
>>> np.gradient(x, 2)
array([ 0.5 , 0.75, 1.25, 1.75, 2.25, 2.5 ])
```

```python
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=np.float))
[array([[ 2., 2., -1.],
    [ 2., 2., -1.]]),
 array([[ 1. , 2.5, 4. ],
    [ 1. , 1. , 1. ]])]
```

**numpy**.cross(\( a, b, axisa=-1, axisb=-1, axisc=-1, axis=None \))

Return the cross product of two (arrays of) vectors.

The cross product of \( a \) and \( b \) in \( \mathbb{R}^3 \) is a vector perpendicular to both \( a \) and \( b \). If \( a \) and \( b \) are arrays of vectors, the vectors are defined by the last axis of \( a \) and \( b \) by default, and these axes can have dimensions 2 or 3. Where the dimension of either \( a \) or \( b \) is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

**Parameters**

- **a**: array_like

  Components of the first vector(s).

- **b**: array_like

  Components of the second vector(s).

- **axisa**: int, optional

  Axis of \( a \) that defines the vector(s). By default, the last axis.

- **axisb**: int, optional

  Axis of \( b \) that defines the vector(s). By default, the last axis.

- **axisc**: int, optional

  Axis of \( c \) containing the cross product vector(s). By default, the last axis.

**Returns**

- **c**: ndarray

  Vector cross product(s).

**Raises**

- **ValueError**
When the dimension of the vector(s) in \(a\) and/or \(b\) does not equal 2 or 3.

See Also:

inner
Inner product

outer
Outer product.

ix_
Construct index arrays.

Examples

Vector cross-product.

```python
generate the code here

One vector with dimension 2.

```python
generate the code here

Equivalently:

```python
generate the code here

Both vectors with dimension 2.

```python
generate the code here

Multiple vector cross-products. Note that the direction of the cross product vector is defined by the right-hand rule.

```python
generate the code here

The orientation of \(c\) can be changed using the \texttt{axisc} keyword.

```python
generate the code here

Change the vector definition of \(x\) and \(y\) using \texttt{axisa} and \texttt{axisb}. 

3.20. Mathematical functions
```python
>>> x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4, 5, 6], [1, 2, 3]])
>>> np.cross(x, y)
array([[ -6, 12, -6],
       [ 0,  0,  0],
       [ 6, -12,  6]])
>>> np.cross(x, y, axisa=0, axisb=0)
array([[ -24,  48, -24],
       [-30,  60, -30],
       [-36,  72, -36]])
```

def numpy.trapz(y, x=None, dx=1.0, axis=-1)

Integrate along the given axis using the composite trapezoidal rule.

Integrate $y(x)$ along given axis.

**Parameters**

- `y`: array_like
  - Input array to integrate.

- `x`: array_like, optional
  - If $x$ is None, then spacing between all $y$ elements is $dx$.

- `dx`: scalar, optional
  - If $x$ is None, spacing given by $dx$ is assumed. Default is 1.

- `axis`: int, optional
  - Specify the axis.

**Returns**

- `trapz`: float
  - Definite integral as approximated by trapezoidal rule.

**See Also:**

`sum`, `cumsum`

**Notes**

Image [R249] illustrates trapezoidal rule – y-axis locations of points will be taken from $y$ array, by default x-axis distances between points will be 1.0, alternatively they can be provided with $x$ array or with $dx$ scalar. Return value will be equal to combined area under the red lines.

**References**

[R248], [R249]

**Examples**

```python
>>> np.trapz([1,2,3])
4.0
>>> np.trapz([1,2,3], x=[4,6,8])
8.0
>>> np.trapz([1,2,3], dx=2)
8.0
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```
3.20.5 Exponents and logarithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp(x, out)</td>
<td>Calculate the exponential of all elements in the input array.</td>
</tr>
<tr>
<td>expm1(x, out)</td>
<td>Calculate $e^x - 1$ for all elements in the array.</td>
</tr>
<tr>
<td>exp2(x, out)</td>
<td>Calculate $2^p$ for all $p$ in the input array.</td>
</tr>
<tr>
<td>log(x, out)</td>
<td>Natural logarithm, element-wise.</td>
</tr>
<tr>
<td>log10(x, out)</td>
<td>Return the base 10 logarithm of the input array, element-wise.</td>
</tr>
<tr>
<td>log2(x, out)</td>
<td>Base-2 logarithm of $x$.</td>
</tr>
<tr>
<td>log1p(x, out)</td>
<td>Return the natural logarithm of one plus the input array, element-wise.</td>
</tr>
<tr>
<td>logaddexp(x1, x2, out)</td>
<td>Logarithm of the sum of exponentiations of the inputs.</td>
</tr>
<tr>
<td>logaddexp2(x1, x2, out)</td>
<td>Logarithm of the sum of exponentiations of the inputs in base-2.</td>
</tr>
</tbody>
</table>

**numpy.exp** (x[, out])

Calculate the exponential of all elements in the input array.

Parameters

- **x**: array_like
  - Input values.

Returns

- **out**: ndarray
  - Output array, element-wise exponential of $x$.

See Also:

- **expm1**: Calculate $e^x - 1$ for all elements in the array.
- **exp2**: Calculate $2^p$ for all elements in the array.

Notes

The irrational number $e$ is also known as Euler’s number. It is approximately 2.718281, and is the base of the natural logarithm, \( \ln \) (this means that, if \( x = \ln y = \log_e y \), then \( e^x = y \). For real input, \( \exp(x) \) is always positive.

For complex arguments, \( x = a + ib \), we can write \( e^x = e^a e^{ib} \). The first term, \( e^a \), is already known (it is the real argument, described above). The second term, \( e^{ib} \), is \( \cos b + i \sin b \), a function with magnitude 1 and a periodic phase.

References

[R18], [R19]

Examples

Plot the magnitude and phase of \( \exp(x) \) in the complex plane:
```python
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)

>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
... extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Magnitude of exp(x)')

>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
... extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi])
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

```python
numpy.exp1(x[, out]) = <ufunc 'exp1'>
Calculate \( \exp(x) - 1 \) for all elements in the array.

Parameters
  x : array_like
      Input values.

Returns
  out : ndarray
      Element-wise exponential minus one: \( \text{out} = \exp(x) - 1 \).

See Also:

  log1p
      \( \log(1 + x) \), the inverse of expm1.

Notes
This function provides greater precision than \( \exp(x) - 1 \) for small values of \( x \).
```
Examples

The true value of \( \exp(1e-10) - 1 \) is \( 1.00000000005e-10 \) to about 32 significant digits. This example shows the superiority of \( \expm1 \) in this case.

```python
>>> np.expm1(1e-10)
1.00000000005e-10
>>> np.exp(1e-10) - 1
1.000000082740371e-10
```

```
numpy.exp2(x[, out]) = <ufunc 'exp2'>
Calculate \( 2^p \) for all \( p \) in the input array.

Parameters
  x : array_like
      Input values.

  out : ndarray, optional
      Array to insert results into.

Returns
  out : ndarray
      Element-wise \( 2^x \).

See Also:
  power

Notes
New in version 1.3.0.
```

```
Examples

```python
>>> np.exp2([2, 3])
array([ 4.,  8.])
```

```
numpy.log(x[, out]) = <ufunc 'log'>
Natural logarithm, element-wise.

The natural logarithm \( \log \) is the inverse of the exponential function, so that \( \log(\exp(x)) = x \). The natural logarithm is logarithm in base e.

Parameters
  x : array_like
      Input value.

Returns
  y : ndarray
      The natural logarithm of \( x \), element-wise.

See Also:
  log10, log2, log1p, emath.log

Notes
Logarithm is a multivalued function: for each \( x \) there is an infinite number of \( z \) such that \( \exp(z) = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\).

3.20. Mathematical functions 907
For real-valued input data types, \( \log \) always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \( \text{nans} \) and sets the invalid floating point error flag.

For complex-valued input, \( \log \) is a complex analytical function that has a branch cut \([-\inf, 0]\) and is continuous from above on it. \( \log \) handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References
[R42], [R43]

Examples
```python
good = np.arange(5, dtype=float)
good
array([0., 1., 2., 3., 4.])
```
Returns

\[ y : \text{ndarray} \]

Base-2 logarithm of \( x \).

See Also:

\texttt{log, log10, log1p, emath.log2}

Notes

New in version 1.3.0. Logarithm is a multivalued function: for each \( x \) there is an infinite number of \( z \) such that \( 2^z = x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\).

For real-valued input data types, \texttt{log2} always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \texttt{nan} and sets the \texttt{invalid} floating point error flag.

For complex-valued input, \texttt{log2} is a complex analytical function that has a branch cut \([-\infty, 0]\) and is continuous from above on it. \texttt{log2} handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

Examples

```python
>>> x = np.array([0, 1, 2, 2**4])
>>> np.log2(x)
array([-Inf,  0.,  1.,  4.])

>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
>>> np.log2(xi)
array([ 0.+2.26618007j,  0.+0.j ,  1.+0.j ,  2.+2.26618007j])
```

\texttt{numpy.log1p(x[, out]) = <ufunc \texttt{\textless log1p\textgreater }>

Return the natural logarithm of one plus the input array, element-wise.

Calculates \( \log(1 + x) \).

Parameters

\[ x : \text{array_like} \]

Input values.

Returns

\[ y : \text{ndarray} \]

Natural logarithm of \( 1 + x \), element-wise.

See Also:

\texttt{expm1}

\texttt{exp(x) - 1}, the inverse of \texttt{log1p}.

Notes

For real-valued input, \texttt{log1p} is accurate also for \( x \) so small that \( 1 + x \approx 1 \) in floating-point accuracy.

Logarithm is a multivalued function: for each \( x \) there is an infinite number of \( z \) such that \( \exp(z) = 1 + x \). The convention is to return the \( z \) whose imaginary part lies in \([-\pi, \pi]\).

For real-valued input data types, \texttt{log1p} always returns real output. For each value that cannot be expressed as a real number or infinity, it yields \texttt{nan} and sets the \texttt{invalid} floating point error flag.
For complex-valued input, \texttt{log1p} is a complex analytical function that has a branch cut \([-\infty, -1]\) and is continuous from above on it. \texttt{log1p} handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R46], [R47]

Examples

```python
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

numpy.\texttt{logaddexp}(x1, x2[, out]) = \texttt{ufunc \textquotesingle logaddexp	extquotesingle}

Logarithm of the sum of exponentiations of the inputs.

Calculates \(\log(e^{x1} + e^{x2})\). This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 : array_like

Input values.

Returns

result : ndarray

Logarithm of \(e^{x1} + e^{x2}\).

See Also:

\texttt{logaddexp2}

Logarithm of the sum of exponentiations of inputs in base 2.

Notes

New in version 1.3.0.

Examples

```python
>>> prob1 = np.log(1e-50)
>>> prob2 = np.log(2.5e-50)
>>> prob12 = np.logaddexp(prob1, prob2)
>>> prob12
-113.87649168120691
>>> np.exp(prob12)
3.5000000000000057e-50
```

numpy.\texttt{logaddexp2}(x1, x2[, out]) = \texttt{ufunc \textquotesingle logaddexp2	extquotesingle}

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates \(\log_2(2^{x1} + 2^{x2})\). This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

Parameters

x1, x2 : array_like
Input values.

**out**: ndarray, optional

Array to store results in.

**Returns**

result : ndarray

Base-2 logarithm of \(2^{x1} + 2^{x2}\).

**See Also:**

logaddexp

Logarithm of the sum of exponentiations of the inputs.

**Notes**

New in version 1.3.0.

**Examples**

```python
>>> prob1 = np.log2(1e-50)
>>> prob2 = np.log2(2.5e-50)
>>> prob12 = np.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12
3.4999999999999914e-50
```

### 3.20.6 Other special functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i0(x)</code></td>
<td>Modified Bessel function of the first kind, order 0.</td>
</tr>
<tr>
<td><code>sinc(x)</code></td>
<td>Return the sinc function.</td>
</tr>
</tbody>
</table>

**numpy.i0(x)**

Modified Bessel function of the first kind, order 0.

Usually denoted \(I_0\). This function does broadcast, but will not “up-cast” int dtype arguments unless accompanied by at least one float or complex dtype argument (see Raises below).

**Parameters**

- **x**: array_like, dtype float or complex
  
  Argument of the Bessel function.

**Returns**

- **out**: ndarray, shape = x.shape, dtype = x.dtype
  
  The modified Bessel function evaluated at each of the elements of x.

**Raises**

- **TypeError**: array cannot be safely cast to required type
  
  If argument consists exclusively of int dtypes.

**See Also:**

scipy.special.iv, scipy.special.ive
Notes

We use the algorithm published by Clenshaw [R29] and referenced by Abramowitz and Stegun [R30], for which the function domain is partitioned into the two intervals [0,8] and (8,inf), and Chebyshev polynomial expansions are employed in each interval. Relative error on the domain [0,30] using IEEE arithmetic is documented [R31] as having a peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

References

[R29], [R30], [R31]

Examples

>>> np.i0([0.])
array(1.0)
>>> np.i0([0., 1. + 2j])
array([ 1.00000000+0.j , 0.18785373+0.64616944j])

def numpy.sinc(x):
    Return the sinc function.
    The sinc function is \( \frac{\sin(\pi x)}{\pi x} \).

Parameters

x : ndarray
    Array (possibly multi-dimensional) of values for which to to calculate \( \text{sinc}(x) \).

Returns

out : ndarray
    \( \text{sinc}(x) \), which has the same shape as the input.

Notes

\( \text{sinc}(0) \) is the limit value 1.

The name sinc is short for “sine cardinal” or “sinus cardinalis”.

The sinc function is used in various signal processing applications, including in anti-aliasing, in the construction of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the sinc function.

References

[R244], [R245]

Examples

>>> x = np.linspace(-4, 4, 41)
>>> np.sinc(x)
array([-3.89804309e-17, -4.92362781e-02, -8.40918587e-02, -8.90384387e-02, -5.84680802e-02, 3.89804309e-02, 1.16434881e-01, 1.26137788e-01, 8.5044803e-02, -3.89804309e-02, -1.03943254e-01, -1.89206682e-01, -2.16236208e-01, -1.55914881e-01, 3.89804309e-17, 2.33872321e-01, 5.04551152e-01, 7.56826729e-01, 9.35489284e-01, 1.00000000e+00, 9.35489284e-01, 7.56826729e-01, 5.04551152e-01, 2.33872321e-01, 3.89804309e-17, -1.55914881e-01,
```python
>>> plt.plot(x, np.sinc(x))
[<matplotlib.lines.Line2D object at 0x...>]
```
```
>>> plt.title("Sinc Function")
<matplotlib.text.Text object at 0x...>
```
```
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
```
```
>>> plt.xlabel("X")
<matplotlib.text.Text object at 0x...>
```
```
>>> plt.show()
```
```
It works in 2-D as well:
```
>>> x = np.linspace(-4, 4, 401)
```
```
>>> xx = np.outer(x, x)
```
```
>>> plt.imshow(np.sinc(xx))
<matplotlib.image.AxesImage object at 0x...>
```
```
3.20.7 Floating point routines
```
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>signbit(x[, out])</code></td>
<td>Returns element-wise True where signbit is set (less than zero).</td>
</tr>
<tr>
<td><code>copysign(x1, x2[, out])</code></td>
<td>Change the sign of x1 to that of x2, element-wise.</td>
</tr>
<tr>
<td><code>frexp(x[, out1, out2])</code></td>
<td>Decompose the elements of x into mantissa and twos exponent.</td>
</tr>
<tr>
<td><code>ldexp(x1, x2[, out])</code></td>
<td>Returns x1 * 2**x2, element-wise.</td>
</tr>
</tbody>
</table>
```
```
numpy.signbit(x[, out]) = <ufunc ‘signbit’>

Returns element-wise True where signbit is set (less than zero).

Parameters

- **x**: array_like
  - The input value(s).
- **out**: ndarray, optional
  - Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

- **result**: ndarray of bool
  - Output array, or reference to out if that was supplied.

Examples

```python
>>> np.signbit(-1.2)
True
```
```
numpy.copysign(x1, x2[, out]) = <ufunc ‘copysign’>

Change the sign of x1 to that of x2, element-wise.
If both arguments are arrays or sequences, they have to be of the same length. If \(x_2\) is a scalar, its sign will be copied to all elements of \(x_1\).

**Parameters**

- \(x_1\) : array_like
  Values to change the sign of.

- \(x_2\) : array_like
  The sign of \(x_2\) is copied to \(x_1\).

- **out** : ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

**Returns**

- **out** : array_like
  The values of \(x_1\) with the sign of \(x_2\).

**Examples**

```python
>>> np.copysign(1.3, -1)
-1.3
>>> 1/np.copysign(0, 1)
inf
>>> 1/np.copysign(0, -1)
-inf
```

```python
>>> np.copysign([-1, 0, 1], -1.1)
array([-1., -0., -1.])
>>> np.copysign([-1, 0, 1], np.arange(3)-1)
array([-1., 0., 1.])
```

**numpy.frexp**

Decompose the elements of \(x\) into mantissa and twos exponent.

Returns \((mantissa, exponent)\), where \(x = mantissa * 2**exponent\). The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

**Parameters**

- \(x\) : array_like
  Array of numbers to be decomposed.

- out1: ndarray, optional
  Output array for the mantissa. Must have the same shape as \(x\).

- out2: ndarray, optional
  Output array for the exponent. Must have the same shape as \(x\).

**Returns**

\((mantissa, exponent)\) : tuple of ndarrays, (float, int)

- mantissa is a float array with values between -1 and 1. exponent is an int array which represents the exponent of 2.

**See Also:**

**ldexp**

Compute \(y = x_1 * 2**x_2\), the inverse of frexp.
Notes

Complex dtypes are not supported, they will raise a TypeError.

Examples

```python
>>> x = np.arange(9)
>>> y1, y2 = np.frexp(x)
>>> y1
array([ 0. , 0.5 , 0.5 , 0.75 , 0.5 , 0.625, 0.75 , 0.875,
       0.5 ])
>>> y2
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8.])
```

numpy.ldexp(x1, x2[, out]) = <ufunc 'ldexp'>

Returns x1 * 2**x2, element-wise.

The mantissas x1 and twos exponents x2 are used to construct floating point numbers x1 * 2**x2.

Parameters

- x1 : array_like
  Array of multipliers.
- x2 : array_like, int
  Array of twos exponents.
- out : ndarray, optional
  Output array for the result.

Returns

- y : ndarray or scalar
  The result of x1 * 2**x2.

See Also:

- frexp
  Return (y1, y2) from x = y1 * 2**y2, inverse to ldexp.

Notes

Complex dtypes are not supported, they will raise a TypeError.

ldexp is useful as the inverse of frexp, if used by itself it is more clear to simply use the expression x1 * 2**x2.

Examples

```python
>>> np.ldexp(5, np.arange(4))
array([ 5., 10., 20., 40.], dtype=float32)
>>> x = np.arange(6)
>>> np.ldexp(*np.frexp(x))
array([ 0., 1., 2., 3., 4., 5.])
```
3.20.8 Arithmetic operations
NumPy Reference, Release 1.8.1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x1, x2[, out])</code></td>
<td>Add arguments element-wise.</td>
</tr>
<tr>
<td><code>reciprocal(x[, out])</code></td>
<td>Return the reciprocal of the argument, element-wise.</td>
</tr>
<tr>
<td><code>negative(x[, out])</code></td>
<td>Numerical negative, element-wise.</td>
</tr>
<tr>
<td><code>multiply(x1, x2[, out])</code></td>
<td>Multiply arguments element-wise.</td>
</tr>
<tr>
<td><code>divide(x1, x2[, out])</code></td>
<td>Divide arguments element-wise.</td>
</tr>
<tr>
<td><code>power(x1, x2[, out])</code></td>
<td>First array elements raised to powers from second array, element-wise.</td>
</tr>
<tr>
<td><code>subtract(x1, x2[, out])</code></td>
<td>Subtract arguments, element-wise.</td>
</tr>
<tr>
<td><code>true_divide(x1, x2[, out])</code></td>
<td>Returns a true division of the inputs, element-wise.</td>
</tr>
<tr>
<td><code>floor_divide(x1, x2[, out])</code></td>
<td>Return the largest integer smaller or equal to the division of the inputs.</td>
</tr>
<tr>
<td><code>fmod(x1, x2[, out])</code></td>
<td>Return the element-wise remainder of division.</td>
</tr>
<tr>
<td><code>mod(x1, x2[, out])</code></td>
<td>Return element-wise remainder of division.</td>
</tr>
<tr>
<td><code>modf(x[, out1, out2])</code></td>
<td>Return the fractional and integral parts of an array, element-wise.</td>
</tr>
<tr>
<td><code>remainder(x1, x2[, out])</code></td>
<td>Return element-wise remainder of division.</td>
</tr>
</tbody>
</table>

**numpy.add(x1, x2[, out]) = <ufunc ‘add’>**

Add arguments element-wise.

**Parameters**

- `x1, x2` : array_like
  - The arrays to be added. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

**Returns**

- `add` : ndarray or scalar
  - The sum of `x1` and `x2`, element-wise. Returns a scalar if both `x1` and `x2` are scalars.

**Notes**

Equivalent to `x1 + x2` in terms of array broadcasting.

**Examples**

```python
gnp.add(1.0, 4.0) 5.0
gnp.add(1.0, 4.0) 5.0
```

**numpy.reciprocal(x[, out]) = <ufunc ‘reciprocal’>**

Return the reciprocal of the argument, element-wise.

**Parameters**

- `x` : array_like
  - Input array.

**Returns**

- `y` : ndarray
  - Return array.
Notes

Note: This function is not designed to work with integers.

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

Examples

```python
>>> np.reciprocal(2.)
0.5
>>> np.reciprocal([1, 2., 3.33])
array([ 1. , 0.5 , 0.3003003])
```

```
numpy.negative(x[, out]) = <ufunc ‘negative’>
Numerical negative, element-wise.

Parameters
x : array_like or scalar
   Input array.

Returns
y : ndarray or scalar
   Returned array or scalar: y = -x.
```

```
numpy.multiply(x1, x2[, out]) = <ufunc ‘multiply’>
Multiply arguments element-wise.

Parameters
x1, x2 : array_like
   Input arrays to be multiplied.

Returns
y : ndarray
   The product of x1 and x2, element-wise. Returns a scalar if both x1 and x2 are scalars.
```

Notes
Equivalent to x1 * x2 in terms of array broadcasting.

Examples

```python
>>> np.multiply(2.0, 4.0)
8.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.multiply(x1, x2)
array([[ 0.,  1.,  4.],
       [ 0.,  4., 10.],
       [ 0.,  7., 16.]])
```
numpy.divide(x1, x2[, out]) = <ufunc 'divide'>
Divide arguments element-wise.

Parameters
x1 : array_like
    Dividend array.
x2 : array_like
    Divisor array.
out : ndarray, optional
    Array into which the output is placed. Its type is preserved and it must be of the right
    shape to hold the output. See doc.ufuncs.

Returns
y : {ndarray, scalar}
    The quotient x1/x2, element-wise. Returns a scalar if both x1 and x2 are scalars.

See Also:
seterr
    Set whether to raise or warn on overflow, underflow and division by zero.

Notes
Equivalent to x1 / x2 in terms of array-broadcasting.
Behavior on division by zero can be changed using seterr.
When both x1 and x2 are of an integer type, divide will return integers and throw away the fractional part.
Moreover, division by zero always yields zero in integer arithmetic.

Examples
>>> np.divide(2.0, 4.0)
0.5
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.divide(x1, x2)
array([[ NaN,  1.,  1.],
       [ Inf,  4.,  2.5],
       [ Inf,  7.,  4. ]])

Note the behavior with integer types:
>>> np.divide(2, 4)
0
>>> np.divide(2, 4.)
0.5

Division by zero always yields zero in integer arithmetic, and does not raise an exception or a warning:
>>> np.divide(np.array([0, 1], dtype=int), np.array([0, 0], dtype=int))
array([[0, 0]])

Division by zero can, however, be caught using seterr:
```python
>>> old_err_state = np.seterr(divide='raise')
>>> np.divide(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: divide by zero encountered in divide

>>> ignored_states = np.seterr(**old_err_state)
>>> np.divide(1, 0)
0
```

```python
numpy.power(x1, x2[, out]) = <ufunc 'power'>
```

First array elements raised to powers from second array, element-wise.

Raise each base in `x1` to the positionally-corresponding power in `x2`. `x1` and `x2` must be broadcastable to the same shape.

**Parameters**
- `x1` : array_like
  The bases.
- `x2` : array_like
  The exponents.

**Returns**
- `y` : ndarray
  The bases in `x1` raised to the exponents in `x2`.

**Examples**

Cube each element in a list.

```python
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.power(x1, 3)
array([ 0, 1, 8, 27, 64, 125])
```

Raise the bases to different exponents.

```python
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.power(x1, x2)
array([ 0., 1., 8., 27., 16., 5.])
```

The effect of broadcasting.

```python
>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.power(x1, x2)
array([[ 0, 1, 8, 27, 16,  5],
       [ 0, 1, 8, 27, 16,  5]])
```

```python
numpy.subtract(x1, x2[, out]) = <ufunc 'subtract'>
```

Subtract arguments, element-wise.

**Parameters**
- `x1, x2` : array_like
```
The arrays to be subtracted from each other.

**Returns**

`y` : ndarray

The difference of `x1` and `x2`, element-wise. Returns a scalar if both `x1` and `x2` are scalars.

**Notes**

Equivalent to `x1 - x2` in terms of array broadcasting.

**Examples**

```python
>>> np.subtract(1.0, 4.0)
-3.0

>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.subtract(x1, x2)
array([[ 0., 0., 0.],
       [ 3., 3., 3.],
       [ 6., 6., 6.]])
```

```
numpy.true_divide(x1, x2[, out]) = <ufunc 'true_divide'>
```

Returns a true division of the inputs, element-wise.

Instead of the Python traditional ‘floor division’, this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

**Parameters**

`x1` : array_like

Dividend array.

`x2` : array_like

Divisor array.

**Returns**

`out` : ndarray

Result is scalar if both inputs are scalar, ndarray otherwise.

**Notes**

The floor division operator `//` was added in Python 2.2 making `//` and `/` equivalent operators. The default floor division operation of `/` can be replaced by true division with `from __future__ import division`.

In Python 3.0, `//` is the floor division operator and `/` the true division operator. The `true_divide(x1, x2)` function is equivalent to true division in Python.

**Examples**

```python
>>> x = np.arange(5)
>>> np.true_divide(x, 4)
array([ 0. , 0.25, 0.5 , 0.75, 1. ])
```

```python
>>> x/4
array([0, 0, 0, 0, 1])
```

```python
>>> x//4
array([0, 0, 0, 0, 1])
```
>>> from __future__ import division
>>> x/4
array([ 0. , 0.25, 0.5 , 0.75, 1. ])
>>> x//4
array([0, 0, 0, 0, 1])

numpy.floor_divide(x1, x2[, out]) = <ufunc ‘floor_divide’>
Return the largest integer smaller or equal to the division of the inputs.

Parameters
  x1 : array_like
    Numerator.
  x2 : array_like
    Denominator.

Returns
  y : ndarray
    \lfloor x1/x2 \rfloor

See Also:
  divide
    Standard division.
  floor
    Round a number to the nearest integer toward minus infinity.
  ceil
    Round a number to the nearest integer toward infinity.

Examples
>>> np.floor_divide(7,3)
2
>>> np.floor_divide([1., 2., 3., 4.], 2.5)
array([ 0., 0., 1., 1.])

numpy.fmod(x1, x2[, out]) = <ufunc ‘fmod’>
Return the element-wise remainder of division.
This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the
dividend x1. It is equivalent to the Matlab(TM) rem function and should not be confused with the Python
modulus operator x1 % x2.

Parameters
  x1 : array_like
    Dividend.
  x2 : array_like
    Divisor.

Returns
  y : array_like
    The remainder of the division of x1 by x2.

See Also:
remainder

Equivalent to the Python % operator.

divide

Notes

The result of the modulo operation for negative dividend and divisors is bound by conventions. For fmod, the sign of result is the sign of the dividend, while for remainder the sign of the result is the sign of the divisor. The fmod function is equivalent to the Matlab(TM) rem function.

Examples

```python
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
a = array([-1, 0, -1, 1, 0, 1])
```

```python
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)
a = array([1, 0, 1, 1, 0, 1])
```

```python
>>> np.fmod([5, 3], [2.0, 2.0])
a = array([ 1., 1.])
```

```python
>>> a = np.arange(-3, 3).reshape(3, 2)
>>> a
array([[-3, -2],
       [-1, 0],
       [ 1, 2]])
```

```python
>>> np.fmod(a, [2, 2])
a = array([[-1, 0],
       [-1, 0],
       [ 1, 0]])
```

numpy.mod(x1, x2[, out]) = <ufunc 'remainder'>

Return element-wise remainder of division.

Computes \( x1 - \lfloor x1 / x2 \rfloor \times x2 \), the result has the same sign as the divisor \( x2 \). It is equivalent to the Python modulus operator \( x1 \% x2 \) and should not be confused with the Matlab(TM) rem function.

Parameters

- **x1**: array_like
  - Dividend array.
- **x2**: array_like
  - Divisor array.
- **out**: ndarray, optional
  - Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

- **y**: ndarray
  - The remainder of the quotient \( x1 \div x2 \), element-wise. Returns a scalar if both \( x1 \) and \( x2 \) are scalars.

See Also:

- **fmod**
  - Equivalent of the Matlab(TM) rem function.

divide, floor

3.20. Mathematical functions
Notes
Returns 0 when \(x2\) is 0 and both \(x1\) and \(x2\) are (arrays of) integers.

Examples

```python
```np.remainder([4, 7], [2, 3])
array([0, 1])
```np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

numpy.modf(x[, out1, out2]) = <ufunc ‘modf’>

Return the fractional and integral parts of an array, element-wise.

The fractional and integral parts are negative if the given number is negative.

Parameters

- \(x\) : array_like
  Input array.

Returns

- \(y1\) : ndarray
  Fractional part of \(x\).

- \(y2\) : ndarray
  Integral part of \(x\).

Notes

For integer input the return values are floats.

Examples

```python
```np.modf([0, 3.5])
(array([ 0. , 0.5]), array([ 0., 3.]))
```np.modf(-0.5)
(-0.5, -0)
```

numpy.remainder (x1[, x2[, out]]) = <ufunc ‘remainder’>

Return element-wise remainder of division.

Computes \(x1 - \text{floor}(x1 / x2) \times x2\), the result has the same sign as the divisor \(x2\). It is equivalent to the Python modulus operator \(x1 \% x2\) and should not be confused with the Matlab(TM) \text{rem} function.

Parameters

- \(x1\) : array_like
  Dividend array.

- \(x2\) : array_like
  Divisor array.

- \(out\) : ndarray, optional
  Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns

- \(y\) : ndarray
The remainder of the quotient $x_1/x_2$, element-wise. Returns a scalar if both $x_1$ and $x_2$
are scalars.

See Also:

fmod

Equivalent of the Matlab(TM) rem function.

divide, floor

Notes

Returns 0 when $x_2$ is 0 and both $x_1$ and $x_2$ are (arrays of) integers.

Examples

```python
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

3.20.9 Handling complex numbers

```python
angle(z[, deg])  Return the angle of the complex argument.
real(val)  Return the real part of the elements of the array.
imag(val)  Return the imaginary part of the elements of the array.
conj(x[, out])  Return the complex conjugate, element-wise.
```

numpy.angle(z, deg=0)

Return the angle of the complex argument.

Parameters

- **z**: array_like
  A complex number or sequence of complex numbers.
- **deg**: bool, optional

Returns

- **angle**: {ndarray, scalar}
  The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

See Also:

arctan2, absolute

Examples

```python
>>> np.angle([1.0, 1.0j, 1+1j]) # in radians
array([ 0.        , 1.57079633, 0.78539816])
>>> np.angle(1+1j, deg=True) # in degrees
45.0
```

3.20. Mathematical functions
numpy.real(val)
Return the real part of the elements of the array.

Parameters
val : array_like
Input array.

Returns
out : ndarray
Output array. If val is real, the type of val is used for the output. If val has complex elements, the returned type is float.

See Also:
real_if_close, imag, angle

Examples
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.real
array([ 1., 3., 5.])
>>> a.real = 9
>>> a
array([ 9.+2.j, 9.+4.j, 9.+6.j])
>>> a.real = np.array([9, 8, 7])
>>> a
array([ 9.+2.j, 8.+4.j, 7.+6.j])

numpy.imag(val)
Return the imaginary part of the elements of the array.

Parameters
val : array_like
Input array.

Returns
out : ndarray
Output array. If val is real, the type of val is used for the output. If val has complex elements, the returned type is float.

See Also:
real, angle, real_if_close

Examples
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.imag
array([ 2., 4., 6.])
>>> a.imag = np.array([8, 10, 12])
>>> a
array([ 1. +8.j, 3.+10.j, 5.+12.j])

numpy.conj(x[, out]) = <ufunc 'conjugate'>
Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Parameters
x : array_like
Input value.

Returns

\( y \): ndarray

The complex conjugate of \( x \), with same dtype as \( y \).

Examples

```python
>>> np.conjugate(1+2j)
(1-2j)
```

```python
x = np.eye(2) + 1j * np.eye(2)
np.conjugate(x)
```

```
array([[ 1.-1.j, 0.-0.j],
       [ 0.-0.j, 1.-1.j]])
```

3.20.10 Miscellaneous

- **convolve** (a, v[, mode]) Returns the discrete, linear convolution of two one-dimensional sequences.
- **clip** (a, a_min, a_max[, out]) Clip (limit) the values in an array.
- **sqrt** (x[, out]) Return the positive square-root of an array, element-wise.
- **square** (x[, out]) Return the element-wise square of the input.
- **absolute** (x[, out]) Calculate the absolute value element-wise.
- **fabs** (x[, out]) Compute the absolute values element-wise.
- **sign** (x[, out]) Returns an element-wise indication of the sign of a number.
- **maximum** (x1, x2[, out]) Element-wise maximum of array elements.
- **minimum** (x1, x2[, out]) Element-wise minimum of array elements.
- **fmax** (x1, x2[, out]) Element-wise maximum of array elements.
- **fmin** (x1, x2[, out]) Element-wise minimum of array elements.
- **nan_to_num** (x) Replace nan with zero and inf with finite numbers.
- **real_if_close** (a[, tol]) If complex input returns a real array if complex parts are close to zero.
- **interp** (x, xp, fp[, left, right]) One-dimensional linear interpolation.

The convolution operator is often seen in signal processing, where it models the effect of a linear time-invariant system on a signal. In probability theory, the sum of two independent random variables is distributed according to the convolution of their individual distributions.

Parameters

- a : (N,) array_like
  First one-dimensional input array.
- v : (M,) array_like
  Second one-dimensional input array.
- mode : {'full', 'valid', 'same'}, optional
  ‘full’:
  By default, mode is ‘full’. This returns the convolution at each point of overlap, with an output shape of (N+M-1,). At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.

3.20. Mathematical functions
‘same’:
Mode same returns output of length \max(M, N). Boundary effects are still visible.

‘valid’:
Mode valid returns output of length \max(M, N) - \min(M, N) + 1. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Returns

out : ndarray
Discrete, linear convolution of a and v.

See Also:

scipy.signal.fftconvolve
Convolves two arrays using the Fast Fourier Transform.

scipy.linalg.toeplitz
Used to construct the convolution operator.

Notes

The discrete convolution operation is defined as

\[(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]\]

It can be shown that a convolution \(x(t) * y(t)\) in time/space is equivalent to the multiplication \(X(f)Y(f)\) in the Fourier domain, after appropriate padding (padding is necessary to prevent circular convolution). Since multiplication is more efficient (faster) than convolution, the function scipy.signal.fftconvolve exploits the FFT to calculate the convolution of large data-sets.

References

[R17]

Examples

Note how the convolution operator flips the second array before “sliding” the two across one another:

```python
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([ 0. , 1. , 2.5, 4. , 1.5])
```

Only return the middle values of the convolution. Contains boundary effects, where zeros are taken into account:

```python
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([ 1. , 2.5, 4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```python
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([ 2.5])
```

numpy.clip(a, a_min, a_max, out=None)
Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of \([0, 1]\) is specified, values smaller than 0 become 0, and values larger than 1 become 1.
Parameters

- **a**: array_like
  
  Array containing elements to clip.

- **a_min**: scalar or array_like
  
  Minimum value.

- **a_max**: scalar or array_like
  
  Maximum value. If *a_min* or *a_max* are array_like, they will be broadcasted to the shape of *a*.

- **out**: ndarray, optional
  
  The results will be placed in this array. It may be the input array for in-place clipping. *out* must be of the right shape to hold the output. Its type is preserved.

Returns

- **clipped_array**: ndarray
  
  An array with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

See Also:

- `numpy.doc.ufuncs`
  
  Section "Output arguments"

Examples

```python
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
```

```python
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```python
>>> np.clip(a, [3,4,1,1,1,4,4,4,4,4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

```
numpy.sqrt(x[, out]) = <ufunc 'sqrt'>
```

Return the positive square-root of an array, element-wise.

Parameters

- **x**: array_like
  
  The values whose square-roots are required.

- **out**: ndarray, optional
  
  Alternate array object in which to put the result; if provided, it must have the same shape as *x*.

Returns

- **y**: ndarray
  
  An array of the same shape as *x*, containing the positive square-root of each element in *x*. If any element in *x* is complex, a complex array is returned (and the square-roots of...
negative reals are calculated). If all of the elements in \( x \) are real, so is \( y \), with negative elements returning \( \text{nan} \). If \( \text{out} \) was provided, \( y \) is a reference to it.

See Also:

**lib.scimath.sqrt**

A version which returns complex numbers when given negative reals.

**Notes**

\( \text{sqrt} \) has–consistent with common convention–as its branch cut the real “interval” \([-\infty, 0)\), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

**Examples**

```python
>>> np.sqrt([1, 4, 9])
anarray([ 1., 2., 3.])
```

```python
>>> np.sqrt([4, -1, -3+4j])
anarray([ 2.+0.j, 0.+1.j, 1.+2.j])
```

```python
>>> np.sqrt([4, -1, numpy.inf])
anarray([ 2., NaN, Inf])
```

**numpy.square** \((x[, \text{out}]) = \text{ufunc \text{`square`}车载重点}

Return the element-wise square of the input.

**Parameters**

- \( x \): array_like
  
  Input data.

**Returns**

- \( \text{out} \): ndarray
  
  Element-wise \( x^2 \), of the same shape and dtype as \( x \). Returns scalar if \( x \) is a scalar.

See Also:

**numpy.linalg.matrix_power, sqrt, power**

**Examples**

```python
>>> np.square([-1j, 1])
anarray([-1.-0.j, 1.+0.j])
```

**numpy.absolute** \((x[, \text{out}]) = \text{ufunc \text{`absolute`}车载重点}

Calculate the absolute value element-wise.

**Parameters**

- \( x \): array_like
  
  Input array.

**Returns**

- \( \text{absolute} \): ndarray
  
  An ndarray containing the absolute value of each element in \( x \). For complex input, \( a + ib \), the absolute value is \( \sqrt{a^2 + b^2} \).
Examples

```python
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2, 1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over $[-10, 10]$:

```python
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(start=-10, stop=10, num=101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()
```

Plot the function over the complex plane:

```python
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10])
>>> plt.show()
```
Compute the absolute values element-wise.

This function returns the absolute values (positive magnitude) of the data in x. Complex values are not handled, use absolute to find the absolute values of complex data.

Parameters
  x : array_like
      The array of numbers for which the absolute values are required. If x is a scalar, the result y will also be a scalar.

  out : ndarray, optional
      Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output. See doc.ufuncs.

Returns
  y : {ndarray, scalar}
      The absolute values of x, the returned values are always floats.

See Also:

absolute
    Absolute values including complex types.

Examples

>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([1.2, 1.2])

Returns an element-wise indication of the sign of a number.

The sign function returns -1 if x < 0, 0 if x==0, 1 if x > 0.

Parameters
  x : array_like
Input values.

**Returns**

\[ y : \text{ndarray} \]

The sign of \( x \).

**Examples**

```python
>>> np.sign([-5., 4.5])
array([-1., 1.])
```

```python
>>> np.sign(0)
0
```

**numpy.maximum(\( x1, x2[, out] \))**  
Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

**Parameters**

\[ x1, x2 : \text{array_like} \]

The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

**Returns**

\[ y : \{\text{ndarray, scalar}\} \]

The maximum of \( x1 \) and \( x2 \), element-wise. Returns scalar if both \( x1 \) and \( x2 \) are scalars.

**See Also:**

- **minimum**  
  Element-wise minimum of two arrays, propagates NaNs.

- **fmax**  
  Element-wise maximum of two arrays, ignores NaNs.

- **amax**  
  The maximum value of an array along a given axis, propagates NaNs.

- **nanmax**  
  The maximum value of an array along a given axis, ignores NaNs.

- **fmin, amin, nanmin**

**Notes**

The maximum is equivalent to `np.where(x1 >= x2, x1, x2)` when neither \( x1 \) nor \( x2 \) are nans, but it is faster and does proper broadcasting.

**Examples**

```python
>>> np.maximum([2, 3, 4], [1, 5, 2])
array([2, 5, 4])
```

```python
>>> np.maximum(np.eye(2), [0.5, 2])  # broadcasting
array([[1. , 2. ],
       [0.5, 2. ]])
```
```python
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ NaN, NaN, NaN])
```

```python
>>> np.maximum(np.Inf, 1)
inf
```

numpy.<code>minimum</code>(<code>x1</code>, <code>x2</code>[, <code>out</code>]) = <code>ufunc 'minimum'</code>
Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

**Parameters**

- <code>x1</code>, <code>x2</code>: array_like
  The arrays holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

**Returns**

- <code>y</code>: {ndarray, scalar}
  The minimum of <code>x1</code> and <code>x2</code>, element-wise. Returns scalar if both <code>x1</code> and <code>x2</code> are scalars.

**See Also:**

- <code>maximum</code>
  Element-wise maximum of two arrays, propagates NaNs.
- <code>fmin</code>
  Element-wise minimum of two arrays, ignores NaNs.
- <code>amin</code>
  The minimum value of an array along a given axis, propagates NaNs.
- <code>nanmin</code>
  The minimum value of an array along a given axis, ignores NaNs.

**Notes**

The minimum is equivalent to `np.where(x1 <= x2, x1, x2)` when neither <code>x1</code> nor <code>x2</code> are NaNs, but it is faster and does proper broadcasting.

**Examples**

```python
>>> np.minimum([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```python
>>> np.minimum(np.eye(2), [0.5, 2])  # broadcasting
array([[ 0.5, 0.],
        [ 0., 1.]])
```

```python
>>> np.minimum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ NaN, NaN, NaN])
```

```python
>>> np.minimum(-np.Inf, 1)
-inf
```

numpy.<code>fmax</code>(<code>x1</code>, <code>x2</code>[, <code>out</code>]) = <code>ufunc 'fmax'</code>
Element-wise maximum of array elements.
Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

**Parameters**

- `x1, x2`: array_like

  The arrays holding the elements to be compared. They must have the same shape.

**Returns**

- `y`: {ndarray, scalar}

  The minimum of `x1` and `x2`, element-wise. Returns scalar if both `x1` and `x2` are scalars.

**See Also:**

- **fmin**
  Element-wise minimum of two arrays, ignores NaNs.

- **maximum**
  Element-wise maximum of two arrays, propagates NaNs.

- **amax**
  The maximum value of an array along a given axis, propagates NaNs.

- **nanmax**
  The maximum value of an array along a given axis, ignores NaNs.

- **minimum, amin, nanmin**

**Notes**

New in version 1.3.0. The `fmax` is equivalent to `np.where(x1 >= x2, x1, x2)` when neither `x1` nor `x2` are NaNs, but it is faster and does proper broadcasting.

**Examples**

```python
>>> np.fmax([2, 3, 4], [1, 5, 2])
array([2., 5., 4.])

>>> np.fmax(np.eye(2), [0.5, 2])
array([[1., 2.],
       [0.5, 2.]])

>>> np.fmax([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([0., 0., NaN])
```

```python
numpy.fmin(x1, x2[, out]) = <ufunc 'fmin'>
```

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

**Parameters**

- `x1, x2`: array_like

  The arrays holding the elements to be compared. They must have the same shape.
Returns

\[ y : \{ \text{ndarray, scalar} \} \]

The minimum of \( x_1 \) and \( x_2 \), element-wise. Returns scalar if both \( x_1 \) and \( x_2 \) are scalars.

See Also:

- \texttt{fmax}
  
  Element-wise maximum of two arrays, ignores NaNs.

- \texttt{minimum}
  
  Element-wise minimum of two arrays, propagates NaNs.

- \texttt{amin}
  
  The minimum value of an array along a given axis, propagates NaNs.

- \texttt{nanmin}
  
  The minimum value of an array along a given axis, ignores NaNs.

- \texttt{maximum, amax, nanmax}

Notes

New in version 1.3.0. The \texttt{fmin} is equivalent to \texttt{np.where}(\( x_1 \leq x_2, x_1, x_2 \)) when neither \( x_1 \) nor \( x_2 \) are NaNs, but it is faster and does proper broadcasting.

Examples

\begin{verbatim}
>>> np.fmin([2, 3, 4], [1, 5, 2])
array([2, 5, 4])

>>> np.fmin(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])

>>> np.fmin([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([ 0.,  0., NaN])
\end{verbatim}

\texttt{numpy.nan_to_num}(x)

Replace nan with zero and inf with finite numbers.

Returns an array or scalar replacing Not a Number (NaN) with zero, (positive) infinity with a very large number and negative infinity with a very small (or negative) number.

Parameters

\[ x : \text{array_like} \]

Input data.

Returns

\[ \text{out} : \text{ndarray, float} \]

Array with the same shape as \( x \) and dtype of the element in \( x \) with the greatest precision. NaN is replaced by zero, and infinity (-infinity) is replaced by the largest (smallest or most negative) floating point value that fits in the output dtype. All finite numbers are upcast to the output dtype (default float64).

See Also:

- \texttt{isinf}
  
  Shows which elements are negative or negative infinity.
**isneginf**
Shows which elements are negative infinity.

**isposinf**
Shows which elements are positive infinity.

**isnan**
Shows which elements are Not a Number (NaN).

**isfinite**
Shows which elements are finite (not NaN, not infinity)

Notes
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples
```python
golden
np.set_printoptions(precision=8)
golden
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
golden
>>> np.nan_to_num(x)
golden
array([ 1.79769313e+308, -1.79769313e+308, 0.00000000e+000, -1.28000000e+002, 1.28000000e+002])
golden
```

numpy.real_if_close(a, tol=100)
If complex input returns a real array if complex parts are close to zero.

“Close to zero” is defined as tol * (machine epsilon of the type for a).

Parameters
- **a**: array_like
  Input array.
- **tol**: float
  Tolerance in machine epsilons for the complex part of the elements in the array.

Returns
- **out**: ndarray
  If a is real, the type of a is used for the output. If a has complex elements, the returned type is float.

See Also:
real, imag, angle

Notes
Machine epsilon varies from machine to machine and between data types but Python floats on most platforms have a machine epsilon equal to 2.2204460492503131e-16. You can use ‘np.finfo(np.float).eps’ to print out the machine epsilon for floats.

Examples
```python
golden
>>> np.finfo(np.float).eps
golden
2.2204460492503131e-16
```
```python
golden
>>> np.real_if_close([2.1 + 4e-14j], tol=1000)
golden
array([ 2.1])
golden
```
>>> np.real_if_close([2.1 + 4e-13j], tol=1000)
array([ 2.1 +4.00000000e-13j])

numpy.interp(x, xp, fp, left=None, right=None)
One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given values at discrete data-points.

Parameters
  x: array_like
      The x-coordinates of the interpolated values.
  xp: 1-D sequence of floats
      The x-coordinates of the data points, must be increasing.
  fp: 1-D sequence of floats
      The y-coordinates of the data points, same length as xp.
  left: float, optional
      Value to return for x < xp[0], default is fp[0].
  right: float, optional
      Value to return for x > xp[-1], defaults is fp[-1].

Returns
  y: {float, ndarray}
      The interpolated values, same shape as x.

Raises
  ValueError
      If xp and fp have different length

Notes
Does not check that the x-coordinate sequence xp is increasing. If xp is not increasing, the results are nonsense. A simple check for increasing is:
np.all(np.diff(xp) > 0)

Examples
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([ 3. , 3. , 2.5 , 0.56, 0. ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0

Plot an interpolant to the sine function:
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
3.21 Matrix library (numpy.matlib)

This module contains all functions in the numpy namespace, with the following replacement functions that return matrices instead of ndarrays.

Functions that are also in the numpy namespace and return matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat(data[, dtype])</td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td>matrix</td>
<td>Returns a matrix from an array-like object, or from a string of data.</td>
</tr>
<tr>
<td>asmatrix(data[, dtype])</td>
<td>Interpret the input as a matrix.</td>
</tr>
<tr>
<td>bmat(obj[, ldict, gdict])</td>
<td>Build a matrix object from a string, nested sequence, or array.</td>
</tr>
</tbody>
</table>

Replacement functions in matlib

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty(shape[, dtype, order])</td>
<td>Return a new matrix of given shape and type, without initializing entries.</td>
</tr>
<tr>
<td>zeros(shape[, dtype, order])</td>
<td>Return a matrix of given shape and type, filled with zeros.</td>
</tr>
<tr>
<td>ones(shape[, dtype, order])</td>
<td>Matrix of ones.</td>
</tr>
<tr>
<td>eye(n[, M, k, dtype])</td>
<td>Return a matrix with ones on the diagonal and zeros elsewhere.</td>
</tr>
<tr>
<td>identity(n[, dtype])</td>
<td>Returns the square identity matrix of given size.</td>
</tr>
<tr>
<td>repmat(a, m, n)</td>
<td>Repeat a 0-D to 2-D array or matrix MxN times.</td>
</tr>
<tr>
<td>rand(*args)</td>
<td>Return a matrix of random values with given shape.</td>
</tr>
<tr>
<td>randn(*args)</td>
<td>Return a random matrix with data from the &quot;standard normal&quot; distribution.</td>
</tr>
</tbody>
</table>

numpy.matlib.empty (shape, dtype=None, order='C')

Return a new matrix of given shape and type, without initializing entries.
Parameters

- **shape**: int or tuple of int
  Shape of the empty matrix.

- **dtype**: data-type, optional
  Desired output data-type.

- **order**: {'C', 'F'}, optional
  Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory.

See Also:

- `empty_like`, `zeros`

Notes

`empty`, unlike `zeros`, does not set the matrix values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

```python
>>> import numpy.matlib
>>> np.matlib.empty((2, 2))  # filled with random data
matrix([[ 6.76425276e-320, 9.79033856e-307],
        [ 7.39337286e-309, 3.22135945e-309]])  #random
>>> np.matlib.empty((2, 2), dtype=int)
matrix([[ 6600475, 0],
        [ 6586976, 22740995]])  #random
```

**numpy.matlib.zeros** *(shape, dtype=None, order='C')*

Return a matrix of given shape and type, filled with zeros.

Parameters

- **shape**: int or sequence of ints
  Shape of the matrix

- **dtype**: data-type, optional
  The desired data-type for the matrix, default is float.

- **order**: {'C', 'F'}, optional
  Whether to store the result in C- or Fortran-contiguous order, default is 'C'.

Returns

- **out**: matrix
  Zero matrix of given shape, dtype, and order.

See Also:

- `numpy.zeros`
  Equivalent array function.

- `matlib.ones`
  Return a matrix of ones.

Notes

If `shape` has length one i.e. (N,), or is a scalar N, `out` becomes a single row matrix of shape (1, N).
Examples

```python
>>> import numpy.matlib
>>> np.matlib.zeros((2, 3))
matrix([[ 0., 0., 0.],
        [ 0., 0., 0.]])

>>> np.matlib.zeros(2)
matrix([[ 0., 0.]])
```

`numpy.matlib.ones(shape, dtype=None, order='C')`

Matrix of ones.

Return a matrix of given shape and type, filled with ones.

Parameters

- `shape`: {sequence of ints, int}
  Shape of the matrix
- `dtype`: data-type, optional
  The desired data-type for the matrix, default is np.float64.
- `order`: {'C', 'F'}, optional
  Whether to store matrix in C- or Fortran-contiguous order, default is ‘C’.

Returns

- `out`: matrix
  Matrix of ones of given shape, dtype, and order.

See Also:

- `ones` (Array of ones)
- `matlib.zeros` (Zero matrix)

Notes

If `shape` has length one i.e. (N,), or is a scalar N, `out` becomes a single row matrix of shape (1,N).

Examples

```python
>>> np.matlib.ones((2,3))
matrix([[ 1., 1., 1.],
        [ 1., 1., 1.]])

>>> np.matlib.ones(2)
matrix([[ 1., 1.]])
```

`numpy.matlib.eye(n, M=None, k=0, dtype=<type 'float'>)`

Return a matrix with ones on the diagonal and zeros elsewhere.

Parameters

- `n`: int
  Number of rows in the output.
- `M`: int, optional
Number of columns in the output, defaults to \( n \).

\[ k : \text{int}, \text{optional} \]

Index of the diagonal: 0 refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

\[ \text{dtype} : \text{dtype, optional} \]

Data-type of the returned matrix.

**Returns**

\[ I : \text{matrix} \]

A \( n \times M \) matrix where all elements are equal to zero, except for the \( k \)-th diagonal, whose values are equal to one.

**See Also:**

- `numpy.eye`
  Equivalent array function.

- `identity`
  Square identity matrix.

**Examples**

```python
>>> import numpy.matlib
>>> np.matlib.eye(3, k=1, dtype=float)
matrix([[ 0., 1., 0.],
        [ 0., 0., 1.],
        [ 0., 0., 0.]])
```

```python
numpy.matlib.identity(n, dtype=None)
```

Returns the square identity matrix of given size.

**Parameters**

\[ n : \text{int} \]

Size of the returned identity matrix.

\[ \text{dtype} : \text{data-type, optional} \]

Data-type of the output. Defaults to `float`.

**Returns**

\[ out : \text{matrix} \]

\( n \times n \) matrix with its main diagonal set to one, and all other elements zero.

**See Also:**

- `numpy.identity`
  Equivalent array function.

- `matlib.eye`
  More general matrix identity function.

**Examples**

```python
>>> import numpy.matlib
>>> np.matlib.identity(3, dtype=int)
matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```
numpy.matlib.repmat(a, m, n)
Repeat a 0-D to 2-D array or matrix MxN times.

Parameters
a : array_like
  The array or matrix to be repeated.
m, n : int
  The number of times a is repeated along the first and second axes.

Returns
out : ndarray
  The result of repeating a.

Examples
>>> import numpy.matlib
>>> a0 = np.array(1)
>>> np.matlib.repmat(a0, 2, 3)
array([[1, 1, 1],
       [1, 1, 1]])
>>> a1 = np.arange(4)
>>> np.matlib.repmat(a1, 2, 2)
array([[0, 1, 2, 3, 0, 1, 2, 3],
       [0, 1, 2, 3, 0, 1, 2, 3]])
>>> a2 = np.asmatrix(np.arange(6).reshape(2, 3))
>>> np.matlib.repmat(a2, 2, 3)
matrix([[0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5],
        [0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5]])

numpy.matlib.rand(*args)
Return a matrix of random values with given shape.

Create a matrix of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

Parameters
*args : Arguments
  Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

Returns
out : ndarray
  The matrix of random values with shape given by *args.

See Also:
randn, numpy.random.rand
Examples

```python
>>> import numpy.matlib
>>> np.matlib.rand(2, 3)
matrix([[ 0.68340382, 0.67926887, 0.83271405], [ 0.00793551, 0.20468222, 0.95253525]]) #random
```

```python
>>> np.matlib.rand((2, 3))  #random
matrix([[ 0.84682055, 0.73626594, 0.11308016], [ 0.85429008, 0.32948250, 0.89139555]])
```

If the first argument is a tuple, other arguments are ignored:

```python
>>> np.matlib.rand((2, 3), 4)  #random
matrix([[ 0.46898646, 0.15163588, 0.95188261], [ 0.59208621, 0.09561818, 0.00583606]])
```

numpy.matlib.randn(*args)

Return a random matrix with data from the “standard normal” distribution. randn generates a matrix filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1.

Parameters

*args : Arguments

Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

Returns

Z : matrix of floats

A matrix of floating-point samples drawn from the standard normal distribution.

See Also:

rand, random.randn

Notes

For random samples from \( N(\mu, \sigma^2) \), use:

\[
\sigma \times \text{numpy.matlib.randn}(...) + \mu
\]

Examples

```python
>>> import numpy.matlib
>>> np.matlib.randn(1)
matrix([-0.09542833]) #random
```

```python
>>> np.matlib.randn(1, 2, 3)
matrix([[ 0.16198284, 0.01945710, 0.18312985], [ 0.59208621, 0.09561818, 0.00583606]]) #random
```

Two-by-four matrix of samples from \( N(3, 6.25) \):

```python
>>> 2.5 * np.matlib.randn((2, 4)) + 3
matrix([[ 4.74085004, 8.89381862, 4.09042411, 4.83721922], [ 7.52373709, 5.07933944, -2.64043543, 0.45610557]]) #random
```
3.22 Numarray compatibility (numpy.numarray)

3.23 Old Numeric compatibility (numpy.oldnumeric)

Don’t add these to the __all__ variable though

3.24 Miscellaneous routines

3.24.1 Buffer objects

<table>
<thead>
<tr>
<th>getbuffer(obj [, offset[, size]])</th>
<th>Create a buffer object from the given object referencing a slice of length size starting at offset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>newbuffer(size)</td>
<td>Return a new uninitialized buffer object.</td>
</tr>
</tbody>
</table>

numpy.getbuffer(obj [, offset[, size]])
Create a buffer object from the given object referencing a slice of length size starting at offset.

Default is the entire buffer. A read-write buffer is attempted followed by a read-only buffer.

Parameters
- **obj**: object
- **offset**: int, optional
- **size**: int, optional

Returns
- **buffer_obj**: buffer

Examples

```python
>>> buf = np.getbuffer(np.ones(5), 1, 3)
>>> len(buf)
3
>>> buf[0]
'\x00'
>>> buf
<read-write buffer for 0x8af1e70, size 3, offset 1 at 0x8ba4ec0>
```

numpy.newbuffer(size)
Return a new uninitialized buffer object.

Parameters
- **size**: int

Size in bytes of returned buffer object.

Returns
- **newbuffer**: buffer object

Returned, uninitialized buffer object of size bytes.

3.24.2 Performance tuning
NumPy Reference, Release 1.8.1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alterdot</td>
<td>Change <code>dot</code>, <code>vdot</code>, and <code>inner</code> to use accelerated BLAS functions.</td>
</tr>
<tr>
<td>restoredot</td>
<td>Restore <code>dot</code>, <code>vdot</code>, and <code>innerproduct</code> to the default non-BLAS implementations.</td>
</tr>
<tr>
<td>setbufsize(size)</td>
<td>Set the size of the buffer used in ufuncs.</td>
</tr>
<tr>
<td>getbufsize()</td>
<td>Return the size of the buffer used in ufuncs.</td>
</tr>
</tbody>
</table>

**numpy.alterdot()**

Change `dot`, `vdot`, and `inner` to use accelerated BLAS functions.

Typically, as a user of Numpy, you do not explicitly call this function. If Numpy is built with an accelerated BLAS, this function is automatically called when Numpy is imported.

When Numpy is built with an accelerated BLAS like ATLAS, these functions are replaced to make use of the faster implementations. The faster implementations only affect float32, float64, complex64, and complex128 arrays. Furthermore, the BLAS API only includes matrix-matrix, matrix-vector, and vector-vector products. Products of arrays with larger dimensionalities use the built-in functions and are not accelerated.

**See Also:**

**restoredot**

`restoredot` undoes the effects of `alterdot`.

**numpy.restoredot()**

Restore `dot`, `vdot`, and `innerproduct` to the default non-BLAS implementations.

Typically, the user will only need to call this when troubleshooting and installation problem, reproducing the conditions of a build without an accelerated BLAS, or when being very careful about benchmarking linear algebra operations.

**See Also:**

**alterdot**

`restoredot` undoes the effects of `alterdot`.

**numpy.setbufsize(size)**

Set the size of the buffer used in ufuncs.

**Parameters**

- `size` : int
  
  Size of buffer.

**numpy.getbufsize()**

Return the size of the buffer used in ufuncs.

**Returns**

- `getbufsize` : int
  
  Size of ufunc buffer in bytes.

### 3.25 Padding Arrays

**pad**

`pad(array, pad_width[, mode])` Pads an array.
numpy.pad(array, pad_width, mode=None, **kwargs)

Pads an array.

Parameters

array : array_like of rank N

Input array

pad_width : {sequence, int}

Number of values padded to the edges of each axis. ((before_1, after_1), ... (before_N, after_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.

mode : {str, function}

One of the following string values or a user supplied function.

‘constant’ Pads with a constant value. ‘edge’ Pads with the edge values of array. ‘linear_ramp’ Pads with the linear ramp between end_value and the array edge value.

‘maximum’ Pads with the maximum value of all or part of the vector along each axis.

‘mean’ Pads with the mean value of all or part of the vector along each axis.

‘median’ Pads with the median value of all or part of the vector along each axis.

‘minimum’ Pads with the minimum value of all or part of the vector along each axis.

‘reflect’ Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

‘symmetric’ Pads with the reflection of the vector mirrored along the edge of the array.

‘wrap’ Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

<function> Padding function, see Notes.

stat_length : {sequence, int}, optional

Used in ‘maximum’, ‘mean’, ‘median’, and ‘minimum’. Number of values at edge of each axis used to calculate the statistic value.

((before_1, after_1), ... (before_N, after_N)) unique statistic lengths for each axis.

((before, after),) yields same before and after statistic lengths for each axis.

(stat_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is None, to use the entire axis.

constant_values : {sequence, int}, optional

Used in ‘constant’. The values to set the padded values for each axis.

((before_1, after_1), ... (before_N, after_N)) unique pad constants for each axis.
((before, after),) yields same before and after constants for each axis.
(constant,) or int is a shortcut for before = after = constant for all axes.
Default is 0.

**end_values**: {sequence, int}, optional

Used in ‘linear_ramp’. The values used for the ending value of the linear_ramp and that
will form the edge of the padded array.

((before_1, after_1), ... (before_N, after_N)) unique end values for each axis.
((before, after),) yields same before and after end values for each axis.
(constant,) or int is a shortcut for before = after = end value for all axes.
Default is 0.

**reflect_type**: str {'even', 'odd'}, optional

Used in ‘reflect’, and ‘symmetric’. The ‘even’ style is the default with an unaltered
reflection around the edge value. For the ‘odd’ style, the extented part of the array is
created by subtracting the reflected values from two times the edge value.

**Returns**

**pad**: ndarray

Padded array of rank equal to array with shape increased according to pad_width.

**Notes**

New in version 1.7.0. For an array with rank greater than 1, some of the padding of later axes is calculated from
padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded
array are calculated by using padded values from the first axis.

The padding function, if used, should return a rank 1 array equal in length to the vector argument with padded
values replaced. It has the following signature:

    padding_func(vector, iaxis_pad_width, iaxis, **kwargs)

where

**vector**

[ndarray] A rank 1 array already padded with zeros. Padded values are vector[:pad_tuple[0]]
and vector[-pad_tuple[1]:].

**iaxis_pad_width**

[tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the
beginning of vector where iaxis_pad_width[1] represents the number of values padded at the
end of vector.

**iaxis**

[int] The axis currently being calculated.

**kwargs**

[misc] Any keyword arguments the function requires.

**Examples**

```python
>>> a = [1, 2, 3, 4, 5]
>>> np.lib.pad(a, (2,3), 'constant', constant_values=(4,6))
array([4, 4, 1, 2, 3, 4, 5, 6, 6, 6])
```
```python
>>> np.lib.pad(a, (2, 3), 'edge')
array([1, 1, 1, 2, 3, 4, 5, 5, 5, 5])

>>> np.lib.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])

>>> np.lib.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])

>>> np.lib.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> np.lib.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])

>>> a = [[1,2], [3,4]]
>>> np.lib.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 4, 3, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])

>>> a = [1, 2, 3, 4, 5]
>>> np.lib.pad(a, (2,3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])

>>> np.lib.pad(a, (2,3), 'reflect', reflect_type='odd')
array([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8])

>>> np.lib.pad(a, (2,3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])

>>> np.lib.pad(a, (2,3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 3, 5, 6])

>>> np.lib.pad(a, (2,3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])

>>> def padwithtens(vector, pad_width, iaxis, kwargs):
...     vector[:pad_width[0]] = 10
...     vector[-pad_width[1]:] = 10
...     return vector

>>> a = np.arange(6)
>>> a = a.reshape((2,3))

>>> np.lib.pad(a, 2, padwithtens)
array([[10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
       [10, 10, 0, 1, 2, 10, 10, 10, 10, 10],
       [10, 10, 3, 4, 5, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]])
```

3.25. Padding Arrays
3.26 Polynomials

Polynomials in NumPy can be created, manipulated, and even fitted using the Using the Convenience Classes of the numpy.polynomial package, introduced in NumPy 1.4.

Prior to NumPy 1.4, numpy.poly1d was the class of choice and it is still available in order to maintain backward compatibility. However, the newer Polynomial package is more complete than numpy.poly1d and its convenience classes are better behaved in the numpy environment. Therefore Polynomial is recommended for new coding.

3.26.1 Transition notice

The various routines in the Polynomial package all deal with series whose coefficients go from degree zero upward, which is the reverse order of the Poly1d convention. The easy way to remember this is that indexes correspond to degree, i.e., coef[i] is the coefficient of the term of degree i.

Polynomial Package

New in version 1.4.0.

Using the Convenience Classes

The convenience classes provided by the polynomial package are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial</td>
<td>Power series</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>Chebyshev series</td>
</tr>
<tr>
<td>Legendre</td>
<td>Legendre series</td>
</tr>
<tr>
<td>Laguerre</td>
<td>Laguerre series</td>
</tr>
<tr>
<td>Hermite</td>
<td>Hermite series</td>
</tr>
<tr>
<td>HermiteE</td>
<td>HermiteE series</td>
</tr>
</tbody>
</table>

The series in this context are finite sums of the corresponding polynomial basis functions multiplied by coefficients. For instance, a power series looks like

\[ p(x) = 1 + 2x + 3x^2 \]

and has coefficients \([1, 2, 3]\). The Chebyshev series with the same coefficients looks like

\[ p(x) = 1T_0(x) + 2T_1(x) + 3T_2(x) \]

and more generally

\[ p(x) = \sum_{i=0}^{n} c_iT_i(x) \]

where in this case the \(T_n\) are the Chebyshev functions of degree \(n\), but could just as easily be the basis functions of any of the other classes. The convention for all the classes is that the coefficient \(c[i]\) goes with the basis function of degree \(i\).

All of the classes have the same methods, and especially they implement the Python numeric operators +, -, *, //, %, divmod, **, ==, and !=. The last two can be a bit problematic due to floating point roundoff errors. We now give a quick demonstration of the various operations using Numpy version 1.7.0.
**Basics**  
First we need a polynomial class and a polynomial instance to play with. The classes can be imported directly from the polynomial package or from the module of the relevant type. Here we import from the package and use the conventional Polynomial class because of its familiarity:

```python
>>> from numpy.polynomial import Polynomial as P
>>> p = P([1,2,3])
>>> p
Polynomial([ 1., 2., 3.], [-1., 1.], [-1., 1.])
```

Note that there are three parts to the long version of the printout. The first is the coefficients, the second is the domain, and the third is the window:

```python
>>> p.coef
array([ 1., 2., 3.])
>>> p.domain
array([-1., 1.])
>>> p.window
array([-1., 1.])
```

Printing a polynomial yields a shorter form without the domain and window:

```python
>>> print p
poly([ 1. 2. 3.])
```

We will deal with the domain and window when we get to fitting, for the moment we ignore them and run through the basic algebraic and arithmetic operations.

**Addition and Subtraction:**

```python
>>> p + p
Polynomial([ 2., 4., 6.], [-1., 1.], [-1., 1.])
>>> p - p
Polynomial([ 0.], [-1., 1.], [-1., 1.])
```

**Multiplication:**

```python
>>> p * p
Polynomial([ 1., 4., 10., 12., 9.], [-1., 1.], [-1., 1.])
```

**Powers:**

```python
>>> p**2
Polynomial([ 1., 4., 10., 12., 9.], [-1., 1.], [-1., 1.])
```

**Division:**

Floor division, ‘//' is the division operator for the polynomial classes, polynomials are treated like integers in this regard. For Python versions < 3.x the '/' operator maps to ‘/’, as it does for Python, for later versions the ‘/’ will only work for division by scalars. At some point it will be deprecated:

```python
>>> p // P([-1, 1])
Polynomial([ 5., 3.], [-1., 1.], [-1., 1.])
```

**Remainder:**

```python
>>> p % P([-1, 1])
Polynomial([ 6.], [-1., 1.], [-1., 1.])
```

**Divmod:**

3.26. Polynomials 951
>>> quo, rem = divmod(p, P([-1, 1]))
>>> quo
Polynomial([ 5., 3.], [-1., 1.], [-1., 1.])
>>> rem
Polynomial([ 6.], [-1., 1.], [-1., 1.])

Evaluation:

>>> x = np.arange(5)
>>> p(x)
array([ 1., 6., 17., 34., 57.])

>>> x = np.arange(6).reshape(3,2)
>>> p(x)
array([[ 1.,  6.],
       [17., 34.],
       [57., 86.]]

Substitution:

Substitute a polynomial for x and expand the result. Here we substitute p in itself leading to a new polynomial of degree 4 after expansion. If the polynomials are regarded as functions this is composition of functions:

```python
>>> p(p)
Polynomial([ 6., 16., 36., 36., 27.], [-1., 1.], [-1., 1.])
```

Roots:

```python
>>> p.roots()
array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
```

It isn’t always convenient to explicitly use Polynomial instances, so tuples, lists, arrays, and scalars are automatically cast in the arithmetic operations:

```python
>>> p + [1, 2, 3]
Polynomial([ 2., 4., 6.], [-1., 1.], [-1., 1.])

>>> [1, 2, 3] * p
Polynomial([ 1., 4., 10., 12., 9.], [-1., 1.], [-1., 1.])

>>> p / 2
Polynomial([ 0.5, 1. , 1.5], [-1., 1.], [-1., 1.])
```

Polynomials that differ in domain, window, or class can’t be mixed in arithmetic:

```python
>>> from numpy.polynomial import Chebyshev as T
>>> p + P([1], domain=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>". line 213, in __add__
TypeError: Domains differ

>>> p + P([1], window=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>". line 215, in __add__
TypeError: Windows differ

>>> p + T([1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>". line 211, in __add__
TypeError: Polynomial types differ
But different types can be used for substitution. In fact, this is how conversion of Polynomial classes among themselves is done for type, domain, and window casting:

```python
gp(T([0, 1]))
```

Chebyshev([2.5, 2., 1.5], [-1., 1.], [-1., 1.])

Which gives the polynomial $p$ in Chebyshev form. This works because $T_1(x) = x$ and substituting $x$ for $x$ doesn’t change the original polynomial. However, all the multiplications and divisions will be done using Chebyshev series, hence the type of the result.

**Calculus**  
Polynomial instances can be integrated and differentiated:

```python
from numpy.polynomial import Polynomial as P

p = P([2, 6])
p.integ()
```

Polynomial([0., 2., 3.], [-1., 1.], [-1., 1.])

```python
p.integ(2)
```

Polynomial([0., 0., 1., 1.], [-1., 1.], [-1., 1.])

The first example integrates $p$ once, the second example integrates it twice. By default, the lower bound of the integration and the integration constant are 0, but both can be specified:

```python
p.integ(lbnd=-1)
```

Polynomial([-1., 2., 3.], [-1., 1.], [-1., 1.])

```python
p.integ(lbnd=-1, k=1)
```

Polynomial([0., 2., 3.], [-1., 1.], [-1., 1.])

In the first case the lower bound of the integration is set to -1 and the integration constant is 0. In the second the constant of integration is set to 1 as well. Differentiation is simpler since the only option is the number times the polynomial is differentiated:

```python
p = P([1, 2, 3])
p.deriv(1)
```

Polynomial([2., 6.], [-1., 1.], [-1., 1.])

```python
p.deriv(2)
```

Polynomial([6.], [-1., 1.], [-1., 1.])

**Other Polynomial Constructors**  
Constructing polynomials by specifying coefficients is just one way of obtaining a polynomial instance, they may also be created by specifying their roots, by conversion from other polynomial types, and by least squares fits. Fitting is discussed in its own section, the other methods are demonstrated below:

```python
from numpy.polynomial import Polynomial as P
from numpy.polynomial import Chebyshev as T

p = P.fromroots([1, 2, 3])
p.convert(kind=T)
```

Chebyshev([-2.4375, 2.96875, -0.5625], [0., 1.], [-1., 1.])

The convert method can also convert domain and window:

```python
p.convert(kind=T, domain=[0, 1])
```

Chebyshev([-2.4375, 2.96875, -0.5625, 0.03125], [0., 1.], [-1., 1.])

```python
p.convert(kind=P, domain=[0, 1])
```

Polynomial([-1.875, 2.875, -1.125, 0.125], [0., 1.], [-1., 1.])

In numpy versions >= 1.7.0 the `basis` and `cast` class methods are also available. The cast method works like the convert method while the basis method returns the basis polynomial of given degree:
Conversions between types can be useful, but it is not recommended for routine use. The loss of numerical precision in passing from a Chebyshev series of degree 50 to a Polynomial series of the same degree can make the results of numerical evaluation essentially random.

**Fitting**  Fitting is the reason that the *domain* and *window* attributes are part of the convenience classes. To illustrate the problem, the values of the Chebyshev polynomials up to degree 5 are plotted below.

![Chebyshev Polynomials](image)

In the range $-1 \leq x \leq 1$ they are nice, equiripple functions lying between +/- 1. The same plots over the range $-2 \leq x \leq 2$ look very different:

![Chebyshev Polynomials](image)
As can be seen, the “good” parts have shrunk to insignificance. In using Chebyshev polynomials for fitting we want to use the region where \( x \) is between -1 and 1 and that is what the window specifies. However, it is unlikely that the data to be fit has all its data points in that interval, so we use domain to specify the interval where the data points lie. When the fit is done, the domain is first mapped to the window by a linear transformation and the usual least squares fit is done using the mapped data points. The window and domain of the fit are part of the returned series and are automatically used when computing values, derivatives, and such. If they aren’t specified in the call the fitting routine will use the default window and the smallest domain that holds all the data points. This is illustrated below for a fit to a noisy sine curve.

```python
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> np.random.seed(11)
>>> x = np.linspace(0, 2*np.pi, 20)
>>> y = np.sin(x) + np.random.normal(scale=.1, size=x.shape)
>>> p = T.fit(x, y, 5)
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x2136c10>]
>>> xx, yy = p.linspace()
>>> plt.plot(xx, yy, lw=2)
[<matplotlib.lines.Line2D object at 0x1cf2890>]
>>> p.domain
array([ 0. , 6.28318531])
>>> p.window
array([-1., 1.])
>>> plt.show()
```
Polynomial Module (numpy.polynomial.polynomial)

New in version 1.4.0. This module provides a number of objects (mostly functions) useful for dealing with Polynomial series, including a Polynomial class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).

```
Polynomial(coef[, domain, window])  A Polynomial series class.
```

Polynomial Class
class numpy.polynomial.polynomial.Polynomial (coef, domain=\([-1, 1]\], window=\([-1, 1]\])
A Polynomial series class.

Polynomial instances provide the standard Python numerical methods ‘+’, ‘-’, ‘*’, ‘/’, ‘%', ‘divmod’, ‘**’, and ‘()’ as well as the listed methods.

Parameters

coeff : array_like
Polynomial coefficients, in increasing order. For example, (1, 2, 3) implies \( P_0 + 2P_1 + 3P_2 \) where the \( P_i \) are a graded polynomial basis.

domain : (2,) array_like, optional
Domain to use. The interval \([domain[0], domain[1]]\) is mapped to the interval \([window[0], window[1]]\) by shifting and scaling. The default value is \([-1,1]\).

window : (2,) array_like, optional
Window, see domain for its use. The default value is \([-1,1]\). .. versionadded:: 1.6.0

Notes

It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.
Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coef</td>
<td>((N,) ndarray) Polynomial coefficients, from low to high.</td>
</tr>
<tr>
<td>domain</td>
<td>((2,) ndarray) Domain that is mapped to window.</td>
</tr>
<tr>
<td>window</td>
<td>((2,) ndarray) Window that domain is mapped to.</td>
</tr>
</tbody>
</table>

Methods

```python
Polynomial.__call__(arg)
```

Polynomial.basis(deg[, domain, window]) Polynomial polynomial of degree `deg`.

Returns an instance of the Polynomial polynomial of degree `d`.

**Parameters**

- **deg**: int
  - Degree of the Polynomial polynomial. Must be >= 0.

- **domain**: array_like
  - The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain.

- **window**: array_like
  - The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the window.

**Returns**

- **p**: Polynomial instance
Notes

New in version 1.7.0.

static Polynomial.cast (series, domain=[-1, 1], window=[-1, 1])
Convert instance to equivalent Polynomial series.

The series is expected to be an instance of some polynomial series of one of the types supported by the numpy.polynomial module, but could be some other class that supports the convert method.

Parameters

series : series
The instance series to be converted.

domain : array_like
The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

window : array_like
The resulting array must be if the form [beg, end], where beg and end are the endpoints of the window.

Returns

p : Polynomial instance
A Polynomial instance equal to the poly series.

See Also:

convert

Notes

New in version 1.7.0.

Polynomial.convert (domain=None, kind=None, window=None)
Convert to different class and/or domain.

Parameters

domain : array_like, optional
The domain of the converted series. If the value is None, the default domain of kind is used.

kind : class, optional
The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

window : array_like, optional
The window of the converted series. If the value is None, the default window of kind is used.

Returns

new_series_instance : kind
The returned class can be of different type than the current instance and/or have a different domain.
Notes
Conversion between domains and class types can result in numerically ill defined series.

Polynomial.copy()
Return a copy.

Returns
new_instance : Polynomial
Copy of current instance.

Polynomial.cutdeg(deg)
Truncate series to the given degree.
Reduce the degree of the Polynomial series to \textit{deg} by discarding the high order terms. If \textit{deg} is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters
\textit{deg} : non-negative int
The series is reduced to degree \textit{deg} by discarding the high order terms. The value of \textit{deg} must be a non-negative integer.

Returns
new_instance : Polynomial
New instance of Polynomial with reduced degree.

Notes
New in version 1.5.0.

Polynomial.degree()
The degree of the series.

Notes
New in version 1.5.0.

Polynomial.deriv(m=1)
Differentiate.
Return an instance of Polynomial that is the derivative of the current series. Refer to \texttt{polyder} for full documentation.

Parameters
\textit{m} : non-negative int
The number of integrations to perform.

Returns
derivative : Polynomial
The derivative of the series using the same domain.

See Also:

polyder
similar function.
**polyint**

similar function for integration.

**static Polynomial.fit** *(x, y, deg, domain=None, rcond=None, full=False, w=None, window=[-1, 1])* 

Least squares fit to data.

Return a Polynomial instance that is the least squares fit to the data y sampled at x. Unlike polyfit, the domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning. Support for NA was added in version 1.7.0. See polyfit for full documentation of the implementation.

**Parameters**

- **x**: array_like, shape (M,)
x-coordinates of the M sample points \((x[i], y[i])\).

- **y**: array_like, shape (M,) or (M, K)
y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- **deg**: int
  Degree of the fitting polynomial.

- **domain**: {None, [beg, end], []}, optional
  Domain to use for the returned Polynomial instance. If None, then a minimal domain that covers the points \(x\) is chosen. If [] the default domain [-1, 1] is used. The default value is [-1,1] in numpy 1.4.x and None in later versions. The '[]' value was added in numpy 1.5.0.

- **rcond**: float, optional
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

- **full**: bool, optional
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

- **w**: array_like, shape (M,), optional
  Weights. If not None the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i]y[i]\) all have the same variance. The default value is None. .. versionadded:: 1.5.0

- **window**: {[beg, end]}, optional
  Window to use for the returned Polynomial instance. The default value is [-1, 1] .. versionadded:: 1.6.0

**Returns**

- **least_squares_fit**: instance of Polynomial
  The Polynomial instance is the least squares fit to the data and has the domain specified in the call.
[residuals, rank, singular_values, rcond] : only if full = True

Residuals of the least squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of rcond. For more details, see linalg.lstsq.

See Also:

polyfit
similar function

static Polynomial.fromroots (roots, domain=[-1, 1], window=[-1, 1])
Return Polynomial instance with specified roots.

Returns an instance of Polynomial representing the product \((x - r[0])*(x - r[1])*...*(x - r[n-1])\), where \(r\) is the list of roots.

Parameters
   roots : array_like
      List of roots.
   domain : {array_like, None}, optional
      Domain for the resulting instance of Polynomial. If none the domain is the interval from the smallest root to the largest. The default is [-1,1].
   window : array_like, optional
      Window for the resulting instance of Polynomial. The default value is [-1,1].

Returns
   object : Polynomial instance
      Series with the specified roots.

See Also:

polyfromroots
equivalent function

Polynomial.has_samecoef (other)
Check if coefficients match.

Parameters
   other : class instance
      The other class must have the coef attribute.

Returns
   bool : boolean
      True if the coefficients are the same, False otherwise.

Notes
New in version 1.6.0.

Polynomial.has_samedomain (other)
Check if domains match.

Parameters
   other : class instance
The other class must have the `domain` attribute.

**Returns**

`bool`: boolean

True if the domains are the same, False otherwise.

**Notes**

New in version 1.6.0.

`Polynomial.has_sametype(other)`

Check if types match.

**Parameters**

`other`: object

Class instance.

**Returns**

`bool`: boolean

True if other is same class as self

**Notes**

New in version 1.7.0.

`Polynomial.has_samewindow(other)`

Check if windows match.

**Parameters**

`other`: class instance

The other class must have the `window` attribute.

**Returns**

`bool`: boolean

True if the windows are the same, False otherwise.

**Notes**

New in version 1.6.0.

`static Polynomial.identity(domain=[-1, 1], window=[-1, 1])`

Identity function.

If `p` is the returned Polynomial object, then \( p(x) = x \) for all values of \( x \).

**Parameters**

`domain`: array_like

The resulting array must be of the form \([\text{beg}, \text{end}]\), where \text{beg} and \text{end} are the endpoints of the domain.

`window`: array_like

The resulting array must be if the form \([\text{beg}, \text{end}]\), where \text{beg} and \text{end} are the endpoints of the window.

**Returns**

`identity`: Polynomial instance
Polynomial.integ(m=1, k=[], lbnd=None)

Integrate.

Return an instance of Polynomial that is the definite integral of the current series. Refer to polyint for full documentation.

Parameters

m : non-negative int
  The number of integrations to perform.

k : array_like
  Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to m in length and any missing values are set to zero.

lbnd : Scalar
  The lower bound of the definite integral.

Returns

integral : Polynomial
  The integral of the series using the same domain.

See Also:

polyint
  similar function.

polyder
  similar function for derivative.

Polynomial.linspace(n=100, domain=None)

Return x,y values at equally spaced points in domain.

Returns x, y values at n linearly spaced points across domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the Polynomial instance. This method is intended mostly as a plotting aid.

Parameters

n : int, optional
  Number of point pairs to return. The default value is 100.

domain : {None, array_like}
  If not None, the specified domain is used instead of that of the calling instance. It should be of the form [beg, end]. The default is None.

Returns

x, y : ndarrays
  x is equal to linspace(self.domain[0], self.domain[1], n) y is the polynomial evaluated at x.

New in version 1.5.0.

Polynomial.mapparms()

Return the mapping parameters.

The returned values define a linear map off + scl*x that is applied to the input arguments before the series is evaluated. The map depends on the domain and window; if the current domain is equal to
the window the resulting map is the identity. If the coefficients of the Polynomial instance are to be
used by themselves outside this class, then the linear function must be substituted for the \( x \) in the standard
representation of the base polynomials.

Returns
off, scl : floats or complex

The mapping function is defined by \( \text{off} + \text{scl} \cdot x \).

Notes
If the current domain is the interval \([l_1, r_1]\) and the window is \([l_2, r_2]\), then the linear
mapping function \( L \) is defined by the equations:

\[
L(l_1) = l_2 \\
L(r_1) = r_2
\]

Polynomial.roots()
Return list of roots.
Return ndarray of roots for this series. See polyroots for full documentation. Note that the accuracy of
the roots is likely to decrease the further outside the domain they lie.

See Also:
polyroots
similar function
polyfromroots
function to go generate series from roots.

Polynomial.trim(tol=0)
Remove small leading coefficients
Remove leading coefficients until a coefficient is reached whose absolute value greater than \( tol \) or the
beginning of the series is reached. If all the coefficients would be removed the series is set to \([0]\). A new
Polynomial instance is returned with the new coefficients. The current instance remains unchanged.

Parameters
tol : non-negative number.
All trailing coefficients less than \( tol \) will be removed.

Returns
new_instance : Polynomial
Contains the new set of coefficients.

Polynomial.truncate(size)
Truncate series to length \( size \).
Reduce the Polynomial series to length \( size \) by discarding the high degree terms. The value of \( size \) must be
a positive integer. This can be useful in least squares where the coefficients of the high degree terms may
be very small.

Parameters
size : positive int
The series is reduced to length \( size \) by discarding the high degree terms. The value of
\( size \) must be a positive integer.

Returns
new_instance : Polynomial
New instance of Polynomial with truncated coefficients.

- `polyval(x, c[, tensor])`: Evaluate a polynomial at points \(x\).
- `polyval2d(x, y, c)`: Evaluate a 2-D polynomial at points \((x, y)\).
- `polyval3d(x, y, z, c)`: Evaluate a 3-D polynomial at points \((x, y, z)\).
- `polygrid2d(x, y, c)`: Evaluate a 2-D polynomial on the Cartesian product of \(x\) and \(y\).
- `polygrid3d(x, y, z, c)`: Evaluate a 3-D polynomial on the Cartesian product of \(x\), \(y\) and \(z\).
- `polyroots(c)`: Compute the roots of a polynomial.
- `polyfromroots(roots)`: Generate a monic polynomial with given roots.

**Basics**

```python
numpy.polynomial.polynomial.polyval(x, c, tensor=True)
```
Evaluate a polynomial at points \(x\).

If \(c\) is of length \(n + 1\), this function returns the value

\[
p(x) = c_0 + c_1 * x + ... + c_n * x^n
\]

The parameter \(x\) is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either \(x\) or its elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) is a 1-D array, then \(p(x)\) will have the same shape as \(x\). If \(c\) is multidimensional, then the shape of the result depends on the value of \(tensor\). If \(tensor\) is true the shape will be \(c.shape[1:] + x.shape\). If \(tensor\) is false the shape will be \(c.shape[1:]\). Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

- \(x\) : array_like, compatible object
  
  If \(x\) is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, \(x\) or its elements must support addition and multiplication with with themselves and with the elements of \(c\).

- \(c\) : array_like
  
  Array of coefficients ordered so that the coefficients for terms of degree \(n\) are contained in \(c[n]\). If \(c\) is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of \(c\).

- \(tensor\) : boolean, optional
  
  If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of \(x\). Scalars have dimension 0 for this action. The result is that every column of coefficients in \(c\) is evaluated for every element of \(x\). If False, \(x\) is broadcast over the columns of \(c\) for the evaluation. This keyword is useful when \(c\) is multidimensional. The default value is True. New in version 1.7.0.

**Returns**

- \(values\) : ndarray, compatible object
  
  The shape of the returned array is described above.
See Also:
polyval2d, polygrid2d, polyval3d, polygrid3d

Notes
The evaluation uses Horner's method.

Examples
```python
>>> from numpy.polynomial.polynomial import polyval
>>> polyval(1, [1,2,3])
6.0
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> polyval(a, [1,2,3])
array([[ 1., 6.],
       [17., 34.]])
>>> coef = np.arange(4).reshape(2,2) # multidimensional coefficients
>>> coef
array([[0, 1],
       [2, 3]])
>>> polyval([1,2], coef, tensor=True)
array([[ 2., 4.],
       [ 4., 7.]])
>>> polyval([1,2], coef, tensor=False)
array([ 2., 7.])
```

numpy.polynomial.polynomial.polyval2d(x, y, c)
Evaluate a 2-D polynomial at points (x, y).
This function returns the value
\[ p(x, y) = \sum_{i,j} c_{i,j} \times x^i \times y^j \]

The parameters \(x\) and \(y\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either \(x\) and \(y\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \(c.shape[2:] + x.shape\).

Parameters
\(x, y\) : array_like, compatible objects

The two dimensional series is evaluated at the points \((x, y)\), where \(x\) and \(y\) must have the same shape. If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

\(c\) : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j\) is contained in \(c[i,j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns
values : ndarray, compatible object
The values of the two dimensional polynomial at points formed with pairs of corresponding values from \(x\) and \(y\).

**See Also:**

polyval, polygrid2d, polyval3d, polygrid3d

**Notes**

```
numpy.polynomial.polynomial.polyval3d(x, y, z, c)
```

Evaluate a 3-D polynomial at points \((x, y, z)\).

This function returns the values:

\[
p(x, y, z) = \sum_{i,j,k} c_{i,j,k} \cdot x^i \cdot y^j \cdot z^k
\]

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c\).shape[3:] + x.shape.

**Parameters**

- \(x, y, z\) : array_like, compatible object
  
The three dimensional series is evaluated at the points \((x, y, z)\), where \(x\), \(y\), and \(z\) must have the same shape. If any of \(x\), \(y\), or \(z\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

- \(c\) : array_like
  
  Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j,k\) is contained in \(c[i, j, k]\). If \(c\) has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

- \(values\) : ndarray, compatible object
  
The values of the multidimensional polynomial on points formed with triples of corresponding values from \(x\), \(y\), and \(z\).

**See Also:**

polyval, polyval2d, polygrid2d, polygrid3d

**Notes**

```
numpy.polynomial.polynomial.polygrid2d(x, y, c)
```

Evaluate a 2-D polynomial on the Cartesian product of \(x\) and \(y\).

This function returns the values:

\[
p(a, b) = \sum_{i,j} c_{i,j} \cdot a^i \cdot b^j
\]

where the points \((a, b)\) consist of all pairs formed by taking \(a\) from \(x\) and \(b\) from \(y\). The resulting points form a grid with \(x\) in the first dimension and \(y\) in the second.
The parameters \( x \) and \( y \) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \( x \) and \( y \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \( c.shape[2:] + x.shape + y.shape \).

**Parameters**

- **x, y**: array_like, compatible objects

  The two dimensional series is evaluated at the points in the Cartesian product of \( x \) and \( y \). If \( x \) or \( y \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- **c**: array_like

  Array of coefficients ordered so that the coefficients for terms of degree \( i,j \) are contained in \( c[i,j] \). If \( c \) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

- **values**: ndarray, compatible object

  The values of the two dimensional polynomial at points in the Cartesian product of \( x \) and \( y \).

**See Also:**

- `polyval`, `polyval2d`, `polyval3d`, `polygrid3d`

**Notes**

`numpy.polynomial.polynomial.polygrid3d(x, y, z, c)`

Evaluate a 3-D polynomial on the Cartesian product of \( x \), \( y \) and \( z \).

This function returns the values:

\[
p(a, b, c) = \sum_{i,j,k} c_{i,j,k} * a^i * b^j * c^k
\]

where the points \((a, b, c)\) consist of all triples formed by taking \( a \) from \( x \), \( b \) from \( y \), and \( c \) from \( z \). The resulting points form a grid with \( x \) in the first dimension, \( y \) in the second, and \( z \) in the third.

The parameters \( x \), \( y \), and \( z \) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \( x \), \( y \), and \( z \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \( c.shape[3:] + x.shape + y.shape + z.shape \).

**Parameters**

- **x, y, z**: array_like, compatible objects

  The three dimensional series is evaluated at the points in the Cartesian product of \( x \), \( y \), and \( z \). If \( x \), \('y'\), or \( z \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- **c**: array_like

  Array of coefficients ordered so that the coefficients for terms of degree \( i,j \) are contained in \( c[i,j] \). If \( c \) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.
Returns
values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of x and y.

See Also:
polyval, polyval2d, polygrid2d, polyval3d

Notes

numpy.polynomial.polynomial.polyroots(c)
Compute the roots of a polynomial.

Return the roots (a.k.a. “zeros”) of the polynomial

\[ p(x) = \sum_{i} c[i] \cdot x^i. \]

Parameters
c : 1-D array_like

1-D array of polynomial coefficients.

Returns
out : ndarray

Array of the roots of the polynomial. If all the roots are real, then out is also real, otherwise it is complex.

See Also:
chebroots

Notes

The root estimates are obtained as the eigenvalues of the companion matrix. Roots far from the origin of the complex plane may have large errors due to the numerical instability of the power series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s method.

Examples

```python
>>> import numpy.polynomial.polynomial as poly
>>> poly.polyroots(poly.polyfromroots((-1,0,1)))
array([-1., 0., 1.])

>>> poly.polyroots(poly.polyfromroots((-1,0,1))).dtype
dtype('float64')

>>> j = complex(0,1)
>>> poly.polyroots(poly.polyfromroots((-j,0,j)))
array([ 0.00000000e+00+0.j,  0.00000000e+00+0.1.j,  2.77555756e-17-1.j])
```

numpy.polynomial.polynomial.polyfromroots(roots)
Generate a monic polynomial with given roots.

Return the coefficients of the polynomial

\[ p(x) = (x - r_0) \cdot (x - r_1) \cdot ... \cdot (x - r_n), \]
where the \( r_n \) are the roots specified in \( \text{roots} \). If a zero has multiplicity \( n \), then it must appear in \( \text{roots} \) \( n \) times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then \( \text{roots} \) looks something like \([2, 2, 2, 3, 3]\). The roots can appear in any order.

If the returned coefficients are \( c \), then

\[
p(x) = c_0 + c_1 \times x + \ldots + x^n
\]

The coefficient of the last term is 1 for monic polynomials in this form.

**Parameters**
- \( \text{roots} : \text{array_like} \)
  Sequence containing the roots.

**Returns**
- \( \text{out} : \text{ndarray} \)
  1-D array of the polynomial’s coefficients If all the roots are real, then \( \text{out} \) is also real, otherwise it is complex. (see Examples below).

**See Also:**
- `chebfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`

**Notes**

The coefficients are determined by multiplying together linear factors of the form \((x - r_i)\), i.e.

\[
p(x) = (x - r_0)(x - r_1)\ldots(x - r_n)
\]

where \( n = \text{len(roots)} - 1 \); note that this implies that 1 is always returned for \( a_n \).

**Examples**

```python
>>> import numpy.polynomial as P

>>> P.polyfromroots((-1,0,1)) # x(x - 1)(x + 1) = x^3 - x
array([ 0., -1., 0., 1.])

>>> j = complex(0,1)

>>> P.polyfromroots((-j,j)) # complex returned, though values are real
array([ 1.+0.j, 0.+0.j, 1.+0.j])
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>polyfit(x, y, deg[, rcond, full, w])</code></td>
<td>Least-squares fit of a polynomial to data.</td>
</tr>
<tr>
<td><code>polyvander(x, deg)</code></td>
<td>Vandermonde matrix of given degree.</td>
</tr>
<tr>
<td><code>polyvander2d(x, y, deg)</code></td>
<td>Pseudo-Vandermonde matrix of given degrees.</td>
</tr>
<tr>
<td><code>polyvander3d(x, y, z, deg)</code></td>
<td>Pseudo-Vandermonde matrix of given degrees.</td>
</tr>
</tbody>
</table>

**Fitting**

```python
numpy.polynomial.polynomial.polyfit (x, y, deg, rcond=None, full=False, w=None)
```

Least-squares fit of a polynomial to data.

Return the coefficients of a polynomial of degree \( \text{deg} \) that is the least squares fit to the data values \( y \) given at points \( x \). If \( y \) is 1-D the returned coefficients will also be 1-D. If \( y \) is 2-D multiple fits are done, one for each column of \( y \), and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted
polynomial(s) are in the form
\[ p(x) = c_0 + c_1 x + \ldots + c_n x^n, \]

where \( n \) is \( \text{deg} \).

Since numpy version 1.7.0, polyfit also supports NA. If any of the elements of \( x, y, \) or \( w \) are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If \( y \) is 2-D, then an NA in any row of \( y \) invalidates that whole row.

**Parameters**
- **x**: array_like, shape \((M,)\)
  
  x-coordinates of the \( M \) sample (data) points \((x[i], y[i])\).

- **y**: array_like, shape \((M,)\) or \((M, K)\)
  
  y-coordinates of the sample points. Several sets of sample points sharing the same x-coordinates can be (independently) fit with one call to `polyfit` by passing in for \( y \) a 2-D array that contains one data set per column.

- **deg**: int
  
  Degree of the polynomial(s) to be fit.

- **rcond**: float, optional
  
  Relative condition number of the fit. Singular values smaller than \( rcond \), relative to the largest singular value, will be ignored. The default value is \( \text{len}(x) \times \text{eps} \), where \( \text{eps} \) is the relative precision of the platform’s float type, about 2\(\times\)16 in most cases.

- **full**: bool, optional
  
  Switch determining the nature of the return value. When `False` (the default) just the coefficients are returned; when `True`, diagnostic information from the singular value decomposition (used to solve the fit’s matrix equation) is also returned.

- **w**: array_like, shape \((M,)\), optional
  
  Weights. If not `None`, the contribution of each point \((x[i], y[i])\) to the fit is weighted by \( w[i] \). Ideally the weights are chosen so that the errors of the products \( w[i] \times y[i] \) all have the same variance. The default value is `None`. New in version 1.5.0.

**Returns**
- **coef**: ndarray, shape \((\text{deg} + 1,)\) or \((\text{deg} + 1, K)\)
  
  Polynomial coefficients ordered from low to high. If \( y \) was 2-D, the coefficients in column \( k \) of \( \text{coef} \) represent the polynomial fit to the data in \( y \)'s \( k \)-th column.

- **[residuals, rank, singular_values, rcond]**: present when `full == True`
  
  Sum of the squared residuals (SSR) of the least-squares fit; the effective rank of the scaled Vandermonde matrix; its singular values; and the specified value of \( rcond \). For more information, see `linalg.lstsq`.

**Raises**
- **RankWarning**
  
  Raised if the matrix in the least-squares fit is rank deficient. The warning is only raised if `full == False`. The warnings can be turned off by:
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)

See Also:
chebfit, legfit, lagfit, hermfit, hermefit
polyval
Evaluates a polynomial.
polyvander
Vandermonde matrix for powers.
linalg.lstsq
Computes a least-squares fit from the matrix.
scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes
The solution is the coefficients of the polynomial \( p \) that minimizes the sum of the weighted squared errors

\[
E = \sum_j w_j^2 \cdot |y_j - p(x_j)|^2,
\]

where the \( w_j \) are the weights. This problem is solved by setting up the (typically) over-determined matrix equation:

\[
V(x) \cdot c = w \cdot y,
\]

where \( V \) is the weighted pseudo Vandermonde matrix of \( x \), \( c \) are the coefficients to be solved for, \( w \) are the weights, and \( y \) are the observed values. This equation is then solved using the singular value decomposition of \( V \).

If some of the singular values of \( V \) are so small that they are neglected (and \( full == \textit{False} \)), a \textit{RankWarning} will be raised. This means that the coefficient values may be poorly determined. Fitting to a lower order polynomial will usually get rid of the warning (but may not be what you want, of course; if you have independent reason(s) for choosing the degree which isn’t working, you may have to: a) reconsider those reasons, and/or b) reconsider the quality of your data). The \textit{rcond} parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Polynomial fits using double precision tend to “fail” at about (polynomial) degree 20. Fits using Chebyshev or Legendre series are generally better conditioned, but much can still depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate, splines may be a good alternative.

Examples

```python
>>> from numpy import polynomial as P
>>> x = np.linspace(-1,1,51)  # x "data": [-1, -0.96, ..., 0.96, 1]
>>> y = x**3 - x + np.random.randn(len(x))  # x^3 - x + N(0,1) "noise"
>>> c, stats = P.polyfit(x,y,3,full=True)
>>> c  # c[0], c[2] should be approx. 0, c[1] approx. -1, c[3] approx. 1
array([0.01909725, -1.30598256, -0.00577963, 1.02644286])
>>> stats # note the large SSR, explaining the rather poor results
[array([ 38.06116253]), 4, array([ 1.38446749, 1.32119158, 0.50443316, 0.28853036]), 1.1324274851176597e-014]
```
Same thing without the added noise

```python
>>> y = x**3 - x
>>> c, stats = np.polyfit(x, y, 3, full=True)
>>> c  # c[0], c[2] should be "very close to 0", c[1] ~= -1, c[3] ~= 1
array([-1.73362882e-17, -1.00000000e+00, -2.67471909e-16,
       1.00000000e+00])
>>> stats  # note the minuscule SSR
(array([ 7.46346754e-31]), 4, array([ 1.38446749, 1.32119158,
       0.50443316, 0.28853036]), 1.1324274851176597e-014)
```

numpy.polynomial.polynomial.polyvander(x, deg)

Vandermonde matrix of given degree.

Returns the Vandermonde matrix of degree `deg` and sample points `x`. The Vandermonde matrix is defined by

\[ V[...i] = x^i, \]

where 0 <= i <= deg. The leading indices of `V` index the elements of `x` and the last index is the power of `x`.

If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the matrix `V = polyvander(x, n)`, then `np.dot(V, c)` and `polyval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of polynomials of the same degree and sample points.

Parameters

- **x**: array_like
  - Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If `x` is scalar it is converted to a 1-D array.

- **deg**: int
  - Degree of the resulting matrix.

Returns

- **vander**: ndarray.
  - The Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the power of `x`. The dtype will be the same as the converted `x`.

See Also:

- polyvander2d, polyvander3d

numpy.polynomial.polynomial.polyvander2d(x, y, deg)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees `deg` and sample points `(x, y)`. The pseudo-Vandermonde matrix is defined by

\[ V[... i * j] = x^i * y^j, \]

where 0 <= i <= `deg[0]` and 0 <= j <= `deg[1]`. The leading indices of `V` index the points `(x, y)` and the last index encodes the powers of `x` and `y`.

If `V = polyvander2d(x, y, [xdeg, ydeg])`, then the columns of `V` correspond to the elements of a 2-D coefficient array `c` of shape `(xdeg + 1, ydeg + 1)` in the order

\[ c_{00}, c_{01}, c_{02}, ..., c_{10}, c_{11}, c_{12}, ... \]
and \( \text{np.dot}(V, \ c.\text{flat}) \) and \( \text{polyval2d}(x, \ y, \ c) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D polynomials of the same degrees and sample points.

**Parameters**

- **x, y**: array_like
  Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

- **deg**: list of ints
  List of maximum degrees of the form \([x\_deg, y\_deg]\).

**Returns**

- **vander2d**: ndarray
  The shape of the returned matrix is \(x.\text{shape} + (order,)\), where \(order = (deg[0] + 1) \times (deg[1] + 1)\). The dtype will be the same as the converted \(x\) and \(y\).

**See Also:**

`polyvander`, `polyvander3d`, `polyval3d`  
`numpy.polynomial.polynomial.polyvander3d(x, y, z, deg)`  
Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees \(deg\) and sample points \((x, y, z)\). If \(l, m, n\) are the given degrees in \(x, y, z\), then The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, (m+1)(n+1)i + (n+1)j + k] = x^i \cdot y^j \cdot z^k,
\]

where \(0 \leq i \leq l\), \(0 \leq j \leq m\), and \(0 \leq k \leq n\). The leading indices of \(V\) index the points \((x, y, z)\) and the last index encodes the powers of \(x, y,\) and \(z\).

If \(V = \text{polyvander3d}(x, y, z, [x\_deg, y\_deg, z\_deg])\), then the columns of \(V\) correspond to the elements of a 3-D coefficient array \(c\) of shape \((x\_deg + 1, y\_deg + 1, z\_deg + 1)\) in the order

\[
c_{0,0,0}, c_{0,0,1}, c_{0,0,2}, \ldots, c_{0,1,0}, c_{0,1,1}, c_{0,1,2}, \ldots
\]

and \(\text{np.dot}(V, \ c.\text{flat})\) and \(\text{polyval3d}(x, y, z, c)\) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D polynomials of the same degrees and sample points.

**Parameters**

- **x, y, z**: array_like
  Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

- **deg**: list of ints
  List of maximum degrees of the form \([x\_deg, y\_deg, z\_deg]\).

**Returns**

- **vander3d**: ndarray
  The shape of the returned matrix is \(x.\text{shape} + (order,)\), where \(order = (deg[0] + 1) \times (deg[1] + 1) \times (deg[2] + 1)\). The dtype will be the same as the converted \(x, y,\) and \(z\).
See Also:

polyvander, polyvander3d, polyval3d

Notes

<table>
<thead>
<tr>
<th>polyder(c[, m, scl, axis])</th>
<th>Differentiate a polynomial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>polyint(c[, m, k, lbnd, scl, axis])</td>
<td>Integrate a polynomial.</td>
</tr>
</tbody>
</table>

Calculus

numpy.polynomial.polynomial.polyder(c, m=1, scl=1, axis=0)

Differentiate a polynomial.

Returns the polynomial coefficients c differentiated m times along axis. At each iteration the result is multiplied by scl (the scaling factor is for use in a linear change of variable). The argument c is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the polynomial \(1 + 2x + 3x^2\) while [[1,2],[1,2]] represents \(1 + 1x + 2y + 2x*y\) if axis=0 is x and axis=1 is y.

Parameters

c : array_like

Array of polynomial coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by scl. The end result is multiplication by scl**m. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.

Returns

der : ndarray

Polynomial coefficients of the derivative.

See Also:

polyint

Examples

```python
>>> from numpy import polynomial as P
>>> c = (1,2,3,4) # 1 + 2x + 3x**2 + 4x**3
>>> P.polyder(c) # (d/dx) (c) = 2 + 6x + 12x**2
array([ 2.,  6., 12.])
>>> P.polyder(c, 3) # (d**3/dx**3) (c) = 24
array([ 24.])
>>> P.polyder(c, scl=-1) # (d/d(-x)) (c) = -2 - 6x - 12x**2
array([ -2., -6., -12.])
>>> P.polyder(c, 2, -1) # (d**2/d(-x)**2) (c) = 6 + 24x
array([  6.,  24.])
```

eumpy.polynomial.polynomial.polyint(c, m=1, k=[], lbnd=0, scl=1, axis=0)

Integrate a polynomial.
Returns the polynomial coefficients $c$ integrated $m$ times from $lbnd$ along $axis$. At each iteration the resulting series is multiplied by $scl$ and an integration constant, $k$, is added. The scaling factor is for use in a linear change of variable. (“Buyer beware”: note that, depending on what one is doing, one may want $scl$ to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument $c$ is an array of coefficients, from low to high degree along each axis, e.g., $[1,2,3]$ represents the polynomial $1 + 2x + 3x^2$ while $[[1,2],[1,2]]$ represents $1 + 1x + 2y + 2xy$ if axis=0 is $x$ and axis=1 is $y$.

**Parameters**

- $c$: array_like
  1-D array of polynomial coefficients, ordered from low to high.
- $m$: int, optional
  Order of integration, must be positive. (Default: 1)
- $k$: {[], list, scalar}, optional
  Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If $k == []$ (the default), all constants are set to zero. If $m == 1$, a single scalar can be given instead of a list.
- $lbnd$: scalar, optional
  The lower bound of the integral. (Default: 0)
- $scl$: scalar, optional
  Following each integration the result is multiplied by $scl$ before the integration constant is added. (Default: 1)
- $axis$: int, optional
  Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

**Returns**

- $S$: ndarray
  Coefficient array of the integral.

**Raises**

- ValueError
  If $m < 1, len(k) > m$.

**See Also:**

- polyder

**Notes**

Note that the result of each integration is multiplied by $scl$. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then .. math:: dx = du/a, so one will need to set $scl$ equal to $1/a$ - perhaps not what one would have first thought.

**Examples**

```python
>>> from numpy import polynomial as P
>>> c = (1,2,3)
>>> P.polyint(c)  # should return array([0, 1, 1, 1])
array([ 0., 1., 1., 1.])
>>> P.polyint(c,3)  # should return array([0, 0, 0, 1/6, 1/12, 1/20])
array([ 0. , 0. , 0. , 0.16666667, 0.08333333, 9.375e-02])
```
0.05 

```python
>>> P.polyint(c, k=3)  # should return array([3, 1, 1, 1])
array([ 3., 1., 1., 1.])
>>> P.polyint(c, ibnd=-2)  # should return array([6, 1, 1, 1])
array([ 6., 1., 1., 1.])
>>> P.polyint(c, scl=-2)  # should return array([0, -2, -2, -2])
array([ 0., -2., -2., -2.])
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>polyadd(c1, c2)</td>
<td>Add one polynomial to another.</td>
</tr>
<tr>
<td>polysub(c1, c2)</td>
<td>Subtract one polynomial from another.</td>
</tr>
<tr>
<td>polymul(c1, c2)</td>
<td>Multiply one polynomial by another.</td>
</tr>
<tr>
<td>polymulx(c)</td>
<td>Multiply a polynomial by x.</td>
</tr>
<tr>
<td>polydiv(c1, c2)</td>
<td>Divide one polynomial by another.</td>
</tr>
<tr>
<td>polypow(c, pow[, maxpower])</td>
<td>Raise a polynomial to a power.</td>
</tr>
</tbody>
</table>

**Algebra**

```python
numpy.polynomial.polynomial.polyadd(c1, c2)
```

Add one polynomial to another.

Returns the sum of two polynomials \( c1 + c2 \). The arguments are sequences of coefficients from lowest order term to highest, i.e., \([1,2,3]\) represents the polynomial \( 1 + 2x + 3x^2 \).

**Parameters**

- `c1, c2`: array_like
  - 1-D arrays of polynomial coefficients ordered from low to high.

**Returns**

- `out`: ndarray
  - The coefficient array representing their sum.

**See Also:**

polysub, polymul, polydiv, polypow

**Examples**

```python
>>> from numpy import polynomial as P
>>> c1 = (1, 2, 3)
>>> c2 = (3, 2, 1)
>>> sum = P.polyadd(c1, c2); sum
array([ 4., 4., 4.])
>>> P.polyval(2, sum)  # 4 + 4(2) + 4(2**2)
28.0
```

```python
numpy.polynomial.polynomial.polysub(c1, c2)
```

Subtract one polynomial from another.

Returns the difference of two polynomials \( c1 - c2 \). The arguments are sequences of coefficients from lowest order term to highest, i.e., \([1,2,3]\) represents the polynomial \( 1 + 2x + 3x^2 \).

**Parameters**

- `c1, c2`: array_like
  - 1-D arrays of polynomial coefficients ordered from low to high.

**Returns**

- `out`: ndarray
Of coefficients representing their difference.

See Also:
polyadd, polymul, polydiv, polypow

Examples

```python
>>> from numpy import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polysub(c1,c2)
array([-2., 0., 2.])
>>> P.polysub(c2,c1)  # -P.polysub(c1,c2)
array([ 2., 0., -2.])
```

`numpy.polynomial.polynomial.polymul(c1, c2)`
Multiply one polynomial by another.

Returns the product of two polynomials $c1 * c2$. The arguments are sequences of coefficients, from lowest order term to highest, e.g., $[1,2,3]$ represents the polynomial $1 + 2x + 3x^2$.

Parameters
- `c1, c2`: array_like

1-D arrays of coefficients representing a polynomial, relative to the “standard” basis, and ordered from lowest order term to highest.

Returns
- `out`: ndarray

Of the coefficients of their product.

See Also:
polyadd, polysub, polydiv, polypow

Examples

```python
>>> import numpy.polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polymul(c1,c2)
array([ 3., 8., 14., 8., 3.])
```

`numpy.polynomial.polynomial.polymulx(c)`
Multiply a polynomial by $x$.

Multiply the polynomial $c$ by $x$, where $x$ is the independent variable.

Parameters
- `c`: array_like

1-D array of polynomial coefficients ordered from low to high.

Returns
- `out`: ndarray

Array representing the result of the multiplication.

Notes
New in version 1.5.0.
numpy.polynomial.polynomial.polydiv(c1, c2)

Divide one polynomial by another.

Returns the quotient-with-remainder of two polynomials \(c1 / c2\). The arguments are sequences of coefficients, from lowest order term to highest, e.g., [1,2,3] represents \(1 + 2x + 3x^2\).

Parameters
   c1, c2 : array_like

1-D arrays of polynomial coefficients ordered from low to high.

Returns
   [quo, rem] : ndarrays

Of coefficient series representing the quotient and remainder.

See Also:
   polyadd, polysub, polymul, polypow

Examples
   >>> import numpy.polynomial as P
   >>> c1 = (1,2,3)
   >>> c2 = (3,2,1)
   >>> P.polydiv(c1,c2)
   (array([ 3.]), array([-8., -4.]))
   >>> P.polydiv(c2,c1)
   (array([ 0.33333333]), array([ 2.66666667, 1.33333333]))

numpy.polynomial.polynomial.polypow(c, pow, maxpower=None)

Raise a polynomial to a power.

Returns the polynomial \(c\) raised to the power \(pow\). The argument \(c\) is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series \(1 + 2x + 3x^2\).

Parameters
   c : array_like

1-D array of array of series coefficients ordered from low to high degree.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns
   coef : ndarray

Power series of power.

See Also:
   polyadd, polysub, polymul, polydiv

polycompanion(c) Return the companion matrix of c.
polydomain
polyzero

Continued on next page
Table 3.129 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>polyone</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>polynx</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>polytrim(c[, tol])</td>
<td>Remove “small” “trailing” coefficients from a polynomial.</td>
</tr>
<tr>
<td>polyline(off, scl)</td>
<td>Returns an array representing a linear polynomial.</td>
</tr>
</tbody>
</table>

**Miscellaneous**

numpy.polynomial.polynomial.polycompanion(c)

Return the companion matrix of c.

The companion matrix for power series cannot be made symmetric by scaling the basis, so this function differs from those for the orthogonal polynomials.

**Parameters**

c : array_like

1-D array of polynomial coefficients ordered from low to high degree.

**Returns**

mat : ndarray

Companion matrix of dimensions (deg, deg).

**Notes**

New in version 1.7.0.

numpy.polynomial.polynomial.polydomain = array([-1, 1])

numpy.polynomial.polynomial.polyzero = array([0])

numpy.polynomial.polynomial.polyone = array([1])

numpy.polynomial.polynomial.polyx = array([0, 1])

numpy.polynomial.polynomial.polytrim(c, tol=0)

Remove “small” “trailing” coefficients from a polynomial.

“Small” means “small in absolute value” and is controlled by the parameter tol; “trailing” means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be “trimmed.”

**Parameters**

c : array_like

1-d array of coefficients, ordered from lowest order to highest.

tol : number, optional

Trailing (i.e., highest order) elements with absolute value less than or equal to tol (default value is zero) are removed.

**Returns**

trimmed : ndarray

1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.
Raises

ValueError

If \( tol < 0 \)

See Also:

trimseq

Examples

```python
>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([ 0., 0., 3., 0., 5.])
```

```python
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([ 0.])
```

```python
i = complex(0,1) # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([ 0.0003+0.j , 0.0010-0.001j])
```

```python
numpy.polynomial.polynomial.polyline(off, scl)
```

Returns an array representing a linear polynomial.

Parameters

off, scl : scalars

The “y-intercept” and “slope” of the line, respectively.

Returns

y : ndarray

This module’s representation of the linear polynomial \( off + scl \times x \).

See Also:

chebline

Examples

```python
>>> from numpy import polynomial as P
>>> P.polyline(1,-1)
array([ 1, -1])
```

```python
>>> P.polyval(1, P.polyline(1,-1)) # should be 0
0.0
```

Chebyshev Module (numpy.polynomial.chebyshev)

New in version 1.4.0. This module provides a number of objects (mostly functions) useful for dealing with Chebyshev series, including a \texttt{Chebyshev} class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, \texttt{numpy.polynomial}).

\texttt{Chebyshev} class

```
numpy.polynomial.chebyshev.Chebyshev(coef[, domain, window])
```

A Chebyshev series class.

Chebyshev Class

```
class numpy.polynomial.chebyshev.Chebyshev(coef, domain=[-1, 1], window=[-1, 1])
```

A Chebyshev series class.

Chebyshev instances provide the standard Python numerical methods \`+\', \`-\', \`*\', \`/\', \`%\', \`divmod\', \`**\', and \`0\' as well as the listed methods.
Parameters

- **coef**: array_like
  
  Chebyshev coefficients, in increasing order. For example, \((1, 2, 3)\) implies \(P_0 + 2P_1 + 3P_2\) where the \(P_i\) are a graded polynomial basis.

- **domain**: (2,) array_like, optional
  
  Domain to use. The interval \([\text{domain}[0], \text{domain}[1]]\) is mapped to the interval \([\text{window}[0], \text{window}[1]]\) by shifting and scaling. The default value is \([-1,1]\).

- **window**: (2,) array_like, optional
  
  Window, see domain for its use. The default value is \([-1,1]\). .. versionadded:: 1.6.0

Notes

It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coef</td>
<td>((N,) \text{ndarray}) Chebyshev coefficients, from low to high.</td>
</tr>
<tr>
<td>domain</td>
<td>((2,) \text{ndarray}) Domain that is mapped to window.</td>
</tr>
<tr>
<td>window</td>
<td>((2,) \text{ndarray}) Window that domain is mapped to.</td>
</tr>
</tbody>
</table>

Methods

- **__call__**(arg)
  
  Chebyshev polynomial of degree deg.

- **basis**(deg[, domain, window])
  
  Chebyshev polynomial of degree deg.

- **cast**(series[, domain, window])
  
  Convert instance to equivalent Chebyshev series.

- **convert**([domain, kind, window])
  
  Convert to different class and/or domain.

- **copy**()
  
  Return a copy.

- **cutdeg**(deg)
  
  Truncate series to the given degree.

- **degree**()
  
  The degree of the series.

- **deriv**([m])
  
  Differentiate.

- **fit**(x, y, deg[, domain, rcond, full, w, window])
  
  Least squares fit to data.

- **fromroots**(roots[, domain, window])
  
  Return Chebyshev instance with specified roots.

- **has_samecoef**(other)
  
  Check if coefficients match.

- **has_samedomain**(other)
  
  Check if domains match.

- **has_sametype**(other)
  
  Check if types match.

- **has_samewindow**(other)
  
  Check if windows match.

- **identity**([domain, window])
  
  Identity function.

- **integ**([m, k, lbnd])
  
  Integrate.

- **linspace**(in, domain)
  
  Return x,y values at equally spaced points in domain.

- **mapparms**()
  
  Return the mapping parameters.

- **roots**()
  
  Return list of roots.

- **trim**(itol)
  
  Remove small leading coefficients

- **truncate**(size)
  
  Truncate series to length size.

Chebyshev.__call__(arg)

static Chebyshev.basis(deg, domain=[-1, 1], window=[-1, 1])

Chebyshev polynomial of degree deg.
Returns an instance of the Chebyshev polynomial of degree \( d \).

**Parameters**

- **deg**: int
  Degree of the Chebyshev polynomial. Must be \( \geq 0 \).

- **domain**: array_like
  The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the domain.

- **window**: array_like
  The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the window.

**Returns**

- **p**: Chebyshev instance

**Notes**

New in version 1.7.0.

**static** `Chebyshev.cast(series, domain=[-1, 1], window=[-1, 1])`

Convert instance to equivalent Chebyshev series.

The `series` is expected to be an instance of some polynomial series of one of the types supported by the numpy.polynomial module, but could be some other class that supports the convert method.

**Parameters**

- **series**: series
  The instance series to be converted.

- **domain**: array_like
  The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the domain.

- **window**: array_like
  The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the window.

**Returns**

- **p**: Chebyshev instance
  A Chebyshev series equal to the `poly` series.

**See Also:**

`convert`

**Notes**

New in version 1.7.0.

`Chebyshev.convert(domain=None, kind=None, window=None)`

Convert to different class and/or domain.

**Parameters**

- **domain**: array_like, optional
The domain of the converted series. If the value is None, the default domain of kind is used.

kind : class, optional

The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

window : array_like, optional

The window of the converted series. If the value is None, the default window of kind is used.

Returns

new_series_instance : kind

The returned class can be of different type than the current instance and/or have a different domain.

Notes

Conversion between domains and class types can result in numerically ill defined series.

Chebyshev.copy()

Return a copy.

Returns

new_instance : Chebyshev

Copy of current instance.

Chebyshev.cutdeg(deg)

Truncate series to the given degree.

Reduce the degree of the Chebyshev series to deg by discarding the high order terms. If deg is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters

deg : non-negative int

The series is reduced to degree deg by discarding the high order terms. The value of deg must be a non-negative integer.

Returns

new_instance : Chebyshev

New instance of Chebyshev with reduced degree.

Notes

New in version 1.5.0.

Chebyshev.degree()

The degree of the series.

Notes

New in version 1.5.0.

Chebyshev.deriv(m=L)

Differentiate.
Return an instance of Chebyshev that is the derivative of the current series. Refer to `chebder` for full documentation.

**Parameters**

- `m`: non-negative int
  
  The number of integrations to perform.

**Returns**

- `derivative`: Chebyshev
  
  The derivative of the series using the same domain.

**See Also:**

- `chebder` similar function.
- `chebint` similar function for integration.

**static** `Chebyshev.fit(x, y, deg, domain=None, rcond=None, full=False, w=None, window=[-1, 1])`

Least squares fit to data.

Return a `Chebyshev` instance that is the least squares fit to the data \( y \) sampled at \( x \). Unlike `chebfit`, the domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning. Support for NA was added in version 1.7.0. See `chebfit` for full documentation of the implementation.

**Parameters**

- `x`: array_like, shape (M,)
  
  x-coordinates of the M sample points \( (x[i], y[i]) \).

- `y`: array_like, shape (M,) or (M, K)
  
  y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- `deg`: int
  
  Degree of the fitting polynomial.

- `domain`: {None, [beg, end], []}, optional
  
  Domain to use for the returned Chebyshev instance. If None, then a minimal domain that covers the points \( x \) is chosen. If [] the default domain \([-1, 1]\) is used. The default value is \([-1,1]\) in numpy 1.4.x and None in later versions. The `[]` value was added in numpy 1.5.0.

- `rcond`: float, optional
  
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \( \text{len}(x) \times \text{eps} \), where \( \text{eps} \) is the relative precision of the float type, about 2e-16 in most cases.

- `full`: bool, optional
  
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

- `w`: array_like, shape (M,), optional
Weights. If not None the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i] \cdot y[i]\) all have the same variance. The default value is None. .. versionadded:: 1.5.0

\text{window} : \{\text{[beg, end]}, \text{optional}\}

Window to use for the returned Chebyshev instance. The default value is \([-1, 1]\) .. versionadded:: 1.6.0

Returns

\text{least_squares_fit} : \text{instance of Chebyshev}

The Chebyshev instance is the least squares fit to the data and has the domain specified in the call.

[\text{residuals, rank, singular_values, rcond}] : only if \text{full} = True

Residuals of the least squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of \text{rcond}. For more details, see \text{linalg.lstsq}.

See Also:

\text{chebfit}

similar function

static \text{Chebyshev}\text{.fromroots}(\text{roots, domain=[-1, 1], window=[-1, 1]})

Return Chebyshev instance with specified roots.

Returns an instance of Chebyshev representing the product \((x - r[0]) \cdot (x - r[1]) \cdot \ldots \cdot (x - r[n-1])\), where \(r\) is the list of roots.

Parameters

\text{roots} : \text{array_like}

List of roots.

\text{domain} : \{\text{array_like, None}, \text{optional}\}

Domain for the resulting instance of Chebyshev. If none the domain is the interval from the smallest root to the largest. The default is \([-1,1]\).

\text{window} : \text{array_like, optional}

Window for the resulting instance of Chebyshev. The default value is \([-1,1]\).

Returns

\text{object} : \text{Chebyshev instance}

Series with the specified roots.

See Also:

\text{chebfromroots}

equivalent function

\text{Chebyshev}\text{.has_samecoef}(\text{other})

Check if coefficients match.

Parameters

\text{other} : \text{class instance}
The other class must have the `coef` attribute.

**Returns**

bool : boolean

True if the coefficients are the same, False otherwise.

**Notes**

New in version 1.6.0.

```python
Chebyshev.has_samedomain(other)
```

Check if domains match.

**Parameters**

other : class instance

The other class must have the `domain` attribute.

**Returns**

bool : boolean

True if the domains are the same, False otherwise.

**Notes**

New in version 1.6.0.

```python
Chebyshev.has_sametype(other)
```

Check if types match.

**Parameters**

other : object

Class instance.

**Returns**

bool : boolean

True if other is same class as self

**Notes**

New in version 1.7.0.

```python
Chebyshev.has_samewindow(other)
```

Check if windows match.

**Parameters**

other : class instance

The other class must have the `window` attribute.

**Returns**

bool : boolean

True if the windows are the same, False otherwise.

**Notes**

New in version 1.6.0.

```python
static Chebyshev.identity(domain=[-1, 1], window=[-1, 1])
```

Identity function.

If $p$ is the returned Chebyshev object, then $p(x) = x$ for all values of $x$. 

### 3.26. Polynomials

987
Parameters

- **domain**: array_like
  The resulting array must be of the form \([\text{beg, end}]\), where \(\text{beg}\) and \(\text{end}\) are the endpoints of the domain.

- **window**: array_like
  The resulting array must be if the form \([\text{beg, end}]\), where \(\text{beg}\) and \(\text{end}\) are the endpoints of the window.

Returns

- **identity**: Chebyshev instance

\[
\text{Chebyshev}.\text{integ}(m=1, k=[], \text{lbnd}=\text{None})
\]

Integrate.

Return an instance of Chebyshev that is the definite integral of the current series. Refer to **chebint** for full documentation.

Parameters

- **m**: non-negative int
  The number of integrations to perform.

- **k**: array_like
  Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to \(m\) in length and any missing values are set to zero.

- **lbnd**: Scalar
  The lower bound of the definite integral.

Returns

- **integral**: Chebyshev
  The integral of the series using the same domain.

See Also:

- **chebint**
  similar function.

- **chebder**
  similar function for derivative.

\[
\text{Chebyshev}.\text{linpace}(n=100, \text{domain}=\text{None})
\]

Return \(x, y\) values at equally spaced points in domain.

Returns \(x, y\) values at \(n\) linearly spaced points across domain. Here \(y\) is the value of the polynomial at the points \(x\). By default the domain is the same as that of the Chebyshev instance. This method is intended mostly as a plotting aid.

Parameters

- **n**: int, optional
  Number of point pairs to return. The default value is 100.

- **domain**: \{None, array_like\}
  If not None, the specified domain is used instead of that of the calling instance. It should be of the form \([\text{beg, end}]\). The default is None.
Returns

\( x, y \) : ndarrays

\( x \) is equal to linspace(self.domain[0], self.domain[1], n) \( y \) is the polynomial evaluated at \( x \).

New in version 1.5.0.

**Chebyshev.mapparms()**

Return the mapping parameters.

The returned values define a linear map \( \text{off} + \text{scl} \times x \) that is applied to the input arguments before the series is evaluated. The map depends on the domain and window; if the current domain is equal to the window the resulting map is the identity. If the coefficients of the Chebyshev instance are to be used by themselves outside this class, then the linear function must be substituted for the \( x \) in the standard representation of the base polynomials.

**Returns**

\( \text{off}, \text{scl} \) : floats or complex

The mapping function is defined by \( \text{off} + \text{scl} \times x \).

**Notes**

If the current domain is the interval \([l_1, r_1]\) and the window is \([l_2, r_2]\), then the linear mapping function \( L \) is defined by the equations:

\[
\begin{align*}
L(l_1) &= l_2 \\
L(r_1) &= r_2
\end{align*}
\]

**Chebyshev.roots()**

Return list of roots.

Return ndarray of roots for this series. See **chebroots** for full documentation. Note that the accuracy of the roots is likely to decrease the further outside the domain they lie.

**See Also:**

- **chebroots**
  similar function
- **chebfroots**
  function to go generate series from roots.

**Chebyshev.trim(tol=0)**

Remove small leading coefficients

Remove leading coefficients until a coefficient is reached whose absolute value greater than \( \text{tol} \) or the beginning of the series is reached. If all the coefficients would be removed the series is set to \([0]\). A new Chebyshev instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

\( \text{tol} \) : non-negative number.

All trailing coefficients less than \( \text{tol} \) will be removed.

**Returns**

\( \text{new_instance} \) : Chebyshev

Contains the new set of coefficients.

3.26. Polynomials 989
Chebyshev.truncate(size)

Truncate series to length size.

Reduce the Chebyshev series to length size by discarding the high degree terms. The value of size must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters

size : positive int

The series is reduced to length size by discarding the high degree terms. The value of size must be a positive integer.

Returns

new_instance : Chebyshev

New instance of Chebyshev with truncated coefficients.

chebval(x, c[, tensor])  Evaluate a Chebyshev series at points x.

chebval2d(x, y, c)  Evaluate a 2-D Chebyshev series at points (x, y).

chebval3d(x, y, z, c)  Evaluate a 3-D Chebyshev series at points (x, y, z).

chebgrid2d(x, y, c)  Evaluate a 2-D Chebyshev series on the Cartesian product of x and y.

chebgrid3d(x, y, z, c)  Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.

chebroots(c)  Compute the roots of a Chebyshev series.

chebfromroots(roots)  Generate a Chebyshev series with given roots.

Basics

numpy.polynomial.chebyshev.chebval(x, c, tensor=True)

Evaluate a Chebyshev series at points x.

If c is of length n + 1, this function returns the value:

\[ p(x) = c_0 \times T_0(x) + c_1 \times T_1(x) + \ldots + c_n \times T_n(x) \]

The parameter x is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either x or its elements must support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array, then \( p(x) \) will have the same shape as x. If c is multidimensional, then the shape of the result depends on the value of tensor. If tensor is true the shape will be c.shape[1:] + x.shape. If tensor is false the shape will be c.shape[1:]. Note that scalars have shape (). Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

x : array_like, compatible object

If x is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, x or its elements must support addition and multiplication with with themselves and with the elements of c.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If c is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of c.
tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each
dimension of x. Scalars have dimension 0 for this action. The result is that every column
of coefficients in c is evaluated for every element of x. If False, x is broadcast over the
columns of c for the evaluation. This keyword is useful when c is multidimensional.
The default value is True. New in version 1.7.0.

Returns
values : ndarray, algebra_like

The shape of the return value is described above.

See Also:
chebval2d, chebgrid2d, chebval3d, chebgrid3d

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.
numpy.polynomial.chebyshev.chebval2d(x, y, c)
Evaluate a 2-D Chebyshev series at points (x, y).
This function returns the values:

\[ p(x, y) = \sum_{i,j} c_{i,j} \cdot T_i(x) \cdot T_j(y) \]

The parameters x and y are converted to arrays only if they are tuples or a lists, otherwise they are treated as a
scalars and they must have the same shape after conversion. In either case, either x and y or their elements must
support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be
c.shape[2:] + x.shape.

Parameters
x, y : array_like, compatible objects

The two dimensional series is evaluated at the points (x, y), where x and y must have the
same shape. If x or y is a list or tuple, it is first converted to an ndarray, otherwise it is
left unchanged and if it isn’t an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is con-
tained in c[i, j]. If c has dimension greater than 2 the remaining indices enumerate
multiple sets of coefficients.

Returns
values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points formed from pairs of
 corresponding values from x and y.

See Also:
chebval, chebgrid2d, chebval3d, chebgrid3d

Notes

numpy.polynomial.chebyshev.chebval3d(x, y, z, c)
Evaluate a 3-D Chebyshev series at points (x, y, z).
This function returns the values:

\[ p(x, y, z) = \sum_{i,j,k} c_{i,j,k} \times T_i(x) \times T_j(y) \times T_k(z) \]

The parameters \( x, y, \) and \( z \) are converted to arrays only if they are tuples or lists, otherwise they are treated as scalars and they must have the same shape after conversion. In either case, either \( x, y, \) and \( z \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \( c.shape[3:] + x.shape \).

**Parameters**

- \( x, y, z \) : array_like, compatible object
  The three dimensional series is evaluated at the points \( (x, y, z) \), where \( x, y, \) and \( z \) must have the same shape. If any of \( x, y, \) or \( z \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

- \( c \) : array_like
  Array of coefficients ordered so that the coefficient of the term of multi-degree \( i,j,k \) is contained in \( c[i, j, k] \). If \( c \) has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

- \( values \) : ndarray, compatible object
  The values of the multidimensional polynomial on points formed with triples of corresponding values from \( x, y, \) and \( z \).

**See Also:**

- chebval
- chebval2d
- chebgrid2d
- chebgrid3d

**Notes**

numpy.polynomial.chebyshev.chebgrid2d \((x, y, c)\)

Evaluate a 2-D Chebyshev series on the Cartesian product of \( x \) and \( y \).

This function returns the values:

\[ p(a, b) = \sum_{i,j} c_{i,j} \times T_i(a) \times T_j(b), \]

where the points \( (a, b) \) consist of all pairs formed by taking \( a \) from \( x \) and \( b \) from \( y \). The resulting points form a grid with \( x \) in the first dimension and \( y \) in the second.

The parameters \( x \) and \( y \) are converted to arrays only if they are tuples or lists, otherwise they are treated as scalars. In either case, either \( x \) and \( y \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \( c.shape[2:] + x.shape + y.shape \).

**Parameters**

- \( x, y \) : array_like, compatible objects
  The two dimensional series is evaluated at the points in the Cartesian product of \( x \) and \( y \). If \( x \) or \( y \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.
c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is
contained in c[i,j]. If c has dimension greater than two the remaining indices enumerate
multiple sets of coefficients.

Returns
values : ndarray, compatible object

The values of the two dimensional Chebyshev series at points in the Cartesian product
of x and y.

See Also:
chebval, chebval2d, chebval3d, chebgrid3d

Notes
numpy.polynomial.chebyshev.chebgrid3d(x, y, z, c)

Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.

This function returns the values:

\[ p(a, b, c) = \sum_{i,j,k} c_{i,j,k} \cdot T_i(a) \cdot T_j(b) \cdot T_k(c) \]

where the points \((a, b, c)\) consist of all triples formed by taking \(a\) from \(x\), \(b\) from \(y\), and \(c\) from \(z\). The resulting
points form a grid with \(x\) in the first dimension, \(y\) in the second, and \(z\) in the third.

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as
as scalars. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with
themselves and with the elements of \(c\).

If \(c\) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the
result will be \(c.shape[3:] + x.shape + y.shape + z.shape\).

Parameters
- x, y, z : array_like, compatible objects
  The three dimensional series is evaluated at the points in the Cartesian product of \(x\), \(y\),
  and \(z\). If \(x\), \(y\), or \(z\) is a list or tuple, it is first converted to an ndarray, otherwise it is left
  unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- c : array_like
  Array of coefficients ordered so that the coefficients for terms of degree i,j are contained
  in \(c[i, j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple
  sets of coefficients.

Returns
values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of \(x\)
and \(y\).

See Also:
chebval, chebval2d, chebgrid2d, chebval3d
NumPy Reference, Release 1.8.1

Notes

numpy.polynomial.chebyshev.chebroots(c)
Compute the roots of a Chebyshev series.

Return the roots (a.k.a. “zeros”) of the polynomial

\[ p(x) = \sum_i c[i] \ast T_i(x). \]

Parameters

- c : 1-D array_like
  1-D array of coefficients.

Returns

- out : ndarray
  Array of the roots of the series. If all the roots are real, then out is also real, otherwise it is complex.

See Also:

polyroots, legroots, lagroots, hermroots, hermeroots

Notes

The root estimates are obtained as the eigenvalues of the companion matrix. Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s method.

The Chebyshev series basis polynomials aren’t powers of x so the results of this function may seem unintuitive.

Examples

```python
>>> import numpy.polynomial.chebyshev as cheb
>>> cheb.chebroots((-1, 1,-1, 1)) # T3 - T2 + T1 - T0 has real roots
array([-5.00000000e-01, 2.60860684e-17, 1.00000000e+00])
```

numpy.polynomial.chebyshev.chebfromroots(roots)
Generate a Chebyshev series with given roots.

The function returns the coefficients of the polynomial

\[ p(x) = (x - r_0) \ast (x - r_1) \ast ... \ast (x - r_n), \]

in Chebyshev form, where the \( r_n \) are the roots specified in roots. If a zero has multiplicity n, then it must appear in roots n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then roots looks something like \([2, 2, 2, 3, 3]\). The roots can appear in any order.

If the returned coefficients are c, then

\[ p(x) = c_0 + c_1 \ast T_1(x) + ... + c_n \ast T_n(x) \]

The coefficient of the last term is not generally 1 for monic polynomials in Chebyshev form.
Parameters
roots : array_like
Sequence containing the roots.

Returns
out : ndarray
1-D array of coefficients. If all roots are real then out is a real array, if some of the roots are complex, then out is complex even if all the coefficients in the result are real (see Examples below).

See Also:
polyfromroots, legfromroots, lagfromroots, hermfromroots, hermefromroots.

Examples

```python
c.chebfromroots((-1, 0, 1)) # x^3 - x relative to the standard basis
array([0., -0.25, 0., 0.25])
c.chebfromroots((-j, j)) # x^2 + 1 relative to the standard basis
array([1.5+0.j, 0.0+0.j, 0.5+0.j])
```

chebfit(x, y, deg[, rcond, full, w])  Least squares fit of Chebyshev series to data.

Return the coefficients of a Legendre series of degree deg that is the least squares fit to the data values y given at points x. If y is 1-D the returned coefficients will also be 1-D. If y is 2-D multiple fits are done, one for each column of y, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

\[ p(x) = c_0 + c_1 \cdot T_1(x) + \ldots + c_n \cdot T_n(x), \]

where \( n \) is deg.

Since numpy version 1.7.0, chebfit also supports NA. If any of the elements of x, y, or w are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If y is 2-D, then an NA in any row of y invalidates that whole row.

Parameters
x : array_like, shape (M,)
x-coordinates of the M sample points \((x[i], y[i])\).

y : array_like, shape (M,) or (M, K)
y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.
deg : int
    Degree of the fitting series

rcond : float, optional
    Relative condition number of the fit. Singular values smaller than this relative to the
    largest singular value will be ignored. The default value is len(x)*eps, where eps is the
    relative precision of the float type, about 2e-16 in most cases.

full : bool, optional
    Switch determining nature of return value. When it is False (the default) just the coeffi-
    cients are returned, when True diagnostic information from the singular value decom-
    position is also returned.

w : array_like, shape (M,), optional
    Weights. If not None, the contribution of each point \((x[i],y[i])\) to the fit is
    weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products
    \(w[i]*y[i]\) all have the same variance. The default value is None. New in version
    1.5.0.

Returns

coef : ndarray, shape (M,) or (M, K)
    Chebyshev coefficients ordered from low to high. If \(y\) was 2-D, the coefficients for the
    data in column \(k\) of \(y\) are in column \(k\).

[residuals, rank, singular_values, rcond] : present when full = True
    Residuals of the least-squares fit, the effective rank of the scaled Vandermonde ma-
   trix and its singular values, and the specified value of \(rcond\). For more details, see
    linalg.lstsq.

Warns

RankWarning
    The rank of the coefficient matrix in the least-squares fit is deficient. The warning is
    only raised if full = False. The warnings can be turned off by

    >>> import warnings
    >>> warnings.simplefilter('ignore', RankWarning)

See Also:

polyfit, legfit, lagfit, hermfit, hermefit

chebval
    Evaluates a Chebyshev series.

chebvander
    Vandermonde matrix of Chebyshev series.

chebweight
    Chebyshev weight function.

linalg.lstsq
    Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline
    Computes spline fits.
Notes

The solution is the coefficients of the Chebyshev series \( p \) that minimizes the sum of the weighted squared errors

\[
E = \sum_j w_j^2 \left| y_j - p(x_j) \right|^2,
\]

where \( w_j \) are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

\[
V(x) \ast c = w \ast y,
\]

where \( V \) is the weighted pseudo Vandermonde matrix of \( x \), \( c \) are the coefficients to be solved for, \( w \) are the weights, and \( y \) are the observed values. This equation is then solved using the singular value decomposition of \( V \).

If some of the singular values of \( V \) are so small that they are neglected, then a `RankWarning` will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The `rcond` parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Chebyshev series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

References

[R58] numpy.polynomial.chebyshev.chebvander(x, deg)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree \( deg \) and sample points \( x \). The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, i] = T_i(x),
\]

where \( 0 \leq i \leq deg \). The leading indices of \( V \) index the elements of \( x \) and the last index is the degree of the Chebyshev polynomial.

If \( c \) is a 1-D array of coefficients of length \( n + 1 \) and \( V \) is the matrix

\[
V = \text{chebvander}(x, n),
\]

then \( \text{np.dot}(V, c) \) and \( \text{chebval}(x, c) \) are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Chebyshev series of the same degree and sample points.

Parameters

- \( x \) : array_like
  - Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If \( x \) is scalar it is converted to a 1-D array.

- \( \text{deg} \) : int
  - Degree of the resulting matrix.

Returns

- \( \text{vander} \) : ndarray
  - The pseudo Vandermonde matrix. The shape of the returned matrix is \( x.\text{shape} + (\text{deg} + 1,) \), where The last index is the degree of the corresponding Chebyshev polynomial. The dtype will be the same as the converted \( x \).
numpy.polynomial.chebyshev.chebvander2d(x, y, deg)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees \( \text{deg} \) and sample points \((x, y)\). The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, \text{deg}[1] \cdot i + j] = T_i(x) \cdot T_j(y),
\]

where \(0 \leq i \leq \text{deg}[0]\) and \(0 \leq j \leq \text{deg}[1]\). The leading indices of \( V \) index the points \((x, y)\) and the last index encodes the degrees of the Chebyshev polynomials.

If \( V = \text{chebvander2d}(x, y, \text{xdeg}, \text{ydeg}) \), then the columns of \( V \) correspond to the elements of a 2-D coefficient array \( c \) of shape \((\text{xdeg} + 1, \text{ydeg} + 1)\) in the order

\[
c_{00}, c_{01}, c_{02}, \ldots, c_{10}, c_{11}, c_{12}, \ldots
\]

and \( \text{np.dot}(V, \text{c.flat}) \) and \( \text{chebval2d}(x, y, c) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Chebyshev series of the same degrees and sample points.

**Parameters**

- \( x, y \): array_like
  - Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex.
  - Scalars are converted to 1-D arrays.
- \( \text{deg} \): list of ints
  - List of maximum degrees of the form \([x\_\text{deg}, y\_\text{deg}]\).

**Returns**

- \( \text{vander2d} \): ndarray
  - The shape of the returned matrix is \(x.\text{shape} + (\text{order},)\), where \( \text{order} = (\text{deg}[0] + 1) \cdot (\text{deg}[1] + 1) \). The dtype will be the same as the converted \( x \) and \( y \).

**See Also:**

- \( \text{chebvander, chebvander3d, chebval3d} \)

**Notes**

numpy.polynomial.chebyshev.chebvander3d(x, y, z, deg)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees \( \text{deg} \) and sample points \((x, y, z)\). If \( l, m, n \) are the given degrees in \( x, y, z \), then The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, (m + 1)(n + 1)i + (n + 1)j + k] = T_i(x) \cdot T_j(y) \cdot T_k(z),
\]

where \(0 \leq i \leq l\), \(0 \leq j \leq m\), and \(0 \leq k \leq n\). The leading indices of \( V \) index the points \((x, y, z)\) and the last index encodes the degrees of the Chebyshev polynomials.

If \( V = \text{chebvander3d}(x, y, z, \text{xdeg}, \text{ydeg}, \text{zdeg}) \), then the columns of \( V \) correspond to the elements of a 3-D coefficient array \( c \) of shape \((\text{xdeg} + 1, \text{ydeg} + 1, \text{zdeg} + 1)\) in the order

\[
c_{000}, c_{001}, c_{002}, \ldots, c_{010}, c_{011}, c_{012}, \ldots
\]
and \( \text{np.dot}(V, \text{c.flat}) \) and \( \text{chebval3d}(x, y, z, \text{c}) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Chebyshev series of the same degrees and sample points.

**Parameters**

- **x, y, z**: array_like
  - Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

- **deg**: list of ints
  - List of maximum degrees of the form \([x_{\text{deg}}, y_{\text{deg}}, z_{\text{deg}}]\).

**Returns**

- **vander3d**: ndarray
  - The shape of the returned matrix is \(x.\text{shape} + (\text{order},)\), where \(\text{order} = (\text{deg}[0] + 1) \times (\text{deg}[1] + 1) \times (\text{deg}[2] + 1)\). The dtype will be the same as the converted \(x, y,\) and \(z\).

**See Also:**

\(\text{chebval}, \text{chebval3d}, \text{chebval3d}\)

**Notes**

\[
\begin{align*}
\text{chebder}(c[, m, scl, axis]) & \quad \text{Differentiate a Chebyshev series.} \\
\text{chebint}(c[, m, k, lbnd, scl, axis]) & \quad \text{Integrate a Chebyshev series.}
\end{align*}
\]

**Calculus**

\texttt{numpy.polynomial.chebyshev.chebder(c, }m=1, scl=1, axis=0)\texttt{)}

Differentiate a Chebyshev series.

Returns the Chebyshev series coefficients \(c\) differentiated \(m\) times along \(axis\). At each iteration the result is multiplied by \(scl\) (the scaling factor is for use in a linear change of variable). The argument \(c\) is an array of coefficients from low to high degree along each axis, e.g., \([1,2,3]\) represents the series \(1\times T_0 + 2\times T_1 + 3\times T_2\) while \([[1,2],[1,2]]\) represents \(1\times T_0(x) \times T_0(y) + 1\times T_1(x) \times T_0(y) + 2\times T_0(x) \times T_1(y) + 2\times T_1(x) \times T_1(y)\) if \(axis=0\) is \(x\) and \(axis=1\) is \(y\).

**Parameters**

- **c**: array_like
  - Array of Chebyshev series coefficients. If \(c\) is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

- **m**: int, optional
  - Number of derivatives taken, must be non-negative. (Default: 1)

- **scl**: scalar, optional
  - Each differentiation is multiplied by \(scl\). The end result is multiplication by \(scl^{\times m}\). This is for use in a linear change of variable. (Default: 1)

- **axis**: int, optional
  - Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.
Returns

der : ndarray
    Chebyshev series of the derivative.

See Also:

chebint

Notes

In general, the result of differentiating a C-series needs to be “reprojected” onto the C-series basis set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples

```python
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3,4)
>>> C.chebder(c)
array([[ 14.,  12.,  24.]])
>>> C.chebder(c,3)
array([  96.])
>>> C.chebder(c,scl=-1)
array([-14., -12., -24.])
>>> C.chebder(c,2,-1)
array([ 12.,  96.])
```

Integrate a Chebyshev series.

Integrate a Chebyshev series.

```python
numpy.polynomial.chebyshev.chebint(c, m=1, k=[], lbnd=0, scl=1, axis=0)
```

Parameters

c : array_like
    Array of Chebyshev series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional
    Order of integration, must be positive. (Default: 1)

k : list, scalar, optional
    Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If k == [] (the default), all constants are set to zero. If m == 1, a single scalar can be given instead of a list.

lbnd : scalar, optional
    The lower bound of the integral. (Default: 0)

scl : scalar, optional
    Following each integration the result is multiplied by scl before the integration constant is added. (Default: 1)
axis : int, optional
Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

Returns
S : ndarray
C-series coefficients of the integral.

Raises
ValueError
If m < 1, len(k) > m, np.isscalar(lbnd) == False, or
np.isscalar(scl) == False.

See Also:
chebder

Notes
Note that the result of each integration is multiplied by scl. Why is this important to note? Say one is making a
linear change of variable u = ax + b in an integral relative to x. Then .. math:: dx = du/a, so one will need to set
scl equal to 1/a perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be “reprojected” onto the C-series basis
set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3)
>>> C.chebint(c)
array([ 0.5, -0.5, 0.5, 0.5])
>>> C.chebint(c,3)
array([ 0.03125 , -0.1875 , 0.04166667, -0.05208333, 0.01041667,
0.00625 ])
>>> C.chebint(c, k=3)
array([ 3.5, -0.5, 0.5, 0.5])
>>> C.chebint(c, lbnd=-2)
array([ 8.5, -0.5, 0.5, 0.5])
>>> C.chebint(c, scl=-2)
array([-1., 1., -1., -1.])

Algebra
numpy.polynomial.chebyshev.chebadd(c1, c2)
Add one Chebyshev series to another.

Returns the sum of two Chebyshev series c1 + c2. The arguments are sequences of coefficients ordered from
lowest order term to highest, i.e., [1,2,3] represents the series T_0  + 2*T_1  + 3*T_2.
c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the Chebyshev series of their sum.

See Also:
chebsub, chebmul, chebdiv, chebpow

Notes

Unlike multiplication, division, etc., the sum of two Chebyshev series is a Chebyshev series (without having to “reproject” the result onto the basis set) so addition, just like that of “standard” polynomials, is simply “component-wise.”

Examples

>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebadd(c1,c2)
array([ 4., 4., 4.])

numpy.polynomial.chebyshev.chebsub(c1,c2)
Subtract one Chebyshev series from another.

Returns the difference of two Chebyshev series \( c_1 - c_2 \). The sequences of coefficients are from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( T_0 + 2*T_1 + 3*T_2 \).

Parameters

c1, c2 : array_like

1-D arrays of Chebyshev series coefficients ordered from low to high.

Returns

out : ndarray

Of Chebyshev series coefficients representing their difference.

See Also:
chebadd, chebmul, chebdiv, chebpow

Notes

Unlike multiplication, division, etc., the difference of two Chebyshev series is a Chebyshev series (without having to “reproject” the result onto the basis set) so subtraction, just like that of “standard” polynomials, is simply “component-wise.”

Examples

>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebsub(c1,c2)
array([-2., 0., 2.])
>>> C.chebsub(c2,c1)  # -C.chebsub(c1,c2)
array([2., 0., -2.])
numpy.polynomial.chebyshev.chebmul(c1, c2)
Multiply one Chebyshev series by another.

Returns the product of two Chebyshev series \( c1 \times c2 \). The arguments are sequences of coefficients, from lowest order “term” to highest, e.g., \([1,2,3]\) represents the series \( T_0 + 2T_1 + 3T_2 \).

**Parameters**

- **c1, c2**: array_like
  
  1-D arrays of Chebyshev series coefficients ordered from low to high.

**Returns**

- **out**: ndarray
  
  Of Chebyshev series coefficients representing their product.

**See Also:**
chebadd, chebsub, chebdiv, chebpow

**Notes**

In general, the (polynomial) product of two C-series results in terms that are not in the Chebyshev polynomial basis set. Thus, to express the product as a C-series, it is typically necessary to “reproject” the product onto said basis set, which typically produces “unintuitive live” (but correct) results; see Examples section below.

**Examples**

```python
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebmul(c1,c2)  # multiplication requires "reprojection"
array([ 6.5, 12. , 12. , 4. , 1.5])
```

numpy.polynomial.chebyshev.chebmulx(c)
Multiply a Chebyshev series by \( x \).

Multiply the polynomial \( c \) by \( x \), where \( x \) is the independent variable.

**Parameters**

- **c**: array_like
  
  1-D array of Chebyshev series coefficients ordered from low to high.

**Returns**

- **out**: ndarray
  
  Array representing the result of the multiplication.

**Notes**

New in version 1.5.0.

numpy.polynomial.chebyshev.chebdiv(c1, c2)
Divide one Chebyshev series by another.

Returns the quotient-with-remainder of two Chebyshev series \( c1 / c2 \). The arguments are sequences of coefficients from lowest order “term” to highest, e.g., \([1,2,3]\) represents the series \( T_0 + 2T_1 + 3T_2 \).

**Parameters**

- **c1, c2**: array_like
  
  1-D arrays of Chebyshev series coefficients ordered from low to high.
Returns

[quo, rem] : ndarrays

Of Chebyshev series coefficients representing the quotient and remainder.

See Also:

chebadd, chebsub, chebmul, chebpow

Notes

In general, the (polynomial) division of one C-series by another results in quotient and remainder terms that are not in the Chebyshev polynomial basis set. Thus, to express these results as C-series, it is typically necessary to “reproject” the results onto said basis set, which typically produces “unintuitive” (but correct) results; see Examples section below.

Examples

```python
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1, 2, 3)
>>> c2 = (3, 2, 1)
>>> C.chebdiv(c1, c2)  # quotient "intuitive," remainder not
(array([ 3.]), array([-8., -4.]))
>>> c2 = (0, 1, 2, 3)
>>> C.chebdiv(c2, c1)  # neither "intuitive"
(array([ 0., 2.]), array([-2., -4.]))
```

numpy.polynomial.chebyshev.chebpow(c, pow, maxpower=16)

Raise a Chebyshev series to a power.

Returns the Chebyshev series c raised to the power pow. The argument c is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series \(T_0 + 2 \cdot T_1 + 3 \cdot T_2\).

Parameters

c : array_like

1-D array of Chebyshev series coefficients ordered from low to high.

pow : integer

Power to which the series will be raised

maxpower : integer, optional

Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns

c : ndarray

Chebyshev series of power.

See Also:

chebadd, chebsub, chebmul, chebdiv

chebgauss(deg) Gauss-Chebyshev quadrature.

chebweight(x) The weight function of the Chebyshev polynomials.

Quadrature

numpy.polynomial.chebyshev.chebgauss(deg)
Gauss-Chebyshev quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. These sample points and weights will correctly integrate polynomials of degree $2 \times \text{deg} - 1$ or less over the interval $[-1, 1]$ with the weight function $f(x) = 1/\sqrt{1 - x^2}$.

**Parameters**

- **deg**: int
  Number of sample points and weights. It must be $\geq 1$.

**Returns**

- **x**: ndarray
  1-D ndarray containing the sample points.

- **y**: ndarray
  1-D ndarray containing the weights.

**Notes**

New in version 1.7.0. The results have only been tested up to degree 100, higher degrees may be problematic. For Gauss-Chebyshev there are closed form solutions for the sample points and weights. If $n = \text{deg}$, then

$$x_i = \cos(\pi(2i - 1)/(2n))$$

$$w_i = \pi/n$$

### numpy.polynomial.chebyshev.chebweight(x)

The weight function of the Chebyshev polynomials.

The weight function is $1/\sqrt{1 - x^2}$ and the interval of integration is $[-1, 1]$. The Chebyshev polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

- **x**: array_like
  Values at which the weight function will be computed.

**Returns**

- **w**: ndarray
  The weight function at $x$.

**Notes**

New in version 1.7.0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chebcompanion(c)</td>
<td>Return the scaled companion matrix of $c$.</td>
</tr>
<tr>
<td>chebdomain</td>
<td></td>
</tr>
<tr>
<td>chebzero</td>
<td></td>
</tr>
<tr>
<td>chebone</td>
<td></td>
</tr>
<tr>
<td>chebx</td>
<td></td>
</tr>
<tr>
<td>chebtrim(c[, tol])</td>
<td>Remove “small” “trailing” coefficients from a polynomial.</td>
</tr>
<tr>
<td>cheb1line(off, scl)</td>
<td>Chebyshev series whose graph is a straight line.</td>
</tr>
<tr>
<td>cheb2poly(c)</td>
<td>Convert a Chebyshev series to a polynomial.</td>
</tr>
<tr>
<td>poly2cheb(pol)</td>
<td>Convert a polynomial to a Chebyshev series.</td>
</tr>
</tbody>
</table>
Miscellaneous

numpy.polynomial.chebyshev.chebcompanion(c)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when c is an Chebyshev basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if numpy.linalg.eigvalsh is used to obtain them.

Parameters

c : array_like

1-D array of Chebyshev series coefficients ordered from low to high degree.

Returns

mat : ndarray

Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.chebyshev.chebdomain = array([-1, 1])
numpy.polynomial.chebyshev.chebzero = array([0])
numpy.polynomial.chebyshev.chebone = array([1])
numpy.polynomial.chebyshev.chebx = array([0, 1])

numpy.polynomial.chebyshev.chebtrim(c, tol=0)

Remove “small” “trailing” coefficients from a polynomial.

“Small” means “small in absolute value” and is controlled by the parameter tol; “trailing” means highest order coefficient(s), e.g., in [0, 1, 1, 0, 0] (which represents 0 + x + x**2 + 0*x**3 + 0*x**4) both the 3-rd and 4-th order coefficients would be “trimmed.”

Parameters

c : array_like

1-d array of coefficients, ordered from lowest order to highest.

tol : number, optional

Trailing (i.e., highest order) elements with absolute value less than or equal to tol (default value is zero) are removed.

Returns

trimmed : ndarray

1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

Raises

ValueError

If tol < 0

See Also:

trimseq
Examples

```python
>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([ 0., 3., 5.])
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([ 0.])
>>> i = complex(0,1) # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([ 0.0003+0.j , 0.0010-0.001j])
```

calculates the polynomial coefficients, which is a straight line.

**Parameters**

- `off`, `scl` : scalars
  
The specified line is given by `off + scl*x`.

**Returns**

- `y` : ndarray
  This module’s representation of the Chebyshev series for `off + scl*x`.

**See Also:**

polyline

**Notes**

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.
Examples

```python
>>> from numpy import polynomial as P
>>> c = P.Chebyshev(range(4))
>>> c
Chebyshev([ 0., 1., 2., 3.], [-1., 1.])
>>> p = c.convert(kind=P.Polynomial)
>>> p
Polynomial([-2., -8., 4., 12.], [-1., 1.])
```

```
numpy.polynomial.chebyshev.poly2cheb(pol)
Convert a polynomial to a Chebyshev series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Chebyshev series, ordered from lowest to highest degree.

Parameters
pol : array_like
1-D array containing the polynomial coefficients

Returns
c : ndarray
1-D array containing the coefficients of the equivalent Chebyshev series.

See Also:
cheb2poly

Notes
The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```python
>>> from numpy import polynomial as P
>>> p = P.Polynomial(range(4))
>>> p
Polynomial([ 0., 1., 2., 3.], [-1., 1.])
>>> c = p.convert(kind=P.Chebyshev)
>>> c
Chebyshev([ 1. , 3.25, 1. , 0.75], [-1., 1.])
>>> P.poly2cheb(range(4))
array([ 1. , 3.25, 1. , 0.75])
```

Legendre Module (numpy.polynomial.legendre)

New in version 1.6.0. This module provides a number of objects (mostly functions) useful for dealing with Legendre series, including a Legendre class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).

Legendre Class

Legendre(coef[, domain, window])  A Legendre series class.
class numpy.polynomial.legendre.Legendre (coef, domain=[-1, 1], window=[-1, 1])
A Legendre series class.

Legendre instances provide the standard Python numerical methods ‘+’, ‘-’, ‘*’, ‘/’, ‘%’, ‘divmod’, ‘**’, and ‘()’ as well as the listed methods.

Parameters

coef : array_like
Legendre coefficients, in increasing order. For example, (1, 2, 3) implies \( P_0 + 2P_1 + 3P_2 \) where the \( P_i \) are a graded polynomial basis.

domain : (2,) array_like, optional
Domain to use. The interval \([\text{domain}[0], \text{domain}[1]]\) is mapped to the interval \([\text{window}[0], \text{window}[1]]\) by shifting and scaling. The default value is \([-1,1]\).

window : (2,) array_like, optional
Window, see domain for its use. The default value is \([-1,1]\). .. versionadded:: 1.6.0

Notes

It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.

Attributes

<table>
<thead>
<tr>
<th>coef</th>
<th>((N,) ndarray) Legendre coefficients, from low to high.</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>((2,) ndarray) Domain that is mapped to window.</td>
</tr>
<tr>
<td>window</td>
<td>((2,) ndarray) Window that domain is mapped to.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th><strong>call</strong></th>
<th>(arg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>basis</td>
<td>(deg[, domain, window])</td>
</tr>
<tr>
<td>cast</td>
<td>(series[, domain, window])</td>
</tr>
<tr>
<td>convert</td>
<td>([domain, kind, window])</td>
</tr>
<tr>
<td>copy</td>
<td>()</td>
</tr>
<tr>
<td>cutdeg</td>
<td>(deg)</td>
</tr>
<tr>
<td>degree</td>
<td>()</td>
</tr>
<tr>
<td>deriv</td>
<td>([m])</td>
</tr>
<tr>
<td>fit</td>
<td>(x, y, deg[, domain, rcond, full, w, window])</td>
</tr>
<tr>
<td>fromroots</td>
<td>(roots[, domain, window])</td>
</tr>
<tr>
<td>has_samecoef</td>
<td>(other)</td>
</tr>
<tr>
<td>has_samedomain</td>
<td>(other)</td>
</tr>
<tr>
<td>has_sametype</td>
<td>(other)</td>
</tr>
<tr>
<td>has_samewindow</td>
<td>(other)</td>
</tr>
<tr>
<td>identity</td>
<td>([domain, window])</td>
</tr>
<tr>
<td>integ</td>
<td>([m, k, lbnd])</td>
</tr>
<tr>
<td>linspace</td>
<td>([n, domain])</td>
</tr>
<tr>
<td>mapparms</td>
<td>()</td>
</tr>
<tr>
<td>roots</td>
<td>()</td>
</tr>
<tr>
<td>trim</td>
<td>([tol])</td>
</tr>
<tr>
<td>truncate</td>
<td>(size)</td>
</tr>
</tbody>
</table>

3.26. Polynomials 1009
Legendre.__call__(arg)

static Legendre.basis(deg, domain=[-1, 1], window=[-1, 1])
Legendre polynomial of degree deg.

Returns an instance of the Legendre polynomial of degree d.

Parameters
deg : int
   Degree of the Legendre polynomial. Must be >= 0.
domain : array_like
   The resulting array must be of the form [beg, end], where beg and end are the
   endpoints of the domain.
window : array_like
   The resulting array must be if the form [beg, end], where beg and end are the
   endpoints of the window.

Returns
p : Legendre instance

Notes
New in version 1.7.0.

static Legendre.cast(series, domain=[-1, 1], window=[-1, 1])
Convert instance to equivalent Legendre series.

The series is expected to be an instance of some polynomial series of one of the types supported by the
numpy.polynomial module, but could be some other class that supports the convert method.

Parameters
series : series
   The instance series to be converted.
domain : array_like
   The resulting array must be of the form [beg, end], where beg and end are the
   endpoints of the domain.
window : array_like
   The resulting array must be if the form [beg, end], where beg and end are the
   endpoints of the window.

Returns
p : Legendre instance
   A Legendre series equal to the poly series.

See Also:
convert
Notes
New in version 1.7.0.

Legendre.convert (domain=None, kind=None, window=None)
Convert to different class and/or domain.

Parameters

domain : array_like, optional
The domain of the converted series. If the value is None, the default domain of kind is used.

kind : class, optional
The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

window : array_like, optional
The window of the converted series. If the value is None, the default window of kind is used.

Returns

new_series_instance : kind
The returned class can be of different type than the current instance and/or have a different domain.

Notes
Conversion between domains and class types can result in numerically ill defined series.

Legendre.copy ()
Return a copy.

Return a copy of the current Legendre instance.

Returns

new_instance : Legendre
Copy of current instance.

Legendre.cutdeg (deg)
Truncate series to the given degree.

Reduce the degree of the Legendre series to deg by discarding the high order terms. If deg is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters

deg : non-negative int
The series is reduced to degree deg by discarding the high order terms. The value of deg must be a non-negative integer.

Returns

new_instance : Legendre
New instance of Legendre with reduced degree.
**Notes**

New in version 1.5.0.

**Legendre**.degree()

The degree of the series.

**Notes**

New in version 1.5.0.

**Legendre**.deriv(m=1)

Differentiate.

Return an instance of Legendre that is the derivative of the current series. Refer to legder for full documentation.

**Parameters**

m : non-negative int

The number of integrations to perform.

**Returns**

derivative : Legendre

The derivative of the series using the same domain.

**See Also:**

legder

similar function.

legint

similar function for integration.

**static** Legendre.fit(x, y, deg, domain=None, rcond=None, full=False, w=None, window=[-1, 1])

Least squares fit to data.

Return a Legendre instance that is the least squares fit to the data y sampled at x. Unlike legfit, the domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning. Support for NA was added in version 1.7.0. See legfit for full documentation of the implementation.

**Parameters**

x : array_like, shape (M,)

x-coordinates of the M sample points ([x[i], y[i]].

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int

Degree of the fitting polynomial.

domain : [None, [beg, end], []], optional

Domain to use for the returned Legendre instance. If None, then a minimal domain that covers the points x is chosen. If [] the default domain [-1,1] is used. The default value is [-1,1] in numpy 1.4.x and None in later versions. The [] value was added in numpy 1.5.0.
**rcond** : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \( \text{len}(x) \times \text{eps} \), where \( \text{eps} \) is the relative precision of the float type, about \( 2 \times 10^{-16} \) in most cases.

**full** : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w** : array_like, shape (M,), optional

Weights. If None the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i] \times y[i]\) all have the same variance. The default value is None. .. versionadded:: 1.5.0

**window** : {[beg, end]}, optional

Window to use for the returned Legendre instance. The default value is \([-1, 1]\) .. versionadded:: 1.6.0

Returns

**least_squares_fit** : instance of Legendre

The Legendre instance is the least squares fit to the data and has the domain specified in the call.

**[residuals, rank, singular_values, rcond]** : only if **full** = True

Residuals of the least squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of **rcond**. For more details, see `linalg.lstsq`.

See Also:

**legfit**

similar function

**static** Legendre. fromroots *(roots, domain=[-1, 1], window=[-1, 1])*

Return Legendre instance with specified roots.

Returns an instance of Legendre representing the product \((x - r[0]) \times (x - r[1]) \times \ldots \times (x - r[n-1])\), where \( r \) is the list of roots.

Parameters

**roots** : array_like

List of roots.

**domain** : {array_like, None}, optional

Domain for the resulting instance of Legendre. If none the domain is the interval from the smallest root to the largest. The default is [-1,1].

**window** : array_like, optional

Window for the resulting instance of Legendre. The default value is [-1,1].

Returns

**object** : Legendre instance
Series with the specified roots.

See Also:

`legfromroots` equivalent function

Legendre has_samecoef(other)
Check if coefficients match.

Parameters

other : class instance

The other class must have the coef attribute.

Returns

bool : boolean

True if the coefficients are the same, False otherwise.

Notes

New in version 1.6.0.

Legendre has_samedomain(other)
Check if domains match.

Parameters

other : class instance

The other class must have the domain attribute.

Returns

bool : boolean

True if the domains are the same, False otherwise.

Notes

New in version 1.6.0.

Legendre has_sametype(other)
Check if types match.

Parameters

other : object

Class instance.

Returns

bool : boolean

True if other is same class as self

Notes

New in version 1.7.0.

Legendre has_samewindow(other)
Check if windows match.

Parameters

other : class instance

The other class must have the window attribute.
Returns

**bool**: boolean

True if the windows are the same, False otherwise.

**Notes**

New in version 1.6.0.

```python
static Legendre.identity( domain=[-1, 1], window=[-1, 1])
```

Identity function.

If `p` is the returned Legendre object, then `p(x) == x` for all values of `x`.

**Parameters**

- `domain`: array_like
  - The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain.
- `window`: array_like
  - The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the window.

**Returns**

- `identity`: Legendre instance

```python
Legendre.integ(m=1, k=[], lbnd=None)
```

Integrate.

Return an instance of Legendre that is the definite integral of the current series. Refer to `legint` for full documentation.

**Parameters**

- `m`: non-negative int
  - The number of integrations to perform.
- `k`: array_like
  - Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to `m` in length and any missing values are set to zero.
- `lbnd`: Scalar
  - The lower bound of the definite integral.

**Returns**

- `integral`: Legendre
  - The integral of the series using the same domain.

**See Also:**

- `legint`  similar function.
- `legder`  similar function for derivative.

```python
Legendre.linspace(n=100, domain=None)
```

Return x,y values at equally spaced points in domain.
Returns x, y values at n linearly spaced points across domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the Legendre instance. This method is intended mostly as a plotting aid.

**Parameters**

- **n**: int, optional
  - Number of point pairs to return. The default value is 100.

- **domain**: {None, array_like}
  - If not None, the specified domain is used instead of that of the calling instance. It should be of the form [beg, end]. The default is None.

**Returns**

- **x, y**: ndarrays
  - x is equal to linspace(self.domain[0], self.domain[1], n) y is the polynomial evaluated at x.

New in version 1.5.0.

**Legendre.mapparms()**

Return the mapping parameters.

The returned values define a linear map off + scl*x that is applied to the input arguments before the series is evaluated. The map depends on the domain and window; if the current domain is equal to the window the resulting map is the identity. If the coefficients of the Legendre instance are to be used by themselves outside this class, then the linear function must be substituted for the x in the standard representation of the base polynomials.

**Returns**

- **off, scl**: floats or complex
  - The mapping function is defined by off + scl*x.

**Notes**

If the current domain is the interval [l_1, r_1] and the window is [l_2, r_2], then the linear mapping function L is defined by the equations:

\[
L(l_1) = l_2 \\
L(r_1) = r_2
\]

**Legendre.roots()**

Return list of roots.

Return ndarray of roots for this series. See legroots for full documentation. Note that the accuracy of the roots is likely to decrease the further outside the domain they lie.

**See Also:**

- **legroots**
  - similar function

- **legfromroots**
  - function to go generate series from roots.

**Legendre.trim(tol=0)**

Remove small leading coefficients
Remove leading coefficients until a coefficient is reached whose absolute value greater than tol or the beginning of the series is reached. If all the coefficients would be removed the series is set to [0]. A new Legendre instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**
- **tol**: non-negative number.
  
  All trailing coefficients less than tol will be removed.

**Returns**
- **new_instance**: Legendre
  
  Contains the new set of coefficients.

```python
Legendre.truncate(size)
```

Truncate series to length size.

Reduce the Legendre series to length size by discarding the high degree terms. The value of size must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**
- **size**: positive int

  The series is reduced to length size by discarding the high degree terms. The value of size must be a positive integer.

**Returns**
- **new_instance**: Legendre

  New instance of Legendre with truncated coefficients.

---

**Basics**

```python
numpy.polynomial.legendre.legval(x, c[, tensor])
```

Evaluate a Legendre series at points x.

If c is of length n + 1, this function returns the value:

\[ p(x) = c_0 * L_0(x) + c_1 * L_1(x) + ... + c_n * L_n(x) \]

The parameter x is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either x or its elements must support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array, then p(x) will have the same shape as x. If c is multidimensional, then the shape of the result depends on the value of tensor. If tensor is true the shape will be c.shape[1:] + x.shape. If tensor is false the shape will be c.shape[1:]. Note that scalars have shape (1,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.
Parameters
  x : array_like, compatible object
    If x is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, x or its elements must support addition and multiplication with with themselves and with the elements of c.

c : array_like
    Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If c is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of c.

tensor : boolean, optional
    If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of x. Scalars have dimension 0 for this action. The result is that every column of coefficients in c is evaluated for every element of x. If False, x is broadcast over the columns of c for the evaluation. This keyword is useful when c is multidimensional. The default value is True. New in version 1.7.0.

Returns
  values : ndarray, algebra_like
    The shape of the return value is described above.

See Also:
  legval2d, leggrid2d, legval3d, leggrid3d

Notes
The evaluation uses Clenshaw recursion, aka synthetic division.

numpy.polynomial.legendre.legval2d(x, y, c)
Evaluate a 2-D Legendre series at points (x, y).
This function returns the values:

\[ p(x, y) = \sum_{i,j} c_{i,j} \times L_i(x) \times L_j(y) \]

The parameters x and y are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either x and y or their elements must support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

Parameters
  x, y : array_like, compatible objects
    The two dimensional series is evaluated at the points (x, y), where x and y must have the same shape. If x or y is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

c : array_like
    Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in c[i, j]. If c has dimension greater than two the remaining indices enumerate multiple sets of coefficients.
Returns

values : ndarray, compatible object

The values of the two dimensional Legendre series at points formed from pairs of corresponding values from x and y.

See Also:

`legval`, `leggrid2d`, `legval3d`, `leggrid3d`

Notes

numpy.polynomial.legendre.legval3d(x, y, z, c)

Evaluate a 3-D Legendre series at points (x, y, z).

This function returns the values:

\[ p(x, y, z) = \sum_{i,j,k} c_{i,j,k} \cdot L_i(x) \cdot L_j(y) \cdot L_k(z) \]

The parameters x, y, and z are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either x, y, and z or their elements must support multiplication and addition both with themselves and with the elements of c.

If c has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

Parameters

x, y, z : array_like, compatible object

The three dimensional series is evaluated at the points (x, y, z), where x, y, and z must have the same shape. If any of x, y, or z is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in c[i, j, k]. If c has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from x, y, and z.

See Also:

`legval`, `legval2d`, `leggrid2d`, `leggrid3d`

Notes

numpy.polynomial.legendre.leggrid2d(x, y, c)

Evaluate a 2-D Legendre series on the Cartesian product of x and y.

This function returns the values:

\[ p(a, b) = \sum_{i,j} c_{i,j} \cdot L_i(a) \cdot L_j(b) \]
where the points \((a, b)\) consist of all pairs formed by taking \(a\) from \(x\) and \(b\) from \(y\). The resulting points form a grid with \(x\) in the first dimension and \(y\) in the second.

The parameters \(x\) and \(y\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either \(x\) and \(y\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \(c.shape[2:] + x.shape + y.shape\).

**Parameters**

- \(x, y\) : array_like, compatible objects
  
The two dimensional series is evaluated at the points in the Cartesian product of \(x\) and \(y\). If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- \(c\) : array_like
  
  Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j\) is contained in \(c[i,j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

- \(values\) : ndarray, compatible object

  The values of the two dimensional Chebyshev series at points in the Cartesian product of \(x\) and \(y\).

**See Also:**

- `legval`, `legval2d`, `legval3d`, `leggrid3d`

**Notes**

`numpy.polynomial.legendre.leggrid3d(x, y, z, c)`

Evaluate a 3-D Legendre series on the Cartesian product of \(x\), \(y\), and \(z\).

This function returns the values:

\[
p(a, b, c) = \sum_{i,j,k} c_{i,j,k} L_i(a) L_j(b) L_k(c)
\]

where the points \((a, b, c)\) consist of all triples formed by taking \(a\) from \(x\), \(b\) from \(y\), and \(c\) from \(z\). The resulting points form a grid with \(x\) in the first dimension, \(y\) in the second, and \(z\) in the third.

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c.shape[3:] + x.shape + y.shape + z.shape\).

**Parameters**

- \(x, y, z\) : array_like, compatible objects

  The three dimensional series is evaluated at the points in the Cartesian product of \(x\), \(y\), and \(z\). If \(x\), \(y\), or \(z\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- \(c\) : array_like
Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in \( c[i, j] \). If \( c \) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**
- **values**: ndarray, compatible object
  - The values of the two dimensional polynomial at points in the Cartesian product of \( x \) and \( y \).

**See Also:**
- `legval`, `legval2d`, `leggrid2d`, `legval3d`

**Notes**

```python
def legroots(c):
    # Compute the roots of a Legendre series.
    # Return the roots (a.k.a. “zeros”) of the polynomial
    # \( p(x) = \sum_i c[i] \cdot L_i(x) \).
```

**Parameters**
- **c**: 1-D array_like
  - 1-D array of coefficients.

**Returns**
- **out**: ndarray
  - Array of the roots of the series. If all the roots are real, then \( out \) is also real, otherwise it is complex.

**See Also:**
- `polyroots`, `chebroots`, `lagroots`, `hermroots`, `hermeroots`

**Notes**

The root estimates are obtained as the eigenvalues of the companion matrix. Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s method.

The Legendre series basis polynomials aren’t powers of \( x \) so the results of this function may seem unintuitive.

**Examples**

```python
>>> import numpy.polynomial.legendre as leg
>>> leg.legroots((1, 2, 3, 4))
# 4L_3 + 3L_2 + 2L_1 + L_0 has only real roots
array([-0.85099543, -0.11407192, 0.51506735])
```

```python
def legfromroots(roots):
    # Generate a Legendre series with given roots.
    # The function returns the coefficients of the polynomial
    # \( p(x) = (x - r_0) \cdot (x - r_1) \cdot ... \cdot (x - r_n) \),
```

3.26. Polynomials
in Legendre form, where the \( r_n \) are the roots specified in \( \text{roots} \). If a zero has multiplicity \( n \), then it must appear in \( \text{roots} \) \( n \) times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then \( \text{roots} \) looks something like \([2, 2, 2, 3, 3]\). The roots can appear in any order.

If the returned coefficients are \( c \), then

\[
p(x) = c_0 + c_1 \cdot L_1(x) + ... + c_n \cdot L_n(x)
\]

The coefficient of the last term is not generally \( 1 \) for monic polynomials in Legendre form.

**Parameters**
- \( \text{roots} \): array_like
  Sequence containing the roots.

**Returns**
- \( \text{out} \): ndarray
  1-D array of coefficients. If all roots are real then \( \text{out} \) is a real array, if some of the roots are complex, then \( \text{out} \) is complex even if all the coefficients in the result are real (see Examples below).

**See Also:**
polyfromroots, chebfromroots, lagfromroots, hermfromroots, hermefromroots.

**Examples**

```python
>>> import numpy.polynomial.legendre as L

>>> L.legfromroots((-1,0,1)) # x^3 - x relative to the standard basis
array([ 0. , -0.4, 0. , 0.4])

>>> j = complex(0,1)
>>> L.legfromroots((-j,j)) # x^2 + 1 relative to the standard basis
array([ 1.33333333+0.j, 0.00000000+0.j, 0.66666667+0.j])
```

**Fitting**

numpy.polynomial.legendre.legfit \( x, y, \text{deg}[\text{rcond}, \text{full}, \text{w}] \) Least squares fit of Legendre series to data.

Return the coefficients of a Legendre series of degree \( \text{deg} \) that is the least squares fit to the data values \( y \) given at points \( x \). If \( y \) is 1-D the returned coefficients will also be 1-D. If \( y \) is 2-D multiple fits are done, one for each column of \( y \), and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

\[
p(x) = c_0 + c_1 \cdot L_1(x) + ... + c_n \cdot L_n(x),
\]

where \( n \) is \( \text{deg} \).

Since numpy version 1.7.0, legfit also supports NA. If any of the elements of \( x \), \( y \), or \( w \) are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If \( y \) is 2-D, then an NA in any row of \( y \) invalidates that whole row.
**Parameters**

- **x**: array_like, shape (M,)
  
  x-coordinates of the M sample points (x[i], y[i]).

- **y**: array_like, shape (M,) or (M, K)
  
  y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- **deg**: int
  
  Degree of the fitting polynomial

- **rcond**: float, optional
  
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

- **full**: bool, optional
  
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

- **w**: array_like, shape (M,), optional
  
  Weights. If not None, the contribution of each point (x[i], y[i]) to the fit is weighted by w[i]. Ideally the weights are chosen so that the errors of the products w[i]*y[i] all have the same variance. The default value is None. New in version 1.5.0.

**Returns**

- **coef**: ndarray, shape (M,) or (M, K)
  
  Legendre coefficients ordered from low to high. If y was 2-D, the coefficients for the data in column k of y are in column k.

- **[residuals, rank, singular_values, rcond]**: present when **full** = True
  
  Residuals of the least-squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of rcond. For more details, see linalg.lstsq.

**Warns**

- **RankWarning**
  
  The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if full = False. The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

**See Also:**

- chebfit, polyfit, lagfit, hermfit, hermefit
- legval
  
  Evaluates a Legendre series.
- legvander
  
  Vandermonde matrix of Legendre series.
**legweight**
Legendre weight function (= 1).

**linalg.lstsq**
Computes a least-squares fit from the matrix.

**scipy.interpolate.UnivariateSpline**
Computes spline fits.

**Notes**
The solution is the coefficients of the Legendre series \( p \) that minimizes the sum of the weighted squared errors

\[
E = \sum_j w_j^2 |y_j - p(x_j)|^2,
\]

where \( w_j \) are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

\[
V(x) \cdot c = w \cdot y,
\]

where \( V \) is the weighted pseudo Vandermonde matrix of \( x \), \( c \) are the coefficients to be solved for, \( w \) are the weights, and \( y \) are the observed values. This equation is then solved using the singular value decomposition of \( V \).

If some of the singular values of \( V \) are so small that they are neglected, then a \textit{RankWarning} will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The \textit{rcond} parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Legendre series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

**References**

[R62]

**numpy.polynomial.legendre.legvander**(\( x, \ deg \))

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree \( deg \) and sample points \( x \). The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, i] = L_i(x)
\]

where \( 0 <= i <= deg \). The leading indices of \( V \) index the elements of \( x \) and the last index is the degree of the Legendre polynomial.

If \( c \) is a 1-D array of coefficients of length \( n + 1 \) and \( V \) is the array \( V = \text{legvander}(x, n) \), then \( \text{np.dot}(V, c) \) and \( \text{legval}(x, c) \) are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Legendre series of the same degree and sample points.

**Parameters**

\textbf{x} : array_like

Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If \( x \) is scalar it is converted to a 1-D array.
deg : int

Degree of the resulting matrix.

Returns

vander : ndarray

The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the degree of the corresponding Legendre polynomial. The dtype will be the same as the converted x.

numpy.polynomial.legendre.legvander2d(x, y, deg)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees `deg` and sample points `(x, y)`. The pseudo-Vandermonde matrix is defined by

\[ V[\ldots, deg[1] * i + j] = L_i(x) * L_j(y), \]

where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of `V` index the points `(x, y)` and the last index encodes the degrees of the Legendre polynomials.

If `V = legvander2d(x, y, [xdeg, ydeg])`, then the columns of `V` correspond to the elements of a 2-D coefficient array `c` of shape `(xdeg + 1, ydeg + 1)` in the order

\[ c_{00}, c_{01}, c_{02}, \ldots, c_{10}, c_{11}, c_{12}, \ldots \]

and `np.dot(V, c.flat)` and `legval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Legendre series of the same degrees and sample points.

Parameters

x, y : array_like

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

deg : list of ints

List of maximum degrees of the form `[x_deg, y_deg].`

Returns

vander2d : ndarray

The shape of the returned matrix is `x.shape + (order,)`, where `order = (deg[0] + 1) * (deg[1] + 1)`. The dtype will be the same as the converted x and y.

See Also:

legvander, legvander3d, legval3d

Notes

numpy.polynomial.legendre.legvander3d(x, y, z, deg)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees `deg` and sample points `(x, y, z)`. If `l, m, n` are the given degrees in `x, y, z`, then The pseudo-Vandermonde matrix is defined by

\[ V[\ldots, (m + 1)(n + 1)i + (n + 1)j + k] = L_i(x) * L_j(y) * L_k(z), \]
where 0 <= i <= l, 0 <= j <= m, and 0 <= j <= n. The leading indices of V index the points (x, y, z) and the last index encodes the degrees of the Legendre polynomials.

If V = legvander3d(x, y, z, [xdeg, ydeg, zdeg]), then the columns of V correspond to the elements of a 3-D coefficient array c of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

\[ c_{000}, c_{001}, c_{002}, \ldots, c_{010}, c_{011}, c_{012}, \ldots \]

and np.dot(V, c.flat) and legval3d(x, y, z, c) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Legendre series of the same degrees and sample points.

### Parameters
- **x, y, z**: array_like
  - Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.
- **deg**: list of ints
  - List of maximum degrees of the form [x_deg, y_deg, z_deg].

### Returns
- **vander3d**: ndarray
  - The shape of the returned matrix is x.shape + (order,), where order = (deg[0] + 1) * (deg[1] + 1) * (deg[2] + 1). The dtype will be the same as the converted x, y, and z.

### See Also:
- legvander, legvander3d, legval3d

### Notes

- legder(c[, m, scl, axis])
  - Differentiate a Legendre series.
- legint(c[, m, k, lbnd, scl, axis])
  - Integrate a Legendre series.

### Calculus

numpy.polynomial.legendre.legder(c, m=1, scl=1, axis=0)

- Differentiate a Legendre series.

Returns the Legendre series coefficients c differentiated m times along axis. At each iteration the result is multiplied by scl (the scaling factor is for use in a linear change of variable). The argument c is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series \(1*\text{L}_0 + 2*\text{L}_1 + 3*\text{L}_2\) while \([[1,2],[1,2]]\) represents \(1*\text{L}_0(x)*\text{L}_0(y) + 1*\text{L}_1(x)*\text{L}_0(y) + 2*\text{L}_0(x)*\text{L}_1(y) + 2*\text{L}_1(x)*\text{L}_1(y)\) if axis=0 is x and axis=1 is y.

### Parameters
- **c**: array_like
  - Array of Legendre series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.
- **m**: int, optional
  - Number of derivatives taken, must be non-negative. (Default: 1)
scl : scalar, optional

Each differentiation is multiplied by scl. The end result is multiplication by \( scl^m \).
This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.

Returns
der : ndarray

Legendre series of the derivative.

See Also:
legint

Notes

In general, the result of differentiating a Legendre series does not resemble the same operation on a power series.
Thus the result of this function may be “unintuitive,” albeit correct; see Examples section below.

Examples

```python
>>> from numpy.polynomial import legendre as L
>>> c = (1,2,3,4)
>>> L.legder(c)
array([ 6., 9., 20.])
>>> L.legder(c, 3)
array([ 60.])
>>> L.legder(c, scl=-1)
array([-6., -9., -20.])
>>> L.legder(c, 2,-1)
array([ 9., 60.])
```

Integrate a Legendre series.

Returns the Legendre series coefficients c integrated m times from lbnd along axis. At each iteration the result-
ing series is multiplied by scl and an integration constant, k, is added. The scaling factor is for use in a
linear change of variable. (“Buyer beware”: note that, depending on what one is doing, one may want scl to
be the reciprocal of what one might expect; for more information, see the Notes section below.) The argu-
ment c is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series
\( L_0 + 2*L_1 + 3*L_2 \) while [[1,2],[1,2]] represents \( 1*L_0(x) * L_0(y) + 1*L_1(x) * L_0(y) + 2*L_0(x) * L_1(y) + 2*L_1(x) * L_1(y) \) if axis=0 is x and axis=1 is y.

Parameters
c : array_like

Array of Legendre series coefficients. If c is multidimensional the different axis cor-
respond to different variables with the degree in each axis given by the corresponding
index.

m : int, optional

Order of integration, must be positive. (Default: 1)

k : ([], list, scalar), optional

Integration constant(s). The value of the first integral at lbnd is the first value in the
list, the value of the second integral at lbnd is the second value, etc. If k == [] (the
default), all constants are set to zero. If m == 1, a single scalar can be given instead of
a list.
lbnd : scalar, optional

The lower bound of the integral. (Default: 0)

scl : scalar, optional

Following each integration the result is multiplied by scl before the integration constant is added. (Default: 1)

axis : int, optional

Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

Returns

S : ndarray

Legendre series coefficient array of the integral.

Raises

ValueError

If m < 0, len(k) > m, np.isscalar(lbnd) == False, or np.isscalar(scl) == False.

See Also:

legder

Notes

Note that the result of each integration is multiplied by scl. Why is this important to note? Say one is making a linear change of variable \( u = ax + b \) in an integral relative to \( x \). Then .. math:: dx = du/a, so one will need to set scl equal to \( 1/a \) - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be “reprojected” onto the C-series basis set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples

```python
>>> from numpy.polynomial import legendre as L
>>> c = (1, 2, 3)
>>> L.legint(c)
array([ 0.33333333, 0.66666667, 0.66666667, 0.66666667])
>>> L.legint(c, 3)
array([ 1.66666667e-02, -1.78571429e-02, 4.76190476e-02,
       -1.73472348e-18, 1.90476190e-02, 9.52380952e-03])
>>> L.legint(c, k=3)
array([ 3.33333333, 0.66666667, 0.66666667, 0.66666667])
>>> L.legint(c, lbnd=-2)
array([ 3.33333333, 0.66666667, 0.66666667, 0.66666667])
>>> L.legint(c, scl=2)
array([ 0.66666667, 1.33333333, 1.2, 1.2])
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>legadd(c1, c2)</td>
<td>Add one Legendre series to another.</td>
</tr>
<tr>
<td>legsub(c1, c2)</td>
<td>Subtract one Legendre series from another.</td>
</tr>
<tr>
<td>legmul(c1, c2)</td>
<td>Multiply one Legendre series by another.</td>
</tr>
<tr>
<td>legmulx(c)</td>
<td>Multiply a Legendre series by x.</td>
</tr>
<tr>
<td>legdiv(c1, c2)</td>
<td>Divide one Legendre series by another.</td>
</tr>
<tr>
<td>legpow(c, pow[, maxpower])</td>
<td>Raise a Legendre series to a power.</td>
</tr>
</tbody>
</table>
Algebra

```
numpy.polynomial.legendre.legadd(c1, c2)
```

Add one Legendre series to another.

Returns the sum of two Legendre series \( c_1 + c_2 \). The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( P_0 + 2*P_1 + 3*P_2 \).

- **Parameters**
  - `c1, c2`: array_like
    - 1-D arrays of Legendre series coefficients ordered from low to high.

- **Returns**
  - `out`: ndarray
    - Array representing the Legendre series of their sum.

See Also:

`legsub`, `legmul`, `legdiv`, `legpow`

Notes

Unlike multiplication, division, etc., the sum of two Legendre series is a Legendre series (without having to “reproject” the result onto the basis set) so addition, just like that of “standard” polynomials, is simply “component-wise.”

Examples

```python
>>> from numpy.polynomial import legendre as L
>>> c1 = (1, 2, 3)
>>> c2 = (3, 2, 1)
>>> L.legadd(c1, c2)
array([ 4., 4., 4.])
```

```
numpy.polynomial.legendre.legsub(c1, c2)
```

Subtract one Legendre series from another.

Returns the difference of two Legendre series \( c_1 - c_2 \). The sequences of coefficients are from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( P_0 + 2*P_1 + 3*P_2 \).

- **Parameters**
  - `c1, c2`: array_like
    - 1-D arrays of Legendre series coefficients ordered from low to high.

- **Returns**
  - `out`: ndarray
    - Of Legendre series coefficients representing their difference.

See Also:

`legadd`, `legmul`, `legdiv`, `legpow`

Notes

Unlike multiplication, division, etc., the difference of two Legendre series is a Legendre series (without having to “reproject” the result onto the basis set) so subtraction, just like that of “standard” polynomials, is simply “component-wise.”
Examples

```python
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legsub(c1,c2)
array([-2., 0., 2.])
>>> L.legsub(c2,c1)  # -C.legsub(c1,c2)
array([ 2., 0., -2.])
```

```
numpy.polynomial.legendre.legmul(c1, c2)
Multiply one Legendre series by another.

Returns the product of two Legendre series c1 * c2. The arguments are sequences of coefficients, from lowest
order “term” to highest, e.g., [1,2,3] represents the series P_0 + 2*P_1 + 3*P_2.

Parameters
   c1, c2 : array_like
      1-D arrays of Legendre series coefficients ordered from low to high.

Returns
   out : ndarray
      Array representing the result of the multiplication.

See Also:
   legadd, legsub, legdiv, legpow
```

```
numpy.polynomial.legendre.legmulx(c)
Multiply a Legendre series by x.

Multiply the Legendre series c by x, where x is the independent variable.

Parameters
   c : array_like
      1-D array of Legendre series coefficients ordered from low to high.

Returns
   out : ndarray
      Array representing the result of the multiplication.
```

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Legendre polynomial
basis set. Thus, to express the product as a Legendre series, it is necessary to “reproject” the product onto said
basis set, which may produce “unintuitive” (but correct) results; see Examples section below.

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2)
>>> P.legmul(c1,c2)  # multiplication requires "reprojection"
array([[ 4.33333333, 10.4 , 11.66666667, 3.6 ]])
```

```
numpy.polynomial.legendre.legmulx(c)
Multiply a Legendre series by x.

Multiply the Legendre series c by x, where x is the independent variable.

Parameters
   c : array_like
      1-D array of Legendre series coefficients ordered from low to high.

Returns
   out : ndarray
      Array representing the result of the multiplication.
```

Notes

The multiplication uses the recursion relationship for Legendre polynomials in the form

\[ xP_i(x) = ((i + 1) * P_{i+1}(x) + i * P_{i-1}(x))/(2i + 1) \]
numpy.polynomial.legendre.\texttt{legdiv}(c_1,c_2)
Divide one Legendre series by another.
Returns the quotient-with-remainder of two Legendre series $c_1/c_2$. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., $[1,2,3]$ represents the series $P_0 + 2*P_1 + 3*P_2$.

**Parameters**
- $c_1, c_2$ : array_like
  1-D arrays of Legendre series coefficients ordered from low to high.

**Returns**
- quo, rem : ndarrays
  Of Legendre series coefficients representing the quotient and remainder.

**See Also:**
- \texttt{legadd}, \texttt{legsub}, \texttt{legmul}, \texttt{legpow}

**Notes**
In general, the (polynomial) division of one Legendre series by another results in quotient and remainder terms that are not in the Legendre polynomial basis set. Thus, to express these results as a Legendre series, it is necessary to "reproject" the results onto the Legendre basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

**Examples**
```python
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legdiv(c1,c2) # quotient "intuitive," remainder not
(array([ 3.]), array([-8., -4.]))
>>> c2 = (0,1,2,3)
>>> L.legdiv(c2,c1) # neither "intuitive"
(array([-0.07407407, 1.66666667]), array([-1.03703704, -2.51851852]))
```

numpy.polynomial.legendre.\texttt{legpow}(c, pow, maxpower=16)
Raise a Legendre series to a power.
Returns the Legendre series $c$ raised to the power $pow$. The argument $c$ is a sequence of coefficients ordered from low to high. i.e., $[1,2,3]$ is the series $P_0 + 2*P_1 + 3*P_2$.

**Parameters**
- $c$ : array_like
  1-D array of Legendre series coefficients ordered from low to high.
- $pow$ : integer
  Power to which the series will be raised
- $maxpower$ : integer, optional
  Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**
- coef : ndarray
  Legendre series of power.
Quadrature

**NumPy Reference, Release 1.8.1**

See Also:

*legadd*, *legsub*, *legmul*, *legdiv*

---

**leggauss**(deg)  Gauss-Legendre quadrature.

**legweight**(x)  Weight function of the Legendre polynomials.

Quadrature

numpy.polynomial.legendre.**leggauss**(deg)

Gauss-Legendre quadrature.

Computes the sample points and weights for Gauss-Legendre quadrature. These sample points and weights will correctly integrate polynomials of degree $2 \times deg - 1$ or less over the interval $[-1, 1]$ with the weight function $f(x) = 1$.

**Parameters**

- **deg**: int
  - Number of sample points and weights. It must be $\geq 1$.

**Returns**

- **x**: ndarray
  - 1-D ndarray containing the sample points.
- **y**: ndarray
  - 1-D ndarray containing the weights.

**Notes**

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c / (L'_n(x_k) \times L_{n-1}(x_k))$$

where $c$ is a constant independent of $k$ and $x_k$ is the $k$’th root of $L_n$, and then scaling the results to get the right value when integrating 1.

numpy.polynomial.legendre.**legweight**(x)

Weight function of the Legendre polynomials.

The weight function is 1 and the interval of integration is $[-1, 1]$. The Legendre polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

- **x**: array_like
  - Values at which the weight function will be computed.

**Returns**

- **w**: ndarray
  - The weight function at $x$.

**Notes**

- **legcompanion**(c)  Return the scaled companion matrix of c.
### Table 3.145 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>legdomain</td>
<td></td>
</tr>
<tr>
<td>legzero</td>
<td></td>
</tr>
<tr>
<td>legone</td>
<td></td>
</tr>
<tr>
<td>legx</td>
<td></td>
</tr>
<tr>
<td>legtrim(c[, tol])</td>
<td>Remove “small” “trailing” coefficients from a polynomial.</td>
</tr>
<tr>
<td>legline(off, scl)</td>
<td>Legendre series whose graph is a straight line.</td>
</tr>
<tr>
<td>leg2poly(c)</td>
<td>Convert a Legendre series to a polynomial.</td>
</tr>
<tr>
<td>poly2leg(pol)</td>
<td>Convert a polynomial to a Legendre series.</td>
</tr>
</tbody>
</table>

### Miscellaneous

**numpy.polynomial.legendre.legcompanion(c)**

Return the scaled companion matrix of $c$.

The basis polynomials are scaled so that the companion matrix is symmetric when $c$ is an Legendre basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

**Parameters**

- $c$ : array_like
  
  1-D array of Legendre series coefficients ordered from low to high degree.

**Returns**

- mat : ndarray
  
  Scaled companion matrix of dimensions (deg, deg).

**Notes**

- `numpy.polynomial.legendre.legdomain = array([-1, 1])`

- `numpy.polynomial.legendre.legzero = array([0])`

- `numpy.polynomial.legendre.legone = array([1])`

- `numpy.polynomial.legendre.legx = array([0, 1])`

**numpy.polynomial.legendre.legtrim(c, tol=0)**

Remove “small” “trailing” coefficients from a polynomial.

“Small” means “small in absolute value” and is controlled by the parameter `tol`; “trailing” means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents $0 + x + x^2 + 0*x^3 + 0*x^4$) both the 3-rd and 4-th order coefficients would be “trimmed.”

**Parameters**

- $c$ : array_like
  
  1-d array of coefficients, ordered from lowest order to highest.

- tol : number, optional
  
  Trailing (i.e., highest order) elements with absolute value less than or equal to `tol` (default value is zero) are removed.

**Returns**

- trimmed : ndarray
1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

Raises

ValueError

If tol < 0

See Also:

trimseq

Examples

```python
>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([0., 0., 3., 0., 5.])
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1)  # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j, 0.0010-0.001j])
```

numpy.polynomial.legendre.legline(off, scl)

Legendre series whose graph is a straight line.

Parameters

off, scl : scalars

The specified line is given by off + scl*x.

Returns

y : ndarray

This module’s representation of the Legendre series for off + scl*x.

See Also:

polyline, chebline

Examples

```python
>>> import numpy.polynomial.legendre as L
>>> L.legline(3,2)
array([3, 2])
>>> L.legval(-3, L.legline(3,2))  # should be -3
-3.0
```

numpy.polynomial.legendre.leg2poly(c)

Convert a Legendre series to a polynomial.

Convert an array representing the coefficients of a Legendre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Legendre series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray
1-D array containing the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest order term to highest.

See Also:
poly2leg

Notes
The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples
```python
>>> c = P.Legendre(range(4))
>>> c
Legendre([ 0., 1., 2., 3.], [-1., 1.])
>>> p = c.convert(kind=P.Polynomial)
>>> p
Polynomial([-1. , -3.5, 3. , 7.5], [-1., 1.])
>>> P.leg2poly(range(4))
array([-1. , -3.5, 3. , 7.5])
```

```
numpy.polynomial.legendre.poly2leg(pol)
Convert a polynomial to a Legendre series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Legendre series, ordered from lowest to highest degree.

Parameters
pol : array_like

Returns
c : ndarray

1-D array containing the polynomial coefficients

See Also:
leg2poly

Notes
The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples
```python
>>> from numpy import polynomial as P
>>> p = P.Polynomial(np.arange(4))
>>> p
Polynomial([ 0., 1., 2., 3.], [-1., 1.])
>>> c = P.Legendre(P.poly2leg(p.coef))
>>> c
Legendre([ 1. , 3.25, 1. , 0.75], [-1., 1.])
```

Laguerre Module (numpy.polynomial.laguerre)
New in version 1.6.0. This module provides a number of objects (mostly functions) useful for dealing with Laguerre series, including a Laguerre class that encapsulates the usual arithmetic operations. (General information
on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).

```
Laguerre(coef[, domain, window])  A Laguerre series class.
```

### Laguerre Class

class numpy.polynomial.laguerre.Laguerre(coef, domain=[-1, 1], window=[-1, 1])

A Laguerre series class.

Laguerre instances provide the standard Python numerical methods ‘+’, ‘-’, ‘*’, ‘/’, ‘%’, ‘divmod’, ‘**’, and ‘()' as well as the listed methods.

**Parameters**

- `coef`: array_like
  Laguerre coefficients, in increasing order. For example, (1, 2, 3) implies \( P_0 + 2P_1 + 3P_2 \) where the \( P_i \) are a graded polynomial basis.

- `domain`: (2,) array_like, optional
  Domain to use. The interval \([domain[0], domain[1]]\) is mapped to the interval \([window[0], window[1]]\) by shifting and scaling. The default value is \([-1,1]\).

- `window`: (2,) array_like, optional
  Window that `domain` is mapped to. The default value is \([-1,1]\). .. versionadded:: 1.6.0

**Notes**

It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.

**Attributes**

- `coef`: ((N,) ndarray) Laguerre coefficients, from low to high.
- `domain`: ((2,) ndarray) Domain that is mapped to `window`.
- `window`: ((2,) ndarray) Window that `domain` is mapped to.

**Methods**

- `__call__(arg)`
- `basis(deg[, domain, window])`  Laguerre polynomial of degree `deg`.
- `cast(series[, domain, window])`  Convert instance to equivalent Laguerre series.
- `convert([domain, kind, window])`  Convert to different class and/or domain.
- `copy()`  Return a copy.
- `cutdeg(deg)`  Truncate series to the given degree.
- `degree()`  The degree of the series.
- `deriv([m])`  Differentiate.
- `fit(x, y, deg[, domain, rcond, full, w, window])`  Least squares fit to data.
- `fromroots(roots[, domain, window])`  Return Laguerre instance with specified roots.
- `has_samecoef(other)`  Check if coefficients match.
- `has_samedomain(other)`  Check if domains match.
- `has_sametype(other)`  Check if types match.
- `has_samewindow(other)`  Check if windows match.

Continued on next page
### Table 3.147 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity</td>
<td>Identity function.</td>
</tr>
<tr>
<td>integ</td>
<td>Integrate.</td>
</tr>
<tr>
<td>linspace</td>
<td>Return x,y values at equally spaced points in domain.</td>
</tr>
<tr>
<td>mapparms</td>
<td>Return the mapping parameters.</td>
</tr>
<tr>
<td>roots</td>
<td>Return list of roots.</td>
</tr>
<tr>
<td>trim</td>
<td>Remove small leading coefficients</td>
</tr>
<tr>
<td>truncate</td>
<td>Truncate series to length size.</td>
</tr>
</tbody>
</table>

**Laguerre**

**__call__**(arg)

**static** Laguerre.basis**(deg, domain=[-1, 1], window=[-1, 1])**

Laguerre polynomial of degree deg.

Returns an instance of the Laguerre polynomial of degree d.

**Parameters**

- **deg** : int
  
  Degree of the Laguerre polynomial. Must be >= 0.

- **domain** : array_like
  
  The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

- **window** : array_like
  
  The resulting array must be if the form [beg, end], where beg and end are the endpoints of the window.

**Returns**

- **p** : Laguerre instance

**Notes**

New in version 1.7.0.

**static** Laguerre.cast**(series, domain=[-1, 1], window=[-1, 1])**

Convert instance to equivalent Laguerre series.

The series is expected to be an instance of some polynomial series of one of the types supported by the numpy.polynomial module, but could be some other class that supports the convert method.

**Parameters**

- **series** : series
  
  The instance series to be converted.

- **domain** : array_like
  
  The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

- **window** : array_like
  
  The resulting array must be if the form [beg, end], where beg and end are the endpoints of the window.

**Returns**

- **p** : Laguerre instance
A Laguerre series equal to the *poly* series.

**See Also:**

*convert*

**Notes**

New in version 1.7.0.

*Laguerre*. *convert* (*domain=None, kind=None, window=None*)

Convert to different class and/or domain.

**Parameters**

- **domain**: array_like, optional
  The domain of the converted series. If the value is None, the default domain of *kind* is used.

- **kind**: class, optional
  The polynomial series type class to which the current instance should be converted. If *kind* is None, then the class of the current instance is used.

- **window**: array_like, optional
  The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

*new_series_instance*: *kind*

The returned class can be of different type than the current instance and/or have a different domain.

**Notes**

Conversion between domains and class types can result in numerically ill defined series.

*Laguerre*. *copy* ()

Return a copy.

**Returns**

*new_instance*: Laguerre

Copy of current instance.

*Laguerre*. *cutdeg* (*deg*)

Truncate series to the given degree.

Reduce the degree of the Laguerre series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

- **deg**: non-negative int
  The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

*new_instance*: Laguerre

New instance of Laguerre with reduced degree.
Notes

New in version 1.5.0.

Laguerre.\texttt{degree}()

The degree of the series.

Notes

New in version 1.5.0.

Laguerre.\texttt{deriv}(m=1)

Differentiate.

Return an instance of Laguerre that is the derivative of the current series. Refer to \texttt{lagder} for full documentation.

Parameters

\texttt{m} : non-negative int

The number of integrations to perform.

Returns

\texttt{derivative} : Laguerre

The derivative of the series using the same domain.

See Also:

\texttt{lagder}

similar function.

\texttt{lagint}

similar function for integration.

\texttt{static} Laguerre.\texttt{fit}(x, y, deg, domain=None, rcond=None, full=False, w=None, window=[-1, 1])

Least squares fit to data.

Return a \texttt{Laguerre} instance that is the least squares fit to the data \texttt{y} sampled at \texttt{x}. Unlike \texttt{lagfit}, the domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning. Support for NA was added in version 1.7.0. See \texttt{lagfit} for full documentation of the implementation.

Parameters

\texttt{x} : array_like, shape (M,)

\texttt{x}-coordinates of the M sample points \((x[i], y[i])\).

\texttt{y} : array_like, shape (M,) or (M, K)

\texttt{y}-coordinates of the sample points. Several data sets of sample points sharing the same \texttt{x}-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

\texttt{deg} : int

Degree of the fitting polynomial.

domain : \{None, [beg, end], []\}, optional

Domain to use for the returned Laguerre instance. If \texttt{None}, then a minimal domain that covers the points \texttt{x} is chosen. If [] the default domain \([-1, 1]\) is used. The default value is [-1,1] in numpy 1.4.x and \texttt{None} in later versions. The ’[]’ value was added in numpy 1.5.0.
rcond : float, optional

Relative condition number of the fit. Singular values smaller than this relative to
the largest singular value will be ignored. The default value is len(x)*eps, where eps is the
relative precision of the float type, about 2e-16 in most cases.

full : bool, optional

Switch determining nature of return value. When it is False (the default) just the coef-
ficients are returned, when True diagnostic information from the singular value decom-
position is also returned.

w : array_like, shape (M,), optional

Weights. If not None the contribution of each point \((x[i], y[i])\) to the fit is
weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products
\(w[i] \ast y[i]\) all have the same variance. The default value is None. .. versionadded::
1.5.0

window : \([\text{beg, end}]\), optional

Window to use for the returned Laguerre instance. The default value is \([-1, 1]\) ..
versionadded: 1.6.0

Returns

least_squares_fit : instance of Laguerre

The Laguerre instance is the least squares fit to the data and has the domain specified in
the call.

[residuals, rank, singular_values, rcond] : only if full = True

Residuals of the least squares fit, the effective rank of the scaled Vandermonde ma-
trix and its singular values, and the specified value of rcond. For more details, see
linalg.lstsq.

See Also:

lagfit

similar function

static Laguerre.fromroots (roots, domain=[-1, 1], window=[-1, 1])

Return Laguerre instance with specified roots.

Returns an instance of Laguerre representing the product \((x - r[0]) \ast (x - r[1]) \ast \ldots \ast (x -
r[n-1])\), where \(r\) is the list of roots.

Parameters

roots : array_like

List of roots.

domain : \{array_like, None\}, optional

Domain for the resulting instance of Laguerre. If none the domain is the interval from
the smallest root to the largest. The default is [-1,1].

window : array_like, optional

Window for the resulting instance of Laguerre. The default value is [-1,1].

Returns

object : Laguerre instance
Series with the specified roots.

See Also:

**lagfromroots**
Equivalent function

Laguerre.\texttt{has\_samecoef}(other)
Check if coefficients match.

**Parameters**
- \texttt{other} : class instance
  
The other class must have the \texttt{coef} attribute.

**Returns**
- \texttt{bool} : boolean
  
  True if the coefficients are the same, False otherwise.

**Notes**
New in version 1.6.0.

Laguerre.\texttt{has\_samedomain}(other)
Check if domains match.

**Parameters**
- \texttt{other} : class instance
  
The other class must have the \texttt{domain} attribute.

**Returns**
- \texttt{bool} : boolean
  
  True if the domains are the same, False otherwise.

**Notes**
New in version 1.6.0.

Laguerre.\texttt{has\_sametype}(other)
Check if types match.

**Parameters**
- \texttt{other} : object
  
  Class instance.

**Returns**
- \texttt{bool} : boolean
  
  True if other is same class as self

**Notes**
New in version 1.7.0.

Laguerre.\texttt{has\_samewindow}(other)
Check if windows match.

**Parameters**
- \texttt{other} : class instance
  
The other class must have the \texttt{window} attribute.
Returns

bool : boolean

True if the windows are the same, False otherwise.

Notes

New in version 1.6.0.

static Laguerre.identity (domain=[-1, 1], window=[-1, 1])

Identity function.

If p is the returned Laguerre object, then p(x) == x for all values of x.

Parameters

domain : array_like

The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

window : array_like

The resulting array must be of the form [beg, end], where beg and end are the endpoints of the window.

Returns

identity : Laguerre instance

Laguerre.integ (m=1, k=[], lbnd=None)

Integrate.

Return an instance of Laguerre that is the definite integral of the current series. Refer to lagint for full documentation.

Parameters

m : non-negative int

The number of integrations to perform.

k : array_like

Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to m in length and any missing values are set to zero.

lbnd : Scalar

The lower bound of the definite integral.

Returns

integral : Laguerre

The integral of the series using the same domain.

See Also:

lagint similar function.

lagder similar function for derivative.

Laguerre.linspace (n=100, domain=None)

Return x,y values at equally spaced points in domain.
Returns \( x, y \) values at \( n \) linearly spaced points across domain. Here \( y \) is the value of the polynomial at the points \( x \). By default the domain is the same as that of the Laguerre instance. This method is intended mostly as a plotting aid.

**Parameters**

- \( n \): int, optional
  
  Number of point pairs to return. The default value is 100.

- \( \text{domain} \): \{None, array_like\}
  
  If not None, the specified domain is used instead of that of the calling instance. It should be of the form \([\text{beg}, \text{end}]\). The default is None.

**Returns**

- \( x, y \): ndarrays
  
  \( x \) is equal to linspace(self.domain[0], self.domain[1], n) \( y \) is the polynomial evaluated at \( x \).

New in version 1.5.0.

**Laguerre.mapparms()**

Return the mapping parameters.

The returned values define a linear map \( \text{off} + \text{scl} \times x \) that is applied to the input arguments before the series is evaluated. The map depends on the domain and window; if the current domain is equal to the window the resulting map is the identity. If the coefficients of the Laguerre instance are to be used by themselves outside this class, then the linear function must be substituted for the \( x \) in the standard representation of the base polynomials.

**Returns**

- \( \text{off}, \text{scl} \): floats or complex
  
  The mapping function is defined by \( \text{off} + \text{scl} \times x \).

**Notes**

If the current domain is the interval \([l_1, r_1]\) and the window is \([l_2, r_2]\), then the linear mapping function \( L \) is defined by the equations:

\[
L(l_1) = l_2
\]
\[
L(r_1) = r_2
\]

**Laguerre.roots()**

Return list of roots.

Return ndarray of roots for this series. See lagroots for full documentation. Note that the accuracy of the roots is likely to decrease the further outside the domain they lie.

**See Also:**

- lagroots
  
  similar function

- lagfromroots
  
  function to go generate series from roots.

**Laguerre.trim(tol=0)**

Remove small leading coefficients
Remove leading coefficients until a coefficient is reached whose absolute value greater than \( tol \) or the beginning of the series is reached. If all the coefficients would be removed the series is set to \([0]\). A new Laguerre instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

- **tol**: non-negative number.
  
  All trailing coefficients less than \( tol \) will be removed.

**Returns**

- **new_instance**: Laguerre
  
  Contains the new set of coefficients.

```
Laguerre.truncate(size)
```

Truncate series to length \( size \).

Reduce the Laguerre series to length \( size \) by discarding the high degree terms. The value of \( size \) must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

- **size**: positive int
  
  The series is reduced to length \( size \) by discarding the high degree terms. The value of \( size \) must be a positive integer.

**Returns**

- **new_instance**: Laguerre
  
  New instance of Laguerre with truncated coefficients.

```
lagval(x, c[, tensor])
```

Evaluate a Laguerre series at points \( x \).

If \( c \) is of length \( n + 1 \), this function returns the value:

\[
p(x) = c_0 \cdot L_0(x) + c_1 \cdot L_1(x) + \ldots + c_n \cdot L_n(x)
\]

The parameter \( x \) is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either \( x \) or its elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) is a 1-D array, then \( p(x) \) will have the same shape as \( x \). If \( c \) is multidimensional, then the shape of the result depends on the value of \( tensor \). If \( tensor \) is true the shape will be \( c.shape[1:] + x.shape \). If \( tensor \) is false the shape will be \( c.shape[1:] \). Note that scalars have shape \((,)\).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.
Parameters

\(x\) : array_like, compatible object

If \(x\) is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, \(x\) or its elements must support addition and multiplication with with themselves and with the elements of \(c\).

\(c\) : array_like

Array of coefficients ordered so that the coefficients for terms of degree \(n\) are contained in \(c[n]\). If \(c\) is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of \(c\).

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of \(x\). Scalars have dimension 0 for this action. The result is that every column of coefficients in \(c\) is evaluated for every element of \(x\). If False, \(x\) is broadcast over the columns of \(c\) for the evaluation. This keyword is useful when \(c\) is multidimensional. The default value is True. New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See Also:

\texttt{lagval2d, laggrid2d, lagval3d, laggrid3d}

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

Examples

```python
>>> from numpy.polynomial.laguerre import lagval
>>> coef = [1,2,3]
>>> lagval(1, coef)
-0.5
>>> lagval([[1,2],[3,4]], coef)
array([[-0.5, -4.],
       [-4.5, -2.]])
```

**numpy.polynomial.laguerre.\texttt{lagval2d}(x, y, c)**

Evaluate a 2-D Laguerre series at points \((x, y)\).

This function returns the values:

\[
p(x, y) = \sum_{i,j} c_{i,j} \cdot L_i(x) \cdot L_j(y)
\]

The parameters \(x\) and \(y\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either \(x\) and \(y\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) is a 1-D array one is implicitly appended to its shape to make it 2-D. The shape of the result will be \(c\).shape[2:] + \(x\).shape.
The two dimensional series is evaluated at the points \((x, y)\), where \(x\) and \(y\) must have the same shape. If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

\[c : \text{array_like}\]

Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j\) is contained in \(c[i, j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

\[values : \text{ndarray, compatible object}\]

The values of the two dimensional polynomial at points formed with pairs of corresponding values from \(x\) and \(y\).

See Also:

\[\text{lagval, laggrid2d, lagval3d, laggrid3d}\]

Notes

\[\text{numpy.polynomial.laguerre.lagval3d}(x, y, z, c)\]

Evaluate a 3-D Laguerre series at points \((x, y, z)\).

This function returns the values:

\[p(x, y, z) = \sum_{i,j,k} c_{i,j,k} \cdot L_i(x) \cdot L_j(y) \cdot L_k(z)\]

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars and they must have the same shape after conversion. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c.shape[3:] + x.shape\).

Parameters

\[x, y, z : \text{array_like, compatible object}\]

The three dimensional series is evaluated at the points \((x, y, z)\), where \(x\), \(y\), and \(z\) must have the same shape. If any of \(x\), \(y\), or \(z\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

\[c : \text{array_like}\]

Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j,k\) is contained in \(c[i, j, k]\). If \(c\) has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns

\[values : \text{ndarray, compatible object}\]

The values of the multidimension polynomial on points formed with triples of corresponding values from \(x\), \(y\), and \(z\).

See Also:

\[\text{lagval, lagval2d, laggrid2d, laggrid3d}\]
numpy.polynomial.laguerre.laggrid2d(x, y, c)
Evaluate a 2-D Laguerre series on the Cartesian product of x and y.
This function returns the values:
\[ p(a, b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b) \]
where the points \((a, b)\) consist of all pairs formed by taking \(a\) from \(x\) and \(b\) from \(y\). The resulting points form a grid with \(x\) in the first dimension and \(y\) in the second.

The parameters \(x\) and \(y\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \(x\) and \(y\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).
If \(c\) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \(c\).shape[2:] + \(x\).shape + \(y\).shape.

Parameters
\(x, y, c: \text{array}_\text{like}, \text{compatible objects}\)
The two dimensional series is evaluated at the points in the Cartesian product of \(x\) and \(y\). If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.
\(c: \text{array}_\text{like}\)
Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j\) is contained in \(c[i,j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns
\(\text{values: \text{ndarray}, \text{compatible object}}\)
The values of the two dimensional Chebyshev series at points in the Cartesian product of \(x\) and \(y\).

See Also:
lagval, lagval2d, lagval3d, laggrid3d

numpy.polynomial.laguerre.laggrid3d(x, y, z, c)
Evaluate a 3-D Laguerre series on the Cartesian product of \(x\), \(y\), and \(z\).
This function returns the values:
\[ p(a, b, c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c) \]
where the points \((a, b, c)\) consist of all triples formed by taking \(a\) from \(x\), \(b\) from \(y\), and \(c\) from \(z\). The resulting points form a grid with \(x\) in the first dimension, \(y\) in the second, and \(z\) in the third.

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).
If \(c\) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c\).shape[3:] + \(x\).shape + \(y\).shape + \(z\).shape.
**Parameters**

- `x, y, z`: array_like, compatible objects
  
  The three dimensional series is evaluated at the points in the Cartesian product of `x, y,` and `z`. If `x, y,` or `z` is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- `c`: array_like
  
  Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i, j]`. If `c` has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

- `values`: ndarray, compatible object
  
  The values of the two dimensional polynomial at points in the Cartesian product of `x` and `y`.

**See Also:**

- `lagval`, `lagval2d`, `laggrid2d`, `lagval3d`

**Notes**

- `numpy.polynomial.laguerre.lagroots(c)`

  Compute the roots of a Laguerre series.

  Return the roots (a.k.a. “zeros”) of the polynomial

  \[ p(x) = \sum_i c[i] \ast L_i(x). \]

**Parameters**

- `c`: 1-D array_like
  
  1-D array of coefficients.

**Returns**

- `out`: ndarray
  
  Array of the roots of the series. If all the roots are real, then `out` is also real, otherwise it is complex.

**See Also:**

- `polyroots`, `legroots`, `chebroots`, `hermroots`, `hermeroots`

**Notes**

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s method.

The Laguerre series basis polynomials aren’t powers of `x` so the results of this function may seem unintuitive.

**Examples**
>>> from numpy.polynomial.laguerre import lagroots, lagfromroots
>>> coef = lagfromroots([0, 1, 2])
>>> coef
array([ 2., -8., 12., -6.])
>>> lagroots(coef)
array([-4.44089210e-16, 1.00000000e+00, 2.00000000e+00])

numpy.polynomial.laguerre.lagfromroots(roots)
Generate a Laguerre series with given roots.

The function returns the coefficients of the polynomial

\[ p(x) = (x - r_0) \cdot (x - r_1) \cdot \ldots \cdot (x - r_n), \]

in Laguerre form, where the \( r_n \) are the roots specified in \( roots \). If a zero has multiplicity \( n \), then it must appear in \( roots \) \( n \) times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then \( roots \) looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are \( c \), then

\[ p(x) = c_0 + c_1 \cdot L_1(x) + \ldots + c_n \cdot L_n(x) \]

The coefficient of the last term is not generally 1 for monic polynomials in Laguerre form.

Parameters

- **roots**: array_like
  Sequence containing the roots.

Returns

- **out**: ndarray
  1-D array of coefficients. If all roots are real then \( out \) is a real array, if some of the roots are complex, then \( out \) is complex even if all the coefficients in the result are real (see Examples below).

See Also:

- polyfromroots, legfromroots, chebfromroots, hermfromroots, hermefromroots.

Examples

```python
>>> from numpy.polynomial.laguerre import lagfromroots, lagval
>>> coef = lagfromroots((-1, 0, 1))
>>> lagval((-1, 0, 1), coef)
array([ 0., 0., 0.])
>>> coef = lagfromroots((-1j, 1j))
>>> lagval((-1j, 1j), coef)
array([ 0.+0.j, 0.+0.j])
```
Fitting

\texttt{numpy.polynomial.laguerre.lagfit}(x, y, \texttt{deg}, \texttt{rcond=\textit{None}}, \texttt{full=\textit{False}}, \texttt{w=\textit{None}})

Least squares fit of Laguerre series to data.

Return the coefficients of a Laguerre series of degree \textit{deg} that is the least squares fit to the data values \textit{y} given at points \textit{x}. If \textit{y} is 1-D the returned coefficients will also be 1-D. If \textit{y} is 2-D multiple fits are done, one for each column of \textit{y}, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

\[ p(x) = c_0 + c_1 \cdot L_1(x) + \ldots + c_n \cdot L_n(x), \]

where \( n \) is \textit{deg}.

Since numpy version 1.7.0, \texttt{lagfit} also supports NA. If any of the elements of \textit{x}, \textit{y}, or \textit{w} are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If \textit{y} is 2-D, then an NA in any row of \textit{y} invalidates that whole row.

**Parameters**

- \texttt{x} : array_like, shape (M,)
  
x-coordinates of the M sample points \((x[i], y[i])\).

- \texttt{y} : array_like, shape (M,) or (M, K)
  
y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- \texttt{deg} : int
  
  Degree of the fitting polynomial

- \texttt{rcond} : float, optional
  
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \textit{len(x)}*\textit{eps}, where \textit{eps} is the relative precision of the float type, about 2e-16 in most cases.

- \texttt{full} : bool, optional
  
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

- \texttt{w} : array_like, shape (M,), optional
  
  Weights. If not None, the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i] \cdot y[i]\) all have the same variance. The default value is None.

**Returns**

- \texttt{coef} : ndarray, shape (M,) or (M, K)
  
  Laguerre coefficients ordered from low to high. If \textit{y} was 2-D, the coefficients for the data in column \textit{k} of \textit{y} are in column \textit{k}.

- \texttt{[residuals, rank, singular_values, rcond]} : present when \texttt{full = True}

  Residuals of the least-squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of \textit{rcond}. For more details, see \texttt{linalg.lstsq}.
**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if `full = False`. The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

**See Also:**

- chebfit, legfit, polyfit, hermfit, hermefit
- `lagval`
  - Evaluates a Laguerre series.
- `lagvander`
  - pseudo Vandermonde matrix of Laguerre series.
- `lagweight`
  - Laguerre weight function.
- `linalg.lstsq`
  - Computes a least-squares fit from the matrix.
- `scipy.interpolate.UnivariateSpline`
  - Computes spline fits.

**Notes**

The solution is the coefficients of the Laguerre series $p$ that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 \cdot |y_j - p(x_j)|^2,$$

where the $w_j$ are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) \cdot c = w \cdot y,$$

where $V$ is the weighted pseudo Vandermonde matrix of $x$, $c$ are the coefficients to be solved for, $w$ are the weights, and $y$ are the observed values. This equation is then solved using the singular value decomposition of $V$.

If some of the singular values of $V$ are so small that they are neglected, then a `RankWarning` will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The `rcond` parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Laguerre series are probably most useful when the data can be approximated by $\sqrt{w(x)} \cdot p(x)$, where $w(x)$ is the Laguerre weight. In that case the weight $\sqrt{w(x[i])}$ should be used together with data values $y[i]/\sqrt{w(x[i])}$. The weight function is available as `lagweight`.

**References**

[R61]
Examples

```python
>>> from numpy.polynomial.laguerre import lagfit, lagval
>>> x = np.linspace(0, 10)
>>> err = np.random.randn(len(x))/10
>>> y = lagval(x, [1, 2, 3]) + err
>>> lagfit(x, y, 2)
array([ 0.96971004, 2.00193749, 3.00288744])
```

**numpy.polynomial.laguerre.lagvander**

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree `deg` and sample points `x`. The pseudo-Vandermonde matrix is defined by

\[ V[..., i] = L_i(x) \]

where 0 <= i <= deg. The leading indices of V index the elements of `x` and the last index is the degree of the Laguerre polynomial.

If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the array `V = lagvander(x, n)`, then `np.dot(V, c)` and `lagval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Laguerre series of the same degree and sample points.

**Parameters**

- `x`: array_like
  - Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If `x` is scalar it is converted to a 1-D array.

- `deg`: int
  - Degree of the resulting matrix.

**Returns**

- `vander`: ndarray
  - The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the degree of the corresponding Laguerre polynomial. The dtype will be the same as the converted `x`.

Examples

```python
>>> from numpy.polynomial.laguerre import lagvander
>>> x = np.array([0, 1, 2])
>>> lagvander(x, 3)
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  0., -0.5, -0.66666667],
       [ 1., -1., -1., -0.33333333]])
```

**numpy.polynomial.laguerre.lagvander2d**

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees `deg` and sample points `(x, y)`. The pseudo-Vandermonde matrix is defined by

\[ V[... , deg[1] * i + j] = L_i(x) * L_j(y), \]

where 0 <= i <= `deg[0]` and 0 <= j <= `deg[1]`. The leading indices of V index the points `(x, y)` and the last index encodes the degrees of the Laguerre polynomials.
If $V = \text{lagvander2d}(x, y, [\text{xdeg}, \text{ydeg}])$, then the columns of $V$ correspond to the elements of a 2-D coefficient array $c$ of shape $(\text{xdeg} + 1, \text{ydeg} + 1)$ in the order

$$c_{00}, c_{01}, c_{02}, ..., c_{10}, c_{11}, c_{12}, ...$$

and $\text{np.dot}(V, c.\text{flat})$ and $\text{lagval2d}(x, y, c)$ will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Laguerre series of the same degrees and sample points.

**Parameters**

- $x, y$: array_like
  - Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex.
  - Scalars are converted to 1-D arrays.

- $\text{deg}$: list of ints
  - List of maximum degrees of the form $[\text{x_deg}, \text{y_deg}]$.

**Returns**

- $\text{lagvander2d}$: ndarray
  - The shape of the returned matrix is $x.\text{shape} + (\text{order},)$, where $\text{order} = (\text{deg}[0] + 1) \times (\text{deg}[1] + 1)$. The dtype will be the same as the converted $x$ and $y$.

**See Also:**

- $\text{lagvander}$, $\text{lagvander3d}$, $\text{lagval}$

**Notes**

$\text{numpy.polynomial.laguerre.\text{lagvander3d}}(x, y, z, \text{deg})$

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees $\text{deg}$ and sample points $(x, y, z)$. If $l, m, n$ are the given degrees in $x, y, z$, then The pseudo-Vandermonde matrix is defined by

$$V[...,(m + 1)(n + 1)i + (n + 1)j + k] = L_i(x) \times L_j(y) \times L_k(z),$$

where $0 <= i <= l$, $0 <= j <= m$, and $0 <= j <= n$. The leading indices of $V$ index the points $(x, y, z)$ and the last index encodes the degrees of the Laguerre polynomials.

If $V = \text{lagvander3d}(x, y, z, [\text{xdeg}, \text{ydeg}, \text{zdeg}])$, then the columns of $V$ correspond to the elements of a 3-D coefficient array $c$ of shape $(\text{xdeg} + 1, \text{ydeg} + 1, \text{zdeg} + 1)$ in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and $\text{np.dot}(V, c.\text{flat})$ and $\text{lagval3d}(x, y, z, c)$ will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Laguerre series of the same degrees and sample points.

**Parameters**

- $x, y, z$: array_like
  - Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex.
  - Scalars are converted to 1-D arrays.

3.26. Polynomials 1053
deg : list of ints
    List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns
vander3d : ndarray
    The shape of the returned matrix is x.shape + (order,), where order =
    (deg[0] + 1) * (deg[1] + 1) * (deg[2] + 1). The dtype will be the same as the con-
    verted x, y, and z.

See Also:
lagvander, lagvander3d., lagval3d

Notes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lagder</td>
<td>Differentiate a Laguerre series.</td>
</tr>
<tr>
<td>lagint</td>
<td>Integrate a Laguerre series.</td>
</tr>
</tbody>
</table>

Calculus

numpy.polynomial.laguerre.**lagder**(c[, m=1, scl=1, axis=0)
    Differentiate a Laguerre series.

Returns the Laguerre series coefficients c differentiated m times along
axis. At each iteration the result is multi-
plied by scl (the scaling factor is for use in a linear change of variable). The argument c is an array of coeffi-
cients from low to high degree along each axis, e.g., [1,2,3] represents the series 1*L_0 + 2*L_1 + 3*L_2
while [[1,2],[1,2]] represents 1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) +
2*L_1(x)*L_1(y) if axis=0 is x and axis=1 is y.

Parameters
c : array_like
    Array of Laguerre series coefficients. If c is multidimensional the different axis cor-
    respond to different variables with the degree in each axis given by the corresponding
    index.

m : int, optional
    Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional
    Each differentiation is multiplied by scl. The end result is multiplication by scl**m.
    This is for use in a linear change of variable. (Default: 1)

axis : int, optional
    Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.

Returns
der : ndarray
    Laguerre series of the derivative.

See Also:
lagint

Notes
In general, the result of differentiating a Laguerre series does not resemble the same operation on a power series.
Thus the result of this function may be “unintuitive,” albeit correct; see Examples section below.
Examples

```python
>>> from numpy.polynomial.laguerre import lagder
>>> lagder([ 1., 1., 1., -3.])
array([ 1., 2., 3.])
>>> lagder([ 1., 0., 0., -4., 3.], m=2)
array([ 1., 2., 3.])
```

`numpy.polynomial.laguerre.lagint(c, m=1, k=[], lbnd=0, scl=1, axis=0)`

Integrate a Laguerre series.

Returns the Laguerre series coefficients `c` integrated `m` times from `lbnd` along `axis`. At each iteration the resulting series is multiplied by `scl` and an integration constant, `k`, is added. The scaling factor is for use in a linear change of variable. (“Buyer beware”: note that, depending on what one is doing, one may want `scl` to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument `c` is an array of coefficients from low to high degree along each axis, e.g., `[1,2,3]` represents the series \( L_0 + 2L_1 + 3L_2 \) while `[[1,2],[1,2]]` represents \( 1L_0(x) * L_0(y) + 1L_1(x) * L_0(y) + 2L_0(x) * L_1(y) + 2L_1(x) * L_1(y) \) if `axis=0` is `x` and `axis=1` is `y`.

Parameters

- `c`: array_like
  Array of Laguerre series coefficients. If `c` is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.
- `m`: int, optional
  Order of integration, must be positive. (Default: 1)
- `k`: [ ], list, scalar, optional
  Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == [ ]` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.
- `lbnd`: scalar, optional
  The lower bound of the integral. (Default: 0)
- `scl`: scalar, optional
  Following each integration the result is multiplied by `scl` before the integration constant is added. (Default: 1)
- `axis`: int, optional
  Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

Returns

- `S`: ndarray
  Laguerre series coefficients of the integral.

Raises

- `ValueError`
  If \( m < 0 \), \( \text{len}(k) > m \), `np.isscalar(lbnd) == False`, or `np.isscalar(scl) == False`.

See Also:

- `lagder`
Notes

Note that the result of each integration is multiplied by \( scl \). Why is this important to note? Say one is making a linear change of variable \( u = ax + b \) in an integral relative to \( x \). Then \( dx = du/a \), so one will need to set \( scl \) equal to \( 1/a \) - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be “reprojected” onto the C-series basis set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples

```python
g1 = [1, 2, 3]
g2 = [1, 2, 3]
g3 = [1, 2]
g1[0] + 2*g1[1] + 3*g1[2]
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lagadd(c1, c2)</code></td>
<td>Add one Laguerre series to another.</td>
</tr>
<tr>
<td><code>lagsub(c1, c2)</code></td>
<td>Subtract one Laguerre series from another.</td>
</tr>
<tr>
<td><code>lagmul(c1, c2)</code></td>
<td>Multiply one Laguerre series by another.</td>
</tr>
<tr>
<td><code>lagmulx(c)</code></td>
<td>Multiply a Laguerre series by ( x ).</td>
</tr>
<tr>
<td><code>lagdiv(c1, c2)</code></td>
<td>Divide one Laguerre series by another.</td>
</tr>
<tr>
<td><code>lagpow(c, pow[, maxpower])</code></td>
<td>Raise a Laguerre series to a power.</td>
</tr>
</tbody>
</table>

Algebra

```
numpy.polynomial.laguerre
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lagadd(c1, c2)</code></td>
<td>Add one Laguerre series to another.</td>
</tr>
</tbody>
</table>

Returns the sum of two Laguerre series \( c1 + c2 \). The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( P_0 + 2P_1 + 3P_2 \).

Parameters

- \( c1, c2 \) : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

- \( out \) : ndarray

Array representing the Laguerre series of their sum.

See Also:

`lagsub, lagmul, lagdiv, lagpow`

Notes

Unlike multiplication, division, etc., the sum of two Laguerre series is a Laguerre series (without having to “reproject” the result onto the basis set) so addition, just like that of “standard” polynomials, is simply “component-wise.”
Examples

```python
>>> from numpy.polynomial.laguerre import lagadd
>>> lagadd([1, 2, 3], [1, 2, 3, 4])
array([ 2., 4., 6., 4.])
```

`numpy.polynomial.laguerre.lagsub(cl, c2)`
Subtract one Laguerre series from another.

Returns the difference of two Laguerre series $c1 - c2$. The sequences of coefficients are from lowest order term to highest, i.e., $[1,2,3]$ represents the series $P_0 + 2*P_1 + 3*P_2$.

Parameters
- `c1`, `c2`: array_like
  1-D arrays of Laguerre series coefficients ordered from low to high.

Returns
- `out`: ndarray
  Of Laguerre series coefficients representing their difference.

See Also:
- `lagadd`, `lagmul`, `lagdiv`, `lagpow`

Notes
Unlike multiplication, division, etc., the difference of two Laguerre series is a Laguerre series (without having to “reproject” the result onto the basis set) so subtraction, just like that of “standard” polynomials, is simply “component-wise.”

Examples

```python
>>> from numpy.polynomial.laguerre import lagsub
>>> lagsub([1, 2, 3, 4], [1, 2, 3])
array([0., 0., 0., 4.])
```

`numpy.polynomial.laguerre.lagmul(cl, c2)`
Multiply one Laguerre series by another.

Returns the product of two Laguerre series $c1 * c2$. The arguments are sequences of coefficients, from lowest order “term” to highest, e.g., $[1,2,3]$ represents the series $P_0 + 2*P_1 + 3*P_2$.

Parameters
- `c1`, `c2`: array_like
  1-D arrays of Laguerre series coefficients ordered from low to high.

Returns
- `out`: ndarray
  Of Laguerre series coefficients representing their product.

See Also:
- `lagadd`, `lagsub`, `lagdiv`, `lagpow`

Notes
In general, the (polynomial) product of two C-series results in terms that are not in the Laguerre polynomial basis set. Thus, to express the product as a Laguerre series, it is necessary to “reproject” the product onto said basis set, which may produce “unintuitive” (but correct) results; see Examples section below.
Examples

```python
>>> from numpy.polynomial.laguerre import lagmul
>>> lagmul([1, 2, 3], [0, 1, 2])
array([ 8., -13., 38., -51., 36.])
```

```
numpy.polynomial.laguerre.lagmulx(c)
Multiply a Laguerre series by x.

Multiply the Laguerre series c by x, where x is the independent variable.

Parameters

c : array_like

1-D array of Laguerre series coefficients ordered from low to high.

Returns

out : ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Laguerre polynomials in the form

\[ xP_i(x) = -(i + 1)P_{i + 1}(x) + (2i + 1)P_i(x) - iP_{i - 1}(x) \]

Examples

```python
>>> from numpy.polynomial.laguerre import lagmulx
>>> lagmulx([1, 2, 3])
array([-1., -1., 11., -9.])
```
```
numpy.polynomial.laguerre.lagdiv(c1, c2)
Divide one Laguerre series by another.

Returns the quotient-with-remainder of two Laguerre series c1 / c2. The arguments are sequences of coefficients from lowest order “term” to highest, e.g., [1,2,3] represents the series \( P_0 + 2P_1 + 3P_2 \).

Parameters

c1, c2 : array_like

1-D arrays of Laguerre series coefficients ordered from low to high.

Returns

[quo, rem] : ndarrays

Of Laguerre series coefficients representing the quotient and remainder.

See Also:

lagadd, lagsub, lagmul, lagpow

Notes

In general, the (polynomial) division of one Laguerre series by another results in quotient and remainder terms that are not in the Laguerre polynomial basis set. Thus, to express these results as a Laguerre series, it is necessary to “reproject” the results onto the Laguerre basis set, which may produce “unintuitive” (but correct) results; see Examples section below.
Examples
>>> from numpy.polynomial.laguerre import lagdiv
>>> lagdiv([  8., -13.,  38., -51.,  36.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> lagdiv([  9., -12.,  38., -51.,  36.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 1.]))

numpy.polynomial.laguerre.lagpow(c, pow, maxpower=16)
Raise a Laguerre series to a power.

Returns the Laguerre series \( c \) raised to the power \( pow \). The argument \( c \) is a sequence of coefficients ordered from low to high. i.e., \([1,2,3]\) is the series \( P_0 + 2*P_1 + 3*P_2 \).

Parameters
- \( c \) : array_like
  1-D array of Laguerre series coefficients ordered from low to high.
- \( pow \) : integer
  Power to which the series will be raised
- \( maxpower \) : integer, optional
  Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns
- \( coef \) : ndarray
  Laguerre series of power.

See Also:
lagadd, lagsub, lagmul, lagdiv

Examples
>>> from numpy.polynomial.laguerre import lagpow
>>> lagpow([1, 2, 3, 4], 2)
array([ 14., -16.,  56., -72.,  54.])

Quadrature
numpy.polynomial.laguerre.laggauss(deg)
Gauss-Laguerre quadrature.

Computes the sample points and weights for Gauss-Laguerre quadrature. These sample points and weights will correctly integrate polynomials of degree \( 2 * deg - 1 \) or less over the interval \([0, \infty)\] with the weight function \( f(x) = \exp(-x) \).

Parameters
- \( deg \) : int
  Number of sample points and weights. It must be >= 1.

Returns
- \( x \) : ndarray

3.26. Polynomials
1-D ndarray containing the sample points.

\[ y : \text{ndarray} \]
1-D ndarray containing the weights.

**Notes**
The results have only been tested up to degree 100 higher degrees may be problematic. The weights are determined by using the fact that

\[ w_k = c / (L_n'(x_k) \ast L_{n-1}(x_k)) \]

where \( c \) is a constant independent of \( k \) and \( x_k \) is the \( k \)'th root of \( L_n \), and then scaling the results to get the right value when integrating 1.

**numpy.polynomial.laguerre.lagweight(x)**
Weight function of the Laguerre polynomials.

The weight function is \( \exp(-x) \) and the interval of integration is \([0, \infty)\). The Laguerre polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**
\[ x : \text{array_like} \]
Values at which the weight function will be computed.

**Returns**
\[ w : \text{ndarray} \]
The weight function at \( x \).

**Notes**

**numpy.polynomial.laguerre.lagcompanion(c)**
Return the companion matrix of \( c \).

The usual companion matrix of the Laguerre polynomials is already symmetric when \( c \) is a basis Laguerre polynomial, so no scaling is applied.

**Parameters**
\[ c : \text{array_like} \]
1-D array of Laguerre series coefficients ordered from low to high degree.

**Returns**
\[ mat : \text{ndarray} \]
Companion matrix of dimensions (deg, deg).

**Miscellaneous**

**numpy.polynomial.laguerre.poly2lag(pol)**
Convert a polynomial to a Laguerre series.
NumPy Reference, Release 1.8.1

Notes

numpy.polynomial.laguerre.lagdomain = array([0, 1])
numpy.polynomial.laguerre.lagzero = array([0])
numpy.polynomial.laguerre.lagone = array([1])
numpy.polynomial.laguerre.lagx = array([1, -1])
numpy.polynomial.laguerre.lagtrim(c, tol=0)
  Remove “small” “trailing” coefficients from a polynomial.
  “Small” means “small in absolute value” and is controlled by the parameter tol; “trailing” means highest order coefficient(s), e.g., in [0, 1, 1, 0, 0] (which represents 0 + x + x**2 + 0*x**3 + 0*x**4) both the 3-rd and 4-th order coefficients would be “trimmed.”

Parameters
  c : array_like
    1-d array of coefficients, ordered from lowest order to highest.
  tol : number, optional
    Trailing (i.e., highest order) elements with absolute value less than or equal to tol (default value is zero) are removed.

Returns

trimmed : ndarray
  1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

Raises

ValueError
  If tol < 0

See Also:

trimseq

Examples

>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([ 0., 0., 3., 0., 5.])
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([ 0.])
>>> i = complex(0,1) # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i), 1e-3)
array([ 0.0003+0.j , 0.0010-0.001j])

numpy.polynomial.laguerre.lagline(off, scl)
  Laguerre series whose graph is a straight line.

Parameters
  off, scl : scalars
  The specified line is given by off + scl*x.
Returns

y : ndarray

This module’s representation of the Laguerre series for \( \text{off} + \text{scl} \times x \).

See Also:

polyline, chebline

Examples

```python
>>> from numpy.polynomial.laguerre import lagline, lagval
>>> lagval(0, lagline(3, 2))
3.0
>>> lagval(1, lagline(3, 2))
5.0
```

```
numpy.polynomial.laguerre.lag2poly(c)

Convert a Laguerre series to a polynomial.

Convert an array representing the coefficients of a Laguerre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest to highest degree.

Parameters

c : array_like

1-D array containing the Laguerre series coefficients, ordered from lowest order term to highest.

Returns

pol : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest order term to highest.

See Also:

poly2lag

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```python
>>> from numpy.polynomial.laguerre import lag2poly
>>> lag2poly([23., -63., 58., -18.])
array([0., 1., 2., 3.])
```

```
numpy.polynomial.laguerre.poly2lag(pol)

Convert a polynomial to a Laguerre series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Laguerre series, ordered from lowest to highest degree.

Parameters

pol : array_like

1-D array containing the polynomial coefficients

Returns

c : ndarray

1-D array containing the polynomial coefficients
1-D array containing the coefficients of the equivalent Laguerre series.

**See Also:**

lag2poly

**Notes**

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

**Examples**

```python
>>> from numpy.polynomial.laguerre import poly2lag
>>> poly2lag(np.arange(4))
array([ 23., -63., 58., -18.])
```

**Hermite Module, “Physicists”** *(numpy.polynomial.hermite)*

New in version 1.6.0. This module provides a number of objects (mostly functions) useful for dealing with Hermite series, including a Hermite class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).

```python
Hermite(coef[, domain, window])  A Hermite series class.
```

**Hermite Class**

```python
class numpy.polynomial.hermite.Hermite(coef, domain=[-1, 1], window=[-1, 1])
A Hermite series class.
```

Hermite instances provide the standard Python numerical methods ‘+’, ‘-‘, ‘*‘, ‘/‘, ‘%‘, ‘divmod‘, ‘**‘, and ‘()‘ as well as the listed methods.

**Parameters**

- `coef`: array_like
  
  Hermite coefficients, in increasing order. For example, (1, 2, 3) implies $P_0 + 2P_1 + 3P_2$ where the $P_i$ are a graded polynomial basis.

- `domain`: (2,) array_like, optional
  
  Domain to use. The interval [domain[0], domain[1]] is mapped to the interval [window[0], window[1]] by shifting and scaling. The default value is [-1,1].

- `window`: (2,) array_like, optional
  
  Window, see domain for its use. The default value is [-1,1]. .. versionadded:: 1.6.0

**Notes**

It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.

**Attributes**

- `coef`: (N,) ndarray
  Hermite coefficients, from low to high.

- `domain`: (2,) ndarray
  Domain that is mapped to `window`.

- `window`: (2,) ndarray
  Window that `domain` is mapped to.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(arg)</td>
<td></td>
</tr>
<tr>
<td>basis(deg[, domain, window])</td>
<td>Hermite polynomial of degree deg.</td>
</tr>
<tr>
<td>cast(series[, domain, window])</td>
<td>Convert instance to equivalent Hermite series.</td>
</tr>
<tr>
<td>convert([domain, kind, window])</td>
<td>Convert to different class and/or domain.</td>
</tr>
<tr>
<td>copy()</td>
<td>Return a copy.</td>
</tr>
<tr>
<td>cutdeg(deg)</td>
<td>Truncate series to the given degree.</td>
</tr>
<tr>
<td>degree()</td>
<td>The degree of the series.</td>
</tr>
<tr>
<td>deriv([m])</td>
<td>Differentiate.</td>
</tr>
<tr>
<td>fit(x, y, deg[, domain, rcond, full, w, window])</td>
<td>Least squares fit to data.</td>
</tr>
<tr>
<td>fromroots(roots[, domain, window])</td>
<td>Return Hermite instance with specified roots.</td>
</tr>
<tr>
<td>has_samecoef(other)</td>
<td>Check if coefficients match.</td>
</tr>
<tr>
<td>has_samedomain(other)</td>
<td>Check if domains match.</td>
</tr>
<tr>
<td>has_sametype(other)</td>
<td>Check if types match.</td>
</tr>
<tr>
<td>has_samewindow(other)</td>
<td>Check if windows match.</td>
</tr>
<tr>
<td>identity([domain, window])</td>
<td>Identity function.</td>
</tr>
<tr>
<td>integ([m, k, lbnd])</td>
<td>Integrate.</td>
</tr>
<tr>
<td>linspace([n, domain])</td>
<td>Return x,y values at equally spaced points in domain.</td>
</tr>
<tr>
<td>mapparms()</td>
<td>Return the mapping parameters.</td>
</tr>
<tr>
<td>roots()</td>
<td>Return list of roots.</td>
</tr>
<tr>
<td>trim([tol])</td>
<td>Remove small leading coefficients.</td>
</tr>
<tr>
<td>truncate(size)</td>
<td>Truncate series to length size.</td>
</tr>
</tbody>
</table>

### static

**Hermite.basis**(deg, domain=[-1, 1], window=[-1, 1])

Hermite polynomial of degree deg.

Returns an instance of the Hermite polynomial of degree d.

**Parameters**

- **deg**: int
  
  Degree of the Hermite polynomial. Must be >= 0.

- **domain**: array_like
  
  The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

- **window**: array_like
  
  The resulting array must be of the form [beg, end], where beg and end are the endpoints of the window.

**Returns**

- **p**: Hermite instance

**Notes**

New in version 1.7.0.

**static** **Hermite.cast**(series, domain=[-1, 1], window=[-1, 1])

Convert instance to equivalent Hermite series.
The `series` is expected to be an instance of some polynomial series of one of the types supported by the `numpy.polynomial` module, but could be some other class that supports the convert method.

**Parameters**

- **series**: series
  The instance series to be converted.

- **domain**: array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain.

- **window**: array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the window.

**Returns**

- **p**: Hermite instance
  A Hermite series equal to the `poly` series.

**See Also:**

- `convert`

**Notes**

New in version 1.7.0.

```python
class Hermite:
    convert(domain=None, kind=None, window=None)
```

Convert to different class and/or domain.

**Parameters**

- **domain**: array_like, optional
  The domain of the converted series. If the value is None, the default domain of `kind` is used.

- **kind**: class, optional
  The polynomial series type class to which the current instance should be converted. If `kind` is None, then the class of the current instance is used.

- **window**: array_like, optional
  The window of the converted series. If the value is None, the default window of `kind` is used.

**Returns**

- **new_series_instance**: `kind`
  The returned class can be of different type than the current instance and/or have a different domain.

**Notes**

Conversion between domains and class types can result in numerically ill defined series.

```python
class Hermite:
    copy()
```

Return a copy.

Return a copy of the current Hermite instance.
Returns

new_instance : Hermite
Copy of current instance.

Hermite.cutdeg(deg)
Truncate series to the given degree.

Reduce the degree of the Hermite series to deg by discarding the high order terms. If deg is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters

deg : non-negative int
The series is reduced to degree deg by discarding the high order terms. The value of deg must be a non-negative integer.

Returns

new_instance : Hermite
New instance of Hermite with reduced degree.

Notes
New in version 1.5.0.

Hermite.degree()
The degree of the series.

Notes
New in version 1.5.0.

Hermite.deriv(m=1)
Differentiate.

Return an instance of Hermite that is the derivative of the current series. Refer to hermder for full documentation.

Parameters

m : non-negative int
The number of integrations to perform.

Returns

derivative : Hermite
The derivative of the series using the same domain.

See Also:

hermder
similar function.

hermint
similar function for integration.

static Hermite.fit(x, y, deg, domain=None, rcond=None, full=False, w=None, window=[-1, 1])
Least squares fit to data.

Return a Hermite instance that is the least squares fit to the data y sampled at x. Unlike hermfit, the domain of the returned instance can be specified and this will often result in a superior fit with less chance
of ill conditioning. Support for NA was added in version 1.7.0. See `hermfit` for full documentation of the implementation.

**Parameters**

- **x**: array_like, shape (M,)
  - x-coordinates of the M sample points \((x[i], y[i])\).

- **y**: array_like, shape (M,) or (M, K)
  - y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- **deg**: int
  - Degree of the fitting polynomial.

- **domain**: [None, [beg, end], []], optional
  - Domain to use for the returned Hermite instance. If `None`, then a minimal domain that covers the points x is chosen. If `[]` the default domain \([-1, 1]\) is used. The default value is \([-1,1]\) in numpy 1.4.x and `None` in later versions. The `[]` value was added in numpy 1.5.0.

- **rcond**: float, optional
  - Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is \(\text{len}(x) \times \text{eps}\), where \(\text{eps}\) is the relative precision of the float type, about 2e-16 in most cases.

- **full**: bool, optional
  - Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

- **w**: array_like, shape (M,), optional
  - Weights. If not `None` the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i] \times y[i]\) all have the same variance. The default value is `None`. .. versionadded:: 1.5.0

- **window**: [{[beg, end]}], optional
  - Window to use for the returned Hermite instance. The default value is \([-1,1]\). .. versionadded:: 1.6.0

**Returns**

- **least_squares_fit**: instance of Hermite
  - The Hermite instance is the least squares fit to the data and has the domain specified in the call.

- **[residuals, rank, singular_values, rcond]**: only if `full` = True
  - Residuals of the least squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of `rcond`. For more details, see `linalg.lstsq`.

**See Also:**

---

3.26. Polynomials 1067
**hermfit**

similar function

**static** `Hermite.fromroots(roots, domain=[-1, 1], window=[-1, 1])`

Return Hermite instance with specified roots.

Returns an instance of Hermite representing the product \((x - r[0])*(x - r[1])*\ldots*(x - r[n-1])\), where \(r\) is the list of roots.

**Parameters**

- **roots**: array_like
  - List of roots.

- **domain**: {array_like, None}, optional
  - Domain for the resulting instance of Hermite. If none the domain is the interval from the smallest root to the largest. The default is [-1,1].

- **window**: array_like, optional
  - Window for the resulting instance of Hermite. The default value is [-1,1].

**Returns**

- **object**: Hermite instance
  - Series with the specified roots.

**See Also:**

- **hermfromroots**
  - equivalent function

**Hermite.has_samecoef(other)**

Check if coefficients match.

**Parameters**

- **other**: class instance
  - The other class must have the `coef` attribute.

**Returns**

- **bool**: boolean
  - True if the coefficients are the same, False otherwise.

**Notes**

New in version 1.6.0.

**Hermite.has_samedomain(other)**

Check if domains match.

**Parameters**

- **other**: class instance
  - The other class must have the `domain` attribute.

**Returns**

- **bool**: boolean
  - True if the domains are the same, False otherwise.
**Notes**

New in version 1.6.0.

Hermite.**has_sametype**(other)
Check if types match.

**Parameters**

other : object
Class instance.

**Returns**

bool : boolean
True if other is same class as self

**Notes**

New in version 1.7.0.

Hermite.**has_samewindow**(other)
Check if windows match.

**Parameters**

other : class instance
The other class must have the window attribute.

**Returns**

bool : boolean
True if the windows are the same, False otherwise.

**Notes**

New in version 1.6.0.

**static** Hermite.**identity**(domain=[-1, 1], window=[-1, 1])
Identity function.

If p is the returned Hermite object, then p(x) == x for all values of x.

**Parameters**

domain : array_like
The resulting array must be of the form [beg, end], where beg and end are the endpoints of the domain.

window : array_like
The resulting array must be if the form [beg, end], where beg and end are the endpoints of the window.

**Returns**

identity : Hermite instance

Hermite.**integ**(m=1, k=[], lbnd=None)
Integrate.

Return an instance of Hermite that is the definite integral of the current series. Refer to hermint for full documentation.

**Parameters**

m : non-negative int
The number of integrations to perform.

\( k \) : array_like

Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to \( m \) in length and any missing values are set to zero.

\( \text{lbnd} \) : Scalar

The lower bound of the definite integral.

**Returns**

\( \text{integral} \) : Hermite

The integral of the series using the same domain.

**See Also:**

- \texttt{hermint}
  - similar function.
- \texttt{hermder}
  - similar function for derivative.

\texttt{Hermite.linspace}(n=100, domain=None)

Return \( x,y \) values at equally spaced points in domain.

Returns \( x, y \) values at \( n \) linearly spaced points across domain. Here \( y \) is the value of the polynomial at the points \( x \). By default the domain is the same as that of the Hermite instance. This method is intended mostly as a plotting aid.

**Parameters**

- \( n \) : int, optional
  - Number of point pairs to return. The default value is 100.
- \( \text{domain} \) : \{None, array_like\}
  - If not None, the specified domain is used instead of that of the calling instance. It should be of the form \([\text{beg}, \text{end}]\). The default is None.

**Returns**

\( x, y \) : ndarrays

\( x \) is equal to \texttt{linspace}(self.domain[0], self.domain[1], n) \( y \) is the polynomial evaluated at \( x \).

New in version 1.5.0.

\texttt{Hermite.mapparms}()

Return the mapping parameters.

The returned values define a linear map \( \text{off} + \text{scl} \times x \) that is applied to the input arguments before the series is evaluated. The map depends on the \texttt{domain} and \texttt{window}; if the current \texttt{domain} is equal to the \texttt{window} the resulting map is the identity. If the coefficients of the \texttt{Hermite} instance are to be used by themselves outside this class, then the linear function must be substituted for the \( x \) in the standard representation of the base polynomials.

**Returns**

\( \text{off}, \text{scl} \) : floats or complex

The mapping function is defined by \( \text{off} + \text{scl} \times x \).
Notes

If the current domain is the interval \([l_1, r_1]\) and the window is \([l_2, r_2]\), then the linear mapping function \(L\) is defined by the equations:

\[
L(l_1) = l_2 \\
L(r_1) = r_2
\]

**Hermite.roots()**

Return list of roots.

Return ndarray of roots for this series. See `hermroots` for full documentation. Note that the accuracy of the roots is likely to decrease the further outside the domain they lie.

**See Also:**

- **hermroots**
  - similar function
- **hermfromroots**
  - function to generate series from roots.

**Hermite.trim(tol=0)**

Remove small leading coefficients

Remove leading coefficients until a coefficient is reached whose absolute value greater than \(tol\) or the beginning of the series is reached. If all the coefficients would be removed the series is set to \([0]\). A new Hermite instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

- **tol** : non-negative number.
  - All trailing coefficients less than \(tol\) will be removed.

**Returns**

- **new_instance** : Hermite
  - Contains the new set of coefficients.

**Hermite.truncate(size)**

Truncate series to length \(size\).

Reduce the Hermite series to length \(size\) by discarding the high degree terms. The value of \(size\) must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

- **size** : positive int
  - The series is reduced to length \(size\) by discarding the high degree terms. The value of \(size\) must be a positive integer.

**Returns**

- **new_instance** : Hermite
  - New instance of Hermite with truncated coefficients.

---

<table>
<thead>
<tr>
<th>hermval(x, c[, tensor])</th>
<th>Evaluate an Hermite series at points (x).</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermval2d(x, y, c)</td>
<td>Evaluate a 2-D Hermite series at points ((x, y)).</td>
</tr>
<tr>
<td>hermval3d(x, y, z, c)</td>
<td>Evaluate a 3-D Hermite series at points ((x, y, z)).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.156 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermgrid2d(x, y, c)</td>
<td>Evaluate a 2-D Hermite series on the Cartesian product of x and y.</td>
</tr>
<tr>
<td>hermgrid3d(x, y, z, c)</td>
<td>Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z.</td>
</tr>
<tr>
<td>hermroots(c)</td>
<td>Compute the roots of a Hermite series.</td>
</tr>
<tr>
<td>hermfromroots(roots)</td>
<td>Generate a Hermite series with given roots.</td>
</tr>
</tbody>
</table>

Basics

numpy.polynomial.hermite.hermval(x, c, tensor=True)

Evaluate an Hermite series at points x.

If \( c \) is of length \( n + 1 \), this function returns the value:

\[
p(x) = c_0 \* H_0(x) + c_1 \* H_1(x) + \ldots + c_n \* H_n(x)
\]

The parameter \( x \) is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either \( x \) or its elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) is a 1-D array, then \( p(x) \) will have the same shape as \( x \). If \( c \) is multidimensional, then the shape of the result depends on the value of tensor. If tensor is true the shape will be c.shape[1:] + x.shape. If tensor is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

Parameters

- **x**: array_like, compatible object
  - If \( x \) is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, \( x \) or its elements must support addition and multiplication with with themselves and with the elements of \( c \).

- **c**: array_like
  - Array of coefficients ordered so that the coefficients for terms of degree \( n \) are contained in \( c[n] \). If \( c \) is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of \( c \).

- **tensor**: boolean, optional
  - If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of \( x \). Scalars have dimension 0 for this action. The result is that every column of coefficients in \( c \) is evaluated for every element of \( x \). If False, \( x \) is broadcast over the columns of \( c \) for the evaluation. This keyword is useful when \( c \) is multidimensional. The default value is True. New in version 1.7.0.

Returns

- **values**: ndarray, algebra_like
  - The shape of the return value is described above.

See Also:

hermval2d, hermgrid2d, hermval3d, hermgrid3d

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.
Examples

```python
>>> from numpy.polynomial.hermite import hermval
>>> c = [1, 2, 3]
>>> hermval(1, c)
11.0
>>> hermval([[1, 2], [3, 4]], c)
array([[ 11.,  51.],
       [135., 203.]])
```

The parameters x and y are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars and they must have the same shape after conversion. In either case, either x and y or their elements must support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

- x, y : array_like, compatible objects
  The two dimensional series is evaluated at the points (x, y), where x and y must have the same shape. If x or y is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

- c : array_like
  Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in c[i, j]. If c has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

- values : ndarray, compatible object
  The values of the two dimensional polynomial at points formed with pairs of corresponding values from x and y.

**See Also:**

- hermval, hermgrid2d, hermval3d, hermgrid3d

**Notes**

```python
>>> from numpy.polynomial.hermite import hermval3d
>>> hermval3d(x, y, z, c)
```

The parameters x, y, and z are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars and they must have the same shape after conversion. In either case, either x, y, and z or their elements must support multiplication and addition both with themselves and with the elements of c.
If \( c \) has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \( c.shape[3:] + x.shape \).

**Parameters**
- \( x, y, z \) : array_like, compatible object
  
The three dimensional series is evaluated at the points \((x, y, z)\), where \( x, y, \) and \( z \) must have the same shape. If any of \( x, y, \) or \( z \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.
- \( c \) : array_like
  
  Array of coefficients ordered so that the coefficient of the term of multi-degree \( i,j,k \) is contained in \( c[i,j,k] \). If \( c \) has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**
- \( \text{values} \) : ndarray, compatible object
  
  The values of the multidimensional polynomial on points formed with triples of corresponding values from \( x, y, \) and \( z \).

**See Also:**
- `hermval`, `hermval2d`, `hermgrid2d`, `hermgrid3d`

**Notes**

```
numpy.polynomial.hermite.hermgrid2d(x, y, c)
```

Evaluate a 2-D Hermite series on the Cartesian product of \( x \) and \( y \).

This function returns the values:

\[
p(a, b) = \sum_{i,j} c_{i,j} \cdot H_i(a) \cdot H_j(b)
\]

where the points \((a, b)\) consist of all pairs formed by taking \( a \) from \( x \) and \( b \) from \( y \). The resulting points form a grid with \( x \) in the first dimension and \( y \) in the second.

The parameters \( x \) and \( y \) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \( x \) and \( y \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \( c.shape[2:] + x.shape \).

**Parameters**
- \( x, y \) : array_like, compatible objects
  
The two dimensional series is evaluated at the points in the Cartesian product of \( x \) and \( y \). If \( x \) or \( y \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.
- \( c \) : array_like
  
  Array of coefficients ordered so that the coefficients for terms of degree \( i,j \) are contained in \( c[i,j] \). If \( c \) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**
- \( \text{values} \) : ndarray, compatible object
The values of the two dimensional polynomial at points in the Cartesian product of \( x \) and \( y \).

See Also:

hermval, hermval2d, hermval3d, hermgrid3d

Notes

numpy.polynomial.hermite.hermgrid3d(x, y, z, c)
Evaluate a 3-D Hermite series on the Cartesian product of \( x \), \( y \), and \( z \).
This function returns the values:

\[
p(a, b, c) = \sum_{i,j,k} c_{i,j,k} \ast H_i(a) \ast H_j(b) \ast H_k(c)
\]

where the points \((a, b, c)\) consist of all triples formed by taking \( a \) from \( x \), \( b \) from \( y \), and \( c \) from \( z \). The resulting points form a grid with \( x \) in the first dimension, \( y \) in the second, and \( z \) in the third.

The parameters \( x \), \( y \), and \( z \) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars. In either case, either \( x \), \( y \), and \( z \) or their elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \( c.shape[3:] + x.shape + y.shape + z.shape \).

Parameters

\( x, y, z : \) array_like, compatible objects
The three dimensional series is evaluated at the points in the Cartesian product of \( x \), \( y \), and \( z \). If \( x, y, \) or \( z \) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

\( c : \) array_like
Array of coefficients ordered so that the coefficients for terms of degree \( i \), \( j \) are contained in \( c[i, j] \). If \( c \) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

\( values : \) ndarray, compatible object
The values of the two dimensional polynomial at points in the Cartesian product of \( x \) and \( y \).

See Also:

hermval, hermval2d, hermgrid2d, hermval3d

Notes

numpy.polynomial.hermite.hermroots(c)
Compute the roots of a Hermite series.
Return the roots (a.k.a. “zeros”) of the polynomial

\[
p(x) = \sum_i c[i] \ast H_i(x).
\]
Parameters
   c : 1-D array_like
       1-D array of coefficients.

Returns
   out : ndarray
       Array of the roots of the series. If all the roots are real, then out is also real, otherwise
       it is complex.

See Also:
   polyroots, legroots, lagroots, chebroots, hermeroots

Notes
   The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the
   complex plane may have large errors due to the numerical instability of the series for such values. Roots with
   multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively
   insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s
   method.

   The Hermite series basis polynomials aren’t powers of x so the results of this function may seem unintuitive.

Examples
   >>> from numpy.polynomial.hermite import hermroots, hermfromroots
   >>> coef = hermfromroots([-1, 0, 1])
   >>> coef
   array([ 0. , 0.25 , 0. , 0.125])
   >>> hermroots(coef)
   array([-1.00000000e+00, -1.38777878e-17, 1.00000000e+00])

   numpy.polynomial.hermite.hermfromroots(roots)
   Generate a Hermite series with given roots.

   The function returns the coefficients of the polynomial
   \[ p(x) = (x - r_0) \cdot (x - r_1) \cdot \ldots \cdot (x - r_n), \]
   in Hermite form, where the \( r_n \) are the roots specified in \( \text{roots} \). If a zero has multiplicity \( n \), then it must appear
   in \( \text{roots} \) \( n \) times. For instance, if \( 2 \) is a root of multiplicity three and \( 3 \) is a root of multiplicity 2, then \( \text{roots} \) looks
   something like \([2, 2, 2, 3] \). The roots can appear in any order.

   If the returned coefficients are \( c \), then
   \[ p(x) = c_0 + c_1 \cdot H_1(x) + \ldots + c_n \cdot H_n(x) \]

   The coefficient of the last term is not generally 1 for monic polynomials in Hermite form.

   Parameters
   roots : array_like
       Sequence containing the roots.

   Returns
   out : ndarray
1-D array of coefficients. If all roots are real then out is a real array, if some of the roots are complex, then out is complex even if all the coefficients in the result are real (see Examples below).

See Also:

polyfromroots, legfromroots, lagfromroots, chebfromroots, hermfromroots.

Examples

```python
from numpy.polynomial.hermite import hermfromroots, hermval

coef = hermfromroots((-1, 0, 1))
hermval((-1, 0, 1), coef)
array([ 0., 0., 0.])

coef = hermfromroots((-1j, 1j))
hermval((-1j, 1j), coef)
array([ 0.+0.j, 0.+0.j])
```

```
hermfit(x, y, deg[, rcond, full, w]) Least squares fit of Hermite series to data.
hermvander(x, deg) Pseudo-Vandermonde matrix of given degree.
hermvander2d(x, y, deg) Pseudo-Vandermonde matrix of given degrees.
hermvander3d(x, y, z, deg) Pseudo-Vandermonde matrix of given degrees.
```

Fitting

numpy.polynomial.hermite.hermfit(x, y, deg, rcond=None, full=False, w=None)

Least squares fit of Hermite series to data.

Return the coefficients of a Hermite series of degree deg that is the least squares fit to the data values y given at points x. If y is 1-D the returned coefficients will also be 1-D. If y is 2-D multiple fits are done, one for each column of y, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

\[ p(x) = c_0 + c_1 \times H_1(x) + \ldots + c_n \times H_n(x), \]

where \( n \) is \( deg \).

Since numpy version 1.7.0, hermfit also supports NA. If any of the elements of x, y, or w are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If y is 2-D, then an NA in any row of y invalidates that whole row.

Parameters

- `x`: array_like, shape (M,)
  - x-coordinates of the M sample points (x[i], y[i]).

- `y`: array_like, shape (M,) or (M, K)
  - y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

- `deg`: int
  - Degree of the fitting polynomial

- `rcond`: float, optional
Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**: bool, optional
Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**: array_like, shape (M,), optional
Weights. If not None, the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i]*y[i]\) all have the same variance. The default value is None.

**Returns**

**coef**: ndarray, shape (M,) or (M, K)
Hermite coefficients ordered from low to high. If \(y\) was 2-D, the coefficients for the data in column \(k\) of \(y\) are in column \(k\).

**[residuals, rank, singular_values, rcond]**: present when \(full = True\)
Residuals of the least-squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of \(rcond\). For more details, see linalg.lstsq.

**Warns**

**RankWarning**
The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if \(full = False\). The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

**See Also:**
chebfit, legfit, lagfit, polyfit, hermfit

**hermval**
Evaluates a Hermite series.

**hermvander**
Vandermonde matrix of Hermite series.

**hermweight**
Hermite weight function

**linalg.lstsq**
Computes a least-squares fit from the matrix.

**scipy.interpolate.UnivariateSpline**
Computes spline fits.

**Notes**
The solution is the coefficients of the Hermite series \(p\) that minimizes the sum of the weighted squared errors

\[
E = \sum_j w_j^2 \cdot |y_j - p(x_j)|^2,
\]
where the $w_j$ are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where $V$ is the weighted pseudo Vandermonde matrix of $x$, $c$ are the coefficients to be solved for, $w$ are the weights, $y$ are the observed values. This equation is then solved using the singular value decomposition of $V$.

If some of the singular values of $V$ are so small that they are neglected, then a RankWarning will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The $rcond$ parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Hermite series are probably most useful when the data can be approximated by $\sqrt{w(x)} * p(x)$, where $w(x)$ is the Hermite weight. In that case the weight $\sqrt{w(x[i])}$ should be used together with data values $y[i]/\sqrt{w(x[i])}$. The weight function is available as hermweight.

References

[R59]

Examples

```python
>>> from numpy.polynomial.hermite import hermfit, hermval
>>> x = np.linspace(-10, 10)
>>> err = np.random.randn(len(x))/10
>>> y = hermval(x, [1, 2, 3]) + err
>>> hermfit(x, y, 2)
array([ 0.97902637, 1.99849131, 3.00006000])
```

**numpy.polynomial.hermite.\texttt{hermvander}(x, deg)**

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree $deg$ and sample points $x$. The pseudo-Vandermonde matrix is defined by

$$V[\ldots, i] = H_i(x),$$

where $0 <= i <= deg$. The leading indices of $V$ index the elements of $x$ and the last index is the degree of the Hermite polynomial.

If $c$ is a 1-D array of coefficients of length $n + 1$ and $V$ is the array $V = \text{hermvander}(x, n)$, then $\text{np.dot}(V, c)$ and $\text{hermval}(x, c)$ are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Hermite series of the same degree and sample points.

**Parameters**

- $x$ : array_like
  
  Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If $x$ is scalar it is converted to a 1-D array.

- $deg$ : int
  
  Degree of the resulting matrix.

**Returns**

- $vander$ : ndarray
The pseudo-Vandermonde matrix. The shape of the returned matrix is \(x.\text{shape} + (\text{deg} + 1,)\), where the last index is the degree of the corresponding Hermite polynomial. The dtype will be the same as the converted \(x\).

**Examples**

```python
>>> from numpy.polynomial.hermite import hermvander

x = np.array([-1, 0, 1])
hermvander(x, 3)
```

```
array([[ 1., -2., 2., 4.],
       [ 1., 0., -2., -0.],
       [ 1., 2., 2., -4.]])
```

`numpy.polynomial.hermite.hermvander2d(x, y, deg)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees \(deg\) and sample points \((x, y)\). The pseudo-Vandermonde matrix is defined by

\[
V[\ldots, \text{deg}[1]*i + j] = H_i(x) * H_j(y),
\]

where \(0 \leq i \leq \text{deg}[0]\) and \(0 \leq j \leq \text{deg}[1]\). The leading indices of \(V\) index the points \((x, y)\) and the last index encodes the degrees of the Hermite polynomials.

If \(V = \text{hermvander2d}(x, y, [\text{xdeg}, \text{ydeg}])\), then the columns of \(V\) correspond to the elements of a 2-D coefficient array \(c\) of shape \((\text{xdeg} + 1, \text{ydeg} + 1)\) in the order

\[
c_{00}, c_{01}, c_{02}, \ldots, c_{10}, c_{11}, c_{12}, \ldots
\]

and \(\text{np.dot}(V, c.\text{flat})\) and \(\text{hermval2d}(x, y, c)\) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Hermite series of the same degrees and sample points.

**Parameters**

\(- x, y : \text{array_like}\)

Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

\(- \text{deg} : \text{list of ints}\)

List of maximum degrees of the form \([\text{x\_deg}, \text{y\_deg}]\).

**Returns**

\(- \text{vander2d} : \text{ndarray}\)

The shape of the returned matrix is \(x.\text{shape} + (\text{order},)\), where \(\text{order} = (\text{deg}[0] + 1) * (\text{deg}[1] + 1)\). The dtype will be the same as the converted \(x\) and \(y\).

**See Also:**

`hermvander`, `hermvander3d`, `hermval3d`

**Notes**

`numpy.polynomial.hermite.hermvander3d(x, y, z, deg)`

Pseudo-Vandermonde matrix of given degrees.
Returns the pseudo-Vandermonde matrix of degrees \( deg \) and sample points \((x, y, z)\). If \( l, m, n \) are the given degrees in \( x, y, z \), then the pseudo-Vandermonde matrix is defined by

\[
V[\ldots, (m + 1)(n + 1)i + (n + 1)j + k] = H_i(x) * H_j(y) * H_k(z),
\]

where \( 0 <= i <= l \), \( 0 <= j <= m \), and \( 0 <= j <= n \). The leading indices of \( V \) index the points \((x, y, z)\) and the last index encodes the degrees of the Hermite polynomials.

If \( V = \text{hermvander3d}(x, y, z, [xdeg, ydeg, zdeg]) \), then the columns of \( V \) correspond to the elements of a 3-D coefficient array \( c \) of shape \((xdeg + 1, ydeg + 1, zdeg + 1)\) in the order

\[
c_{000}, c_{001}, c_{002}, \ldots, c_{010}, c_{011}, c_{012}, \ldots
\]

and \( \text{np.dot}(V, c\text{.flat}) \) and \( \text{hermval3d}(x, y, z, c) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Hermite series of the same degrees and sample points.

**Parameters**

- \( x, y, z \): array_like
  
  Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

- \( \text{deg} \): list of ints
  
  List of maximum degrees of the form \([x\_deg, y\_deg, z\_deg]\).

**Returns**

- \( \text{vander3d} \): ndarray
  
  The shape of the returned matrix is \( x\_shape + (\text{order},) \), where \( \text{order} = (\text{deg}[0] + 1) * (\text{deg}[1] + 1) * (\text{deg}[2] + 1) \). The dtype will be the same as the converted \( x, y, \) and \( z \).

**See Also:**

- \( \text{hermvander} \), \( \text{hermvander3d} \), \( \text{hermval3d} \)

**Notes**

- \( \text{hermder}(c, m=1, s=1, a=0) \)
  
  Differentiate a Hermite series.

- \( \text{hermint}(c, m, k, lbnd, s=1, a=0) \)
  
  Integrate a Hermite series.

**Calculus**

- \( \text{numpy.polynomial.hermite.\text{hermder}}(c, m=1, s=1, a=0) \)
  
  Differentiate a Hermite series.

- \( \text{Returns the Hermite series coefficients} c \) differentiated \( m \) times along \( a \). At each iteration the result is multiplied by \( s \) (the scaling factor is for use in a linear change of variable). The argument \( c \) is an array of coefficients from low to high degree along each axis, e.g., \([1,2,3]\) represents the series \(1*H_0 + 2*H_1 + 3*H_2\) while \([1,2],[1,2]\) represents \(1*H_0(x) * H_0(y) + 1*H_1(x) * H_0(y) + 2*H_0(x) * H_1(y) + 2*H_1(x) * H_1(y)\) if \( a=0 \) is \( x \) and \( a=1 \) is \( y \).

**Parameters**

- \( c \): array_like
Array of Hermite series coefficients. If `c` is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m** : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

**scl** : scalar, optional

Each differentiation is multiplied by `scl`. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis** : int, optional

Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.

**Returns**

**der** : ndarray

Hermite series of the derivative.

**See Also:**

`hermder`

**Notes**

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be “unintuitive,” albeit correct; see Examples section below.

**Examples**

```python
>>> from numpy.polynomial.hermite import hermder
>>> hermder([ 1. , 0.5, 0.5, 0.5])
array([ 1., 2., 3.])
>>> hermder([-0.5, 1./2., 1./8., 1./12., 1./16.], m=2)
array([ 1., 2., 3.])
```

`numpy.polynomial.hermite.hermint(c, m=1, k=[], lbnd=0, scl=1, axis=0)`

Integrate a Hermite series.

Returns the Hermite series coefficients `c` integrated `m` times from `lbnd` along `axis`. At each iteration the resulting series is multiplied by `scl` and an integration constant, `k`, is added. The scaling factor is for use in a linear change of variable. (“Buyer beware”: note that, depending on what one is doing, one may want `scl` to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument `c` is an array of coefficients from low to high degree along each axis, e.g., `[1,2,3]` represents the series \( H_0 + 2 \cdot H_1 + 3 \cdot H_2 \) while `[1,2],[1,2]` represents \( 1 \cdot H_0(x) \cdot H_0(y) + 1 \cdot H_1(x) \cdot H_0(y) + 2 \cdot H_0(x) \cdot H_1(y) + 2 \cdot H_1(x) \cdot H_1(y) \) if `axis=0` is `x` and `axis=1` is `y`.

**Parameters**

**c** : array_like

Array of Hermite series coefficients. If `c` is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m** : int, optional

Order of integration, must be positive. (Default: 1)

**k** : [[]], list, scalar], optional
Integration constant(s). The value of the first integral at \( lbnd \) is the first value in the list, the value of the second integral at \( lbnd \) is the second value, etc. If \( k == [] \) (the default), all constants are set to zero. If \( m == 1 \), a single scalar can be given instead of a list.

\[ \text{lbnd} : \text{scalar, optional} \]

The lower bound of the integral. (Default: 0)

\[ \text{scl} : \text{scalar, optional} \]

Following each integration the result is multiplied by \( \text{scl} \) before the integration constant is added. (Default: 1)

\[ \text{axis} : \text{int, optional} \]

Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

Returns

\[ S : \text{ndarray} \]

Hermite series coefficients of the integral.

Raises

ValueError

If \( m < 0 \), \( \text{len}(k) > m \), \( \text{np.isscalar}(\text{lbnd}) == \text{False} \), or \( \text{np.isscalar}(\text{scl}) == \text{False} \).

See Also:

hermdet

Notes

Note that the result of each integration is multiplied by \( \text{scl} \). Why is this important to note? Say one is making a linear change of variable \( u = ax + b \) in an integral relative to \( x \). Then .. math:: dx = du/a, so one will need to set \( \text{scl} \) equal to \( 1/a \) - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be “reprojected” onto the C-series basis set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples

```
>>> from numpy.polynomial.hermite import hermint
>>> hermint([1,2,3])  # integrate once, value 0 at 0.
array([ 1. , 0.5, 0.5, 0.5])
>>> hermint([1,2,3], m=2)  # integrate twice, value & deriv 0 at 0
array([-0.5, 0.5, 0.125, 0.08333333, 0.0625])
>>> hermint([1,2,3], k=1)  # integrate once, value 1 at 0.
array([ 2. , 0.5, 0.5, 0.5])
>>> hermint([1,2,3], lbnd=-1)  # integrate once, value 0 at -1
array([-2. , 0.5, 0.5, 0.5])
>>> hermint([1,2,3], m=2, k=[1,2], lbnd=-1)  # integrate once, value 0 at -1
array([ 1.66666667, -0.5, 0.125, 0.08333333, 0.0625])
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermadd(c1, c2)</td>
<td>Add one Hermite series to another.</td>
</tr>
<tr>
<td>hermsub(c1, c2)</td>
<td>Subtract one Hermite series from another.</td>
</tr>
<tr>
<td>hermmul(c1, c2)</td>
<td>Multiply one Hermite series by another.</td>
</tr>
<tr>
<td>hermmulx(c)</td>
<td>Multiply a Hermite series by x.</td>
</tr>
</tbody>
</table>

Continued on next page
Algebra

**numpy.polynomial.hermite.**

- `hermadd(c1, c2)`
  
  Add one Hermite series to another.
  
  Returns the sum of two Hermite series \( c_1 + c_2 \). The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( P_0 + 2P_1 + 3P_2 \).

  **Parameters**
  
  - `c1, c2`: array_like
    
    1-D arrays of Hermite series coefficients ordered from low to high.

  **Returns**
  
  - `out`: ndarray
    
    Array representing the Hermite series of their sum.

- `hermsub(c1, c2)`
  
  Subtract one Hermite series from another.
  
  Returns the difference of two Hermite series \( c_1 - c_2 \). The sequences of coefficients are from lowest order term to highest, i.e., \([1,2,3]\) represents the series \( P_0 + 2P_1 + 3P_2 \).

  **Parameters**
  
  - `c1, c2`: array_like
    
    1-D arrays of Hermite series coefficients ordered from low to high.

  **Returns**
  
  - `out`: ndarray
    
    Of Hermite series coefficients representing their difference.
Examples

```python
>>> from numpy.polynomial.hermite import hermsub
>>> hermsub([1, 2, 3, 4], [1, 2, 3])
array([ 0., 0., 0., 4.])
```

cnumpy.polynomial.hermite.hermmul(c1, c2)

Multiply one Hermite series by another.

Returns the product of two Hermite series c1 * c2. The arguments are sequences of coefficients, from lowest order “term” to highest, e.g., [1,2,3] represents the series \( P_0 + 2*P_1 + 3*P_2 \).

Parameters

- **c1, c2**: array_like

Returns

- **out**: ndarray

Of Hermite series coefficients representing their product.

See Also:

hermadd, hermsub, hermdiv, hermpow

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis set. Thus, to express the product as a Hermite series, it is necessary to “reproject” the product onto said basis set, which may produce “unintuitive” (but correct) results; see Examples section below.

Examples

```python
>>> from numpy.polynomial.hermite import hermmul
>>> hermmul([1, 2, 3], [0, 1, 2])
array([ 52., 29., 52., 7., 6.])
```

ncnumpy.polynomial.hermite.hermmulx(c)

Multiply a Hermite series by x.

Multiply the Hermite series c by x, where x is the independent variable.

Parameters

- **c**: array_like

Returns

- **out**: ndarray

Array representing the result of the multiplication.

Notes

The multiplication uses the recursion relationship for Hermite polynomials in the form

\[
xP_i(x) = (P_{i+1}(x)/2 + i*P_{i-1}(x))
\]
Examples

```python
>>> from numpy.polynomial.hermite import hermmulx
>>> hermmulx([1, 2, 3])
array([ 2. , 6.5, 1. , 1.5])
```

numpy.polynomial.hermite.**hermdiv**\((c1, c2)\)

Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series \(c1 / c2\). The arguments are sequences of coefficients from lowest order “term” to highest, e.g., [1,2,3] represents the series \(P_0 + 2\cdot P_1 + 3\cdot P_2\).

**Parameters**

- **c1, c2**: array_like
  1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

- **[quo, rem]**: ndarrays
  Of Hermite series coefficients representing the quotient and remainder.

**See Also:**
hermadd, hermsub, hermmul, hermpow

**Notes**

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to “reproject” the results onto the Hermite basis set, which may produce “unintuitive” (but correct) results; see Examples section below.

Examples

```python
>>> from numpy.polynomial.hermite import hermdiv
>>> hermdiv([ 52., 29., 52., 7., 6.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> hermdiv([ 54., 31., 52., 7., 6.], [0, 1, 2])
(array([1., 2., 3.]), array([2., 2.]))
>>> hermdiv([ 53., 30., 52., 7., 6.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 1.]))
```

numpy.polynomial.hermite.**hermpow**\((c, pow, maxpower=16)\)

Raise a Hermite series to a power.

Returns the Hermite series \(c\) raised to the power \(pow\). The argument \(c\) is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series \(P_0 + 2\cdot P_1 + 3\cdot P_2\).

**Parameters**

- **c**: array_like
  1-D array of Hermite series coefficients ordered from low to high.

- **pow**: integer
  Power to which the series will be raised

- **maxpower**: integer, optional
  Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16
Returns

```
coef : ndarray

Hermite series of power.
```

See Also:

```
hermadd, hermsub, hermmul, hermdiv
```

Examples

```python
>>> from numpy.polynomial.hermite import hermpow

>>> hermpow([1, 2, 3], 2)
array([ 81., 52., 82., 12., 9.])
```

### Quaadrature

**numpy.polynomial.hermite.hermgauss(deg)**

Gauss-Hermite quadrature.

Computes the sample points and weights for Gauss-Hermite quadrature. These sample points and weights will correctly integrate polynomials of degree \(2 \times \text{deg} - 1\) or less over the interval \([-\infty, \infty]\) with the weight function \(f(x) = \exp(-x^2)\).

**Parameters**

```
deg : int

Number of sample points and weights. It must be \(\geq 1\).
```

**Returns**

```
x : ndarray

1-D ndarray containing the sample points.

y : ndarray

1-D ndarray containing the weights.
```

**Notes**

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

\[
w_k = c/(H'_n(x_k) \ast H_{n-1}(x_k))
\]

where \(c\) is a constant independent of \(k\) and \(x_k\) is the \(k\)'th root of \(H_n\), and then scaling the results to get the right value when integrating 1.

**numpy.polynomial.hermite.hermweight(x)**

Weight function of the Hermite polynomials.

The weight function is \(\exp(-x^2)\) and the interval of integration is \([-\infty, \infty]\). the Hermite polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

```
x : array_like

Values at which the weight function will be computed.
```
NumPy Reference, Release 1.8.1

Returns

- `w`: ndarray
  
The weight function at `x`.

Notes

- `hermcompanion(c)`: Return the scaled companion matrix of `c`.
- `hermdomain`: 
- `hermzero`: 
- `hermone`: 
- `hermx`: 
- `hermtrim(c[, tol])`: Remove “small” “trailing” coefficients from a polynomial.
- `hermline(off, scl)`: Hermite series whose graph is a straight line.
- `herm2poly(c)`: Convert a Hermite series to a polynomial.
- `poly2herm(pol)`: Convert a polynomial to a Hermite series.

Miscellaneous

- `numpy.polynomial.hermite.hermcompanion(c)`: 
  
  Return the scaled companion matrix of `c`.

  The basis polynomials are scaled so that the companion matrix is symmetric when `c` is an Hermite basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

  Parameters

  - `c`: array_like
    
    1-D array of Hermite series coefficients ordered from low to high degree.

  Returns

  - `mat`: ndarray
    
    Scaled companion matrix of dimensions (deg, deg).

  Notes

- `numpy.polynomial.hermite.hermdomain`: 
- `numpy.polynomial.hermite.hermzero`: 
- `numpy.polynomial.hermite.hermone`: 
- `numpy.polynomial.hermite.hermx`: 
- `numpy.polynomial.hermite.hermtrim(c[, tol=0])`: Remove “small” “trailing” coefficients from a polynomial.

  “Small” means “small in absolute value” and is controlled by the parameter `tol`; “trailing” means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be “trimmed.”

  Parameters

  - `c`: array_like
    
    1-d array of coefficients, ordered from lowest order to highest.
tol : number, optional

Trailing (i.e., highest order) elements with absolute value less than or equal to tol (default value is zero) are removed.

Returns
trimmed : ndarray

1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

Raises
ValueError

If tol < 0

See Also:
trimseq

Examples

```python
>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([ 0., 0., 3., 0., 5.])
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j, 0.0010-0.001j])
```

numpy.polynomial.hermite.**hermline**(off, scl)

Hermite series whose graph is a straight line.

Parameters
off, scl : scalars

The specified line is given by off + scl*x.

Returns
y : ndarray

This module’s representation of the Hermite series for off + scl*x.

See Also:
polyline, chebline

Examples

```python
>>> from numpy.polynomial.hermite import hermline, hermval
>>> hermval(0,hermline(3, 2))
3.0
>>> hermval(1,hermline(3, 2))
5.0
```

numpy.polynomial.hermite.**herm2poly**(c)

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest to highest degree.
Parameters

\(c\) : array_like

1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

Returns

\(pol\) : ndarray

1-D array containing the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest order term to highest.

See Also:

poly2herm

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```python
>>> from numpy.polynomial.hermite import herm2poly
>>> herm2poly([ 1. , 2.75 , 0.5 , 0.375])
array([ 0., 1., 2., 3.])
```

```

numpy.polynomial.hermite.poly2herm(pol)

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

Parameters

\(pol\) : array_like

1-D array containing the polynomial coefficients

Returns

\(c\) : ndarray

1-D array containing the coefficients of the equivalent Hermite series.

See Also:

herm2poly

Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```python
>>> from numpy.polynomial.hermite_e import poly2herme
>>> poly2herme(np.arange(4))
array([ 1. , 2.75 , 0.5 , 0.375])
```

HermiteE Module, “Probabilists” (numpy.polynomial.hermite_e)

New in version 1.6.0. This module provides a number of objects (mostly functions) useful for dealing with HermiteE series, including a HermiteE class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, numpy.polynomial).
HermiteE Class
class numpy.polynomial.hermite_e.HermiteE(coef, domain=[-1, 1], window=[-1, 1])
A HermiteE series class.

HermiteE instances provide the standard Python numerical methods ‘+’, ‘-’, ‘*’, ‘/’, ‘%’, ‘divmod’, ‘***’, and ‘()’ as well as the listed methods.

Parameters
coef : array_like
    HermiteE coefficients, in increasing order. For example, (1, 2, 3) implies P_0 + 2P_1 + 3P_2 where the P_i are a graded polynomial basis.

domain : (2,) array_like, optional
    Domain to use. The interval [domain[0], domain[1]] is mapped to the interval [window[0], window[1]] by shifting and scaling. The default value is [-1,1].

window : (2,) array_like, optional
    Window, see domain for its use. The default value is [-1,1]. .. versionadded:: 1.6.0

Notes
It is important to specify the domain in many cases, for instance in fitting data, because many of the important properties of the polynomial basis only hold in a specified interval and consequently the data must be mapped into that interval in order to benefit.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coef</td>
<td>((N,) ndarray) HermiteE coefficients, from low to high.</td>
</tr>
<tr>
<td>domain</td>
<td>((2,) ndarray) Domain that is mapped to window.</td>
</tr>
<tr>
<td>window</td>
<td>((2,) ndarray) Window that domain is mapped to.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>(arg)</td>
</tr>
<tr>
<td>basis</td>
<td>(deg[, domain, window]) HermiteE polynomial of degree deg.</td>
</tr>
<tr>
<td>cast</td>
<td>(series[, domain, window]) Convert instance to equivalent HermiteE series.</td>
</tr>
<tr>
<td>convert</td>
<td>([domain, kind, window]) Convert to different class and/or domain.</td>
</tr>
<tr>
<td>copy</td>
<td>() Return a copy.</td>
</tr>
<tr>
<td>cutdeg</td>
<td>(deg) Truncate series to the given degree.</td>
</tr>
<tr>
<td>degree</td>
<td>() The degree of the series.</td>
</tr>
<tr>
<td>deriv</td>
<td>([m]) Differentiate.</td>
</tr>
<tr>
<td>fit</td>
<td>(x, y, deg[, domain, rcond, full, w, window]) Least squares fit to data.</td>
</tr>
<tr>
<td>fromroots</td>
<td>(roots[, domain, window]) Return HermiteE instance with specified roots.</td>
</tr>
<tr>
<td>has_samecoef</td>
<td>(other) Check if coefficients match.</td>
</tr>
<tr>
<td>has_samedomain</td>
<td>(other) Check if domains match.</td>
</tr>
<tr>
<td>has_sametype</td>
<td>(other) Check if types match.</td>
</tr>
<tr>
<td>has_samewindow</td>
<td>(other) Check if windows match.</td>
</tr>
<tr>
<td>identity</td>
<td>([domain, window]) Identity function.</td>
</tr>
<tr>
<td>integ</td>
<td>([m, k, lbnd]) Integrate.</td>
</tr>
<tr>
<td>linspace</td>
<td>([n, domain]) Return x,y values at equally spaced points in domain.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.163 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapparms()</code></td>
<td>Return the mapping parameters.</td>
</tr>
<tr>
<td><code>roots()</code></td>
<td>Return list of roots.</td>
</tr>
<tr>
<td><code>trim([tol])</code></td>
<td>Remove small leading coefficients</td>
</tr>
<tr>
<td><code>truncate(size)</code></td>
<td>Truncate series to length <code>size</code>.</td>
</tr>
</tbody>
</table>

```
HermiteE.__call__(arg)
```

```
static HermiteE.basis(deg, domain=[-1, 1], window=[-1, 1])
```

HermiteE polynomial of degree `deg`.

Returns an instance of the HermiteE polynomial of degree `d`.

**Parameters**

- `deg` : int
  Degree of the HermiteE polynomial. Must be >= 0.

- `domain` : array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain.

- `window` : array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the window.

**Returns**

- `p` : HermiteE instance

**Notes**

New in version 1.7.0.

```
static HermiteE.cast(series, domain=[-1, 1], window=[-1, 1])
```

Convert instance to equivalent HermiteE series.

The `series` is expected to be an instance of some polynomial series of one of the types supported by the `numpy.polynomial` module, but could be some other class that supports the `convert` method.

**Parameters**

- `series` : series
  The instance series to be converted.

- `domain` : array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain.

- `window` : array_like
  The resulting array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the window.

**Returns**

- `p` : HermiteE instance
  A HermiteE series equal to the `poly` series.
See Also:
convert

Notes
New in version 1.7.0.

HermiteE.convert (domain=None, kind=None, window=None)
Convert to different class and/or domain.

Parameters

domain : array_like, optional
  The domain of the converted series. If the value is None, the default domain of kind is used.

kind : class, optional
  The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

window : array_like, optional
  The window of the converted series. If the value is None, the default window of kind is used.

Returns

new_series_instance : kind
  The returned class can be of different type than the current instance and/or have a different domain.

Notes
Conversion between domains and class types can result in numerically ill-posed series.

HermiteE.copy ()
Return a copy.

Returns

new_instance : HermiteE
  Copy of current instance.

HermiteE.cutdeg (deg)
Truncate series to the given degree.

Reduce the degree of the HermiteE series to deg by discarding the high order terms. If deg is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

Parameters

deg : non-negative int
  The series is reduced to degree deg by discarding the high order terms. The value of deg must be a non-negative integer.

Returns

new_instance : HermiteE
  New instance of HermiteE with reduced degree.
Notes
New in version 1.5.0.

HermiteE\texttt{.degree}()

The degree of the series.

Notes
New in version 1.5.0.

HermiteE\texttt{.deriv}(m=1)

Differentiate.

Return an instance of HermiteE that is the derivative of the current series. Refer to hermeder for full documentation.

Parameters
m : non-negative int

The number of integrations to perform.

Returns
derivative : HermiteE

The derivative of the series using the same domain.

See Also:

hermeder
similar function.

hermeint
similar function for integration.

static HermiteE\texttt{.fit}(x, y, deg, domain=\texttt{None}, rcond=\texttt{None}, full=\texttt{False}, w=\texttt{None}, window=\texttt{[-1, 1]})

Least squares fit to data.

Return a HermiteE instance that is the least squares fit to the data $y$ sampled at $x$. Unlike hermefit, the domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning. Support for NA was added in version 1.7.0. See hermefit for full documentation of the implementation.

Parameters
x : array_like, shape (M,)

x-coordinates of the $M$ sample points ($x[i]$, $y[i]$).

y : array_like, shape (M,) or (M, K)

y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

deg : int

Degree of the fitting polynomial.

domain : {\texttt{None}, [beg, end], []}, optional

Domain to use for the returned HermiteE instance. If \texttt{None}, then a minimal domain that covers the points $x$ is chosen. If [] the default domain [-1, 1] is used. The default value is [-1,1] in numpy 1.4.x and \texttt{None} in later versions. The ‘[]’ value was added in numpy 1.5.0.
**rcond** : float, optional

Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full** : bool, optional

Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w** : array_like, shape (M,), optional

Weights. If not None the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i]*y[i]\) all have the same variance. The default value is None. .. versionadded:: 1.5.0

**window** : ([beg, end]), optional

Window to use for the returned HermiteE instance. The default value is \([-1, 1]\) .. versionadded:: 1.6.0

**Returns**

- **least_squares_fit** : instance of HermiteE

  The HermiteE instance is the least squares fit to the data and has the domain specified in the call.

- **[residuals, rank, singular_values, rcond]** : only if full = True

  Residuals of the least squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of rcond. For more details, see linalg.lstsq.

**See Also:**

- **hermefit**

  similar function

**static** `HermiteE.fromroots(roots, domain=[-1, 1], window=[-1, 1])`

Return HermiteE instance with specified roots.

Returns an instance of HermiteE representing the product \((x - r[0])*(x - r[1])*\ldots*(x - r[n-1])\), where \(r\) is the list of roots.

**Parameters**

- **roots** : array_like

  List of roots.

- **domain** : {array_like, None}, optional

  Domain for the resulting instance of HermiteE. If none the domain is the interval from the smallest root to the largest. The default is [-1,1].

- **window** : array_like, optional

  Window for the resulting instance of HermiteE. The default value is [-1,1].

**Returns**

- **object** : HermiteE instance
Series with the specified roots.

See Also:

```
hermefromroots
```
equivalent function

```
HermiteE.has_samecoef(other)
```
Check if coefficients match.

**Parameters**

- `other`: class instance

  The other class must have the `coef` attribute.

**Returns**

- `bool`: boolean

  True if the coefficients are the same, False otherwise.

**Notes**

New in version 1.6.0.

```
HermiteE.has_samedomain(other)
```
Check if domains match.

**Parameters**

- `other`: class instance

  The other class must have the `domain` attribute.

**Returns**

- `bool`: boolean

  True if the domains are the same, False otherwise.

**Notes**

New in version 1.6.0.

```
HermiteE.has_sametype(other)
```
Check if types match.

**Parameters**

- `other`: object

  Class instance.

**Returns**

- `bool`: boolean

  True if other is same class as self

**Notes**

New in version 1.7.0.

```
HermiteE.has_samewindow(other)
```
Check if windows match.

**Parameters**

- `other`: class instance

  The other class must have the `window` attribute.
Returns

boolean: boolean

True if the windows are the same, False otherwise.

Notes

New in version 1.6.0.

static HermiteE. identity (domain=[-1, 1], window=[-1, 1])

Identity function.

If \( p \) is the returned HermiteE object, then \( p(x) = x \) for all values of \( x \).

Parameters

domain: array_like

The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the domain.

window: array_like

The resulting array must be of the form \([\text{beg}, \text{end}]\), where \( \text{beg} \) and \( \text{end} \) are the endpoints of the window.

Returns

identity: HermiteE instance

HermiteE. integ (m=1, k=[], lbnd=None)

Integrate.

Return an instance of HermiteE that is the definite integral of the current series. Refer to hermeint for full documentation.

Parameters

m: non-negative int

The number of integrations to perform.

k: array_like

Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to \( m \) in length and any missing values are set to zero.

lbnd: Scalar

The lower bound of the definite integral.

Returns

integral: HermiteE

The integral of the series using the same domain.

See Also:

hermeint

similar function.

hermeder

similar function for derivative.

HermiteE.linspace (n=100, domain=None)

Return \( x,y \) values at equally spaced points in domain.
Returns x, y values at n linearly spaced points across domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the HermiteE instance. This method is intended mostly as a plotting aid.

**Parameters**

- **n**: int, optional
  
  Number of point pairs to return. The default value is 100.

- **domain**: {None, array_like}
  
  If not None, the specified domain is used instead of that of the calling instance. It should be of the form \([\text{beg}, \text{end}]\). The default is None.

**Returns**

- **x**, **y**: ndarrays
  
  x is equal to linspace(self.domain[0], self.domain[1], n)
  y is the polynomial evaluated at x.

New in version 1.5.0.

**HermiteE.mapparms**()

Return the mapping parameters.

The returned values define a linear map \(\text{off} + \text{scl} \times x\) that is applied to the input arguments before the series is evaluated. The map depends on the domain and window; if the current domain is equal to the window the resulting map is the identity. If the coefficients of the HermiteE instance are to be used by themselves outside this class, then the linear function must be substituted for the \(x\) in the standard representation of the base polynomials.

**Returns**

- **off**, **scl**: floats or complex

  The mapping function is defined by \(\text{off} + \text{scl} \times x\).

**Notes**

If the current domain is the interval \([l_1, r_1]\) and the window is \([l_2, r_2]\), then the linear mapping function \(L\) is defined by the equations:

\[
\begin{align*}
L(l_1) &= l_2 \\
L(r_1) &= r_2
\end{align*}
\]

**HermiteE.roots**()

Return list of roots.

Return ndarray of roots for this series. See `hermeroots` for full documentation. Note that the accuracy of the roots is likely to decrease the further outside the domain they lie.

**See Also:**

- `hermeroots` similar function
- `hermefromroots` function to go generate series from roots.

**HermiteE.trim**(*tol=0*)

Remove small leading coefficients
Remove leading coefficients until a coefficient is reached whose absolute value greater than \( tol \) or the beginning of the series is reached. If all the coefficients would be removed the series is set to \([0]\). A new HermiteE instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

tol : non-negative number.

All trailing coefficients less than \( tol \) will be removed.

**Returns**

new_instance : HermiteE

Contains the new set of coefficients.

HermiteE\(\text{.truncate}(\text{size})\)

Truncate series to length \( size \).

Reduce the HermiteE series to length \( size \) by discarding the high degree terms. The value of \( size \) must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

size : positive int

The series is reduced to length \( size \) by discarding the high degree terms. The value of \( size \) must be a positive integer.

**Returns**

new_instance : HermiteE

New instance of HermiteE with truncated coefficients.

**Basics**

numpy.polynomial.hermite_e.\(\text{hermeval}(x, c[, \text{tensor}])\)  Evaluate an HermiteE series at points \( x \).

If \( c \) is of length \( n + 1 \), this function returns the value:

\[
p(x) = c_0 \ast H_{e_0}(x) + c_1 \ast H_{e_1}(x) + \ldots + c_n \ast H_{e_n}(x)
\]

The parameter \( x \) is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either \( x \) or its elements must support multiplication and addition both with themselves and with the elements of \( c \).

If \( c \) is a 1-D array, then \( p(x) \) will have the same shape as \( x \). If \( c \) is multidimensional, then the shape of the result depends on the value of tensor. If tensor is true the shape will be \( c\text{.shape}[1:] + x\text{.shape} \). If tensor is false the shape will be \( c\text{.shape}[1:] \). Note that scalars have shape (1,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.
NumPy Reference, Release 1.8.1

Parameters

x : array_like, compatible object

If x is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and
treated as a scalar. In either case, x or its elements must support addition and multipli-
cation with with themselves and with the elements of c.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree n are contained
in c[n]. If c is multidimensional the remaining indices enumerate multiple polynomials.
In the two dimensional case the coefficients may be thought of as stored in the columns
of c.

tensor : boolean, optional

If True, the shape of the coefficient array is extended with ones on the right, one for each
dimension of x. Scalars have dimension 0 for this action. The result is that every column
of coefficients in c is evaluated for every element of x. If False, x is broadcast over the
columns of c for the evaluation. This keyword is useful when c is multidimensional.
The default value is True. New in version 1.7.0.

Returns

values : ndarray, algebra_like

The shape of the return value is described above.

See Also:

hermeval2d, hermepoly2d, hermeval3d, hermegrid3d

Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

Examples

```python
>>> from numpy.polynomial.hermite_e import hermeval
>>> coef = [1,2,3]
>>> hermeval(1, coef)
3.0
>>> hermeval([[1,2],[3,4]], coef)
array([[ 3., 14.],
       [31., 54.]])
```

Evaluate a 2-D HermiteE series at points (x, y).

This function returns the values:

\[
p(x, y) = \sum_{i,j} c_{i,j} \cdot He_i(x) \cdot He_j(y)
\]

The parameters x and y are converted to arrays only if they are tuples or a lists, otherwise they are treated as a
scalars and they must have the same shape after conversion. In either case, either x and y or their elements must
support multiplication and addition both with themselves and with the elements of c.

If c is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be
c.shape[2:] + x.shape.

Parameters

x, y : array_like, compatible objects
The two dimensional series is evaluated at the points \((x, y)\), where \(x\) and \(y\) must have the same shape. If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

\[ c : \text{array_like} \]

Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j\) is contained in \(c[i, j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns
values : ndarray, compatible object

The values of the two dimensional polynomial at points formed with pairs of corresponding values from \(x\) and \(y\).

See Also:
hermeval, hermegovgrid2d, hermeval3d, hermegrid3d

Notes
numpy.polynomial.hermite_e.hermeval3d(x, y, z, c)

Evaluate a 3-D Hermite_e series at points \((x, y, z)\).

This function returns the values:

\[ p(x, y, z) = \sum_{i,j,k} c_{i,j,k} * He_i(x) * He_j(y) * He_k(z) \]

The parameters \(x\), \(y\), and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as scalars and they must have the same shape after conversion. In either case, either \(x\), \(y\), and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c.shape[3:] + x.shape\).

Parameters
\[ x, y, z : \text{array_like, compatible object} \]

The three dimensional series is evaluated at the points \((x, y, z)\), where \(x\), \(y\), and \(z\) must have the same shape. If any of \(x\), \(y\), or \(z\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn’t an ndarray it is treated as a scalar.

\[ c : \text{array_like} \]

Array of coefficients ordered so that the coefficient of the term of multi-degree \(i,j,k\) is contained in \(c[i, j, k]\). If \(c\) has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

Returns
values : ndarray, compatible object

The values of the multidimensional polynomial on points formed with triples of corresponding values from \(x\), \(y\), and \(z\).

See Also:
hermeval, hermeval2d, hermegrid2d, hermegrid3d

3.26. Polynomials
**Notes**

numpy.polynomial.hermite_e.hermegrid2d(x, y, c)

Evaluate a 2-D HermiteE series on the Cartesian product of x and y.

This function returns the values:

\[ p(a, b) = \sum_{i,j} c_{i,j} \times H_i(a) \times H_j(b) \]

where the points \((a, b)\) consist of all pairs formed by taking \(a\) from \(x\) and \(b\) from \(y\). The resulting points form a grid with \(x\) in the first dimension and \(y\) in the second.

The parameters \(x\) and \(y\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either \(x\) and \(y\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be \(c.shape[2:] + x.shape\).

**Parameters**
- \(x, y\) : array_like, compatible objects
  - The two dimensional series is evaluated at the points in the Cartesian product of \(x\) and \(y\).
  - If \(x\) or \(y\) is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

- \(c\) : array_like
  - Array of coefficients ordered so that the coefficients for terms of degree \(i,j\) are contained in \(c[i, j]\). If \(c\) has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**
- \(values\) : ndarray, compatible object
  - The values of the two dimensional polynomial at points in the Cartesian product of \(x\) and \(y\).

**See Also:**

hermeval, hermeval2d, hermeval3d, hermegrid3d

**Notes**

numpy.polynomial.hermite_e.hermegrid3d(x, y, z, c)

Evaluate a 3-D HermiteE series on the Cartesian product of \(x, y,\) and \(z\).

This function returns the values:

\[ p(a, b, c) = \sum_{i,j,k} c_{i,j,k} \times He_i(a) \times He_j(b) \times He_k(c) \]

where the points \((a, b, c)\) consist of all triples formed by taking \(a\) from \(x\), \(b\) from \(y\), and \(c\) from \(z\). The resulting points form a grid with \(x\) in the first dimension, \(y\) in the second, and \(z\) in the third.

The parameters \(x, y,\) and \(z\) are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either \(x, y,\) and \(z\) or their elements must support multiplication and addition both with themselves and with the elements of \(c\).

If \(c\) has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be \(c.shape[3:] + x.shape + y.shape + z.shape\).
Parameters

x, y, z : array_like, compatible objects

The three dimensional series is evaluated at the points in the Cartesian product of x, y, and z. If x, y, or z is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

c : array_like

Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in c[i, j]. If c has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

Returns

values : ndarray, compatible object

The values of the two dimensional polynomial at points in the Cartesian product of x and y.

See Also:

hermeval, hermeval2d, hermegrid2d, hermeval3d

Notes

numpy.polynomial.hermite_e.hermeroots(c)

Compute the roots of a HermiteE series.

Return the roots (a.k.a. “zeros”) of the polynomial

\[ p(x) = \sum_i c[i] \cdot H_e_i(x). \]

Parameters

c : 1-D array_like

1-D array of coefficients.

Returns

out : ndarray

Array of the roots of the series. If all the roots are real, then out is also real, otherwise it is complex.

See Also:

polyroots, legroots, lagroots, hermroots, chebroots

Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton’s method.

The HermiteE series basis polynomials aren’t powers of x so the results of this function may seem unintuitive.

Examples
>>> from numpy.polynomial.hermite_e import hermeroots, hermefromroots
>>> coef = hermefromroots([-1, 0, 1])
>>> coef
array([ 0., 2., 0., 1.])
>>> hermeroots(coef)
array([-1., 0., 1.])

numpy.polynomial.hermite_e.hermefromroots(roots)
Generate a HermiteE series with given roots.

The function returns the coefficients of the polynomial

\[ p(x) = (x - r_0) \ast (x - r_1) \ast \ldots \ast (x - r_n), \]

in HermiteE form, where the \( r_n \) are the roots specified in \( \text{roots} \). If a zero has multiplicity \( n \), then it must appear in \( \text{roots} \) \( n \) times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then \( \text{roots} \) looks something like \([2, 2, 2, 3, 3]\). The roots can appear in any order.

If the returned coefficients are \( c \), then

\[ p(x) = c_0 + c_1 \ast H_1(x) + \ldots + c_n \ast H_n(x) \]

The coefficient of the last term is not generally 1 for monic polynomials in HermiteE form.

Parameters
- roots : array_like
  Sequence containing the roots.

Returns
- out : ndarray
  1-D array of coefficients. If all roots are real then \( \text{out} \) is a real array, if some of the roots are complex, then \( \text{out} \) is complex even if all the coefficients in the result are real (see Examples below).

See Also:
- polyfromroots, legfromroots, lagfromroots, hermfromroots, chebfromroots.

Examples

```python
>>> from numpy.polynomial.hermite_e import hermefromroots, hermeval
>>> coef = hermefromroots((-1, 0, 1))
>>> hermeval((-1, 0, 1), coef)
array([ 0., 0., 0.])
>>> coef = hermefromroots((-1j, 1j))
>>> hermeval((-1j, 1j), coef)
array([ 0.+0.j, 0.+0.j])
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermefit</td>
<td>Least squares fit of Hermite series to data.</td>
</tr>
<tr>
<td>hermevander</td>
<td>Pseudo-Vandermonde matrix of given degree.</td>
</tr>
<tr>
<td>hermevander2d</td>
<td>Pseudo-Vandermonde matrix of given degrees.</td>
</tr>
<tr>
<td>hermevander3d</td>
<td>Pseudo-Vandermonde matrix of given degrees.</td>
</tr>
</tbody>
</table>
Fitting

```
numpy.polynomial.hermite_e.hermefit(x, y, deg, rcond=None, full=False, w=None)
```

Least squares fit of Hermite series to data.

Return the coefficients of a HermiteE series of degree `deg` that is the least squares fit to the data values `y` given at points `x`. If `y` is 1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple fits are done, one for each column of `y`, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

\[ p(x) = c_0 + c_1 \cdot He_1(x) + \ldots + c_n \cdot He_n(x), \]

where \( n \) is `deg`.

Since numpy version 1.7.0, hermefit also supports NA. If any of the elements of `x`, `y`, or `w` are NA, then the corresponding rows of the linear least squares problem (see Notes) are set to 0. If `y` is 2-D, then an NA in any row of `y` invalidates that whole row.

**Parameters**

- `x` : array_like, shape (M,)
  
  x-coordinates of the M sample points \((x[i], y[i])\).
  
- `y` : array_like, shape (M,) or (M, K)
  
  y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.
  
- `deg` : int
  
  Degree of the fitting polynomial
  
- `rcond` : float, optional
  
  Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is `len(x)*eps`, where `eps` is the relative precision of the float type, about 2e-16 in most cases.
  
- `full` : bool, optional
  
  Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.
  
- `w` : array_like, shape (M,), optional
  
  Weights. If not None, the contribution of each point \((x[i], y[i])\) to the fit is weighted by \(w[i]\). Ideally the weights are chosen so that the errors of the products \(w[i] \cdot y[i]\) all have the same variance. The default value is None.

**Returns**

- `coef` : ndarray, shape (M,) or (M, K)
  
  Hermite coefficients ordered from low to high. If `y` was 2-D, the coefficients for the data in column k of `y` are in column `k`.
  
  `residuals`, `rank`, `singular_values`, `rcond` : present when `full = True`
  
  Residuals of the least-squares fit, the effective rank of the scaled Vandermonde matrix and its singular values, and the specified value of `rcond`. For more details, see `linalg.lstsq`.

3.26. Polynomials 1105
warns

Warns

RankWarning

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if full = False. The warnings can be turned off by

```python
>>> import warnings
>>> warnings.simplefilter('ignore', RankWarning)
```

See Also:

chebfit, legfit, polyfit, hermfit, polyfit

hermeval

Evaluates a Hermite series.

hermevander

pseudo Vandermonde matrix of Hermite series.

hermeweight

HermiteE weight function.

linalg.lstsq

Computes a least-squares fit from the matrix.

scipy.interpolate.UnivariateSpline

Computes spline fits.

Notes

The solution is the coefficients of the HermiteE series \( p \) that minimizes the sum of the weighted squared errors

\[
E = \sum_j w_j^2 \cdot |y_j - p(x_j)|^2,
\]

where the \( w_j \) are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

\[
V(x) \cdot c = w \cdot y,
\]

where \( V \) is the pseudo Vandermonde matrix of \( x \), the elements of \( c \) are the coefficients to be solved for, and the elements of \( y \) are the observed values. This equation is then solved using the singular value decomposition of \( V \).

If some of the singular values of \( V \) are so small that they are neglected, then a \texttt{RankWarning} will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The \texttt{rcond} parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using HermiteE series are probably most useful when the data can be approximated by \( \sqrt{w(x)} \cdot p(x) \), where \( w(x) \) is the HermiteE weight. In that case the weight \( \sqrt{w(x[i])} \) should be used together with data values \( y[i]/\sqrt{w(x[i])} \). The weight function is available as \texttt{hermeweight}.

References

[R60]

Examples
from numpy.polynomial.hermite_e import hermefik, hermeval
x = np.linspace(-10, 10)
err = np.random.randn(len(x))/10
y = hermeval(x, [1, 2, 3]) + err
hermefit(x, y, 2)
array([1.01690445, 1.99951418, 2.99948696])

numpy.polynomial.hermite_e.hermevander(x, deg)
Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree deg and sample points x. The pseudo-Vandermonde matrix is defined by

\[ V[\ldots, i] = He_i(x), \]

where \( 0 \leq i \leq \text{deg} \). The leading indices of \( V \) index the elements of \( x \) and the last index is the degree of the Hermite polynomials.

If \( c \) is a 1-D array of coefficients of length \( n + 1 \) and \( V \) is the array \( V = \text{hermevander}(x, n) \), then \( \text{np.dot}(V, c) \) and \( \text{hermeval}(x, c) \) are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Hermite polynomials of the same degree and sample points.

Parameters
x : array_like
   Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If \( x \) is scalar it is converted to a 1-D array.

deg : int
   Degree of the resulting matrix.

Returns
vander : ndarray
   The pseudo-Vandermonde matrix. The shape of the returned matrix is \( x.shape + (\text{deg} + 1, \) , where the last index is the degree of the corresponding Hermite polynomials. The dtype will be the same as the converted \( x \).

Examples

numpy.polynomial.hermite_e.hermevander2d(x, y, deg)
Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees deg and sample points \((x, y)\). The pseudo-Vandermonde matrix is defined by

\[ V[\ldots, \text{deg}[1] \times i + j] = He_i(x) \times He_j(y), \]

where \( 0 \leq i \leq \text{deg[0]} \) and \( 0 \leq j \leq \text{deg[1]} \). The leading indices of \( V \) index the points \((x, y)\) and the last index encodes the degrees of the Hermite polynomials.
If \( V = \text{hermenvander2d}(x, y, [x_{\text{deg}}, y_{\text{deg}}]) \), then the columns of \( V \) correspond to the elements of a 2-D coefficient array \( c \) of shape \((x_{\text{deg}} + 1, y_{\text{deg}} + 1)\) in the order

\[
    c_{00}, c_{01}, c_{02}, \ldots, c_{10}, c_{11}, c_{12}, \ldots
\]

and \( \text{np.dot}(V, c.\text{flat}) \) and \( \text{hermeval2d}(x, y, c) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D HermiteE series of the same degrees and sample points.

**Parameters**
- \( x, y: \text{array_like} \)
  
  Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

- \( \text{deg} : \text{list of ints} \)
  
  List of maximum degrees of the form \([x_{\text{deg}}, y_{\text{deg}}]\).

**Returns**
- \( \text{vander2d} : \text{ndarray} \)
  
  The shape of the returned matrix is \( x.\text{shape} + (\text{order},) \), where \( \text{order} = (\text{deg}[0] + 1) \ast (\text{deg}[1] + 1) \). The dtype will be the same as the converted \( x \) and \( y \).

**See Also:**
- \( \text{hermenvander}, \text{hermenvander3d}, \text{hermeval3d} \)

**Notes**

\( \text{numpy.polynomial.hermite_e.hermevander3d}(x, y, z, \text{deg}) \)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees \( \text{deg} \) and sample points \((x, y, z)\). If \( l, m, n \) are the given degrees in \( x, y, z \), then HeHe pseudo-Vandermonde matrix is defined by

\[
    V[\ldots,(m + 1)(n + 1)i + (n + 1)j + k] = He_i(x) \ast He_j(y) \ast He_k(z),
\]

where \( 0 <= i <= l, 0 <= j <= m, \) and \( 0 <= j <= n \). The leading indices of \( V \) index the points \((x, y, z)\) and the last index encodes the degrees of the HermiteE polynomials.

If \( V = \text{hermenvander3d}(x, y, z, [x_{\text{deg}}, y_{\text{deg}}, z_{\text{deg}}]) \), then the columns of \( V \) correspond to the elements of a 3-D coefficient array \( c \) of shape \((x_{\text{deg}} + 1, y_{\text{deg}} + 1, z_{\text{deg}} + 1)\) in the order

\[
    c_{000}, c_{001}, c_{002}, \ldots, c_{010}, c_{011}, c_{012}, \ldots
\]

and \( \text{np.dot}(V, c.\text{flat}) \) and \( \text{hermeval3d}(x, y, z, c) \) will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D HermiteE series of the same degrees and sample points.

**Parameters**
- \( x, y, z : \text{array_like} \)
  
  Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.
deg : list of ints

List of maximum degrees of the form [x_deg, y_deg, z_deg].

Returns

vander3d : ndarray

The shape of the returned matrix is x.shape + (order,), where order = (deg[0] + 1) * (deg[1] + 1) * (deg[2] + 1). The.dtype will be the same as the converted x, y, and z.

See Also:
hermevander, hermevander3d, hermeval3d

Notes

Calculus

numpy.polynomial.hermite_e.hermder(c, m=1, scl=1, axis=0)

Differentiate a Hermite_e series.

Returns the series coefficients c differentiated m times along axis. At each iteration the result is multiplied by scl (the scaling factor is for use in a linear change of variable). The argument c is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series 1*He_0 + 2*He_1 + 3*He_2 while [[1,2],[1,2]] represents 1*He_0(x)*He_0(y) + 1*He_1(x)*He_0(y) + 2*He_0(x)*He_1(y) + 2*He_1(x)*He_1(y) if axis=0 is x and axis=1 is y.

Parameters

c : array_like

Array of Hermite_e series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

m : int, optional

Number of derivatives taken, must be non-negative. (Default: 1)

scl : scalar, optional

Each differentiation is multiplied by scl. The end result is multiplication by scl**m. This is for use in a linear change of variable. (Default: 1)

axis : int, optional

Axis over which the derivative is taken. (Default: 0). New in version 1.7.0.

Returns

der : ndarray

Hermite series of the derivative.

See Also:
hermeint

Notes

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be “unintuitive,” albeit correct; see Examples section below.
Examples

```python
>>> from numpy.polynomial.hermite_e import hermeder
>>> hermeder([ 1.,  1.,  1.,  1.])
array([ 1.,  2.,  3.,  4.])
>>> hermeder([-0.25,  1.,  1./2.,  1./3.,  1./4 ], m=2)
array([ 1.,  2.,  3.])
```

numpy.polynomial.hermite_e.
hermeint(c, m=1, k=[], lbnd=0, scl=1, axis=0)

Integrate a Hermite_e series.

Returns the Hermite_e series coefficients c integrated m times from lbnd along axis. At each iteration the resulting series is multiplied by scl and an integration constant, k, is added. The scaling factor is for use in a linear change of variable. (“Buyer beware”: note that, depending on what one is doing, one may want scl to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument c is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series \( H_0 + 2H_1 + 3H_2 \) while [[1,2],[1,2]] represents \( 1H_0(x)H_0(y) + 1H_1(x)H_0(y) + 2H_0(x)H_1(y) + 2H_1(x)H_1(y) \) if axis=0 is x and axis=1 is y.

Parameters

- **c**: array_like
  Array of Hermite_e series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

- **m**: int, optional
  Order of integration, must be positive. (Default: 1)

- **k**: list, scalar, optional
  Integration constant(s). The value of the first integral at lbnd is the first value in the list, the value of the second integral at lbnd is the second value, etc. If k == [] (the default), all constants are set to zero. If m == 1, a single scalar can be given instead of a list.

- **lbnd**: scalar, optional
  The lower bound of the integral. (Default: 0)

- **scl**: scalar, optional
  Following each integration the result is multiplied by scl before the integration constant is added. (Default: 1)

- **axis**: int, optional
  Axis over which the integral is taken. (Default: 0). New in version 1.7.0.

Returns

- **S**: ndarray
  Hermite_e series coefficients of the integral.

Raises

- **ValueError**
  If \( m < 0 \), \( \text{len(k)} > m \), np.isscalar(lbnd) == False, or np.isscalar(scl) == False.

See Also:

hermeder
Notes

Note that the result of each integration is multiplied by scl. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set scl equal to $1/a$—perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be “reprojected” onto the C-series basis set. Thus, typically, the result of this function is “unintuitive,” albeit correct; see Examples section below.

Examples

```python
>>> from numpy.polynomial.hermite_e import hermeint
>>> hermeint([1, 2, 3]) # integrate once, value 0 at 0.
array([ 1., 1., 1., 1.])
>>> hermeint([1, 2, 3], m=2) # integrate twice, value & deriv 0 at 0
array([-0.25 , 1. , 0.5 , 0.33333333, 0.25 ])
>>> hermeint([1, 2, 3], k=1) # integrate once, value 1 at 0.
array([ 2., 1., 1., 1.])
>>> hermeint([1, 2, 3], lbnd=-1) # integrate once, value 0 at -1
array([ 1.83333333, 0. , 0.5 , 0.33333333, 0.25 ])
```

**hermeadd**(c1, c2) Add one Hermite series to another.

```
hermeadd(c1, c2) Add one Hermite series to another.
hermesub(c1, c2) Subtract one Hermite series from another.
hermemul(c1, c2) Multiply one Hermite series by another.
hermemulx(c) Multiply a Hermite series by x.
hermediv(c1, c2) Divide one Hermite series by another.
hermepow(c, pow[, maxpower]) Raise a Hermite series to a power.
```

Algebra

```
numpy.polynomial.hermite_e.hermeadd(c1, c2)
Add one Hermite series to another.

Returns

Returns

1-D arrays of Hermite series coefficients ordered from low to high.
```

Parameters

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

See Also:

hermesub, hermemul, hermediv, hermepow

Notes

Unlike multiplication, division, etc., the sum of two Hermite series is a Hermite series (without having to “re-project” the result onto the basis set) so addition, just like that of “standard” polynomials, is simply “component-wise.”
Examples

```python
>>> from numpy.polynomial.hermite_e import hermeadd
>>> hermeadd([1, 2, 3], [1, 2, 3, 4])
array([ 2., 4., 6., 4.])
```

```
numpy.polynomial.hermite_e.hermesub(cl, c2)

Subtract one Hermite series from another.

Returns the difference of two Hermite series cl - c2. The sequences of coefficients are from lowest order term
to highest, i.e., [1,2,3] represents the series P_0 + 2*P_1 + 3*P_2.

Parameters

    c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

    out : ndarray

Of Hermite series coefficients representing their difference.

See Also:

hermeadd, hermemul, hermediv, hermepow

Notes

Unlike multiplication, division, etc., the difference of two Hermite series is a Hermite series (without having
to “reproject” the result onto the basis set) so subtraction, just like that of “standard” polynomials, is simply
“component-wise.”

Examples

```python
>>> from numpy.polynomial.hermite_e import hermesub
>>> hermesub([1, 2, 3, 4], [1, 2, 3])
array([ 0., 0., 0., 4.])
```

```
numpy.polynomial.hermite_e.hermemul(cl, c2)

Multiply one Hermite series by another.

Returns the product of two Hermite series cl * c2. The arguments are sequences of coefficients, from lowest
order “term” to highest, e.g., [1,2,3] represents the series P_0 + 2*P_1 + 3*P_2.

Parameters

    c1, c2 : array_like

1-D arrays of Hermite series coefficients ordered from low to high.

Returns

    out : ndarray

Of Hermite series coefficients representing their product.

See Also:

hermeadd, hermesub, hermediv, hermepow

Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis
set. Thus, to express the product as a Hermite series, it is necessary to “reproject” the product onto said basis
set, which may produce “unintuitive” (but correct) results; see Examples section below.
Examples

```python
>>> from numpy.polynomial.hermite_e import hermemul
>>> hermemul([1, 2, 3], [0, 1, 2])
array([ 14., 15., 28.,  7.,  6.])
```

**numpy.polynomial.hermite_e.**

**hermemulx(c)**

Multiply a Hermite series by x.

Multiply the Hermite series c by x, where x is the independent variable.

**Parameters**

- **c**: array_like
  - 1-D array of Hermite series coefficients ordered from low to high.

**Returns**

- **out**: ndarray
  - Array representing the result of the multiplication.

**Notes**

The multiplication uses the recursion relationship for Hermite polynomials in the form

\[ xP_i(x) = (P_{i + 1}(x) + iP_{i - 1}(x)) \]

**Examples**

```python
>>> from numpy.polynomial.hermite_e import hermemulx
>>> hermemulx([1, 2, 3])
array([ 2.,  7.,  2.,  3.])
```

**numpy.polynomial.hermite_e.**

**hermediv(c1, c2)**

Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series c1/c2. The arguments are sequences of coefficients from lowest order “term” to highest, e.g., [1,2,3] represents the series \( P_0 + 2*P_1 + 3*P_2 \).

**Parameters**

- **c1, c2**: array_like
  - 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

- **[quo, rem]**: ndarrays
  - Of Hermite series coefficients representing the quotient and remainder.

**See Also:**

hermeadd, hermesub, hermemul, hermepow

**Notes**

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to “reproject” the results onto the Hermite basis set, which may produce “unintuitive” (but correct) results; see Examples section below.
Examples

```python
>>> from numpy.polynomial.hermite_e import hermediv
>>> hermediv([14., 15., 28., 7., 6.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> hermediv([15., 17., 28., 7., 6.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 2.]))
```

numpy.polynomial.hermite_e.hermepow(c, pow, maxpower=16)

Raise a Hermite series to a power.

Returns the Hermite series \( c \) raised to the power \( \text{pow} \). The argument \( c \) is a sequence of coefficients ordered from low to high. i.e., \([1,2,3]\) is the series \( P_0 + 2*P_1 + 3*P_2 \).

Parameters
- `c`: array_like
  1-D array of Hermite series coefficients ordered from low to high.
- `pow`: integer
  Power to which the series will be raised
- `maxpower`: integer, optional
  Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

Returns
- `coef`: ndarray
  Hermite series of power.

See Also:
- hermeadd, hermesub, hermemul, hermediv

Examples

```python
>>> from numpy.polynomial.hermite_e import hermepow
>>> hermepow([1, 2, 3], 2)
array([ 23., 28., 46., 12., 9.])
```

hermegauss(deg) Gauss-HermiteE quadrature.

Computes the sample points and weights for Gauss-HermiteE quadrature. These sample points and weights will correctly integrate polynomials of degree \( 2 \times \text{deg} - 1 \) or less over the interval \([-\infty, \infty]\) with the weight function \( f(x) = \exp(-x^2/2) \).

Parameters
- `deg`: int
  Number of sample points and weights. It must be \( \geq 1 \).

Returns
- `x`: ndarray

hermeweight(x) Weight function of the Hermite_e polynomials.
1-D ndarray containing the sample points.

y : ndarray
1-D ndarray containing the weights.

Notes
The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

\[ w_k = c / (H'_n(x_k) * H_{n-1}(x_k)) \]

where \( c \) is a constant independent of \( k \) and \( x_k \) is the \( k \)’th root of \( H_n \), and then scaling the results to get the right value when integrating 1.

numpy.polynomial.hermite_e.hermeweight(x)
Weight function of the Hermite_e polynomials.
The weight function is \( \exp(-x^2/2) \) and the interval of integration is \([ -\infty, \infty ]\). the HermiteE polynomials are orthogonal, but not normalized, with respect to this weight function.

Parameters
x : array_like
Values at which the weight function will be computed.

Returns
w : ndarray
The weight function at \( x \).

Notes

<table>
<thead>
<tr>
<th>hermecompanion(c)</th>
<th>Return the scaled companion matrix of ( c ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>hermedomain</td>
<td></td>
</tr>
<tr>
<td>hermezero</td>
<td></td>
</tr>
<tr>
<td>hermeone</td>
<td></td>
</tr>
<tr>
<td>hermex</td>
<td></td>
</tr>
<tr>
<td>hermetrim(c[, tol])</td>
<td>Remove “small” “trailing” coefficients from a polynomial.</td>
</tr>
<tr>
<td>hermeline(off, scl)</td>
<td>Hermite series whose graph is a straight line.</td>
</tr>
<tr>
<td>herme2poly(c)</td>
<td>Convert a Hermite series to a polynomial.</td>
</tr>
<tr>
<td>poly2herme(pol)</td>
<td>Convert a polynomial to a Hermite series.</td>
</tr>
</tbody>
</table>

Miscellaneous
numpy.polynomial.hermite_e.hermecompanion(c)
Return the scaled companion matrix of \( c \).

The basis polynomials are scaled so that the companion matrix is symmetric when \( c \) is an HermiteE basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if \( \text{numpy.linalg.eigvalsh} \) is used to obtain them.

Parameters
c : array_like
1-D array of HermiteE series coefficients ordered from low to high degree.

Returns
mat : ndarray
Scaled companion matrix of dimensions (deg, deg).

Notes

numpy.polynomial.hermite_e.hermedomain = array([-1, 1])
numpy.polynomial.hermite_e.hermezero = array([0])
numpy.polynomial.hermite_e.hermeone = array([1])
numpy.polynomial.hermite_e.hermex = array([0, 1])
numpy.polynomial.hermite_e.hermtrim(c, tol=0)

Remove “small” “trailing” coefficients from a polynomial.

“Small” means “small in absolute value” and is controlled by the parameter tol; “trailing” means highest order coefficient(s), e.g., in [0, 1, 1, 0, 0] (which represents 0 + x + x**2 + 0*x**3 + 0*x**4) both the 3-rd and 4-th order coefficients would be “trimmed.”

Parameters

- c : array_like
  1-d array of coefficients, ordered from lowest order to highest.

- tol : number, optional
  Trailing (i.e., highest order) elements with absolute value less than or equal to tol (default value is zero) are removed.

Returns

- trimmed : ndarray
  1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

Raises

ValueError

If tol < 0

See Also:

trimseq

Examples

>>> from numpy import polynomial as P
>>> P.trimcoef((0,0,3,0,5,0,0))
array([ 0., 0., 3., 0., 5.])
>>> P.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3)  # item == tol is trimmed
array([ 0.])
>>> i = complex(0,1)  # works for complex
>>> P.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([ 0.0003+0.j , 0.0010-0.001j])
numpy.polynomial.hermite_e.hermeline(off, scl)

Hermite series whose graph is a straight line.

Parameters

- off, scl : scalars
The specified line is given by \( \text{off} + \text{scl} \times x \).

**Returns**

\( y \) : ndarray

This module’s representation of the Hermite series for \( \text{off} + \text{scl} \times x \).

**See Also:**

polyline, chebline

**Examples**

```python
>>> from numpy.polynomial.hermite_e import hermeline
>>> hermeval(0, hermeline(3, 2))
3.0
>>> hermeval(1, hermeline(3, 2))
5.0
```

**numpy.polynomial.hermite_e.herme2poly(c)**

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest to highest degree.

**Parameters**

- \( c \) : array_like
  
  1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

**Returns**

- \( \text{pol} \) : ndarray
  
  1-D array containing the coefficients of the equivalent polynomial (relative to the “standard” basis) ordered from lowest order term to highest.

**See Also:**

poly2herme

**Notes**

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

**Examples**

```python
>>> from numpy.polynomial.hermite_e import herme2poly
>>> herme2poly([ 2., 10., 2., 3.])
array([ 0., 1., 2., 3.])
```

**numpy.polynomial.hermite_e.poly2herme(pol)**

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

**Parameters**

- \( \text{pol} \) : array_like
  
  1-D array containing the polynomial coefficients
Returns
c : ndarray

1-D array containing the coefficients of the equivalent Hermite series.

See Also:
herme2poly

Notes
The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

Examples

```python
>>> from numpy.polynomial.hermite_e import poly2herme
>>> poly2herme(np.arange(4))
array([ 2., 10., 2., 3.])
```

Poly1d

Basics

```python
c = np.poly1d([1, 2, 3])
c
```

class numpy.poly1d(c_or_r[, r=0, variable=None])

A one-dimensional polynomial class.

A convenience class, used to encapsulate “natural” operations on polynomials so that said operations may take on their customary form in code (see Examples).

Parameters
c_or_r : array_like

The polynomial’s coefficients, in decreasing powers, or if the value of the second parameter is True, the polynomial’s roots (values where the polynomial evaluates to 0). For example, poly1d([1, 2, 3]) returns an object that represents \( x^2 + 2x + 3 \), whereas poly1d([1, 2, 3], True) returns one that represents \((x - 1)(x - 2)(x - 3) = x^3 - 6x^2 + 11x - 6\).

r : bool, optional

If True, c_or_r specifies the polynomial’s roots; the default is False.

variable : str, optional

Changes the variable used when printing p from x to variable (see Examples).

Examples

Construct the polynomial \( x^2 + 2x + 3 \):

```python
>>> p = np.poly1d([1, 2, 3])
>>> print np.poly1d(p)
2
1 x + 2 x + 3
```
Evaluate the polynomial at $x = 0.5$:

```python
>>> p(0.5)
4.25
```

Find the roots:

```python
>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
>>> p(p.r)
array([-4.44089210e-16+0.j, -4.44089210e-16+0.j])
```

These numbers in the previous line represent $(0, 0)$ to machine precision

Show the coefficients:

```python
>>> p.c
array([1, 2, 3])
```

Display the order (the leading zero-coefficients are removed):

```python
>>> p.order
2
```

Show the coefficient of the k-th power in the polynomial (which is equivalent to $p.c[-(i+1)]$):

```python
>>> p[1]
2
```

Polynomials can be added, subtracted, multiplied, and divided (returns quotient and remainder):

```python
>>> p * p
poly1d([1, 4, 10, 12, 9])
>>> (p**3 + 4) / p
(poly1d([1, 4, 10, 12, 9]), poly1d([4.]))
```

`asarray(p)` gives the coefficient array, so polynomials can be used in all functions that accept arrays:

```python
>>> p**2 # square of polynomial
poly1d([1, 4, 10, 12, 9])
>>> np.square(p) # square of individual coefficients
array([1, 4, 9])
```

The variable used in the string representation of $p$ can be modified, using the `variable` parameter:

```python
>>> p = np.poly1d([1,2,3], variable='z')
>>> print p
2
1 z + 2 z + 3
```

Construct a polynomial from its roots:

```python
>>> np.poly1d([1, 2], True)
poly1d([1, -3, 2])
```

This is the same polynomial as obtained by:

```python
>>> np.poly1d([1, -1]) * np.poly1d([1, -2])
poly1d([1, -3, 2])
```
Attributes

- coeffs
- order
- variable

Methods

```python
poly1d.__call__(val)
```

```python
poly1d.deriv(m=1)
```
Return a derivative of this polynomial.

Refer to `polyder` for full documentation.

See Also:

- `polyder`
  equivalent function

```python
poly1d.integ(m=1, k=0)
```
Return an antiderivative (indefinite integral) of this polynomial.

Refer to `polyint` for full documentation.

See Also:

- `polyint`
  equivalent function

```python
numpy.polyval(p, x)
```
Evaluate a polynomial at specific values.

If `p` is of length `N`, this function returns the value:

```
p[0]*x**(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x + p[N-1]
```

If `x` is a sequence, then `p(x)` is returned for each element of `x`. If `x` is another polynomial then the composite polynomial `p(x(t))` is returned.

Parameters

- `p` : array_like or poly1d object
  1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term, or an instance of poly1d.

- `x` : array_like or poly1d object
  A number, a 1D array of numbers, or an instance of poly1d, “at” which to evaluate `p`.

Returns

- `values` : ndarray or poly1d
If $x$ is a `poly1d` instance, the result is the composition of the two polynomials, i.e., $x$ is “substituted” in $p$ and the simplified result is returned. In addition, the type of $x$ - array_like or poly1d - governs the type of the output: $x$ array_like => values array_like, $x$ a `poly1d` object => values is also.

**See Also:**

`poly1d`
A polynomial class.

**Notes**

Horner’s scheme [R63] is used to evaluate the polynomial. Even so, for polynomials of high degree the values may be inaccurate due to rounding errors. Use carefully.

**References**

[R63]

**Examples**

```python
c = np.polyval([3,0,1], 5)  # 3 * 5**2 + 0 * 5**1 + 1
c
76
>>> np.polyval([3,0,1], np.poly1d(5))
poly1d([ 76.])
```

numpy.poly(seq_of_zeros)

Find the coefficients of a polynomial with the given sequence of roots.

Returns the coefficients of the polynomial whose leading coefficient is one for the given sequence of zeros (multiple roots must be included in the sequence as many times as their multiplicity; see Examples). A square matrix (or array, which will be treated as a matrix) can also be given, in which case the coefficients of the characteristic polynomial of the matrix are returned.

**Parameters**

seq_of_zeros : array_like, shape (N,) or (N, N)

A sequence of polynomial roots, or a square array or matrix object.

**Returns**

`c` : ndarray

1D array of polynomial coefficients from highest to lowest degree:

$c[0] * x**(N) + c[1] * x**(N-1) + ... + c[N-1] * x + c[N]$ where $c[0]$ always equals 1.

**Raises**

ValueError

If input is the wrong shape (the input must be a 1-D or square 2-D array).

**See Also:**

`polyval`  
Evaluate a polynomial at a point.
roots
Return the roots of a polynomial.

polyfit
Least squares polynomial fit.

poly1d
A one-dimensional polynomial class.

Notes
Specifying the roots of a polynomial still leaves one degree of freedom, typically represented by an undetermined leading coefficient. In the case of this function, that coefficient - the first one in the returned array - is always taken as one. (If for some reason you have one other point, the only automatic way presently to leverage that information is to use polyfit.)

The characteristic polynomial, \( p_a(t) \), of an \( n \)-by-\( n \) matrix \( A \) is given by

\[
p_a(t) = \det(tI - A),
\]

where \( I \) is the \( n \)-by-\( n \) identity matrix. [R55]

References
[R54], [R55]

Examples
Given a sequence of a polynomial’s zeros:

```python
>>> np.poly((0, 0, 0))  # Multiple root example
array([1, 0, 0, 0])
```

The line above represents \( z^{**3} + 0*z^{**2} + 0*z + 0 \).

```python
>>> np.poly((-1./2, 0, 1./2))
array([ 1. , 0. , -0.25, 0. ])
```

The line above represents \( z^{**3} - z/4 \)

```python
>>> np.poly((np.random.random(1)[0], 0, np.random.random(1)[0]))
array([ 1. , -0.77086955, 0.08618131, 0. ])
```

Given a square array object:

```python
>>> P = np.array([[0, 1./3], [-1./2, 0]])
>>> np.poly(P)
array([ 1. , 0. , 0.16666667])
```

Or a square matrix object:

```python
>>> np.poly(np.matrix(P))
array([ 1. , 0. , 0.16666667])
```

Note how in all cases the leading coefficient is always 1.

numpy.roots(p)
Return the roots of a polynomial with coefficients given in \( p \).

The values in the rank-1 array \( p \) are coefficients of a polynomial. If the length of \( p \) is \( n+1 \) then the polynomial is described by:
\[ p[0] \times x^n + p[1] \times x^{(n-1)} + \ldots + p[n-1] \times x + p[n] \]

**Parameters**

- \( p \): array_like
  
  Rank-1 array of polynomial coefficients.

**Returns**

- \( out \): ndarray
  
  An array containing the complex roots of the polynomial.

**Raises**

- ValueError
  
  When \( p \) cannot be converted to a rank-1 array.

**See Also:**

- poly
  
  Find the coefficients of a polynomial with a given sequence of roots.

- polyval
  
  Evaluate a polynomial at a point.

- polyfit
  
  Least squares polynomial fit.

- poly1d
  
  A one-dimensional polynomial class.

**Notes**

The algorithm relies on computing the eigenvalues of the companion matrix [R241].

**References**

[R241]

**Examples**

```python
>>> coeff = [3.2, 2, 1]
>>> np.roots(coeff)
array([-0.3125+0.46351241j, -0.3125-0.46351241j])
```

**Fitting**

```python
numpy.polyfit(x, y, deg[, rcond, full, w, cov])  
Least squares polynomial fit.
```

```
numpy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)  
Least squares polynomial fit.
```

Fit a polynomial \( p(x) = p[0] \times x^{deg} + \ldots + p[deg] \) of degree \( deg \) to points \( (x, y) \). Returns a vector of coefficients \( p \) that minimises the squared error.

**Parameters**

- \( x \): array_like, shape (M,)
  
  \( x \)-coordinates of the \( M \) sample points \( (x[i], y[i]) \).
y : array_like, shape (M,) or (M, K)
y-coordinates of the sample points. Several data sets of sample points sharing the same
x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset
per column.

deg : int
Degree of the fitting polynomial

rcond : float, optional
Relative condition number of the fit. Singular values smaller than this relative to the
largest singular value will be ignored. The default value is len(x)*eps, where eps is the
relative precision of the float type, about 2e-16 in most cases.

full : bool, optional
Switch determining nature of return value. When it is False (the default) just the coef-
ficients are returned, when True diagnostic information from the singular value decom-
position is also returned.

w : array_like, shape (M,), optional
weights to apply to the y-coordinates of the sample points.

cov : bool, optional
Return the estimate and the covariance matrix of the estimate If full is True, then cov is
not returned.

Returns

p : ndarray, shape (M,) or (M, K)
Polynomial coefficients, highest power first. If y was 2-D, the coefficients for k-th data
set are in p[:, k].

residuals, rank, singular_values, rcond : present only if full = True
Residuals of the least-squares fit, the effective rank of the scaled Vandermonde coeffi-
cient matrix, its singular values, and the specified value of rcond. For more details, see
linalg.lstsq.

V : ndaray, shape (M,M) or (M,M,K)
The covariance matrix of the polynomial coefficient estimates. The diagonal of this ma-
trix are the variance estimates for each coefficient. If y is a 2-d array, then the covariance
matrix for the k-th data set are in V[:, :, k]

Warns

RankWarning
The rank of the coefficient matrix in the least-squares fit is deficient. The warning is
only raised if full = False.
The warnings can be turned off by

>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)

See Also:

polyval
Computes polynomial values.
NumPy Reference, Release 1.8.1

linalg.lstsq
Computes a least-squares fit.

scipy.interpolate.UnivariateSpline
Computes spline fits.

Notes
The solution minimizes the squared error

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2$$

in the equations:

- $x[0] \cdot p[n] + \ldots + x[0] \cdot p[1] + p[0] = y[0]$

... 
- $x[k] \cdot p[n] + \ldots + x[k] \cdot p[1] + p[0] = y[k]$

The coefficient matrix of the coefficients $p$ is a Vandermonde matrix.

polyfit issues a RankWarning when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing $x \cdot x - x \cdot x.mean()$. The rcond parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

References
[R56], [R57]

Examples
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206, 1.69312169, -0.03968254])

It is convenient to use poly1d objects for dealing with polynomials:

```python
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179
>>> p(3.5)
-0.34732142857143039
>>> p(10)
22.579365079365115
```

High-order polynomials may oscillate wildly:

```python
>>> p30 = np.poly1d(np.polyfit(x, y, 30))
.../... RankWarning: Polyfit may be poorly conditioned...
>>> p30(4)
```

3.26. Polynomials 1125
Illustration:

```python
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

Calculus

```
polyder(p, m)  Return the derivative of the specified order of a polynomial.
polyint(p, m, k)  Return an antiderivative (indefinite integral) of a polynomial.
```

numpy.polyder (p, m=1)  
Return the derivative of the specified order of a polynomial.

**Parameters**

- **p**: poly1d or sequence
  - Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see poly1d.
- **m**: int, optional
  - Order of differentiation (default: 1)

**Returns**

- **der**: poly1d
  - A new polynomial representing the derivative.
See Also:

**polyint**
Anti-derivative of a polynomial.

**poly1d**
Class for one-dimensional polynomials.

Examples
The derivative of the polynomial \(x^3 + x^2 + x + 1\) is:

```python
>>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```
which evaluates to:

```python
>>> p2(2.)
17.0
```

We can verify this, approximating the derivative with \((f(x + h) - f(x))/h:\n
```python
>>> (p(2. + 0.001) - p(2.)) / 0.001
17.007000999997857
```
The fourth-order derivative of a 3rd-order polynomial is zero:

```python
>>> np.polyder(p, 3)
poly1d([6])
>>> np.polyder(p, 4)
poly1d([ 0.])
```

```python
numpy.polyint (p, m=1, k=None)
```
Return an antiderivative (indefinite integral) of a polynomial.

The returned order \(m\) antiderivative \(P\) of polynomial \(p\) satisfies \(\frac{d^m}{dx^m}P(x) = p(x)\) and is defined up to \(m - 1\) integration constants \(k\). The constants determine the low-order polynomial part

\[
\frac{k_{m-1}}{0!}x^0 + \cdots + \frac{k_0}{(m-1)!}x^{m-1}
\]

of \(P\) so that \(P^{(j)}(0) = k_{m-j-1}\).

Parameters
\(p\) : {array_like, poly1d}
Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see `poly1d`.

\(m\) : int, optional
Order of the antiderivative. (Default: 1)

\(k\) : {None, list of \(m\) scalars, scalar}, optional
Integration constants. They are given in the order of integration: those corresponding to highest-order terms come first.
If `None` (default), all constants are assumed to be zero. If \( m = 1 \), a single scalar can be given instead of a list.

**See Also:**

- `polyder`  
  derivative of a polynomial

- `poly1d.integ`  
  equivalent method

**Examples**

The defining property of the antiderivative:

```python
g = np.poly1d([1, 1, 1])
g = np.polyint(g)
g
poly1d([ 0.33333333, 0.5 , 1. , 0. ])
g = np.polyder(g) == g
True
```

The integration constants default to zero, but can be specified:

```python
g = np.polyint(g, 3)
g(0)
0.0
g = np.polyint(g, 2)(0)
0.0
g = np.polyint(g, 3, k=[6,5,3])
g
poly1d([ 0.01666667, 0.04166667, 0.16666667, 3. , 5. , 3. ])
```

Note that 3 = 6 / 2!, and that the constants are given in the order of integrations. Constant of the highest-order polynomial term comes first:

```python
g = np.polyder(g, 2)(0)
6.0
g = np.polyder(g, 1)(0)
5.0
g(0)
3.0
```

**Arithmetic**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>numpy.polyadd(a1, a2)</code></td>
<td>Find the sum of two polynomials.</td>
</tr>
<tr>
<td><code>polydiv(u, v)</code></td>
<td>Returns the quotient and remainder of polynomial division.</td>
</tr>
<tr>
<td><code>polymul(a1, a2)</code></td>
<td>Find the product of two polynomials.</td>
</tr>
<tr>
<td><code>polysub(a1, a2)</code></td>
<td>Difference (subtraction) of two polynomials.</td>
</tr>
</tbody>
</table>

`numpy.polyadd(a1, a2)`  
Find the sum of two polynomials.

Returns the polynomial resulting from the sum of two input polynomials. Each input must be either a `poly1d` object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

**Parameters**
**NumPy Reference, Release 1.8.1**

**a1, a2**: array_like or poly1d object

Input polynomials.

**Returns**

- **out**: ndarray or poly1d object

  The sum of the inputs. If either input is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

**See Also:**

- poly1d
  
  A one-dimensional polynomial class.

- poly, polyadd, polyder, polydiv, polyfit, polyint, polymul, polysub, polyval

**Examples**

```python
>>> np.polyadd([1, 2], [9, 5, 4])
array([9, 6, 6])
```

Using poly1d objects:

```python
>>> p1 = np.poly1d([1, 2])
>>> p2 = np.poly1d([9, 5, 4])
>>> print p1
1 x + 2
>>> print p2
2
9 x + 5 x + 4
>>> print np.polyadd(p1, p2)
2
9 x + 6 x + 6
```

**numpy.polydiv(u, v)**

Returns the quotient and remainder of polynomial division.

The input arrays are the coefficients (including any coefficients equal to zero) of the “numerator” (dividend) and “denominator” (divisor) polynomials, respectively.

**Parameters**

- **u**: array_like or poly1d
  
  Dividend polynomial’s coefficients.

- **v**: array_like or poly1d
  
  Divisor polynomial’s coefficients.

**Returns**

- **q**: ndarray
  
  Coefficients, including those equal to zero, of the quotient.

- **r**: ndarray
  
  Coefficients, including those equal to zero, of the remainder.

**See Also:**

- poly, polyadd, polyder, polydiv, polyfit, polyint, polymul, polysub, polyval
Notes

Both u and v must be 0-d or 1-d (ndim = 0 or 1), but u.ndim need not equal v.ndim. In other words, all four possible combinations - u.ndim = v.ndim = 0, u.ndim = v.ndim = 1, u.ndim = 0, and u.ndim = 0, v.ndim = 1 - work.

Examples

\[
\frac{3x^2 + 5x + 2}{2x + 1} = 1.5x + 1.75, \text{remainder} 0.25
\]

```python
>>> x = np.array([3.0, 5.0, 2.0])
>>> y = np.array([2.0, 1.0])
>>> np.polydiv(x, y)
(array([ 1.5,  1.75]), array([ 0.25]))
```

**numpy.polymul(a1, a2)**

Find the product of two polynomials.

Finds the polynomial resulting from the multiplication of the two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

**Parameters**

- `a1, a2`: array_like or poly1d object
  
  Input polynomials.

**Returns**

- `out`: ndarray or poly1d object
  
  The polynomial resulting from the multiplication of the inputs. If either inputs is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

**See Also:**

poly1d

A one-dimensional polynomial class.

poly, polyadd, polyder, polydiv, polyfit, polyint, polysub, polyval

**Examples**

```python
>>> np.polymul([1, 2, 3], [9, 5, 1])
array([ 9, 23, 38, 17,  3])
```

Using poly1d objects:

```python
>>> p1 = np.poly1d([1, 2, 3])
>>> p2 = np.poly1d([9, 5, 1])
>>> print p1
2
1 x + 2 x + 3
>>> print p2
2
9 x + 5 x + 1
>>> print np.polymul(p1, p2)
```

1130 Chapter 3. Routines
numpy.polysub(a1, a2)
Difference (subtraction) of two polynomials.

Given two polynomials $a1$ and $a2$, returns $a1 - a2$. $a1$ and $a2$ can be either array_like sequences of the polynomials’ coefficients (including coefficients equal to zero), or poly1d objects.

Parameters

- **a1, a2**: array_like or poly1d
  Minuend and subtrahend polynomials, respectively.

Returns

- **out**: ndarray or poly1d
  Array or poly1d object of the difference polynomial’s coefficients.

See Also:
polyval, polydiv, polymul, polyadd

Examples

$$(2x^2 + 10x - 2) - (3x^2 + 10x - 4) = (-x^2 + 2)$$

>>> np.polysub([2, 10, -2], [3, 10, -4])
array([-1, 0,  2])

Warnings

- **RankWarning** Issued by polyfit when the Vandermonde matrix is rank deficient.

exception numpy.RankWarning

Issued by polyfit when the Vandermonde matrix is rank deficient.

For more information, a way to suppress the warning, and an example of RankWarning being issued, see polyfit.

### 3.27 Random sampling (numpy.random)

#### 3.27.1 Simple random data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand(d0, d1, ..., dn)</td>
<td>Random values in a given shape.</td>
</tr>
<tr>
<td>randn(d0, d1, ..., dn)</td>
<td>Return a sample (or samples) from the “standard normal” distribution.</td>
</tr>
<tr>
<td>randint(low[, high, size])</td>
<td>Return random integers from low (inclusive) to high (exclusive).</td>
</tr>
<tr>
<td>random_integers(low[, high, size])</td>
<td>Return random integers between low and high, inclusive.</td>
</tr>
<tr>
<td>random_sample([size])</td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
<tr>
<td>random([size])</td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
</tbody>
</table>
Table 3.176 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ranf</td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
<tr>
<td>sample</td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
<tr>
<td>choice</td>
<td>Generates a random sample from a given 1-D array</td>
</tr>
<tr>
<td>bytes</td>
<td>Return random bytes.</td>
</tr>
</tbody>
</table>

**numpy.random.rand**

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over [0, 1).

**Parameters**

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

**Returns**

out : ndarray, shape (d0, d1, ..., dn)

Random values.

**See Also:**

random

**Notes**

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

**Examples**

```python
>>> np.random.rand(3,2)
array([[ 0.14022471, 0.96360618],
       [ 0.37601032, 0.25528411],
       [ 0.49313049, 0.94909878]])
```

**numpy.random.randn**

Return a sample (or samples) from the “standard normal” distribution.

If positive, int_like or int-convertible arguments are provided, randn generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

**Parameters**

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

**Returns**

Z : ndarray or float

A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.
See Also:

random.standard_normal
   Similar, but takes a tuple as its argument.

Notes
For random samples from \( N(\mu, \sigma^2) \), use:
\[
\sigma * \text{np.random.randn}(...) + \mu
\]

Examples
>>> np.random.randn()
2.1923875335537315 #random

Two-by-four array of samples from N(3, 6.25):

>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.49401501, 4.00950034, -1.81814867, 7.29718677], #random
       [ 0.39924804, 4.68456316, 4.99394529, 4.84057254]]) #random

numpy.random.randint (low, high=None, size=None)
   Return random integers from low (inclusive) to high (exclusive).
   Return random integers from the “discrete uniform” distribution in the “half-open” interval [low, high). If high is None (the default), then results are from [0, low).

Parameters
   low : int
      Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the highest such integer).

   high : int, optional
      If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

   size : int or tuple of ints, optional
      Output shape. Default is None, in which case a single int is returned.

Returns
   out : int or ndarray of ints
      size-shaped array of random integers from the appropriate distribution, or a single such random int if size not provided.

See Also:

random.random_integers
   similar to randint, only for the closed interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

Examples
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 0, 0, 1, 0, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```python
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

```
numpy.random.randint(low, high=None, size=None)
Return random integers between low and high, inclusive.
```

Return random integers from the “discrete uniform” distribution in the closed interval [low, high]. If high is None (the default), then results are from [1, low].

**Parameters**
- **low**: int
  Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the highest such integer).
- **high**: int, optional
  If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).
- **size**: int or tuple of ints, optional
  Output shape. Default is None, in which case a single int is returned.

**Returns**
- **out**: int or ndarray of ints
  size-shaped array of random integers from the appropriate distribution, or a single such random int if size not provided.

**See Also:**
- **random.randint**
  Similar to `random_integers`, only for the half-open interval [low, high), and 0 is the lowest value if high is omitted.

**Notes**
To sample from N evenly spaced floating-point numbers between a and b, use:

\[
a + (b - a) \times (\text{np.random.random_integers}(N) - 1) / (N - 1.)
\]

**Examples**

```python
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3, 2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```python
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625, 1.25 , 0.625, 0.625, 2.5 ])
```
Roll two six sided dice 1000 times and sum the results:

```python
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

```python
numpy.random.random_sample(size=None)
```

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample \( U \) in \([a, b), b > a\)
multiply the output of `random_sample` by \((b-a)\) and add \(a\):

\[(b - a) \times \text{random_sample()} + a\]

**Parameters**

- `size`: int or tuple of ints, optional
  
  Defines the shape of the returned array of random floats. If None (the default), returns a single float.

**Returns**

- `out`: float or ndarray of floats
  
  Array of random floats of shape `size` (unless `size=None`, in which case a single float is returned).

**Examples**

```python
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
aarray([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])
```
Three-by-two array of random numbers from [-5, 0):

```python
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`numpy.random.random(size=None)`

Return random floats in the half-open interval [0.0, 1.0). Results are from the “continuous uniform” distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of `random_sample` by $(b-a)$ and add $a$:

$$(b - a) \times \text{random\_sample}() + a$$

**Parameters**

- `size`: int or tuple of ints, optional
  
  Defines the shape of the returned array of random floats. If None (the default), returns a single float.

**Returns**

- `out`: float or ndarray of floats
  
  Array of random floats of shape `size` (unless `size=None`, in which case a single float is returned).

**Examples**

```python
>>> np.random.random_sample() 1.05406802607
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```python
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`numpy.random.ranf(size=None)`

Return random floats in the half-open interval [0.0, 1.0). Results are from the “continuous uniform” distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of `random_sample` by $(b-a)$ and add $a$:

$$(b - a) \times \text{random\_sample}() + a$$

**Parameters**

- `size`: int or tuple of ints, optional
  
  Defines the shape of the returned array of random floats. If None (the default), returns a single float.

**Returns**

- `out`: float or ndarray of floats
Array of random floats of shape size (unless size=None, in which case a single float is returned).

Examples

```python
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```python
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`numpy.random.choice(a, size=None, replace=True, p=None)`

Generates a random sample from a given 1-D array

New in version 1.7.0.
Parameters

- **a**: 1-D array-like or int
  If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a was np.arange(n)

- **size**: int or tuple of ints, optional
  Output shape. Default is None, in which case a single value is returned.

- **replace**: boolean, optional
  Whether the sample is with or without replacement

- **p**: 1-D array-like, optional
  The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

Returns

- **samples**: 1-D ndarray, shape (size,)
  The generated random samples

Returns

- **ValueError**: If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See Also:

- `randint`, `shuffle`, `permutation`

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```python
generate a uniform random sample from np.arange(5) of size 3
>>> np.random.choice(5, 3)
array([0, 3, 4])
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```python
generate a non-uniform random sample from np.arange(5) of size 3
>>> np.random.choice(5, 3, p=[0.1, 0.3, 0.6, 0])
array([3, 3, 0])
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```python
generate a uniform random sample from np.arange(5) of size 3 without replacement
>>> np.random.choice(5, 3, replace=False)
array([3, 1, 0])
>>> #This is equivalent to np.random.shuffle(np.arange(5))[3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```python
generate a non-uniform random sample from np.arange(5) of size 3 without replacement
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0.3, 0.6, 0])
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:
NumPy Reference, Release 1.8.1

>>> aa_milne_arr = [’pooh’, ’rabbit’, ’piglet’, ’Christopher’]
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array([’pooh’, ’pooh’, ’pooh’, ’Christopher’, ’piglet’],
dtype=’|S11’)

numpy.random.bytes(length)
Return random bytes.
Parameters
length : int
Number of random bytes.
Returns
out : str
String of length length.
Examples
>>> np.random.bytes(10)
’ eh\x85\x022SZ\xbf\xa4’ #random

3.27.2 Permutations
shuffle(x)
permutation(x)

Modify a sequence in-place by shuffling its contents.
Randomly permute a sequence, or return a permuted range.

numpy.random.shuffle(x)
Modify a sequence in-place by shuffling its contents.
Parameters
x : array_like
The array or list to be shuffled.
Returns
None
Examples
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]

This function only shuffles the array along the first index of a multi-dimensional array:
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5],
[6, 7, 8],
[0, 1, 2]])

numpy.random.permutation(x)
Randomly permute a sequence, or return a permuted range.

3.27. Random sampling (numpy.random)

1139


If \( x \) is a multi-dimensional array, it is only shuffled along its first index.

**Parameters**

- \( x \): int or array_like

  If \( x \) is an integer, randomly permute \( \text{np.arange}(x) \). If \( x \) is an array, make a copy and shuffle the elements randomly.

**Returns**

- \( \text{out} \): ndarray

  Permuted sequence or array range.

**Examples**

```python
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

>>> np.random.permutation([1, 4, 9, 12, 15])
array([15, 1, 9, 4, 12])

>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([ [6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

### 3.27.3 Distributions

- **beta**(\( a, b, \text{size} \))  
  The Beta distribution over \([0, 1]\).
- **binomial**(\( n, p, \text{size} \))  
  Draw samples from a binomial distribution.
- **chisquare**(\( df, \text{size} \))  
  Draw samples from a chi-square distribution.
- **dirichlet**(\( \alpha, \text{size} \))  
  Draw samples from the Dirichlet distribution.
- **exponential**(\( \text{scale, size} \))  
  Exponential distribution.
- **f**(\( dfnum, dfden, \text{size} \))  
  Draw samples from a F distribution.
- **gamma**(\( \text{shape, scale, size} \))  
  Draw samples from a Gamma distribution.
- **geometric**(\( p, \text{size} \))  
  Draw samples from the geometric distribution.
- **gumbel**(\( \text{loc, scale, size} \))  
  Gumbel distribution.
- **hypergeometric**(\( \text{ngood, nbad, nsample, size} \))  
  Draw samples from a Hypergeometric distribution.
- **laplace**(\( \text{loc, scale, size} \))  
  Draw samples from the Laplace or double exponential distribution.
- **logistic**(\( \text{loc, scale, size} \))  
  Draw samples from a Logistic distribution.
- **lognormal**(\( \text{mean, sigma, size} \))  
  Return samples drawn from a log-normal distribution.
- **logseries**(\( p, \text{size} \))  
  Draw samples from a Logarithmic Series distribution.
- **multinomial**(\( n, pvals, \text{size} \))  
  Draw samples from a multinomial distribution.
- **multivariate_normal**(\( \text{mean, cov, size} \))  
  Draw random samples from a multivariate normal distribution.
- **negative_binomial**(\( n, p, \text{size} \))  
  Draw samples from a negative binomial distribution.
- **noncentral_chisquare**(\( df, \text{nonc, size} \))  
  Draw samples from a noncentral chi-square distribution.
- **noncentral_f**(\( dfnum, dfden, \text{nonc, size} \))  
  Draw samples from the noncentral F distribution.
- **normal**(\( \text{loc, scale, size} \))  
  Draw random samples from a normal (Gaussian) distribution.
- **pareto**(\( \text{a, size} \))  
  Draw samples from a Pareto II or Lomax distribution.
- **poisson**(\( \lambda, \text{size} \))  
  Draw samples from a Poisson distribution.
- **power**(\( \text{a, size} \))  
  Draw samples in \([0, 1]\) from a power distribution with positive exponent \( a - 1 \).
- **rayleigh**(\( \text{scale, size} \))  
  Draw samples from a Rayleigh distribution.

Continued on next page
**Table 3.178 – continued from previous page**

- `standard_cauchy([size])`: Standard Cauchy distribution with mode = 0.
- `standard_exponential([size])`: Draw samples from the standard exponential distribution.
- `standard_gamma(shape[, size])`: Draw samples from a Standard Gamma distribution.
- `standard_normal([size])`: Returns samples from a Standard Normal distribution (mean=0, stdev=1).
- `standard_t(df[, size])`: Standard Student’s t distribution with df degrees of freedom.
- `triangular(left, mode, right[, size])`: Draw samples from the triangular distribution.
- `uniform([low, high, size])`: Draw samples from a uniform distribution.
- `vonmises(mu, kappa[, size])`: Draw samples from a von Mises distribution.
- `wald(mean, scale[, size])`: Draw samples from a Wald, or Inverse Gaussian, distribution.
- `weibull(a[, size])`: Weibull distribution.
- `zipf(a[, size])`: Draw samples from a Zipf distribution.

### `numpy.random.beta(a, b, size=None)`

The Beta distribution over `[0, 1]`.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

\[
f(x; a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1},
\]

where the normalisation, B, is the beta function,

\[
B(a, b) = \int_0^1 t^{a-1} (1-t)^{b-1} dt.
\]

It is often seen in Bayesian inference and order statistics.

**Parameters**

- `a`: float
  - Alpha, non-negative.
- `b`: float
  - Beta, non-negative.
- `size`: tuple of ints, optional
  - The number of samples to draw. The output is packed according to the size given.

**Returns**

- `out`: ndarray
  - Array of the given shape, containing values drawn from a Beta distribution.

### `numpy.random.binomial(n, p, size=None)`

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

**Parameters**

- `n`: float (but truncated to an integer)
  - parameter, >= 0.
NumPy Reference, Release 1.8.1

```
  p : float
      parameter, >= 0 and <=1.
  size : {tuple, int}
      Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are
drawn.

Returns

  samples : {ndarray, scalar}
      where the values are all integers in [0, n].

See Also:

  scipy.stats.distributions.binom
      probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Binomial distribution is

\[ P(N) = \binom{n}{N} p^N (1-p)^{n-N}, \]

where \( n \) is the number of trials, \( p \) is the probability of success, and \( N \) is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal
distribution works well unless the product \( p*n \leq 5 \), where \( p \) = population proportion estimate, and \( n \) = number
of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows
4 who are left handed, and 11 who are right handed. Then \( p = 4/15 = 27\% \). 0.27*15 = 4, so the binominal
distribution should be used in this case.

References

[R182], [R183], [R184], [R185], [R186]

Examples

Draw samples from the distribution:

```python
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of
success of 0.1. All nine wells fail. What is the probability of that happening?

Let’s do 20,000 trials of the model, and count the number that generate zero positive results.

```python
>>> sum(np.random.binomial(9,0.1,20000)==0)/20000.
answer = 0.38885, or 38%. 0.27*15 = 4, so the binomial
distribution should be used in this case.
```
```
```
Parameters

df : int
   Number of degrees of freedom.

size : tuple of ints, int, optional
   Size of the returned array. By default, a scalar is returned.

Returns

output : ndarray
   Samples drawn from the distribution, packed in a size-shaped array.

Raises

ValueError
   When df <= 0 or when an inappropriate size (e.g. size=-1) is given.

Notes

The variable obtained by summing the squares of df independent, standard normally distributed random vari-
ables:

\[ Q = \sum_{i=0}^{\text{df}} X_i^2 \]

is chi-square distributed, denoted

\[ Q \sim \chi^2_k. \]

The probability density function of the chi-squared distribution is

\[ p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2}, \]

where \( \Gamma \) is the gamma function,

\[ \Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt. \]

References

NIST/SEMATECH e-Handbook of Statistical Methods

Examples

>>> np.random.chisquare(2, 4)
array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])

numpy.random.dirichlet(alpha, size=None)
   Draw samples from the Dirichlet distribution.

   Draw size samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be
   seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial
   in Bayesian inference.
Parameters

\textbf{alpha} : array

Parameter of the distribution (k dimension for sample of dimension k).

\textbf{size} : array

Number of samples to draw.

Returns

\textbf{samples} : ndarray,

The drawn samples, of shape (alpha.ndim, size).

Notes

\[ X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1} \]

Uses the following property for computation: for each dimension, draw a random sample \( y_i \) from a standard gamma generator of shape \( \alpha_i \), then \( X = \frac{1}{\sum_{i=1}^{k} y_i} (y_1, \ldots, y_n) \) is Dirichlet distributed.

References

[R187], [R188]

Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

\begin{verbatim}
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()

>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
\end{verbatim}

numpy.random.\texttt{exponential} \texttt{(scale=1.0, size=None)}

Exponential distribution.

Its probability density function is

\[ f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right), \]

for \( x > 0 \) and 0 elsewhere. \( \beta \) is the scale parameter, which is the inverse of the rate parameter \( \lambda = 1/\beta \). The rate parameter is an alternative, widely used parameterization of the exponential distribution [R191].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [R189], or the time between page requests to Wikipedia [R190].

Parameters

\texttt{scale} : float

The scale parameter, \( \beta = 1/\lambda \).
size : tuple of ints

Number of samples to draw. The output is shaped according to size.

References

[R189], [R190], [R191]
numpy.random.f (dfnum, dfden, size=None)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, dfnum (degrees of freedom in numerator) and dfden (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Parameters

- dfnum : float
  Degrees of freedom in numerator. Should be greater than zero.

- dfden : float
  Degrees of freedom in denominator. Should be greater than zero.

- size : {tuple, int}, optional
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. By default only one sample is returned.

Returns

- samples : {ndarray, scalar}
  Samples from the Fisher distribution.

See Also:

scipy.stats.distributions.f

probability density function, distribution or cumulative density function, etc.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable dfnum is the number of samples minus one, the between-groups degrees of freedom, while dfden is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

[R192], [R193]

Examples

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children’s blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)

The lower bound for the top 1% of the samples is:

```plaintext
>>> sort(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

numpy.random.gamma (shape, scale=1.0, size=None)

Draw samples from a Gamma distribution.

Parameters
- shape : scalar > 0
  The shape of the gamma distribution.
- scale : scalar > 0, optional
  The scale of the gamma distribution. Default is equal to 1.
- size : shape_tuple, optional
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
- out : ndarray, float
  Returns one sample unless size parameter is specified.

See Also:
- scipy.stats.distributions.gamma
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

\[ p(x) = x^{k-1} \frac{e^{-x/\theta}}{\theta^k \Gamma(k)}, \]

where \( k \) is the shape and \( \theta \) the scale, and \( \Gamma \) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R194], [R195]
Examples

Draw samples from the distribution:

```python
>>> shape, scale = 2., 2.  # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) / ... (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

```
numpy.random.geometric(p, size=None)
```

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, \( k = 1, 2, \ldots \).

The probability mass function of the geometric distribution is

\[
f(k) = (1 - p)^{k-1} p
\]

where \( p \) is the probability of success of an individual trial.

**Parameters**

- **p**: float
  
The probability of success of an individual trial.

- **size**: tuple of ints
  
  Number of values to draw from the distribution. The output is shaped according to size.

**Returns**

- **out**: ndarray
Samples from the geometric distribution, shaped according to size.

Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```python
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```python
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

`numpy.random.gumbel(loc=0.0, scale=1.0, size=None)`

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Parameters

- `loc` : float
  The location of the mode of the distribution.

- `scale` : float
  The scale parameter of the distribution.

- `size` : tuple of ints
  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns

- `out` : ndarray
  The samples

See Also:

- `scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`

- `scipy.stats.genextreme`

  probability density function, distribution, or cumulative density function, etc. for each of the above

- `weibull`

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with “exponential-like” tails.

The probability density for the Gumbel distribution is

\[ p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}}, \]

where \( \mu \) is the mode, a location parameter, and \( \beta \) is the scale parameter.
The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a “fat-tailed” distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

**References**


**Examples**

Draw samples from the distribution:

```python
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```python
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(np.mean(a))
...    maxima.append(np.max(a))
```
... means.append(a.mean())
... maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(- (bins - mu)/beta)
... * np.exp(-np.exp(- (bins - mu)/beta)),
... linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
... * np.exp(- (bins - mu)**2 / (2 * beta**2)),
... linewidth=2, color='g')
>>> plt.show()

numpy.random.hypergeometric(ngood, nbad, nsample, size=None)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

Parameters

ngood : int or array_like

Number of ways to make a good selection. Must be nonnegative.

nbad : int or array_like

Number of ways to make a bad selection. Must be nonnegative.

nsample : int or array_like

Number of items sampled. Must be at least 1 and at most ngood + nbad.

size : int or tuple of int

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns

samples : ndarray or scalar
The values are all integers in [0, n].

See Also:

```python
scipy.stats.distributions.hypergeom
```
probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Hypergeometric distribution is

\[ P(x) = \binom{m}{x} \binom{N-m}{n-x} / \binom{N}{n}, \]

where \(0 \leq x \leq m\) and \(n + m - N \leq x \leq n\)

for \(P(x)\) the probability of \(x\) successes, \(n = \text{ngood}\), \(m = \text{nbad}\), and \(N = \text{number of samples}\).

Consider an urn with black and white marbles in it, \(\text{ngood}\) of them black and \(\text{nbad}\) are white. If you draw \(\text{nsamp}\) balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

References

[R196], [R197], [R198]

Examples

Draw samples from the distribution:

```python
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
```

```
hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```python
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

**Laplace distribution**

```python
numpy.random.laplace(loc=0.0, scale=1.0, size=None)
```

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Parameters

- **loc**: float

  The position, \(\mu\), of the distribution peak.

- **scale**: float

  The scale, \(\lambda\), of the distribution.
\( \lambda \), the exponential decay.

**Notes**

It has the probability density function

\[
f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).
\]

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

**References**

[R199], [R200], [R201], [R202]

**Examples**

Draw samples from the distribution

```python
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```python
>>> g = (1/(scale * np.sqrt(2 * np.pi)) * 
...     np.exp(- (x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x, g)
```
Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Parameters
- loc : float
- scale : float > 0.

size : {tuple, int}
Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
- samples : {ndarray, scalar}
where the values are all integers in [0, n].

See Also:
- scipy.stats.distributions.logistic
  probability density function, distribution or cumulative density function, etc.

Notes
The probability density for the Logistic distribution is

\[ P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2}, \]

where \( \mu \) = location and \( s \) = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References
- [R203], [R204], [R205]

Examples
Draw samples from the distribution:

```python
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)
# plot against distribution
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale+(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()/logist(bins, loc, scale).max())
>>> plt.show()
```
numpy.random.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

**Parameters**

- **mean**: float
  Mean value of the underlying normal distribution

- **sigma**: float, > 0.
  Standard deviation of the underlying normal distribution

- **size**: tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

**Returns**

- **samples**: ndarray or float
  The desired samples. An array of the same shape as size if given, if size is None a float is returned.

**See Also:**

- **scipy.stats.lognorm**
  probability density function, distribution, cumulative density function, etc.

**Notes**

A variable \( x \) has a log-normal distribution if \( \log(x) \) is normally distributed. The probability density function for the log-normal distribution is:

\[
p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{-\left(\frac{\ln(x) - \mu}{\sigma^2}\right)^2}
\]

where \( \mu \) is the mean and \( \sigma \) is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the product of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the sum of a large number of independent, identically-distributed variables.

**References**


**Examples**

Draw samples from the distribution:

```python
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')

>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.randn(100)
...     b.append(np.product(a))

>>> b = np.array(b) / np.min(b)  # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))

>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))

>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
	numpy.random.logseries(p, size=None)

Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0 < p < 1).

Parameters

loc : float
scale : float > 0.
size : {tuple, int}

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns

samples : {ndarray, scalar}

where the values are all integers in [0, n].

See Also:

scipy.stats.distributions.logser

probability density function, distribution or cumulative density function, etc.
Notes

The probability density for the Log Series distribution is

\[ P(k) = \frac{-p^k}{k \ln(1 - p)}, \]

where \( p \) = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References

[R206], [R207], [R208], [R209]

Examples

Draw samples from the distribution:

```python
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)

# plot against distribution
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/logseries(bins, a).max(), 'r')
>>> plt.show()
```

`numpy.random.multinomial(n, pvals, size=None)`

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of \( p \) possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents \( n \) such experiments. Its values, \( X_i = [X_0, X_1, \ldots, X_p] \), represent the number of times the outcome was \( i \).

Parameters

- **n**: int
  
  Number of experiments.

- **pvals**: sequence of floats, length \( p \)
  
  Probabilities of each of the \( p \) different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as \( \text{sum}(\text{pvals}[:-1]) \leq 1 \)).

- **size**: tuple of ints
  
  Given a size of \( (M, N, K) \), then \( M*N*K \) samples are drawn, and the output shape becomes \( (M, N, K, p) \), since each sample has shape \( (p,) \).

Examples

Throw a dice 20 times:
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 4, 3],
       [2, 4, 3, 4, 0]])

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])

`numpy.random.multivariate_normal(mean, cov[, size])`

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or “center”) and variance (standard deviation, or “width,” squared) of the one-dimensional normal distribution.

**Parameters**

- `mean`: 1-D array_like, of length N
  Mean of the N-dimensional distribution.

- `cov`: 2-D array_like, of shape (N, N)
  Covariance matrix of the distribution. Must be symmetric and positive semi-definite for “physically meaningful” results.

- `size`: int or tuple of ints, optional
  Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

**Returns**

- `out`: ndarray
  The drawn samples, of shape `size`, if that was provided. If not, the shape is `(N, )`.

In other words, each entry `out[i, j, ..., :]` is an N-dimensional value drawn from the distribution.

**Notes**

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, \ldots x_N]$. The covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$ (i.e. its “spread”).

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (`cov` is a multiple of the identity matrix)
•Diagonal covariance (\(\text{cov}\) has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```python
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance, points lie on x or y-axis
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

References


Examples

```python
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```python
>>> print list((x[0, 0] - mean) < 0.6)
[True, True]
```

```python
numpy.random.negative_binomial(n, p, size=None)
```

Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, \(n\) trials and \(p\) probability of success where \(n\) is an integer > 0 and \(p\) is in the interval [0, 1].

Parameters

- **n** : int
  Parameter, > 0.

- **p** : float
  Parameter, >= 0 and <=1.

- **size** : int or tuple of ints
  Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m * n * k\) samples are drawn.

Returns

- **samples** : int or ndarray of ints
  Drawn samples.

Notes

The probability density for the Negative Binomial distribution is

\[
P(N; n, p) = \binom{N + n - 1}{n - 1} p^n (1 - p)^N,
\]
where \( n - 1 \) is the number of successes, \( p \) is the probability of success, and \( N + n - 1 \) is the number of trials.

The negative binomial distribution gives the probability of \( n-1 \) successes and \( N \) failures in \( N+n-1 \) trials, and success on the \( (N+n) \)th trial.

If one throws a die repeatedly until the third time a “1” appears, then the probability distribution of the number of non-“1”s that appear before the third “1” is a negative binomial distribution.

References

[R210], [R211]

Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```python
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...    probability = sum(s<i) / 100000.
...    print i, "wells drilled, probability of one success =", probability
```

```
numpy.random.noncentral_chisquare
```

Draw samples from a noncentral chi-square distribution.

The noncentral \( \chi^2 \) distribution is a generalisation of the \( \chi^2 \) distribution.

Parameters

- **df**: int
  - Degrees of freedom, should be \( \geq 1 \).
- **nonc**: float
  - Non-centrality, should be \( > 0 \).
- **size**: int or tuple of ints
  - Shape of the output.

Notes

The probability density function for the noncentral Chi-square distribution is

\[
P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-\text{nonc}/2}(\text{nonc}/2)^i}{i!} P_{Y_{df+2i}}(x),
\]

where \( Y_q \) is the Chi-square with \( q \) degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

References

[R212], [R213]
Examples

Draw values from the distribution and plot the histogram

```python
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```

![Histogram of Noncentral Chi-Square Distribution](image)

Draw values from a noncentral chi-square with very small noncentrality, and compare to a chisquare.

```python
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                    bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

![Comparison of Noncentral Chi-Square and Chi-Square Distributions](image)
Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```python
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```

```python
numpy.random.noncentral_f(dfnum, dfden, nonc, size=None)
```

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, `dfnum` (degrees of freedom in numerator) and `dfden` (degrees of freedom in denominator), where both parameters > 1. `nonc` is the non-centrality parameter.

**Parameters**
- `dfnum` : int
  - Parameter, should be > 1.
- `dfden` : int
  - Parameter, should be > 1.
- `nonc` : float
  - Parameter, should be >= 0.
- `size` : int or tuple of ints
  - Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

**Returns**
- `samples` : scalar or ndarray
  - Drawn samples.

**Notes**

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.
References


Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We’ll plot the two probability distributions for comparison.

```python
>>> dfnum = 3  # between group deg of freedom
>>> dfden = 20  # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

`numpy.random.normal (loc=0.0, scale=1.0, size=None)`

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [R215], is often called the bell curve because of its characteristic shape (see the example below).

The normal distribution occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [R215].

Parameters

- **loc** : float
  Mean (“centre”) of the distribution.

- **scale** : float
  Standard deviation (spread or “width”) of the distribution.

- **size** : tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

See Also:

- `scipy.stats.distributions.norm`
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

\[
p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},
\]
where $\mu$ is the mean and $\sigma$ the standard deviation. The square of the standard deviation, $\sigma^2$, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [R215]). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.

**References**

[R214], [R215]

**Examples**

Draw samples from the distribution:

```python
>>> mu, sigma = 0, 0.1  # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```python
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
... np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
... linewidth=2, color='r')
>>> plt.show()
```

`numpy.random.pareto(a, size=None)`

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter $m$, see below. The smallest value of
the Lomax distribution is zero while for the classical Pareto distribution it is \( m \), where the standard Pareto distribution has location \( m=1 \). Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the “80-20 rule”. In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

**Parameters**

- **shape**: float, > 0.
  
  Shape of the distribution.

- **size**: tuple of ints
  
  Output shape. If the given shape is, e.g., \( (m, n, k) \), then \( m \times n \times k \) samples are drawn.

**See Also:**

- `scipy.stats.distributions.lomax.pdf`
  
  probability density function, distribution or cumulative density function, etc.

- `scipy.stats.distributions.genpareto.pdf`
  
  probability density function, distribution or cumulative density function, etc.

**Notes**

The probability density for the Pareto distribution is

\[
p(x) = \frac{am^a}{x^{a+1}}
\]

where \( a \) is the shape and \( m \) the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called “fat-tailed” distributions.

**References**

[R216], [R217], [R218], [R219]

**Examples**

Draw samples from the distribution:

```python
g, m = 3., 1. # shape and mode
gs = np.random.pareto(g, 1000) + m

Display the histogram of the samples, along with the probability density function:

```
numpy.random.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

Parameters

- **lam**: float
  - Expectation of interval, should be >= 0.

- **size**: int or tuple of ints, optional
  - Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Notes

The Poisson distribution

\[
f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}
\]

For events with an expected separation \( \lambda \) the Poisson distribution \( f(k; \lambda) \) describes the probability of \( k \) events occurring within the observed interval \( \lambda \).

Because the output is limited to the range of the C long type, a ValueError is raised when lam is within 10 sigma of the maximum representable value.

References

[R220], [R221]

Examples

Draw samples from the distribution:

```python
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```
**numpy.random.power** *(a, size=None)*

Draws samples in [0, 1] from a power distribution with positive exponent \(a - 1\).

Also known as the power function distribution.

**Parameters**
- **a** : float
  parameter, > 0
- **size** : tuple of ints
  Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m * n * k\) samples are drawn.

**Returns**
- **samples** : {ndarray, scalar}
  The returned samples lie in [0, 1].

**Raises**
- **ValueError**
  If \(a<1\).

**Notes**

The probability density function is

\[
P(x; a) = a x^{a-1}, \ 0 \leq x \leq 1, \ a > 0.
\]

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

**References**

[R222], [R223]
Examples

Draw samples from the distribution:

```python
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt

>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```python
>>> from scipy import stats

>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0, 1, 100)
>>> powpdf = stats.powerlaw.pdf(xx, 5)

>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx, powpdf, 'r-')
>>> plt.title('np.random.power(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx, powpdf, 'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```
>>> plt.title('inverse of stats.pareto(5)')

Inverse of 1 + \texttt{np.random.pareto(5)}
numpy.random.rayleigh(scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The $\chi$ and Weibull distributions are generalizations of the Rayleigh.

**Parameters**

- **scale** : scalar
  Scale, also equals the mode. Should be $\geq 0$.

- **size** : int or tuple of ints, optional
  Shape of the output. Default is None, in which case a single value is returned.

**Notes**

The probability density function for the Rayleigh distribution is

$$P(x; \text{scale}) = \frac{x}{\text{scale}^2} e^{\frac{x^2}{\text{scale}^2}}$$

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

**References**

[R224], [R225]

**Examples**

Draw values from the distribution and plot the histogram

```python
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```python
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```
The percentage of waves larger than 3 meters is:

```python
>>> 100.*sum(s>3)/1000000.
0.08730000000000003
```

```{.python}
import numpy as np

np.random.standard_cauchy(size=None)
```

Also known as the Lorentz distribution.

**Parameters**

- `size` : int or tuple of ints
  
  Shape of the output.

**Returns**

- `samples` : ndarray or scalar
  
  The drawn samples.

**Notes**

The probability density function for the full Cauchy distribution is

\[
P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[ 1 + \left( \frac{x-x_0}{\gamma} \right)^2 \right]}
\]

and the Standard Cauchy distribution just sets \( x_0 = 0 \) and \( \gamma = 1 \)

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

**References**

[R227], [R228], [R229]

**Examples**

Draw samples and plot the distribution:

```python
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

```{.python}
import numpy as np

np.random.standard_exponential(size=None)
```

Draw samples from the standard exponential distribution.

**standard_exponential** is identical to the exponential distribution with a scale parameter of 1.

**Parameters**

- `size` : int or tuple of ints
  
  Shape of the output.

**Returns**

- `out` : float or ndarray
Drawn samples.

**Examples**
Output a 3x8000 array:

```python
>>> n = np.random.standard_exponential((3, 8000))
```

`numpy.random.standard_gamma(shape, size=None)`

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated “k”) and scale=1.

**Parameters**
- `shape`: float
  - Parameter, should be > 0.
- `size`: int or tuple of ints
  - Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

**Returns**
- `samples`: ndarray or scalar
  - The drawn samples.

**See Also:**
- `scipy.stats.distributions.gamma`
  - probability density function, distribution or cumulative density function, etc.

**Notes**
The probability density for the Gamma distribution is

\[ p(x) = x^{k-1} e^{-x/\theta} / \theta^k \Gamma(k), \]

where \( k \) is the shape and \( \theta \) the scale, and \( \Gamma \) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

**References**
[R230], [R231]

**Examples**
Draw samples from the distribution:

```python
>>> shape, scale = 2., 1.  # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:
```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

>>> count, bins, ignored = plt.hist(s, 50, normed=True)

>>> y = bins***(shape-1) * ((np.exp(-bins/scale))/
... (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

```python
numpy.random.standard_normal(size=None)
Returns samples from a Standard Normal distribution (mean=0, stdev=1).

Parameters
    size : int or tuple of ints, optional
       Output shape. Default is None, in which case a single value is returned.

Returns
    out : float or ndarray
       Drawn samples.

Examples
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 , 0.78096262, -0.89086505, ..., 0.49876311, #random
        -0.38672696, -0.4685006 ])
>>> s.shape
(8000,)

numpy.random.standard_t(df, size=None)
Standard Student’s t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As df gets large, the result resembles that of the standard normal distribution (standard_normal).
```
Parameters

- **df**: int
  Degrees of freedom, should be > 0.

- **size**: int or tuple of ints, optional
  Output shape. Default is None, in which case a single value is returned.

Returns

- **samples**: ndarray or scalar
  Drawn samples.

Notes

The probability density function for the t distribution is

\[ P(x, df) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df} \Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-(df+1)/2} \]

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

[R232], [R233]

Examples

From Dalgaard page 83 [R232], suppose the daily energy intake for 11 women in Kj is:

```python
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, ..., 7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```python
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
>>> (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```python
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.
numpy.random.\texttt{triangular}(left, mode, right, size=None)

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

\textbf{Parameters}

- \texttt{left} : scalar
  Lower limit.

- \texttt{mode} : scalar
  The value where the peak of the distribution occurs. The value should fulfill the condition \texttt{left} \leq \texttt{mode} \leq \texttt{right}.

- \texttt{right} : scalar
  Upper limit, should be larger than \texttt{left}.

- \texttt{size} : int or tuple of ints, optional
  Output shape. Default is None, in which case a single value is returned.

\textbf{Returns}

- \texttt{samples} : ndarray or scalar
  The returned samples all lie in the interval \texttt{[left, right]}.

\textbf{Notes}

The probability density function for the Triangular distribution is

\[ P(x; l, m, r) = \begin{cases} 
\frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\
\frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\
0 & \text{otherwise.} 
\end{cases} \]

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

\textbf{References}

[R234]

\textbf{Examples}

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200, 
               normed=True)
>>> plt.show()
```
numpy.random.uniform(low=0.0, high=1.0, size=1)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by uniform.

**Parameters**

- **low**: float, optional
  
  Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

- **high**: float
  
  Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

- **size**: int or tuple of ints, optional
  
  Shape of output. If the given size is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

**Returns**

- **out**: ndarray
  
  Drawn samples, with shape size.

**See Also**

- **randint**
  
  Discrete uniform distribution, yielding integers.

- **random_integers**
  
  Discrete uniform distribution over the closed interval [low, high].

- **random_sample**
  
  Floats uniformly distributed over [0, 1).

- **random**
  
  Alias for random_sample.
**rand**

Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over \([0, 1)\).

**Notes**

The probability density function of the uniform distribution is

\[
p(x) = \frac{1}{b-a}
\]

anywhere within the interval \([a, b)\), and zero elsewhere.

**Examples**

Draw samples from the distribution:

```python
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```python
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

**numpy.random.vonmises**

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-\(\pi\), \(\pi\)].
The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

**Parameters**

- **mu**: float
  Mode (“center”) of the distribution.
- **kappa**: float
  Dispersion of the distribution, has to be >=0.
- **size**: int or tuple of int
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

**Returns**

- **samples**: scalar or ndarray
  The returned samples, which are in the interval [-pi, pi].

**See Also:**

*scipy.stats.distributions.vonmises*

probability density function, distribution, or cumulative density function, etc.

**Notes**

The probability density for the von Mises distribution is

\[ p(x) = \frac{e^{\kappa \cos(x - \mu)}}{2\pi I_0(\kappa)}, \]

where \( \mu \) is the mode and \( \kappa \) the dispersion, and \( I_0(\kappa) \) is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

**References**


**Examples**

Draw samples from the distribution:

```python
>>> mu, kappa = 0.0, 4.0  # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```
numpy.random.wald(mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters

mean : scalar
   Distribution mean, should be > 0.

scale : scalar
   Scale parameter, should be >= 0.

size : int or tuple of ints, optional
   Output shape. Default is None, in which case a single value is returned.

Returns

samples : ndarray or scalar
   Drawn sample, all greater than zero.

Notes

The probability density function for the Wald distribution is

\[ P(x; \text{mean}, \text{scale}) = \frac{\text{scale} \cdot e^{-\frac{\text{scale}(x-\text{mean})^2}{2\text{mean}^2}}}{2\pi x^3} \]

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

References

[R235], [R236], [R237]
Examples

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)
>>> plt.show()
```

```
0 10 20 30 40 50 60 70 80
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
```

```python
numpy.random.weibull(a, size=None)
```

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter \( a \).

\[ X = \left(-\ln(U)\right)^{1/a} \]

Here, \( U \) is drawn from the uniform distribution over \((0,1]\).

The more common 2-parameter Weibull, including a scale parameter \( \lambda \) is just \( X = \lambda \left(-\ln(U)\right)^{1/a} \).

Parameters
- `a`: float
  Shape of the distribution.
- `size`: tuple of ints
  Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

See Also:
- `scipy.stats.distributions.weibull_max`
- `scipy.stats.distributions.weibull_min`
- `scipy.stats.distributions.genextreme`
- `gumbel`

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.
The probability density for the Weibull distribution is

\[ p(x) = \frac{a}{\lambda} \left(\frac{x}{\lambda}\right)^{a-1} e^{-(x/\lambda)^a}, \]

where \( a \) is the shape and \( \lambda \) the scale.

The function has its peak (the mode) at \( \lambda \left(\frac{a-1}{a}\right)^{1/a} \).

When \( a = 1 \), the Weibull distribution reduces to the exponential distribution.

**References**

[R238], [R239], [R240]

**Examples**

Draw samples from the distribution:

```python
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1, 100.) / 50.
>>> def weib(x, n, a):
...      return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>>> count, bins, ignored = plt.hist(np.random.weibull(5., 1000))
>>> x = np.arange(1, 100.) / 50.
>>> scale = count.max() / weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

**numpy.random.zipf** (\( a, size=None \))

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter \( a > 1 \).
The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf’s law: the frequency of an item is inversely proportional to its rank in a frequency table.

**Parameters**

- **a** : float > 1
  Distribution parameter.

- **size** : int or tuple of int, optional
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length size is returned. The default is None, in which case a single scalar is returned.

**Returns**

- **samples** : scalar or ndarray
  The returned samples are greater than or equal to one.

**See Also:**

- `scipy.stats.distributions.zipf`
  probability density function, distribution, or cumulative density function, etc.

**Notes**

The probability density for the Zipf distribution is

\[ p(x) = \frac{x^{-a}}{\zeta(a)}. \]

where \( \zeta \) is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

**References**


**Examples**

Draw samples from the distribution:

```python
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```
3.27.4 Random generator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RandomState</code></td>
<td>Container for the Mersenne Twister pseudo-random number generator.</td>
</tr>
<tr>
<td><code>seed(seed)</code></td>
<td>Seed the generator.</td>
</tr>
<tr>
<td><code>get_state()</code></td>
<td>Return a tuple representing the internal state of the generator.</td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Set the internal state of the generator from a tuple.</td>
</tr>
</tbody>
</table>

**class numpy.random.RandomState**

Container for the Mersenne Twister pseudo-random number generator.

RandomState exposes a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument `size` that defaults to None. If `size` is None, then a single value is generated and returned. If `size` is an integer, then a 1-D array filled with generated values is returned. If `size` is a tuple, then an array with that shape is filled and returned.

**Parameters**

- `seed` : {None, int, array_like}, optional
  Random seed initializing the pseudo-random number generator. Can be an integer, an array (or other sequence) of integers of any length, or `None` (the default). If `seed` is None, then `RandomState` will try to read data from `/dev/urandom` (or the Windows analogue) if available or seed from the clock otherwise.

**Notes**

The Python stdlib module “random” also contains a Mersenne Twister pseudo-random number generator with a number of methods that are similar to the ones available in `RandomState`. `RandomState`, besides being NumPy-aware, has the advantage that it provides a much larger number of probability distributions to choose from.

**Methods**

- `beta(a, b[, size])` The Beta distribution over `[0, 1]`.  

Continued on next page
Table 3.180 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binomial(n, p[, size])</code></td>
<td>Draw samples from a binomial distribution.</td>
</tr>
<tr>
<td><code>bytes(length)</code></td>
<td>Return random bytes.</td>
</tr>
<tr>
<td><code>chisquare(df[, size])</code></td>
<td>Draw samples from a chi-square distribution.</td>
</tr>
<tr>
<td><code>choice(a[, size, replace, p])</code></td>
<td>Generates a random sample from a given 1-D array.</td>
</tr>
<tr>
<td><code>dirichlet(alpha[, size])</code></td>
<td>Draw samples from the Dirichlet distribution.</td>
</tr>
<tr>
<td><code>exponential([scale, size])</code></td>
<td>Exponential distribution.</td>
</tr>
<tr>
<td><code>f(dfnum, dfden[, size])</code></td>
<td>Draw samples from a F distribution.</td>
</tr>
<tr>
<td><code>gamma(shape[, scale, size])</code></td>
<td>Draw samples from a Gamma distribution.</td>
</tr>
<tr>
<td><code>geometric(p[, size])</code></td>
<td>Draw samples from the geometric distribution.</td>
</tr>
<tr>
<td><code>get_state()</code></td>
<td>Return a tuple representing the internal state of the generator.</td>
</tr>
<tr>
<td><code>gumbel(loc, scale, size)</code></td>
<td>Gumbel distribution.</td>
</tr>
<tr>
<td><code>hypergeometric(ngood, nbad, nsample[, size])</code></td>
<td>Draw samples from a Hypergeometric distribution.</td>
</tr>
<tr>
<td><code>laplace(loc[, scale, size])</code></td>
<td>Draw samples from the Laplace or double exponential distribution with</td>
</tr>
<tr>
<td><code>logistic(loc[, scale, size])</code></td>
<td>Draw samples from a Logistic distribution.</td>
</tr>
<tr>
<td><code>lognormal(mean, sigma, size)</code></td>
<td>Return samples drawn from a log-normal distribution.</td>
</tr>
<tr>
<td><code>logseries(p[, size])</code></td>
<td>Draw samples from a Logarithmic Series distribution.</td>
</tr>
<tr>
<td><code>multinomial(n, pvals[, size])</code></td>
<td>Draw samples from a multinomial distribution.</td>
</tr>
<tr>
<td><code>multivariate_normal(mean, cov[, size])</code></td>
<td>Draw random samples from a multivariate normal distribution.</td>
</tr>
<tr>
<td><code>negative_binomial(n, p[, size])</code></td>
<td>Draw samples from a negative_binomial distribution.</td>
</tr>
<tr>
<td><code>noncentral_chisquare(df, nonc[, size])</code></td>
<td>Draw samples from a noncentral chi-square distribution.</td>
</tr>
<tr>
<td><code>noncentral_f(dfnum, dfden, nonc[, size])</code></td>
<td>Draw samples from the noncentral F distribution.</td>
</tr>
<tr>
<td><code>normal(loc[, scale, size])</code></td>
<td>Draw random samples from a normal (Gaussian) distribution.</td>
</tr>
<tr>
<td><code>pareto(a[, size])</code></td>
<td>Draw samples from a Pareto II or Lomax distribution with specified shape.</td>
</tr>
<tr>
<td><code>permutation(x)</code></td>
<td>Randomly permute a sequence, or return a permuted range.</td>
</tr>
<tr>
<td><code>poisson(lam[, size])</code></td>
<td>Draw samples from a Poisson distribution.</td>
</tr>
<tr>
<td><code>power(a[, size])</code></td>
<td>Draws samples in [0, 1] from a power distribution with positive exponent a - 1.</td>
</tr>
<tr>
<td><code>rand(d0, d1, ..., dn)</code></td>
<td>Return random integers from low (inclusive) to high (exclusive).</td>
</tr>
<tr>
<td><code>randint(low[, high, size])</code></td>
<td>Return a sample (or samples) from the “standard normal” distribution.</td>
</tr>
<tr>
<td><code>random_integers(low[, high, size])</code></td>
<td>Return random integers between low and high, inclusive.</td>
</tr>
<tr>
<td><code>random_sample([size])</code></td>
<td>Return random floats in the half-open interval [0.0, 1.0).</td>
</tr>
<tr>
<td><code>rayleigh([scale, size])</code></td>
<td>Draw samples from a Rayleigh distribution.</td>
</tr>
<tr>
<td><code>seed([seed])</code></td>
<td>Seed the generator.</td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Set the internal state of the generator from a tuple.</td>
</tr>
<tr>
<td><code>shuffle(x)</code></td>
<td>Modify a sequence in-place by shuffling its contents.</td>
</tr>
<tr>
<td><code>standard_cauchy([size])</code></td>
<td>Standard Cauchy distribution with mode = 0.</td>
</tr>
<tr>
<td><code>standard_exponential([size])</code></td>
<td>Draw samples from the standard exponential distribution.</td>
</tr>
<tr>
<td><code>standard_gamma(shape[, size])</code></td>
<td>Draw samples from a Standard Gamma distribution.</td>
</tr>
<tr>
<td><code>standard_normal([size])</code></td>
<td>Returns samples from a Standard Normal distribution (mean=0, stdev=1).</td>
</tr>
<tr>
<td><code>standard_t(df[, size])</code></td>
<td>Standard Student’s t distribution with df degrees of freedom.</td>
</tr>
<tr>
<td><code>tomaxint([size])</code></td>
<td>Random integers between 0 and sys.maxint, inclusive.</td>
</tr>
<tr>
<td><code>triangular(left, right[, size])</code></td>
<td>Draw samples from the triangular distribution.</td>
</tr>
<tr>
<td><code>uniform([low, high, size])</code></td>
<td>Draw samples from a uniform distribution.</td>
</tr>
<tr>
<td><code>vonmises(mu, kappa[, size])</code></td>
<td>Draw samples from a von Mises distribution.</td>
</tr>
<tr>
<td><code>wald(mean, scale[, size])</code></td>
<td>Draw samples from a Wald, or Inverse Gaussian, distribution.</td>
</tr>
<tr>
<td><code>weibull(a[, size])</code></td>
<td>Weibull distribution.</td>
</tr>
<tr>
<td><code>zipf(a[, size])</code></td>
<td>Draw samples from a Zipf distribution.</td>
</tr>
</tbody>
</table>

RandomState.beta(a, b, size=None)

The Beta distribution over [0, 1].
The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

\[ f(x; \alpha, \beta) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1} (1 - x)^{\beta-1}, \]

where the normalisation, B, is the beta function,

\[ B(\alpha, \beta) = \int_0^1 t^{\alpha-1} (1 - t)^{\beta-1} dt. \]

It is often seen in Bayesian inference and order statistics.

**Parameters**
- \(a\): float
  - Alpha, non-negative.
- \(b\): float
  - Beta, non-negative.
- size: tuple of ints, optional
  - The number of samples to draw. The output is packed according to the size given.

**Returns**
- out: ndarray
  - Array of the given shape, containing values drawn from a Beta distribution.

**RandomState.binomial**

Draw samples from a binomial distribution.

Samples are drawn from a Binomial distribution with specified parameters, \(n\) trials and \(p\) probability of success where \(n\) an integer \(\geq 0\) and \(p\) is in the interval \([0,1]\). (\(n\) may be input as a float, but it is truncated to an integer in use)

**Parameters**
- \(n\): float (but truncated to an integer)
  - Parameter, \(\geq 0\).
- \(p\): float
  - Parameter, \(\geq 0\) and \(\leq 1\).
- size: \{tuple, int\}
  - Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

**Returns**
- samples: \{ndarray, scalar\}
  - where the values are all integers in \([0, n]\).

**See Also:**
- scipy.stats.distributions.binom
  - probability density function, distribution or cumulative density function, etc.
Notes

The probability density for the Binomial distribution is

\[ P(N) = \binom{n}{N} p^N (1 - p)^{n-N}, \]

where \( n \) is the number of trials, \( p \) is the probability of success, and \( N \) is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product \( p^n \leq 5 \), where \( p \) = population proportion estimate, and \( n \) = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then \( p = 4/15 = 27\% \). 0.27*15 = 4, so the binomial distribution should be used in this case.

References

[R123], [R124], [R125], [R126], [R127]

Examples

Draw samples from the distribution:

```python
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)  # result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```python
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
answer = 0.38885, or 38%.
```

RandomState.bytes(length)

Return random bytes.

Parameters

- **length**: int

  Number of random bytes.

Returns

- **out**: str

  String of length length.

Examples

```python
>>> np.random.bytes(10)
'b\xe8\xc4\x85\xf7\x02\x22\xb2\x8b\xf4'  #random
```

RandomState.chisquare(df, size=None)

Draw samples from a chi-square distribution.

When \( df \) independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Parameters

- **df**: int

3.27. Random sampling (numpy.random)
Number of degrees of freedom.

**size** : tuple of ints, int, optional
Size of the returned array. By default, a scalar is returned.

Returns

**output** : ndarray
Samples drawn from the distribution, packed in a size-shaped array.

Raises

**ValueError**
When \( df \leq 0 \) or when an inappropriate size (e.g. size=-1) is given.

Notes

The variable obtained by summing the squares of \( df \) independent, standard normally distributed random variables:

\[
Q = \sum_{i=0}^{df} X_i^2
\]

is chi-square distributed, denoted

\[
Q \sim \chi^2_k.
\]

The probability density function of the chi-squared distribution is

\[
p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},
\]

where \( \Gamma \) is the gamma function,

\[
\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt.
\]

References

NIST/SEMATECH e-Handbook of Statistical Methods

Examples

```python
>>> np.random.chisquare(2,4)
array([ 1.89920014, 9.00867716, 3.13710533, 5.62318272])
```

RandomState.choice(a, size=None, replace=True, p=None)
Generates a random sample from a given 1-D array

New in version 1.7.0.

Parameters

**a** : 1-D array-like or int
If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a was np.arange(n)

**size**: int or tuple of ints, optional
Output shape. Default is None, in which case a single value is returned.

**replace**: boolean, optional
Whether the sample is with or without replacement

**p**: 1-D array-like, optional
The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

**Returns**

- **samples**: 1-D ndarray, shape (size,)
The generated random samples

**Raises**

- **ValueError**
  If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See Also:

- randint, shuffle, permutation

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```python
>>> np.random.choice(5, 3)
array([0, 3, 4])
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```python
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0])
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```python
>>> np.random.choice(5, 3, replace=False)
array([3, 1, 0])
>>> #This is equivalent to np.random.shuffle(np.arange(5))[3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```python
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```python
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'christopher', 'pooh', 'piglet'],
dtype='|S11')
```
RandomState.dirichlet(alpha, size=None)

Draw samples from the Dirichlet distribution.

Draw size samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

Parameters

- alpha : array
  Parameter of the distribution (k dimension for sample of dimension k).
- size : array
  Number of samples to draw.

Returns

- samples : ndarray,
  The drawn samples, of shape (alpha.ndim, size).

Notes

\[ X \approx \prod_{i=1}^{k} x_i^{\alpha_i-1} \]

Uses the following property for computation: for each dimension, draw a random sample y_i from a standard gamma generator of shape alpha_i, then \( X = \frac{1}{\sum_{i=1}^{k} y_i} (y_1, \ldots, y_n) \) is Dirichlet distributed.

References

[R128], [R129]

Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```python
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```

RandomState.exponential(scale=1.0, size=None)

Exponential distribution.

Its probability density function is

\[ f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}), \]

for \( x > 0 \) and 0 elsewhere. \( \beta \) is the scale parameter, which is the inverse of the rate parameter \( \lambda = 1/\beta \). The rate parameter is an alternative, widely used parameterization of the exponential distribution [R132].
The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [R130], or the time between page requests to Wikipedia [R131].

**Parameters**
- **scale**: float
  - The scale parameter, \( \beta = 1/\lambda \).
- **size**: tuple of ints
  - Number of samples to draw. The output is shaped according to size.

**References**
[R130], [R131], [R132]

RandomState.f(dfnum, dfden, size=None)
Draw samples from a F distribution.

Samples are drawn from an F distribution with specified parameters, \( dfnum \) (degrees of freedom in numerator) and \( dfden \) (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

**Parameters**
- **dfnum**: float
  - Degrees of freedom in numerator. Should be greater than zero.
- **dfden**: float
  - Degrees of freedom in denominator. Should be greater than zero.
- **size**: {tuple, int}, optional
  - Output shape. If the given shape is, e.g., \( (m, n, k) \), then \( m \times n \times k \) samples are drawn. By default only one sample is returned.

**Returns**
- **samples**: {ndarray, scalar}
  - Samples from the Fisher distribution.

**See Also:**
scipy.stats.distributions.f
probability density function, distribution or cumulative density function, etc.

**Notes**
The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable \( dfnum \) is the number of samples minus one, the between-groups degrees of freedom, while \( dfden \) is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

**References**
[R133], [R134]
Examples

An example from Glantz[1], pp 47-40. Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children’s blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```python
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is:

```python
>>> sorted(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`RandomState.gamma(shape, scale=1.0, size=None)`

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, `shape` (sometimes designated “k”) and `scale` (sometimes designated “theta”), where both parameters are > 0.

Parameters

- `shape`: scalar > 0
  The shape of the gamma distribution.
- `scale`: scalar > 0, optional
  The scale of the gamma distribution. Default is equal to 1.
- `size`: shape_tuple, optional
  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns

- `out`: ndarray, float
  Returns one sample unless `size` parameter is specified.

See Also:

- `scipy.stats.distributions.gamma`
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gamma distribution is

\[ p(x) = \frac{x^{k-1} e^{-x/\theta}}{\theta^k \Gamma(k)}, \]

where \( k \) is the shape and \( \theta \) the scale, and \( \Gamma \) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.
References

[R135], [R136]

Examples

Draw samples from the distribution:

```python
>>> shape, scale = 2., 2. # mean and dispersion
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) / ... (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

RandomState.geometric(p, size=None)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, \( k = 1, 2, \ldots \).

The probability mass function of the geometric distribution is

\[
f(k) = (1 - p)^{k-1} p
\]

where \( p \) is the probability of success of an individual trial.

Parameters

- **p**: float
  - The probability of success of an individual trial.

- **size**: tuple of ints
Number of values to draw from the distribution. The output is shaped according to `size`.

**Returns**

- `out`: ndarray

  Samples from the geometric distribution, shaped according to `size`.

**Examples**

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```python
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```python
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

**RandomState.get_state()**

Return a tuple representing the internal state of the generator.

For more details, see `set_state`.

**Returns**

- `out`: tuple(str, ndarray of 624 uints, int, int, float)

  The returned tuple has the following items:

  1. the string 'MT19937'.
  2. a 1-D array of 624 unsigned integer keys.
  3. an integer `pos`.
  4. an integer `has_gauss`.
  5. a float `cached_gaussian`.

**See Also:**

- `set_state`

**Notes**

`set_state` and `get_state` are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

**RandomState.gumbel(loc=0.0, scale=1.0, size=None)**

Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

**Parameters**

- `loc`: float

  The location of the mode of the distribution.

- `scale`: float

  The scale parameter of the distribution.

- `size`: tuple of ints

  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.
Returns

out : ndarray
    The samples

See Also:

scipy.stats.gumbel_l, scipy.stats.gumbel_r

scipy.stats.genextreme
    probability density function, distribution, or cumulative density function, etc. for each of the above

weibull

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one
of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems.
The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions
with "exponential-like" tails.

The probability density for the Gumbel distribution is

\[ p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta e^{-e^{-(x-\mu)/\beta}}}, \]

where \( \mu \) is the mode, a location parameter, and \( \beta \) is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology
literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed
and rainfall rates. It is a “fat-tailed” distribution - the probability of an event in the tail of the distribution is
larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods
were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions,
which also includes the Weibull and Frechet.

The function has a mean of \( \mu + 0.57721 \beta \) and a variance of \( \frac{\pi^2}{6} \beta^2 \).

References

Reiss, R.-D. and Thomas, M., *Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology

Examples

Draw samples from the distribution:

```python
count, bins, ignored = plt.hist(s, 30, normed=True)
```

```
import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
... * np.exp( -np.exp( -(bins - mu) /beta) ),
... linewidth=2, color='r')
>>> plt.show()
```
Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```python
>>> means = []
>>> maxima = []
>>> for i in range(0,1000):
...     a = np.random.normal(mu, beta, 1000)
...     means.append(a.mean())
...     maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = np.std(maxima)*np.pi/np.sqrt(6)
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```
Draw samples from a Hypergeometric distribution.

Samples are drawn from a Hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

**Parameters**

- **ngood**: int or array_like
  - Number of ways to make a good selection. Must be nonnegative.

- **nbad**: int or array_like
  - Number of ways to make a bad selection. Must be nonnegative.

- **nsample**: int or array_like
  - Number of items sampled. Must be at least 1 and at most ngood + nbad.

- **size**: int or tuple of int
  - Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

**Returns**

- **samples**: ndarray or scalar
  - The values are all integers in [0, n].

**See Also:**

- `scipy.stats.distributions.hypergeom`
  - probability density function, distribution or cumulative density function, etc.

**Notes**

The probability density for the Hypergeometric distribution is

\[
P(x) = \binom{m}{n} \binom{N-m}{n-x} / \binom{N}{n},
\]

where \(0 \leq x \leq m\) and \(n + m - N \leq x \leq n\)

for \(P(x)\) the probability of \(x\) successes, \(n = \text{ngood}\), \(m = \text{nbad}\), and \(N = \text{number of samples}\).

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the Hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the Binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the Binomial.

**References**

[R137], [R138], [R139]
Examples

Draw samples from the distribution:

```python
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
# note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```python
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
# answer = 0.003 ... pretty unlikely!
```

RandomState.laplace(loc=0.0, scale=1.0, size=None)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

**Parameters**

- `loc` : float
  - The position, \( \mu \), of the distribution peak.

- `scale` : float
  - \( \lambda \), the exponential decay.

**Notes**

It has the probability density function

\[
f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).
\]

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in Economics and Health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

**References**

[R140], [R141], [R142], [R143]

**Examples**

Draw samples from the distribution

```python
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:
```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)

Plot Gaussian for comparison:

```python
>>> g = (1/(scale * np.sqrt(2 * np.pi)) * 
...    np.exp( - (x - loc)**2 / (2 * scale**2) ))
``` python

```python
>>> plt.plot(x,g)
```

RandomState.logistic(loc=0.0, scale=1.0, size=None)
Draw samples from a Logistic distribution.

Samples are drawn from a Logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

**Parameters**

- **loc** : float
- **scale** : float > 0.
- **size** : {tuple, int}

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

**Returns**

- **samples** : {ndarray, scalar}

where the values are all integers in [0, n].

**See Also:**

scipy.stats.distributions.logistic

probability density function, distribution or cumulative density function, etc.
Notes

The probability density for the Logistic distribution is

\[ P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}, \]

where \( \mu \) = location and \( s \) = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References

[R144], [R145], [R146]

Examples

Draw samples from the distribution:

```python
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> count, bins, ignored = plt.hist(s, bins=50)

# plot against distribution
>>> def logist(x, loc, scale):
...     return exp((loc-x)/scale)/(scale*(1+exp((loc-x)/scale))**2)
>>> plt.plot(bins, logist(bins, loc, scale)*count.max()\
...          /logist(bins, loc, scale).max())
>>> plt.show()
```

RandomState.lognormal(mean=0.0, sigma=1.0, size=None)

Return samples drawn from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Parameters

- **mean**: float
  Mean value of the underlying normal distribution
- **sigma**: float, > 0.
  Standard deviation of the underlying normal distribution
- **size**: tuple of ints
  Output shape. If the given shape is e.g., (m, n, k), then \( m \times n \times k \) samples are drawn.

Returns

- **samples**: ndarray or float
  The desired samples. An array of the same shape as \( size \) if given, if \( size \) is None a float is returned.

See Also:
scipy.stats.lognorm
probability density function, distribution, cumulative density function, etc.

Notes
A variable $x$ has a log-normal distribution if $\log(x)$ is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{-\frac{(\log(x) - \mu)^2}{2\sigma^2}}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the product of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the sum of a large number of independent, identically-distributed variables.

References

Examples
Draw samples from the distribution:

```python
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='mid')
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi)))
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```python
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.random(100)
...     b.append(np.product(a))

>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, normed=True, align='center')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2) / (2 * sigma**2))
... / (x * sigma * np.sqrt(2 * np.pi))

>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()

NumPy Reference, Release 1.8.1

RandomState.logseries(p, size=None)
Draw samples from a Logarithmic Series distribution.

Samples are drawn from a Log Series distribution with specified parameter, p (probability, 0 < p < 1).

Parameters
- **loc**: float
- **scale**: float > 0.
- **size**: {tuple, int}
  - Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
- **samples**: {ndarray, scalar}
  - where the values are all integers in [0, n].

See Also:
- scipy.stats.distributions.logser
  - probability density function, distribution or cumulative density function, etc.

Notes
The probability density for the Log Series distribution is

\[ P(k) = \frac{-p^k}{k \ln(1 - p)} \]

where p = probability.

The Log Series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References
[R147], [R148], [R149], [R150]

Examples
Draw samples from the distribution:

```python
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution
```python
>>> def logseries(k, p):
...     return -p**k/(k*log(1-p))

>>> plt.plot(bins, logseries(bins, a)*count.max() /
           logseries(bins, a).max(), 'r')

>>> plt.show()
```

RandomState.multinomial(n, pvals, size=None)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of p possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents n such experiments. Its values, $X_i = \{X_0, X_1, \ldots, X_p\}$, represent the number of times the outcome was i.

**Parameters**

- **n**: int
  - Number of experiments.
- **pvals**: sequence of floats, length p
  - Probabilities of each of the p different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1`).
- **size**: tuple of ints
  - Given a size of (M, N, K), then M*N*K samples are drawn, and the output shape becomes (M, N, K, p), since each sample has shape (p,).

**Examples**

Throw a dice 20 times:

```python
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```python
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 4, 3, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```python
>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])
```

RandomState.multivariate_normal(mean, cov[, size])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or “center”) and variance (standard deviation, “width,” squared) of the one-dimensional normal distribution.

**Parameters**

- **mean**: 1-D array_like, of length N
Mean of the N-dimensional distribution.

cov : 2-D array_like, of shape (N, N)

Covariance matrix of the distribution. Must be symmetric and positive semi-definite for “physically meaningful” results.

size : int or tuple of ints, optional

Given a shape of, for example, (m, n, k), m*n*k samples are generated, and packed in an m-by-n-by-k arrangement. Because each sample is N-dimensional, the output shape is (m, n, k, N). If no shape is specified, a single (N-D) sample is returned.

Returns

out : ndarray

The drawn samples, of shape size, if that was provided. If not, the shape is (N, ). In other words, each entry out[i, j, ..., :] is an N-dimensional value drawn from the distribution.

Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$ (i.e. its “spread”).

Instead of specifying the full covariance matrix, popular approximations include:

•Spherical covariance (cov is a multiple of the identity matrix)
•Diagonal covariance (cov has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```python
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance, points lie on x or y-axis

>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

References


Examples

```python
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```
The following is probably true, given that 0.6 is roughly twice the standard deviation:

```python
>>> print list(x[0,0,:]-mean) < 0.6
[True, True]
```

```
RandomState.negative_binomial(n, p, size=None)
Draw samples from a negative_binomial distribution.

Samples are drawn from a negative_Binomial distribution with specified parameters, n trials and p probability of success where n is an integer > 0 and p is in the interval [0, 1).

Parameters
n : int
    Parameter, > 0.
p : float
    Parameter, >= 0 and <=1.
size : int or tuple of ints
    Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns
samples : int or ndarray of ints
    Drawn samples.

Notes
The probability density for the Negative Binomial distribution is

\[ P(N; n, p) = \binom{N + n - 1}{n - 1} p^n (1 - p)^N, \]

where \( n - 1 \) is the number of successes, \( p \) is the probability of success, and \( N + n - 1 \) is the number of trials.

The negative binomial distribution gives the probability of \( n-1 \) successes and \( N \) failures in \( N+n-1 \) trials, and success on the \( (N+n)th \) trial.

If one throws a die repeatedly until the third time a “1” appears, then the probability distribution of the number of non-“1”’s that appear before the third “1” is a negative binomial distribution.

References
[R151], [R152]

Examples
Draw samples from the distribution:
A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```python
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...    probability = sum(s<i) / 100000.
...    print i, "wells drilled, probability of one success =", probability
```
RandomState.noncentral_chisquare(df, nonc, size=None)

Draw samples from a noncentral chi-square distribution.

The noncentral χ² distribution is a generalisation of the χ² distribution.

Parameters

df : int
    Degrees of freedom, should be >= 1.

nonc : float
    Non-centrality, should be > 0.

size : int or tuple of ints
    Shape of the output.

Notes

The probability density function for the noncentral Chi-square distribution is

\[ P(x; df, nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x), \]

where \( Y_{q} \) is the Chi-square with \( q \) degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

References

[R153], [R154]

Examples

Draw values from the distribution and plot the histogram

```python
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                     bins=200, normed=True)
>>> plt.show()
```
Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```python
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 0.0000001, 100000),
...                   bins=np.arange(0., 25, .1), normed=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                     bins=np.arange(0., 25, .1), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```python
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, normed=True)
>>> plt.show()
```
RandomState.noncentral_f (dfnum, dfden, nonc, size=None)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, dfnum (degrees of freedom in numerator) and dfden (degrees of freedom in denominator), where both parameters > 1. nonc is the non-centrality parameter.

Parameters

- dfnum : int
  Parameter, should be > 1.

- dfden : int
  Parameter, should be > 1.

- nonc : float
  Parameter, should be >= 0.

- size : int or tuple of ints
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Returns

- samples : scalar or ndarray
  Drawn samples.

Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References


Examples
In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We’ll plot the two probability distributions for comparison.

```python
>>> dfnum = 3  # between group deg of freedom
>>> dfden = 20  # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, normed=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

RandomState.normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [R156], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [R156].

Parameters

- **loc**: float
  
  Mean (“centre”) of the distribution.

- **scale**: float
  
  Standard deviation (spread or “width”) of the distribution.

- **size** : tuple of ints
  
  Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

See Also:

- scipy.stats.distributions.norm
  
  probability density function, distribution or cumulative density function, etc.

Notes

The probability density for the Gaussian distribution is

\[ p(x) = \frac{1}{\sqrt{2\pi \sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \]

where \( \mu \) is the mean and \( \sigma \) the standard deviation. The square of the standard deviation, \( \sigma^2 \), is called the variance.
The function has its peak at the mean, and its “spread” increases with the standard deviation (the function reaches 0.607 times its maximum at $x+\sigma$ and $x-\sigma$ [R156]). This implies that `numpy.random.normal` is more likely to return samples lying close to the mean, rather than those far away.

**References**

[R155], [R156]

**Examples**

Draw samples from the distribution:

```python
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```python
>>> abs(mu - np.mean(s)) < 0.01
True
>>> abs(sigma - np.std(s, ddof=1)) < 0.01
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...           np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...           linewidth=2, color='r')
>>> plt.show()
```

RandomState. `pareto`(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding the location parameter $m$, see below. The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is $m$, where the standard Pareto distribution has location $m=1$. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.
The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the “80-20 rule”. In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

**Parameters**
- **shape**: float, > 0.
  Shape of the distribution.
- **size**: tuple of ints
  Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn.

**See Also:**
- `scipy.stats.distributions.lomax.pdf`
  probability density function, distribution or cumulative density function, etc.
- `scipy.stats.distributions.genpareto.pdf`
  probability density function, distribution or cumulative density function, etc.

**Notes**
The probability density for the Pareto distribution is

\[
p(x) = \frac{am^a}{x^{a+1}}
\]

where \(a\) is the shape and \(m\) the location

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called “fat-tailed” distributions.

**References**
[R157], [R158], [R159], [R160]

**Examples**
Draw samples from the distribution:

```python
>>> a, m = 3., 1. # shape and mode
>>> s = np.random.pareto(a, 1000) + m
```

Display the histogram of the samples, along with the probability density function:

```python
import matplotlib.pyplot as plt

>>> count, bins, ignored = plt.hist(s, 100, normed=True, align='center')
>>> fit = a*m**a/bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit),linewidth=2, color='r')
```

RandomState.permutation(x)
Randomly permute a sequence, or return a permuted range.

If \(x\) is a multi-dimensional array, it is only shuffled along its first index.
Parameters

x : int or array_like

If x is an integer, randomly permute np.arange(x). If x is an array, make a copy and shuffle the elements randomly.

Returns

out : ndarray

Permuted sequence or array range.

Examples

>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])

>>> np.random.permutation([1, 4, 9, 12, 15])
array([15, 1, 9, 4, 12])

>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])

RandomState.poisson(lam=1.0, size=None)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

Parameters

lam : float

Expectation of interval, should be >= 0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

Notes

The Poisson distribution

\[ f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \]

For events with an expected separation \( \lambda \) the Poisson distribution \( f(k; \lambda) \) describes the probability of \( k \) events occurring within the observed interval \( \lambda \).

Because the output is limited to the range of the C long type, a ValueError is raised when \( \text{lam} \) is within 10 sigma of the maximum representable value.

References

[R161], [R162]

Examples

Draw samples from the distribution:
```python
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)

Display histogram of the sample:
```n```
```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, normed=True)
>>> plt.show()
```

RandomState.power(a, size=None)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Parameters

a : float
    parameter, > 0

size : tuple of ints

Output shape. If the given shape is, e.g., (m, n, k), then
m * n * k samples are drawn.

Returns

samples : {ndarray, scalar}

The returned samples lie in [0, 1].

Raises

ValueError

If a<1.

Notes

The probability density function is

\[ P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0. \]
The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

References
[R163], [R164]

Examples
Draw samples from the distribution:

```python
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```python
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
>>> plt.figure()
>>> plt.hist(rvs, bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```
```python
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')

>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, normed=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

![Graph of np.random.power(5)](image1)

![Graph of inverse of 1 + np.random.pareto(5)](image2)
RandomState.rnd(d0, d1, ..., dn)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over
[0, 1).

Parameters

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should all be positive. If no argument is given a
single Python float is returned.

Returns

out : ndarray, shape (d0, d1, ..., dn)

Random values.

See Also:

random

Notes

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer
to np.random.random_sample.

Examples

>>> np.random.rand(3,2)
array([[ 0.14022471, 0.96360618],
       [ 0.37601032, 0.25528411],
       [ 0.49313049, 0.94909878]])

RandomState.randint(low, high=None, size=None)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the “discrete uniform” distribution in the “half-open” interval [low, high). If
high is None (the default), then results are from [0, low).

Parameters

low : int
Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the highest such integer).

**high**: int, optional

If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

**size**: int or tuple of ints, optional

Output shape. Default is None, in which case a single int is returned.

**Returns**

- **out**: int or ndarray of ints
  
  size-shaped array of random integers from the appropriate distribution, or a single such random int if size not provided.

**See Also:**

`random.random_integers`

Similar to `randint`, only for the closed interval [low, high], and 1 is the lowest value if high is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

**Examples**

```python
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])

>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```python
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

**RandomState**.random

Return a sample (or samples) from the “standard normal” distribution.

If positive, int_like or int-convertible arguments are provided, random generates an array of shape (d0, d1, ..., dn), filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the di are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use numpy.random.standard_normal instead.

**Parameters**

- **d0, d1, ..., dn**: int, optional
  
  The dimensions of the returned array, should be all positive. If no argument is given a single Python float is returned.

**Returns**

- **Z**: ndarray or float
  
  A (d0, d1, ..., dn)-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

**See Also:**
random.standard_normal

Similar, but takes a tuple as its argument.

Notes

For random samples from \( N(\mu, \sigma^2) \), use:

\[
\text{sigma} * \text{np.random.randn(...) + mu}
\]

Examples

```python
>>> np.random.randn()
2.1923875335537315 #random
```

Two-by-four array of samples from \( N(3, 6.25) \):

```python
>>> 2.5 * np.random.randn(2, 4) + 3
array([[ 0.39924804,  4.68456316,  4.99394529,  4.84057254], #random
        [-4.49401501,  4.00950034, -1.81814867,  7.29718677]]) #random
```

RandomState.random_integers

Return random integers between \( low \) and \( high \), inclusive.

Return random integers from the “discrete uniform” distribution in the closed interval \( [low, high] \). If \( high \) is None (the default), then results are from \( [1, low] \).

Parameters

- **low**: int
  - Lowest (signed) integer to be drawn from the distribution (unless \( high=None \), in which case this parameter is the highest such integer).

- **high**: int, optional
  - If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if \( high=None \)).

- **size**: int or tuple of ints, optional
  - Output shape. Default is None, in which case a single int is returned.

Returns

- **out**: int or ndarray of ints
  - \( size \)-shaped array of random integers from the appropriate distribution, or a single such random int if \( size \) not provided.

See Also:

random.randint

Similar to random_integers, only for the half-open interval \( [low, high) \), and 0 is the lowest value if \( high \) is omitted.

Notes

To sample from \( N \) evenly spaced floating-point numbers between \( a \) and \( b \), use:

\[
a + (b - a) \times (\text{np.random.random_integers}(N) - 1) / (N - 1.)
\]
Examples

```python
>>> np.random.random_integers(5)
4
>>> type(np.random.random_integers(5))
<type 'int'>
>>> np.random.random_integers(5, size=(3,2.))
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (i.e., from the set 0, 5/8, 10/8, 15/8, 20/8):

```python
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625, 1.25 , 0.625, 0.625, 2.5 ])
```

Roll two six sided dice 1000 times and sum the results:

```python
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```python
>>> import matplotlib.pyplot as plt

>>> count, bins, ignored = plt.hist(dsums, 11, normed=True)
>>> plt.show()
```

RandomState.random_sample(size=None)
Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample \( \text{Unif}[a, b) \), \( b > a \) multiply the output of random_sample by \((b-a)\) and add \( a \):

\[
(b - a) \times \text{random_sample()} + a
\]

Parameters

size : int or tuple of ints, optional
Defines the shape of the returned array of random floats. If None (the default), returns a single float.

Returns

out : float or ndarray of floats

Array of random floats of shape size (unless size=None, in which case a single float is returned).

Examples

```python
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```python
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

RandomState . rayleigh ( scale=1.0, size=None)

Draw samples from a Rayleigh distribution.

The Χ and Weibull distributions are generalizations of the Rayleigh.

Parameters

scale : scalar

Scale, also equals the mode. Should be >= 0.

size : int or tuple of ints, optional

Shape of the output. Default is None, in which case a single value is returned.

Notes

The probability density function for the Rayleigh distribution is

\[ P(x; \text{scale}) = \frac{x}{\text{scale}^2} e^{-\frac{x^2}{2 \text{scale}^2}} \]

The Rayleigh distribution arises if the wind speed and wind direction are both gaussian variables, then the vector wind velocity forms a Rayleigh distribution. The Rayleigh distribution is used to model the expected output from wind turbines.

References

[R165], [R166]

Examples

Draw values from the distribution and plot the histogram

```python
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, normed=True)
```
Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```python
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```python
>>> 100.*sum(s>3)/1000000.
0.08730000000000003
```

**RandomState**.seed(seed=None)

Seed the generator.

This method is called when RandomState is initialized. It can be called again to re-seed the generator. For details, see RandomState.

**Parameters**

- **seed**: int or array_like, optional
  Seed for RandomState.

**See Also:**

RandomState

**RandomState**.set_state(state)

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the “Mersenne Twister”[R167] pseudo-random number generating algorithm.

**Parameters**

- **state**: tuple(str, ndarray of 624 uints, int, int, float)
  The state tuple has the following items:
  1. the string ‘MT19937’, specifying the Mersenne Twister algorithm.
  2. a 1-D array of 624 unsigned integers keys.
  3. an integer pos.
  4. an integer has_gauss.
  5. a float cached_gaussian.

**Returns**

- **out**: None
  Returns ‘None’ on success.

**See Also:**

get_state

**Notes**

set_state and get_state are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: state = (‘MT19937’, keys, pos).
RandomState.shuffle(x)
Modify a sequence in-place by shuffling its contents.

Parameters

x : array_like
The array or list to be shuffled.

Returns

None

Examples

>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8]

This function only shuffles the array along the first index of a multi-dimensional array:

>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5],
       [6, 7, 8],
       [0, 1, 2]])

RandomState.standard_cauchy(size=None)
Standard Cauchy distribution with mode = 0.
Also known as the Lorentz distribution.

Parameters

size : int or tuple of ints
Shape of the output.

Returns

samples : ndarray or scalar
The drawn samples.

Notes

The probability density function for the full Cauchy distribution is

\[ P(x; x_0, \gamma) = \frac{1}{\pi \gamma \left[ 1 + \left( \frac{x - x_0}{\gamma} \right)^2 \right] } \]

and the Standard Cauchy distribution just sets \( x_0 = 0 \) and \( \gamma = 1 \)

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.
References

[R168], [R169], [R170]

Examples

Draw samples and plot the distribution:

```python
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

RandomState.standard_exponential(size=None)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Parameters

- `size` : int or tuple of ints
  Shape of the output.

Returns

- `out` : float or ndarray
  Drawn samples.

Examples

Output a 3x8000 array:

```python
>>> n = np.random.standard_exponential((3, 8000))
```

RandomState.standard_gamma(shape, size=None)

Draw samples from a Standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated “k”) and scale=1.

Parameters

- `shape` : float
  Parameter, should be > 0.
- `size` : int or tuple of ints
  Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn.

Returns

- `samples` : ndarray or scalar
  The drawn samples.

See Also:

- `scipy.stats.distributions.gamma`
  probability density function, distribution or cumulative density function, etc.
Notes

The probability density for the Gamma distribution is

\[ p(x) = x^{k-1} e^{-x/\theta} \frac{\theta^k}{\Gamma(k)}, \]

where \( k \) is the shape and \( \theta \) the scale, and \( \Gamma \) is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

[R171], [R172]

Examples

Draw samples from the distribution:

```python
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps

```python
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale)) / (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

RandomState.standard_normal(size=None)

Returns samples from a Standard Normal distribution (mean=0, stdev=1).

Parameters

- size : int or tuple of ints, optional
  Output shape. Default is None, in which case a single value is returned.
Returns

`out` : float or ndarray

Drawn samples.

Examples

```python
>>> s = np.random.standard_normal(8000)
```

```python
array([ 0.6888893 , 0.78096262, -0.89086505, ..., 0.49876311, #random
       -0.38672696, -0.4685006 ]) #random
```

```python
>>> s.shape
(8000, )
```

```python
>>> s = np.random.standard_normal(size=(3, 4, 2))
```

```python
>>> s.shape
(3, 4, 2)
```

RandomState.standard_t(df, size=None)

Standard Student’s t distribution with df degrees of freedom.

A special case of the hyperbolic distribution. As df gets large, the result resembles that of the standard normal distribution (`standard_normal`).

Parameters

- `df` : int
  
  Degrees of freedom, should be > 0.

- `size` : int or tuple of ints, optional
  
  Output shape. Default is None, in which case a single value is returned.

Returns

- `samples` : ndarray or scalar
  
  Drawn samples.

Notes

The probability density function for the t distribution is

\[
P(x, df) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{\pi df} \Gamma\left(\frac{df}{2}\right)} \left(1 + \frac{x^2}{df}\right)^{-\frac{df+1}{2}}
\]

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was forst published in 1908 by William Gisset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

[R173], [R174]

Examples

From Dalgaard page 83 [R173], suppose the daily energy intake for 11 women in Kj is:
Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999 #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

```
RandomState.tomaxint(size=None)
Random integers between 0 and sys.maxint, inclusive.
```

Return a sample of uniformly distributed random integers in the interval [0, sys.maxint].

**Parameters**

- `size`: tuple of ints, int, optional
  
  Shape of output. If this is, for example, (m,n,k), m*n*k samples are generated. If no shape is specified, a single sample is returned.

**Returns**

- `out`: ndarray

  Drawn samples, with shape `size`.

**See Also:**

- `randint`
  
  Uniform sampling over a given half-open interval of integers.

- `random_integers`
  
  Uniform sampling over a given closed interval of integers.

**Examples**

```
>>> RS = np.random.mtrand.RandomState() # need a RandomState object
>>> RS.tomaxint((2,2,2))
array([[1170048599, 1600360186],
      [ 739731006, 1947757578]],
      [[1871712945, 752307660],
       [1601631370, 1479324245]])
```

```
>>> import sys
>>> sys.maxint
2147483647
```
>> RS.tomaxint((2,2,2)) < sys.maxint
array([[ True,  True],
       [ True,  True],
       [[ True,  True],
       [ True,  True]], dtype=bool)

RandomState.triangular(left, mode, right, size=None)

Draw samples from the triangular distribution.

The triangular distribution is a continuous probability distribution with lower limit limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Parameters
left : scalar
    Lower limit.
mode : scalar
    The value where the peak of the distribution occurs. The value should fulfill the condition left <= mode <= right.
right : scalar
    Upper limit, should be larger than left.
size : int or tuple of ints, optional
    Output shape. Default is None, in which case a single value is returned.

Returns
samples : ndarray or scalar
    The returned samples all lie in the interval [left, right].

Notes
The probability density function for the Triangular distribution is

\[
P(x; l, m, r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(m-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}
\]

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

References
[R175]

Examples
Draw values from the distribution and plot the histogram:

>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...               normed=True)
>>> plt.show()
RandomState.uniform\[low=0.0, high=1.0, size=1\]

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval \([low, high)\) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by \texttt{uniform}.

**Parameters**

- **low**: float, optional
  
  Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

- **high**: float
  
  Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

- **size**: int or tuple of ints, optional
  
  Shape of output. If the given size is, for example, \((m,n,k)\), \(m\times n\times k\) samples are generated. If no shape is specified, a single sample is returned.

**Returns**

- **out**: ndarray
  
  Drawn samples, with shape \texttt{size}.

**See Also**

- \texttt{randint}
  
  Discrete uniform distribution, yielding integers.

- \texttt{random_integers}
  
  Discrete uniform distribution over the closed interval \([low, high]\).

- \texttt{random_sample}
  
  Floats uniformly distributed over \([0, 1)\).

- \texttt{random}
  
  Alias for \texttt{random_sample}. 
**rand**

Convenience function that accepts dimensions as input, e.g., `rand(2, 2)` would generate a 2-by-2 array of floats, uniformly distributed over $[0, 1)$.

**Notes**

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval $[a, b)$, and zero elsewhere.

**Examples**

Draw samples from the distribution:

```python
>>> s = np.random.uniform(-1, 0, 1000)
```

All values are within the given interval:

```python
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, normed=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

RandomState.**vonmises** *(mu, kappa, size=None)*

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].
The von Mises distribution (also known as the circular normal distribution) is a continuous probability
distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

**Parameters**
- **mu**: float
  - Mode ("center") of the distribution.
- **kappa**: float
  - Dispersion of the distribution, has to be >=0.
- **size**: int or tuple of int
  - Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are
drawn.

**Returns**
- **samples**: scalar or ndarray
  - The returned samples, which are in the interval [-pi, pi].

**See Also:**
- `scipy.stats.distributions.vonmises`
  - probability density function, distribution, or cumulative density function, etc.

**Notes**
The probability density for the von Mises distribution is

\[ p(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)} , \]

where \( \mu \) is the mode and \( \kappa \) the dispersion, and \( I_0(\kappa) \) is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now
the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in
probability theory, aerodynamics, fluid mechanics, and philosophy of science.

**References**

**Examples**

Draw samples from the distribution:

```python
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> x = np.arange(-np.pi, np.pi, 2*np.pi/50.)
>>> y = -np.exp(kappa*np.cos(x-mu))/(2*np.pi*sps.jn(0,kappa))
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```
RandomState.wald (mean, scale, size=None)

Draw samples from a Wald, or Inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian.

Some references claim that the Wald is an Inverse Gaussian with mean=1, but this is by no means universal.

The Inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Parameters

- mean : scalar
  Distribution mean, should be > 0.

- scale : scalar
  Scale parameter, should be >= 0.

- size : int or tuple of ints, optional
  Output shape. Default is None, in which case a single value is returned.

Returns

- samples : ndarray or scalar
  Drawn sample, all greater than zero.

Notes

The probability density function for the Wald distribution is

\[ P(x; \text{mean}, \text{scale}) = \sqrt{\text{scale}} \cdot \frac{1}{\sqrt{2\pi x^3}} e^{-\frac{\text{scale}(x-\text{mean})^2}{2 \text{mean}^2 x}} \]

As noted above the Inverse Gaussian distribution first arise from attempts to model Brownian Motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.
References

[R176], [R177], [R178]

Examples

Draw values from the distribution and plot the histogram:

```python
>>> import matplotlib.pyplot as plt

>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, normed=True)

>>> plt.show()
```

RandomState.weibull (a, size=None)

Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter $a$.

\[ X = (-\ln(U))^{1/a} \]

Here, $U$ is drawn from the uniform distribution over $(0,1]$. The more common 2-parameter Weibull, including a scale parameter $\lambda$ is just $X = \lambda(-\ln(U))^{1/a}$.

Parameters:

- **a**: float
  Shape of the distribution.

- **size**: tuple of ints
  Output shape. If the given shape is, e.g., $(m, n, k)$, then $m \times n \times k$ samples are drawn.

See Also:

scipy.stats.distributions.weibull_max, scipy.stats.distributions.weibull_min, scipy.stats.distributions.genextreme, gumbel
Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

\[ p(x) = \frac{a}{\lambda} \left( \frac{x}{\lambda} \right)^{a-1} e^{-\left( \frac{x}{\lambda} \right)^a}, \]

where \( a \) is the shape and \( \lambda \) the scale.

The function has its peak (the mode) at \( \lambda \left( \frac{a-1}{a} \right)^{1/a} \).

When \( a = 1 \), the Weibull distribution reduces to the exponential distribution.

References

[R179], [R180], [R181]

Examples

Draw samples from the distribution:

```python
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
... return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)

>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```
RandomState.zipf(a, size=None)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter \( a > 1 \).

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf’s law: the frequency of an item is inversely proportional to its rank in a frequency table.

Parameters

- **a**: float > 1
  - Distribution parameter.
- **size**: int or tuple of int, optional
  - Output shape. If the given shape is, e.g., \((m, n, k)\), then \(m \times n \times k\) samples are drawn; a single integer is equivalent in its result to providing a mono-tuple, i.e., a 1-D array of length size is returned. The default is None, in which case a single scalar is returned.

Returns

- **samples**: scalar or ndarray
  - The returned samples are greater than or equal to one.

See Also:

- scipy.stats.distributions.zipf
  - probability density function, distribution, or cumulative density function, etc.

Notes

The probability density for the Zipf distribution is

\[
p(x) = \frac{x^{-a}}{\zeta(a)},
\]

where \( \zeta \) is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

References


Examples

Draw samples from the distribution:

```python
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```python
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
Truncate s values at 50 so plot is interesting
>>> count, bins, ignored = plt.hist(s[s<50], 50, normed=True)
>>> x = np.arange(1., 50.)
```
```python
>>> y = x**(-a)/sps.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```

**numpy.random.seed** *(seed=None)*

Seed the generator.

This method is called when `RandomState` is initialized. It can be called again to re-seed the generator. For details, see `RandomState`.

**Parameters**

- `seed`: int or array_like, optional
  
  Seed for `RandomState`.

**See Also:**

- `RandomState`

**numpy.random.get_state()**

Return a tuple representing the internal state of the generator.

For more details, see `set_state`.

**Returns**

- `out`: tuple(str, ndarray of 624 uints, int, int, float)

  The returned tuple has the following items:
  1. the string ‘MT19937’.
  2. a 1-D array of 624 unsigned integer keys.
  3. an integer pos.
  4. an integer has_gauss.
  5. a float cached_gaussian.

**See Also:**

- `set_state`
Notes

`set_state` and `get_state` are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

```
numpy.random.set_state(state)
```

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the “Mersenne Twister”[R226] pseudo-random number generating algorithm.

**Parameters**

- `state`: tuple(str, ndarray of 624 uints, int, int, float)
  
  The `state` tuple has the following items:

  1. the string ‘MT19937’, specifying the Mersenne Twister algorithm.
  2. a 1-D array of 624 unsigned integers `keys`.
  3. an integer `pos`.
  4. an integer `has_gauss`.
  5. a float `cached_gaussian`.

**Returns**

- `out`: None
  
  Returns ‘None’ on success.

See Also:

get_state

Notes

`set_state` and `get_state` are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

References

[R226]

### 3.28 Set routines

#### 3.28.1 Making proper sets

**unique(ar[, return_index, return_inverse])** Find the unique elements of an array.

```
numpy.unique(ar, return_index=False, return_inverse=False)
```

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.
Parameters

**ar** : array_like
    Input array. This will be flattened if it is not already 1-D.

**return_index** : bool, optional
    If True, also return the indices of ar that result in the unique array.

**return_inverse** : bool, optional
    If True, also return the indices of the unique array that can be used to reconstruct ar.

Returns

**unique** : ndarray
    The sorted unique values.

**unique_indices** : ndarray, optional
    The indices of the first occurrences of the unique values in the (flattened) original array. Only provided if return_index is True.

**unique_inverse** : ndarray, optional
    The indices to reconstruct the (flattened) original array from the unique array. Only provided if return_inverse is True.

See Also:

**numpy.lib.arraysetops**
Module with a number of other functions for performing set operations on arrays.

Examples

```python
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the indices of the original array that give the unique values:

```python
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'],
     dtype='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
     dtype='|S1')
```

Reconstruct the input array from the unique values:

```python
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
```
3.28.2 Boolean operations

numpy.in1d(ar1, ar2[, assume_unique, invert])  Test whether each element of a 1-D array is also present in a second array.

Parameters

ar1 : (M,) array_like
    Input array.
ar2 : array_like
    The values against which to test each value of ar1.
assume_unique : bool, optional
    If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.
invert : bool, optional
    If True, the values in the returned array are inverted (that is, False where an element of ar1 is in ar2 and True otherwise). Default is False. np.in1d(a, b, invert=True) is equivalent to (but is faster than) np.invert(in1d(a, b)). New in version 1.8.0.

Returns

in1d : (M,) ndarray, bool
    The values ar1[in1d] are in ar2.

See Also:
numpy.lib.arraysetops
    Module with a number of other functions for performing set operations on arrays.

Notes

in1d can be considered as an element-wise function version of the python keyword in, for 1-D sequences. in1d(a, b) is roughly equivalent to np.array([item in b for item in a]). New in version 1.4.0.

Examples

>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
```python
>>> mask
array([ True, False,  True, False,  True], dtype=bool)
>>> test[mask]
array([0, 2, 0])
>>> mask = np.in1d(test, states, invert=True)
>>> mask
array([False,  True, False,  True, False], dtype=bool)
>>> test[mask]
array([1, 5])
```

```python
numpy.intersect1d(ar1, ar2, assume_unique=False)
```
Find the intersection of two arrays.

Return the sorted, unique values that are in both of the input arrays.

**Parameters**

- `ar1, ar2`: array_like
  - Input arrays.

- `assume_unique`: bool
  - If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

- `intersect1d`: ndarray
  - Sorted 1D array of common and unique elements.

**See Also:**

`numpy.lib.arraysetops`
Module with a number of other functions for performing set operations on arrays.

**Examples**

```python
>>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
array([1, 3])
```

```python
numpy.setdiff1d(ar1, ar2, assume_unique=False)
```
Find the set difference of two arrays.

Return the sorted, unique values in `ar1` that are not in `ar2`.

**Parameters**

- `ar1`: array_like
  - Input array.

- `ar2`: array_like
  - Input comparison array.

- `assume_unique`: bool
  - If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

- `setdiff1d`: ndarray
  - Sorted 1D array of values in `ar1` that are not in `ar2`.
See Also:

`numpy.lib.arraysetops`
Module with a number of other functions for performing set operations on arrays.

**Examples**
```python
g = np.array([1, 2, 3, 2, 4, 1])
b = np.array([3, 4, 5, 6])
np.setdiff1d(a, b)
array([1, 2])
```

`numpy.setxor1d(ar1, ar2, assume_unique=False)`
Find the set exclusive-or of two arrays.

Return the sorted, unique values that are in only one (not both) of the input arrays.

**Parameters**
- `ar1, ar2` : array_like
  Input arrays.
- `assume_unique` : bool
  If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**
- `setxor1d` : ndarray
  Sorted 1D array of unique values that are in only one of the input arrays.

**Examples**
```python
g = np.array([1, 2, 3, 2, 4])
b = np.array([3, 4, 5, 7, 5])
np.setxor1d(a, b)
array([1, 4, 5, 7])
```

`numpy.union1d(ar1, ar2)`
Find the union of two arrays.

Return the unique, sorted array of values that are in either of the two input arrays.

**Parameters**
- `ar1, ar2` : array_like
  Input arrays. They are flattened if they are not already 1D.

**Returns**
- `union1d` : ndarray
  Unique, sorted union of the input arrays.

**See Also:**

`numpy.lib.arraysetops`
Module with a number of other functions for performing set operations on arrays.
Examples

```python
>>> np.union1d([-1, 0, 1], [-2, 0, 2])
array([-2, -1,  0,  1,  2])
```

### 3.29 Sorting, searching, and counting

#### 3.29.1 Sorting

```python
numpy.sort(a, axis=-1, kind='quicksort', order=None)
```

Return a sorted copy of an array.

**Parameters**

- `a`: array_like
  - Array to be sorted.
- `axis`: int or None, optional
  - Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.
- `kind`: {'quicksort', 'mergesort', 'heapsort'}, optional
  - Sorting algorithm. Default is 'quicksort'.
- `order`: list, optional
  - When `a` is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

**Returns**

- `sorted_array`: ndarray
  - Array of the same type and shape as `a`.

**See Also:**

- `ndarray.sort`
  - Method to sort an array in-place.
- `argsort`
  - Indirect sort.
- `lexsort`
  - Indirect stable sort on multiple keys.
searchsorted
Find elements in a sorted array.

partition
Partial sort.

Notes
The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'quicksort'</td>
<td>1</td>
<td>O(n^2)</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>'mergesort'</td>
<td>2</td>
<td>O(n*log(n))</td>
<td>~n/2</td>
<td>yes</td>
</tr>
<tr>
<td>'heapsort'</td>
<td>3</td>
<td>O(n*log(n))</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan values led to undefined behaviour. In numpy versions >= 1.4.0 nan values are sorted to the end. The extended sort order is:

- Real: [R, nan]
- Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]

where R is a non-nan real value. Complex values with the same nan placements are sorted according to the non-nan part if it exists. Non-nan values are sorted as before.

Examples

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> np.sort(a)                      # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)           # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)              # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:

```python
>>> dtype = [('', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ...
    ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)    # create a structured array
>>> np.sort(a, order='height')           # sort by height
array([[Galahad', 1.7, 38],
       ['Arthur', 1.8, 41],
       ['Lancelot', 1.8999999999999999, 38]],
       dtye=[[name', |S10'), ('height', '<f8'), ('age', '<i4')]]
```

Sort by age, then height if ages are equal:

```python
>>> np.sort(a, order=['age', 'height'])
array([[Galahad', 1.7, 38],
       ['Lancelot', 1.8999999999999999, 38],
```
numpy.**lexsort** *(keys, axis=-1)*

Perform an indirect sort using a sequence of keys.

Given multiple sorting keys, which can be interpreted as columns in a spreadsheet, lexsort returns an array of integer indices that describes the sort order by multiple columns. The last key in the sequence is used for the primary sort order, the second-to-last key for the secondary sort order, and so on. The keys argument must be a sequence of objects that can be converted to arrays of the same shape. If a 2D array is provided for the keys argument, its rows are interpreted as the sorting keys and sorting is according to the last row, second last row etc.

**Parameters**
- **keys**: *(k, N) array or tuple containing k (N,)-shaped sequences*
  The k different “columns” to be sorted. The last column (or row if keys is a 2D array) is the primary sort key.
- **axis**: int, optional
  Axis to be indirectly sorted. By default, sort over the last axis.

**Returns**
- **indices**: *(N,) ndarray of ints*
  Array of indices that sort the keys along the specified axis.

**See Also**:
- **argsort**
  Indirect sort.
- **ndarray.sort**
  In-place sort.
- **sort**
  Return a sorted copy of an array.

**Examples**

Sort names: first by surname, then by name.

```python
>>> surnames = ('Hertz', 'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))
>>> ind
array([1, 2, 0])
```

```python
>>> [surnames[i] + ', ' + first_names[i] for i in ind]
['Galilei, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
```

Sort two columns of numbers:

```python
>>> a = [1,5,1,4,3,4,4]  # First column
>>> b = [9,4,0,4,0,2,1]  # Second column
>>> ind = np.lexsort((b,a))  # Sort by a, then by b
>>> print ind
[2 0 4 6 5 3 1]
```
>>> [(a[i],b[i]) for i in ind]
[(1, 0), (1, 9), (3, 0), (4, 1), (4, 2), (4, 4), (5, 4)]

Note that sorting is first according to the elements of a. Secondary sorting is according to the elements of b.

A normal argsort would have yielded:

>>> [(a[i],b[i]) for i in np.argsort(a)]
[(1, 9), (1, 0), (3, 0), (4, 4), (4, 2), (4, 1), (5, 4)]

Structured arrays are sorted lexically by argsort:

>>> x = np.array([(1,9), (5,4), (1,0), (4,4), (3,0), (4,2), (4,1)],
... dtype=np.dtype([('x', int), ('y', int)]))

>>> np.argsort(x) # or np.argsort(x, order=('x', 'y'))
array([2, 0, 4, 6, 5, 3, 1])

def numpy.argort(a, axis=-1, kind='quicksort', order=None)

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in sorted order.

Parameters

- a : array_like
  Array to sort.

- axis : int or None, optional
  Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

- kind : {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm.

- order : list, optional
  When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

Returns

- index_array : ndarray, int
  Array of indices that sort a along the specified axis. In other words, a[index_array] yields a sorted a.

See Also:

- sort
  Describes sorting algorithms used.

- lexsort
  Indirect stable sort with multiple keys.

- ndarray.sort
  Inplace sort.

- argpartition
  Indirect partial sort.
Notes

See `sort` for notes on the different sorting algorithms.

As of NumPy 1.4.0 `argsort` works with real/complex arrays containing `nan` values. The enhanced sort order is documented in `sort`.

Examples

One dimensional array:

```python
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])
```

Two-dimensional array:

```python
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])

>>> np.argsort(x, axis=0)
array([[0, 1],
       [1, 0]])

>>> np.argsort(x, axis=1)
array([[0, 1],
       [0, 1]])
```

Sorting with keys:

```python
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])

>>> np.argsort(x, order=('x','y'))
array([0, 1])

>>> np.argsort(x, order=('y','x'))
array([1, 0])
```

ndarray.sort (axis=-1, kind='quicksort', order=None)

Sort an array, in-place.

Parameters

- **axis** : int, optional
  Axis along which to sort. Default is -1, which means sort along the last axis.

- **kind** : {'quicksort', 'mergesort', 'heapsort'}, optional
  Sorting algorithm. Default is 'quicksort'.

- **order** : list, optional
  When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also:
**numpy.sort**
Return a sorted copy of an array.

**argsort**
Indirect sort.

**lexsort**
Indirect stable sort on multiple keys.

**searchsorted**
Find elements in sorted array.

**partition**
Partial sort.

**Notes**
See sort for notes on the different sorting algorithms.

**Examples**

```python
>>> a = np.array([[1, 4], [3, 1]])
>>> a.sort(axis=1)
```

```python
array([[1, 4],
       [1, 3]])
```

```python
>>> a.sort(axis=0)
```

```python
array([[1, 3],
       [1, 4]])
```

Use the order keyword to specify a field to use when sorting a structured array:

```python
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
```

```python
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

**numpy.msort (a)**
Return a copy of an array sorted along the first axis.

**Parameters**

- **a**: array_like
  Array to be sorted.

**Returns**

- **sorted_array**: ndarray
  Array of the same type and shape as `a`.

**See Also**

- **sort**

**Notes**

`np.msort (a)` is equivalent to `np.sort (a, axis=0)`.

**numpy.sort_complex (a)**
Sort a complex array using the real part first, then the imaginary part.
Parameters
   a : array_like
      Input array

Returns
   out : complex ndarray
      Always returns a sorted complex array.

Examples
>>> np.sort_complex([5, 3, 6, 2, 1])
array([ 1.+0.j, 2.+0.j, 3.+0.j, 5.+0.j, 6.+0.j])

>>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
array([ 1.+2.j, 2.-1.j, 3.-3.j, 3.-2.j, 3.+5.j])

numpy.partition(a, kth=-1, kind='introselect', order=None)
    Return a partitioned copy of an array.

    Creates a copy of the array with its elements rearranged in such a way that the value of the element in kth position
    is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this
    element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is
    undefined. New in version 1.8.0.

Parameters
   a : array_like
      Array to be sorted.

   kth : int or sequence of ints
      Element index to partition by. The kth value of the element will be in its final sorted
      position and all smaller elements will be moved before it and all equal or greater ele-
      ments behind it. The order all elements in the partitions is undefined. If provided with
      a sequence of kth it will partition all elements indexed by kth of them into their sorted
      position at once.

   axis : int or None, optional
      Axis along which to sort. If None, the array is flattened before sorting. The default is
      -1, which sorts along the last axis.

   kind : {'introselect'}, optional
      Selection algorithm. Default is ‘introselect’.

   order : list, optional
      When a is a structured array, this argument specifies which fields to compare first, sec-
      ond, and so on. This list does not need to include all of the fields.

Returns
   partitioned_array : ndarray
      Array of the same type and shape as a.

See Also:

ndarray.partition
   Method to sort an array in-place.
argpartition
Indirect partition.

sort
Full sorting

Notes
The various selection algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

<table>
<thead>
<tr>
<th>kind</th>
<th>speed</th>
<th>worst case</th>
<th>work space</th>
<th>stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>'introselect'</td>
<td>1</td>
<td>O(n)</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

All the partition algorithms make temporary copies of the data when partitioning along any but the last axis. Consequently, partitioning along the last axis is faster and uses less space than partitioning along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Examples
>>> a = np.array([3, 4, 2, 1])
>>> np.partition(a, 3)
array([2, 1, 3, 4])

>>> np.partition(a, (1, 3))
array([1, 2, 3, 4])

numpy.argpartition (a, kth, axis=-1, kind='introselect', order=None)
Perform an indirect partition along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in partitioned order. New in version 1.8.0.

Parameters
a : array_like
Array to sort.

kth : int or sequence of ints
Element index to partition by. The kth element will be in its final sorted position and all smaller elements will be moved before it and all larger elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of kth it will partition all of them into their sorted position at once.

axis : int or None, optional
Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

kind : {'introselect'}, optional
Selection algorithm. Default is ‘introselect’

order : list, optional
When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.
Returns

index_array : ndarray, int

Array of indices that partition a along the specified axis. In other words, 
a[index_array] yields a sorted a.

See Also:

partition
Describes partition algorithms used.

ndarray.partition
Inplace partition.

argsort
Full indirect sort

Notes

See partition for notes on the different selection algorithms.

Examples

One dimensional array:

>>> x = np.array([3, 4, 2, 1])
>>> x[np.argpartition(x, 3)]
array([2, 1, 3, 4])
>>> x[np.argpartition(x, (1, 3))]
array([1, 2, 3, 4])

3.29.2 Searching

argmax(a[, axis]) Indices of the maximum values along an axis.
nanargmax(a[, axis]) Return the indices of the maximum values in the specified axis ignoring
argmin(a[, axis]) Return the indices of the minimum values along an axis.
nanargmin(a[, axis]) Return the indices of the minimum values in the specified axis ignoring
argwhere(a) Find the indices of array elements that are non-zero, grouped by element.
nonzero(a) Return the indices of the elements that are non-zero.
flatnonzero(a) Return indices that are non-zero in the flattened version of a.
where(condition, [x, y]) Return elements, either from x or y, depending on condition.
searchsorted(a, v[, side, sorter]) Find indices where elements should be inserted to maintain order.
extract(condition, arr) Return the elements of an array that satisfy some condition.

numpy.argmax(a, axis=None)
Indices of the maximum values along an axis.

Parameters

a : array_like
Input array.

axis : int, optional
By default, the index is into the flattened array, otherwise along the specified axis.

Returns
index_array : ndarray of ints

Array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.

See Also:

ndarray.argmax, argmin

amax

The maximum value along a given axis.

unravel_index

Convert a flat index into an index tuple.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```python
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])

>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b)  # Only the first occurrence is returned.
1
```

numpy.nanargmax(a, axis=None)

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slices ValueError is raised. Warning: the results cannot be trusted if a slice contains only NaNs and -Infs.

Parameters

a : array_like

Input data.

axis : int, optional

Axis along which to operate. By default flattened input is used.

Returns

index_array : ndarray

An array of indices or a single index value.

See Also:

argmax, nanargmin
Examples

```python
code>
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmax(a)
0
>>> np.nanargmax(a)
1
>>> np.nanargmax(a, axis=0)
array([1, 0])
>>> np.nanargmax(a, axis=1)
array([1, 1])
```

`numpy.argmin(a, axis=None)`
Return the indices of the minimum values along an axis.

**See Also:**

`argmax`
Similar function. Please refer to `numpy.argmax` for detailed documentation.

`numpy.nanargmin(a, axis=None)`
Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slices `ValueError` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and Infs.

**Parameters**

- `a`: array_like
  Input data.

- `axis`: int, optional
  Axis along which to operate. By default flattened input is used.

**Returns**

- `index_array`: ndarray
  An array of indices or a single index value.

**See Also:**

`argmin`, `nanargmax`

**Examples**

```python
code>
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmin(a)
0
>>> np.nanargmin(a)
2
>>> np.nanargmin(a, axis=0)
array([1, 1])
>>> np.nanargmin(a, axis=1)
array([1, 0])
```

`numpy.argwhere(a)`
Find the indices of array elements that are non-zero, grouped by element.

**Parameters**

- `a`: array_like
  Input data.
Returns

index_array : ndarray

Indices of elements that are non-zero. Indices are grouped by element.

See Also:

where, nonzero

Notes

np.argwhere(a) is the same as np.transpose(np.nonzero(a)).

The output of argwhere is not suitable for indexing arrays. For this purpose use where(a) instead.

Examples

>>> x = np.arange(6).reshape(2,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argwhere(x>1)
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
	numpy.nonzero(a)
Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of a, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

a[nonzero(a)]

To group the indices by element, rather than dimension, use:

transpose(nonzero(a))

The result of this is always a 2-D array, with a row for each non-zero element.

Parameters

a : array_like
    Input array.

Returns

tuple_of_arrays : tuple
    Indices of elements that are non-zero.

See Also:

flatnonzero
    Return indices that are non-zero in the flattened version of the input array.

ndarray.nonzero
    Equivalent ndarry method.

count_nonzero
    Counts the number of non-zero elements in the input array.
Examples

```python
>>> x = np.eye(3)
```

```python
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
```

```python
>>> np.nonzero(x)
(array([0, 1, 2]), array([0, 1, 2]))
```

```python
>>> x[np.nonzero(x)]
array([ 1., 1., 1.])
```

```python
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```python
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

```python
>>> a > 3
array([[False, False, False],
       [ True, True, True],
       [ True, True, True]], dtype=bool)
```

```python
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the boolean array can also be called.

```python
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`numpy.flatnonzero(a)`

Return indices that are non-zero in the flattened version of `a`.

This is equivalent to `a.ravel().nonzero()[0]`.

**Parameters**

`a` : ndarray

Input array.

**Returns**

`res` : ndarray

Output array, containing the indices of the elements of `a.ravel()` that are non-zero.

**See Also:**

`nonzero`

Return the indices of the non-zero elements of the input array.

`ravel`

Return a 1-D array containing the elements of the input array.

**Examples**

```python
>>> x = np.arange(-2, 3)
```
array([-2, -1, 0, 1, 2])
>>> np.flatnonzero(x)
array([0, 1, 3, 4])

Use the indices of the non-zero elements as an index array to extract these elements:

```python
>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1, 1, 2])
```

**numpy.where(condition[, x, y])**

Return elements, either from x or y, depending on condition.

If only condition is given, return condition.nonzero().

**Parameters**
- **condition**: array_like, bool
  - When True, yield x, otherwise yield y.
- **x, y**: array_like, optional
  - Values from which to choose. x and y need to have the same shape as condition.

**Returns**
- **out**: ndarray or tuple of ndarrays
  - If both x and y are specified, the output array contains elements of x where condition is True, and elements from y elsewhere.
  - If only condition is given, return the tuple condition.nonzero(), the indices where condition is True.

**See Also:**
- nonzero, choose

**Notes**

If x and y are given and input arrays are 1-D, where is equivalent to:

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

**Examples**

```python
>>> np.where([[True, False], [True, True]],
            [[1, 2], [3, 4]],
            [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

```python
>>> np.where([[0, 1], [1, 0]])
(array([0, 1]), array([1, 0]))
```

```python
>>> x = np.arange(9.).reshape(3, 3)
>>> np.where(x > 5)
(array([2, 2, 2]), array([0, 1, 2]))
```

```python
>>> x[np.where(x > 3.0)]
# Note: result is 1D.
array([4., 5., 6., 7., 8.])
```

```python
>>> np.where(x < 5, x, -1)
# Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```
Find the indices of elements of $x$ that are in $\text{goodvalues}$.

```python
>>> goodvalues = [3, 4, 7]
>>> ix = np.in1d(x.ravel(), goodvalues).reshape(x.shape)
>>> ix
array([[False, False, False],
       [ True,  True, False],
       [False,  True, False]], dtype=bool)
```

```python
>>> np.where(ix)
(array([1, 1, 2]), array([0, 1, 1]))
```

`numpy.searchsorted(a, v, side='left', sorter=None)`

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array $a$ such that, if the corresponding elements in $v$ were inserted before the indices, the order of $a$ would be preserved.

**Parameters**

- $a$: 1-D array_like
  - Input array. If $sorter$ is None, then it must be sorted in ascending order, otherwise $sorter$ must be an array of indices that sort it.

- $v$: array_like
  - Values to insert into $a$.

- `side`: {'left', 'right'}, optional
  - If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or $N$ (where $N$ is the length of $a$).

- `sorter`: 1-D array_like, optional
  - New in version 1.7.0. Optional array of integer indices that sort array $a$ into ascending order. They are typically the result of argsort.

**Returns**

- `indices`: array of ints
  - Array of insertion points with the same shape as $v$.

**See Also:**

- `sort`
  - Return a sorted copy of an array.

- `histogram`
  - Produce histogram from 1-D data.

**Notes**

Binary search is used to find the required insertion points.

As of Numpy 1.4.0 `searchsorted` works with real/complex arrays containing `nan` values. The enhanced sort order is documented in `sort`.

**Examples**

```python
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
```
>>> np.searchsorted([1, 2, 3, 4, 5], [-10, 10, 2, 3])
array([5, 1, 2])

numpy.extract(condition, arr)

Return the elements of an array that satisfy some condition.

This is equivalent to np.compress(ravel(condition), ravel(arr)). If condition is boolean
np.extract is equivalent to arr[condition].

Parameters

condition : array_like

An array whose nonzero or True entries indicate the elements of arr to extract.

arr : array_like

Input array of the same size as condition.

Returns

extract : ndarray

Rank 1 array of values from arr where condition is True.

See Also:
take, put, copyto, compress

Examples

>>> arr = np.arange(12).reshape((3, 4))
>>> arr
array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])

>>> condition = np.mod(arr, 3)==0
>>> condition
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]], dtype=bool)

>>> np.extract(condition, arr)
array([0, 3, 6, 9])

If condition is boolean:

>>> arr[condition]
array([0, 3, 6, 9])

3.29.3 Counting

count_nonzero(a) Counts the number of non-zero values in the array a.

count_reduce_items

numpy.count_nonzero(a)

Counts the number of non-zero values in the array a.

Parameters

a : array_like
The array for which to count non-zeros.

**Returns**

- **count**: int or array of int

  Number of non-zero values in the array.

**See Also:**

- **nonzero**

  Return the coordinates of all the non-zero values.

**Examples**

```python
>>> np.count_nonzero(np.eye(4))
4
>>> np.count_nonzero([[0, 1, 7, 0, 0], [3, 0, 0, 2, 19]])
5
```

## 3.30 Statistics

### 3.30.1 Order statistics

- **amin** *(a, axis=None, out=None, keepdims=False)*

  Return the minimum of an array or minimum along an axis.

  **Parameters**

  - **a**: array_like
    
    Input data.
  
  - **axis**: int, optional
    
    Axis along which to operate. By default, flattened input is used.
  
  - **out**: ndarray, optional
    
    Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See `doc.ufuncs` (Section “Output arguments”) for more details.
  
  - **keepdims**: bool, optional
    
    If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.

  **Returns**

  - **amin**: ndarray or scalar
Minimum of \( a \). If \( \text{axis} \) is None, the result is a scalar value. If \( \text{axis} \) is given, the result is an array of dimension \( a.ndim - 1 \).

See Also:

- **amax**
  The maximum value of an array along a given axis, propagating any NaNs.
- **nanmin**
  The minimum value of an array along a given axis, ignoring any NaNs.
- **minimum**
  Element-wise minimum of two arrays, propagating any NaNs.
- **fmin**
  Element-wise minimum of two arrays, ignoring any NaNs.
- **argmin**
  Return the indices of the minimum values.
- **nanmax, maximum, fmax**

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don’t use `amin` for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `minimum(a[0], a[1])` is faster than `amin(a, axis=0)`.

Examples

```python
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)  # Minimum of the flattened array
0
>>> np.amin(a, axis=0)  # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)  # Minima along the second axis
array([0, 2])

>>> b = np.arange(5, dtype=np.float)
>>> np.amin(b)
nan
>>> np.nanmin(b)
0.0
```

`numpy.amax(a, axis=None, out=None, keepdims=False)`

Return the maximum of an array or maximum along an axis.

Parameters

- **a** : array_like
  Input data.

- **axis** : int, optional
  Axis along which to operate. By default, flattened input is used.
out : ndarray, optional
    Alternative output array in which to place the result. Must be of the same shape and
    buffer length as the expected output. See doc.ufuncs (Section “Output arguments”) for
    more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left in the result as dimensions with
    size one. With this option, the result will broadcast correctly against the original arr.

Returns

amax : ndarray or scalar
    Maximum of a. If axis is None, the result is a scalar value. If axis is given, the result is an array of dimension a.ndim - 1.

See Also:

amin
    The minimum value of an array along a given axis, propagating any NaNs.

nanmax
    The maximum value of an array along a given axis, ignoring any NaNs.

maximum
    Element-wise maximum of two arrays, propagating any NaNs.

fmax
    Element-wise maximum of two arrays, ignoring any NaNs.

argmax
    Return the indices of the maximum values.

nanmin, minimum, fmin

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.

Don’t use amax for element-wise comparison of 2 arrays; when a.shape[0] is 2, maximum(a[0], a[1]) is faster than amax(a, axis=0).

Examples

>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a) # Maximum of the flattened array
3
>>> np.amax(a, axis=0) # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1) # Maxima along the second axis
array([1, 3])

>>> b = np.arange(5, dtype=np.float)
>>> np.amax(b)
nan
>>> np.nanmax(b)
4.0

```
numpy.nanmin(a, axis=None, out=None, keepdims=False)
```

Return minimum of an array or minimum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and Nan is returned for that slice.

**Parameters**

- **a**: array_like
  - Array containing numbers whose minimum is desired. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  - Axis along which the minimum is computed. The default is to compute the minimum of the flattened array.

- **out**: ndarray, optional
  - Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See `doc.ufuncs` for details. New in version 1.8.0.

- **keepdims**: bool, optional
  - If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `a`. New in version 1.8.0.

**Returns**

- **nanmin**: ndarray
  - An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if axis is None, an ndarray scalar is returned. The same dtype as `a` is returned.

**See Also:**

- **nanmax**
  - The maximum value of an array along a given axis, ignoring any NaNs.

- **amin**
  - The minimum value of an array along a given axis, propagating any NaNs.

- **fmin**
  - Element-wise minimum of two arrays, ignoring any NaNs.

- **minimum**
  - Element-wise minimum of two arrays, propagating any NaNs.

- **isnan**
  - Shows which elements are Not a Number (NaN).

- **isfinite**
  - Shows which elements are neither NaN nor infinity.

- **amax, fmax, maximum**

**Notes**

Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.
If the input has a integer type the function is equivalent to np.min.

Examples

```python
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([ 1., 2.])
>>> np.nanmin(a, axis=1)
array([ 1., 3.])
```

When positive infinity and negative infinity are present:

```python
>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf
```

```python
numpy.nanmax(a, axis=None, out=None, keepdims=False)
```

Return the maximum of an array or maximum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and NaN is returned for that slice.

Parameters

- `a`: array_like
  - Array containing numbers whose maximum is desired. If `a` is not an array, a conversion is attempted.

- `axis`: int, optional
  - Axis along which the maximum is computed. The default is to compute the maximum of the flattened array.

- `out`: ndarray, optional
  - Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See `doc.ufuncs` for details. New in version 1.8.0.

- `keepdims`: bool, optional
  - If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `a`. New in version 1.8.0.

Returns

- `nanmax`: ndarray
  - An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if axis is `None`, an ndarray scalar is returned. The same dtype as `a` is returned.

See Also:

- `nanmin`
  - The minimum value of an array along a given axis, ignoring any NaNs.

- `amax`
  - The maximum value of an array along a given axis, propagating any NaNs.

- `fmax`
  - Element-wise maximum of two arrays, ignoring any NaNs.
maximum
Element-wise maximum of two arrays, propagating any NaNs.

isnan
Shows which elements are Not a Number (NaN).

isfinite
Shows which elements are neither NaN nor infinity.

amin, fmin, minimum

Notes
Numpy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.max.

Examples
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([ 3.,  2.])
>>> np.nanmax(a, axis=1)
array([ 2.,  3.])

When positive infinity and negative infinity are present:

>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
infty

numpy.ptp(a, axis=None, out=None)

Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

Parameters

a : array_like
    Input values.

axis : int, optional
    Axis along which to find the peaks. By default, flatten the array.

out : array_like
    Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type of the output values will be cast if necessary.

Returns

ptp : ndarray
    A new array holding the result, unless out was specified, in which case a reference to out is returned.
Examples

```python
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])

>>> np.ptp(x, axis=0)
array([2, 2])

>>> np.ptp(x, axis=1)
array([1, 1])
```

`numpy.percentile(a, q, axis=None, out=None, overwrite_input=False)`

Compute the qth percentile of the data along the specified axis.

Returns the qth percentile of the array elements.

**Parameters**

- `a` : array_like
  Input array or object that can be converted to an array.

- `q` : float in range of [0,100] (or sequence of floats)
  Percentile to compute which must be between 0 and 100 inclusive.

- `axis` : int, optional
  Axis along which the percentiles are computed. The default (None) is to compute the median along a flattened version of the array.

- `out` : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

- `overwrite_input` : bool, optional
  If True, then allow use of memory of input array `a` for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if `overwrite_input` is True and the input is not already an array, an error will be raised.

**Returns**

- `percentile` : scalar or ndarray
  If a single percentile `q` is given and `axis=None` a scalar is returned. If multiple percentiles `q` are given an array holding the result is returned. The results are listed in the first axis. (If `out` is specified, in which case that array is returned instead). If the input contains integers, or floats of smaller precision than 64, then the output data-type is float64. Otherwise, the output data-type is the same as that of the input.

**See Also:**

`mean`, `median`

**Notes**

Given a vector V of length N, the q-th percentile of V is the q-th ranked value in a sorted copy of V. The values and distances of the two nearest neighbors as well as the `interpolation` parameter will determine the percentile.
if the normalized ranking does not match q exactly. This function is the same as the median if $q=50$, the same as the minimum if $q=0$, and the same as the maximum if $q=100$.

Examples

```python
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.percentile(a, 50)
3.5
>>> np.percentile(a, 50, axis=0)
array([ 6.5,  4.5,  2.5])
>>> np.percentile(a, 50, axis=1)
array([ 7.,  2.])

>>> m = np.percentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.percentile(a, 50, axis=0, out=m)
array([ 6.5,  4.5,  2.5])
>>> m
array([ 6.5,  4.5,  2.5])

>>> b = a.copy()
>>> np.percentile(b, 50, axis=1, overwrite_input=True)
array([ 7.,  2.])
>>> assert not np.all(a==b)

>>> b = a.copy()
>>> np.percentile(b, 50, axis=None, overwrite_input=True)
3.5
```

### 3.30.2 Averages and variances

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>median(a, axis, out, overwrite_input)</code></td>
<td>Compute the median along the specified axis.</td>
</tr>
<tr>
<td><code>average(a, axis, weights, returned)</code></td>
<td>Compute the weighted average along the specified axis.</td>
</tr>
<tr>
<td><code>mean(a, axis, dtype, out, keepdims)</code></td>
<td>Compute the arithmetic mean along the specified axis.</td>
</tr>
<tr>
<td><code>std(a, axis, dtype, out, ddof, keepdims)</code></td>
<td>Compute the standard deviation along the specified axis.</td>
</tr>
<tr>
<td><code>var(a, axis, dtype, out, ddof, keepdims)</code></td>
<td>Compute the variance along the specified axis.</td>
</tr>
<tr>
<td><code>nanmean(a, axis, dtype, out, keepdims)</code></td>
<td>Compute the arithmetic mean along the specified axis, ignoring NaNs.</td>
</tr>
<tr>
<td><code>nanstd(a, axis, dtype, out, ddof, keepdims)</code></td>
<td>Compute the standard deviation along the specified axis, while ignoring NaNs.</td>
</tr>
<tr>
<td><code>nanvar(a, axis, dtype, out, ddof, keepdims)</code></td>
<td>Compute the variance along the specified axis, while ignoring NaNs.</td>
</tr>
</tbody>
</table>

**numpy.median**

Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

- **a**: array_like
  
  Input array or object that can be converted to an array.

- **axis**: int, optional
  
  Axis along which the medians are computed. The default (axis=None) is to compute the median along a flattened version of the array.
**out**: ndarray, optional

Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**: bool, optional

If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if *overwrite_input* is True and the input is not already an ndarray, an error will be raised.

**Returns**

**median**: ndarray

A new array holding the result (unless *out* is specified, in which case that array is returned instead). If the input contains integers, or floats of smaller precision than 64, then the output data-type is float64. Otherwise, the output data-type is the same as that of the input.

**See Also:**

table mean, percentile

**Notes**

Given a vector V of length N, the median of V is the middle value of a sorted copy of V, V_sorted - i.e., V_sorted[(N-1)/2], when N is odd. When N is even, it is the average of the two middle values of V_sorted.

**Examples**

```python
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10, 7, 4],
       [3, 2, 1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([6.5, 4.5, 2.5])
>>> np.median(a, axis=1)
array([7., 2.])
>>> m = np.median(a, axis=0)
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=out)
array([6.5, 4.5, 2.5])
>>> m
array([6.5, 4.5, 2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)
```

numpy.average (*a*, *axis=None*, *weights=None*, *returned=False*)

Compute the weighted average along the specified axis.
Parameters

- **a**: array_like
  
  Array containing data to be averaged. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  
  Axis along which to average `a`. If `None`, averaging is done over the flattened array.

- **weights**: array_like, optional
  
  An array of weights associated with the values in `a`. Each value in `a` contributes to the average according to its associated weight. The weights array can either be 1-D (in which case its length must be the size of `a` along the given axis) or of the same shape as `a`. If `weights=None`, then all data in `a` are assumed to have a weight equal to one.

- **returned**: bool, optional
  
  Default is `False`. If `True`, the tuple `(average, sum_of_weights)` is returned, otherwise only the average is returned. If `weights=None`, `sum_of_weights` is equivalent to the number of elements over which the average is taken.

Returns

- **average, [sum_of_weights]** : {array_type, double}
  
  Return the average along the specified axis. When returned is `True`, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is `Float` if `a` is of integer type, otherwise it is of the same type as `a`. `sum_of_weights` is of the same type as `average`.

 Raises

- **ZeroDivisionError**
  
  When all weights along axis are zero. See `numpy.ma.average` for a version robust to this type of error.

- **TypeError**
  
  When the length of 1D `weights` is not the same as the shape of `a` along axis.

See Also:

- `mean`

- `ma.average`
  
  average for masked arrays – useful if your data contains “missing” values

Examples

```python
>>> data = range(1,5)
>>> data
[1, 2, 3, 4]
>>> np.average(data)
2.5
>>> np.average(range(1,11), weights=range(10,0,-1))
4.0

>>> data = np.arange(6).reshape((3,2))
>>> data
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> np.average(data, axis=1, weights=[1./4, 3./4])
```

1264 Chapter 3. Routines
array([ 0.75, 2.75, 4.75])
>>> np.average(data, weights=[1./4, 3./4])
Traceback (most recent call last):
  ...
TypeError: Axis must be specified when shapes of a and weights differ.

numpy.mean (a, axis=None, dtype=None, out=None, keepdims=False)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

Parameters

a : array_like
   Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.

axis : int, optional
   Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype : data-type, optional
   Type to use in computing the mean. For integer inputs, the default is float64; for floating point inputs, it is the same as the input dtype.

out : ndarray, optional
   Alternate output array in which to place the result. The default is None; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See doc.ufuncs for details.

keepdims : bool, optional
   If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

Returns

m : ndarray, see dtype parameter above
   If out=None, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also:

average
   Weighted average

std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-precision accumulator using the dtype keyword can alleviate this issue.
Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

In single precision, `mean` can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.546875
```

Computing the mean in float64 is more accurate:

```python
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

`numpy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- **a**: array_like
  
  Calculate the standard deviation of these values.

- **axis**: int, optional
  
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- **dtype**: dtype, optional
  
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- **out**: ndarray, optional
  
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- **ddof**: int, optional
  
  Means Delta Degrees of Freedom. The divisor used in calculations is \( N - ddof \), where \( N \) represents the number of elements. By default \( ddof \) is zero.

- **keepdims**: bool, optional
  
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \( arr \).

Returns

- **standard_deviation**: ndarray, see dtype parameter above.

  If \( out \) is None, return a new array containing the standard deviation, otherwise return a reference to the output array.
See Also:

```
var, mean, nanmean, nanstd, nanvar
```

numpy.doc.ufuncs

Section “Output arguments”

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean()}^2))} \).

The average squared deviation is normally calculated as \( x.\text{sum()} / N \), where \( N = \text{len}(x) \). If, however, \( \text{ddof} \) is specified, the divisor \( N \) is used instead. In standard statistical practice, \( \text{ddof}=1 \) provides an unbiased estimator of the variance of the infinite population. \( \text{ddof}=0 \) provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with \( \text{ddof}=1 \), it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, \( \text{std} \) takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the \( \text{std} \) is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for \( \text{float32} \) (see example below). Specifying a higher-accuracy accumulator using the \( \text{dtype} \) keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1., 1.])
>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

In single precision, \( \text{std()} \) can be inaccurate:

```python
>>> a = np.zeros((2,512*512), dtype=np.float32)
>>> a[0, :]=1.0
>>> a[1, :]=0.1
>>> np.std(a)
0.45172946707416706
```

Computing the standard deviation in \( \text{float64} \) is more accurate:

```python
>>> np.std(a, dtype=np.float64)
0.44999999925552653
```

```
numpy.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
```

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Parameters

- \( a \) : array_like
  
  Array containing numbers whose variance is desired. If \( a \) is not an array, a conversion is attempted.

- \( \text{axis} \) : int, optional
Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype : data-type, optional
Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

out : ndarray, optional
Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof : int, optional
“Delta Degrees of Freedom”: the divisor used in the calculation is \(N - ddof\), where \(N\) represents the number of elements. By default \(ddof = 0\).

keepdims : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original \(arr\).

Returns

variance : ndarray, see dtype parameter above
If out=None, returns a new array containing the variance; otherwise, a reference to the output array is returned.

See Also:
std, mean, nanmean, nanstd, nanvar

numpy.doc.ufuncs
Section “Output arguments”

Notes
The variance is the average of the squared deviations from the mean, i.e., \(\text{var} = \text{mean}(\text{abs}(x - x.\text{mean()}^2)\).

The mean is normally calculated as \(x.\text{sum()} / N\), where \(N = \text{len}(x)\). If, however, \(ddof\) is specified, the divisor \(N - ddof\) is used instead. In standard statistical practice, \(ddof=1\) provides an unbiased estimator of the variance of a hypothetical infinite population. \(ddof=0\) provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1., 1.])
>>> np.var(a, axis=1)
array([0.25, 0.25])
```
In single precision, var() can be inaccurate:

```python
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20405951142311096
```

Computing the variance in float64 is more accurate:

```python
>>> np.var(a, dtype=np.float64)
0.20249999932997387
```

```python
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.20250000000000001
```

```python
numpy.nanmean(a, axis=None, dtype=None, out=None, keepdims=False)
```
Compute the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

For all-NaN slices, NaN is returned and a RuntimeWarning is raised. New in version 1.8.0.

Parameters

- **a**: array_like
  Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

- **axis**: int, optional
  Axis along which the means are computed. The default is to compute the mean of the flattened array.

- **dtype**: data-type, optional
  Type to use in computing the mean. For integer inputs, the default is float64; for inexact inputs, it is the same as the input dtype.

- **out**: ndarray, optional
  Alternate output array in which to place the result. The default is None; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See doc.ufuncs for details.

- **keepdims**: bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

Returns

- **m**: ndarray, see dtype parameter above
  If out=None, returns a new array containing the mean values, otherwise a reference to the output array is returned. Nan is returned for slices that contain only NaNs.

See Also:

- **average**: Weighted average
- **mean**: Arithmetic mean taken while not ignoring NaNs
**Notes**

The arithmetic mean is the sum of the non-NaN elements along the axis divided by the number of non-NaN elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32. Specifying a higher-precision accumulator using the `dtype` keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanmean(a)
2.6666666666666665
>>> np.nanmean(a, axis=0)
array([ 2., 4.])
>>> np.nanmean(a, axis=1)
array([ 1., 3.5])
```

`numpy.nanstd(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

Compute the standard deviation along the specified axis, while ignoring NaNs.

Returns the standard deviation, a measure of the spread of a distribution, of the non-NaN array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a `RuntimeWarning` is raised. New in version 1.8.0.

**Parameters**

- `a` : array_like
  Calculate the standard deviation of the non-NaN values.

- `axis` : int, optional
  Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

- `dtype` : dtype, optional
  Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

- `out` : ndarray, optional
  Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

- `ddof` : int, optional
  Means Delta Degrees of Freedom. The divisor used in calculations is $N - \text{ddof}$, where $N$ represents the number of non-NaN elements. By default $\text{ddof}$ is zero.

- `keepdims` : bool, optional
  If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original `arr`.

**Returns**

- `standard_deviation` : ndarray, see dtype parameter above.
If `out` is None, return a new array containing the standard deviation, otherwise return a reference to the output array. If `ddof` is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

**See Also:**

`var`, `mean`, `std`, `nanvar`, `nanmean`

numpy.doc.ufuncs

Section “Output arguments”

**Notes**

The standard deviation is the square root of the average of the squared deviations from the mean: \( \text{std} = \sqrt{\text{mean}(\text{abs}(x - x.\text{mean})^2)} \).

The average squared deviation is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, `ddof` is specified, the divisor \( N - \text{ddof} \) is used instead. In standard statistical practice, `ddof` takes the divisor \( N - \text{ddof} \) is instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, `std` takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the `std` is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

**Examples**

```python
>>> a = np.array([[1, np.nan], [3, 4]])
```

```python
>>> np.nanstd(a)
1.247219128924647
```

```python
>>> np.nanstd(a, axis=0)
array([ 1., 0.])
```

```python
>>> np.nanstd(a, axis=1)
array([ 0., 0.5])
```

**numpy.nanvar**

Compute the variance along the specified axis, while ignoring NaNs.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a `RuntimeWarning` is raised. New in version 1.8.0.

**Parameters**

- `a`: array_like

  Array containing numbers whose variance is desired. If `a` is not an array, a conversion is attempted.

- `axis`: int, optional

  Axis along which the variance is computed. The default is to compute the variance of the flattened array.

- `dtype`: data-type, optional
Type to use in computing the variance. For arrays of integer type the default is float32; for arrays of float types it is the same as the array type.

out : ndarray, optional
Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddf : int, optional
“Delta Degrees of Freedom”: the divisor used in the calculation is N - ddf, where N represents the number of non-NaN elements. By default ddf is zero.

keepdims : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original arr.

Returns

variance : ndarray, see dtype parameter above
If out is None, return a new array containing the variance, otherwise return a reference to the output array. If ddf is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

See Also:

std
Standard deviation

mean
Average

var
Variance while not ignoring NaNs

nanstd, nanmean

numpy.doc.ufuncs
Section “Output arguments”

Notes

The variance is the average of the squared deviations from the mean, i.e., \( \text{var} = \text{mean}(\text{abs}(x - x.\text{mean}())^2) \).

The mean is normally calculated as \( x.\text{sum}() / N \), where \( N = \text{len}(x) \). If, however, ddf is specified, the divisor \( N - ddf \) is used instead. In standard statistical practice, ddf=1 provides an unbiased estimator of the variance of a hypothetical infinite population. ddf=0 provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the dtype keyword can alleviate this issue.

Examples
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.var(a)
1.5555555555555554
>>> np.nanvar(a, axis=0)
array([ 1., 0.])
>>> np.nanvar(a, axis=1)
array([ 0., 0.25])

3.30.3 Correlating

- `corrcoef(x[, y, rowvar, bias, ddof])`: Return correlation coefficients.
- `correlate(a, v[, mode, old_behavior])`: Cross-correlation of two 1-dimensional sequences.
- `cov(m[, y, rowvar, bias, ddof])`: Estimate a covariance matrix, given data.

NumPy's `corrcoef(x, y=None, rowvar=1, bias=0, ddof=None)`

Return correlation coefficients.

Please refer to the documentation for `cov` for more detail. The relationship between the correlation coefficient matrix, $P$, and the covariance matrix, $C$, is

$$P_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} \cdot C_{jj}}}$$

The values of $P$ are between -1 and 1, inclusive.

- **Parameters**
  - `x`: array_like
    A 1-D or 2-D array containing multiple variables and observations. Each row of $m$ represents a variable, and each column a single observation of all those variables. Also see `rowvar` below.
  - `y`: array_like, optional
    An additional set of variables and observations. $y$ has the same shape as $m$.
  - `rowvar`: int, optional
    If `rowvar` is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.
  - `bias`: int, optional
    Default normalization is by $(N - 1)$, where $N$ is the number of observations (unbiased estimate). If `bias` is 1, then normalization is by $N$. These values can be overridden by using the keyword `ddof` in NumPy versions $\geq$ 1.5.
  - `ddof`: {None, int}, optional
    New in version 1.5. If not `None` normalization is by $(N - ddof)$, where $N$ is the number of observations; this overrides the value implied by `bias`. The default value is `None`.

- **Returns**
  - `out`: ndarray
The correlation coefficient matrix of the variables.

See Also:

`cov`

Covariance matrix

`numpy.correlate(a, v, mode='valid', old_behavior=False)`

Cross-correlation of two 1-dimensional sequences.

This function computes the correlation as generally defined in signal processing texts:

\[ z[k] = \sum_n a[n] \times \overline{v[n+k]} \]

with a and v sequences being zero-padded where necessary and \( \overline{v} \) being the conjugate.

Parameters

- **a, v**: array_like
  - Input sequences.
- **mode**: {'valid', 'same', 'full'}, optional
  - Refer to the `convolve` docstring. Note that the default is `valid`, unlike `convolve`, which uses `full`.
- **old_behavior**: bool
  - If True, uses the old behavior from Numeric, \((\text{correlate}(a,v) == \text{correlate}(v,a))\), and the conjugate is not taken for complex arrays. If False, uses the conventional signal processing definition.

See Also:

`convolve`  
Discrete, linear convolution of two one-dimensional sequences.

Examples

```python
>>> np.correlate([1, 2, 3], [0, 1, 0.5])
array([ 3.5])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
array([ 2. , 3.5, 3. ])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
array([ 0.5, 2. , 3.5, 3. , 0. ])
```

`numpy.cov(m, y=None, rowvar=1, bias=0, ddof=None)`

Estimate a covariance matrix, given data.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, \( X = [x_1, x_2, ... x_N]^T \), then the covariance matrix element \( C_{ij} \) is the covariance of \( x_i \) and \( x_j \). The element \( C_{ii} \) is the variance of \( x_i \).

Parameters

- **m**: array_like
  - A 1-D or 2-D array containing multiple variables and observations. Each row of \( m \) represents a variable, and each column a single observation of all those variables. Also see rowvar below.
- **y**: array_like, optional
  - An additional set of variables and observations. \( y \) has the same form as that of \( m \).
rowvar : int, optional
If rowvar is non-zero (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

bias : int, optional
Default normalization is by \( (N - 1) \), where \( N \) is the number of observations given (unbiased estimate). If bias is 1, then normalization is by \( N \). These values can be overridden by using the keyword ddof in numpy versions >= 1.5.

ddof : int, optional
New in version 1.5. If not None normalization is by \( (N - ddof) \), where \( N \) is the number of observations; this overrides the value implied by bias. The default value is None.

Returns
out : ndarray
The covariance matrix of the variables.

See Also:
corrcoef
Normalized covariance matrix

Examples
Consider two variables, \( x_0 \) and \( x_1 \), which correlate perfectly, but in opposite directions:

```python
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how \( x_0 \) increases while \( x_1 \) decreases. The covariance matrix shows this clearly:

```python
>>> np.cov(x)
anarray([[ 1., -1.],
          [-1., 1.]])
```

Note that element \( C_{0,1} \), which shows the correlation between \( x_0 \) and \( x_1 \), is negative.

Further, note how \( x \) and \( y \) are combined:

```python
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = np.vstack((x,y))
>>> print np.cov(X)
[[ 11.71 -4.286 ]
 [ -4.286 2.14413333]]
>>> print np.cov(x, y)
[[ 11.71 -4.286 ]
 [ -4.286 2.14413333]]
>>> print np.cov(x)
11.71
```
3.30.4 Histograms
**NumPy Reference, Release 1.8.1**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram(a[, bins, range, normed, weights, ...])</td>
<td>Compute the histogram of a set of data.</td>
</tr>
<tr>
<td>histogram2d(x, y[, bins, range, normed, weights])</td>
<td>Compute the bi-dimensional histogram of two data samples.</td>
</tr>
<tr>
<td>histogramdd(sample[, bins, range, normed, ...])</td>
<td>Compute the multidimensional histogram of some data.</td>
</tr>
<tr>
<td>bincount(x[, weights, minlength])</td>
<td>Count number of occurrences of each value in array of non-negative ints.</td>
</tr>
<tr>
<td>digitize(x, bins[, right])</td>
<td>Return the indices of the bins to which each value in input array belongs.</td>
</tr>
</tbody>
</table>

**numpy.histogram**

(a, bins=10, range=None, normed=False, weights=None, density=None)

Compute the histogram of a set of data.

**Parameters**

- **a**: array_like
  
  Input data. The histogram is computed over the flattened array.

- **bins**: int or sequence of scalars, optional
  
  If bins is an int, it defines the number of equal-width bins in the given range (10, by default). If bins is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

- **range**: (float, float), optional
  
  The lower and upper range of the bins. If not provided, range is simply a.min(), a.max(). Values outside the range are ignored.

- **normed**: bool, optional
  
  This keyword is deprecated in Numpy 1.6 due to confusing/buggy behavior. It will be removed in Numpy 2.0. Use the density keyword instead. If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Note that this latter behavior is known to be buggy with unequal bin widths; use density instead.

- **weights**: array_like, optional
  
  An array of weights, of the same shape as a. Each value in a only contributes its associated weight towards the bin count (instead of 1). If normed is True, the weights are normalized, so that the integral of the density over the range remains 1.

- **density**: bool, optional
  
  If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability mass function. Overrides the normed keyword if given.

**Returns**

- **hist**: array
  
  The values of the histogram. See normed and weights for a description of the possible semantics.

- **bin_edges**: array of dtype float
  
  Return the bin edges (length(hist)+1).

**See Also:**

- histogramdd
- bincount
- searchsorted
- digitize

3.30. Statistics
Notes

All but the last (righthand-most) bin is half-open. In other words, if bins is:

\[ [1, 2, 3, 4] \]

then the first bin is \([1, 2)\) (including 1, but excluding 2) and the second \([2, 3)\). The last bin, however, is \([3, 4]\), which includes 4.

Examples

```python
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
>>> np.histogram(np.arange(4), bins=np.arange(5), density=True)
(array([ 0.25, 0.25, 0.25, 0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
>>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, density=True)
>>> hist
array([ 0.5, 0. , 0.5, 0. , 0. , 0. , 0.5, 0. , 0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist*np.diff(bin_edges))
1.0
```

`numpy.histogram2d(x, y, bins=10, range=None, normed=False, weights=None)`

Compute the bi-dimensional histogram of two data samples.

Parameters

- `x`: array_like, shape (N,)  
  An array containing the x coordinates of the points to be histogrammed.

- `y`: array_like, shape (N,)  
  An array containing the y coordinates of the points to be histogrammed.

- `bins`:  
  [int or [int, int] or array_like or [array, array], optional]  
  The bin specification:
  - If int, the number of bins for the two dimensions (nx=ny=bins).
  - If [int, int], the number of bins in each dimension (nx, ny = bins).
  - If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
  - If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

- `range`: array_like, shape(2,2), optional The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

- `normed`: bool, optional If False, returns the number of samples in each bin. If True, returns the bin density \( \text{bin\_count} / \text{sample\_count} / \text{bin\_area} \).

- `weights`: array_like, shape(N,), optional An array of values \( w_i \) weighing each sample
(\(x_i\), \(y_i\)). Weights are normalized to 1 if \textit{normed} is True. If \textit{normed} is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

**Returns**

\(H\): ndarray, shape(nx, ny)

The bi-dimensional histogram of samples \(x\) and \(y\). Values in \(x\) are histogrammed along the first dimension and values in \(y\) are histogrammed along the second dimension.

\textit{xedges}

[ndarray, shape(nx,)] The bin edges along the first dimension.

\textit{yedges}

[ndarray, shape(ny,)] The bin edges along the second dimension.

**See Also:**

codehistogram

1D histogram
codehistogramdd

Multidimensional histogram

**Notes**

When \textit{normed} is True, then the returned histogram is the sample density, defined such that the sum over bins of the product \(\text{bin\_value} \times \text{bin\_area}\) is 1.

Please note that the histogram does not follow the Cartesian convention where \(x\) values are on the abscissa and \(y\) values on the ordinate axis. Rather, \(x\) is histogrammed along the first dimension of the array (vertical), and \(y\) along the second dimension of the array (horizontal). This ensures compatibility with \textit{histogramdd}.

**Examples**

```python
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt

Construct a 2D-histogram with variable bin width. First define the bin edges:

```python
>>> xedges = [0, 1, 1.5, 3, 5]
>>> yedges = [0, 2, 3, 4, 6]
```

Next we create a histogram \(H\) with random bin content:

```python
>>> x = np.random.normal(3, 1, 100)
>>> y = np.random.normal(1, 1, 100)
>>> H, xedges, yedges = np.histogram2d(y, x, bins=(xedges, yedges))
```

Or we fill the histogram \(H\) with a determined bin content:

```python
>>> H = np.ones((4, 4)).cumsum().reshape(4, 4)
>>> print H[::-1]  # This shows the bin content in the order as plotted
[[13. 14. 15. 16.]
 [ 9. 10. 11. 12.]
 [ 5.  6.  7.  8.]
 [ 1.  2.  3.  4.]]
```
Imshow can only do an equidistant representation of bins:

```python
>>> fig = plt.figure(figsize=(7, 3))
>>> ax = fig.add_subplot(131)
>>> ax.set_title('imshow: equidistant')
>>> im = plt.imshow(H, interpolation='nearest', origin='low',
                  extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])
```

Pcolormesh can displaying exact bin edges:

```python
>>> ax = fig.add_subplot(132)
>>> ax.set_title('pcolormesh: exact bin edges')
>>> X, Y = np.meshgrid(xedges, yedges)
>>> ax.pcolormesh(X, Y, H)
>>> ax.set_aspect('equal')
```

NonUniformImage displays exact bin edges with interpolation:

```python
>>> ax = fig.add_subplot(133)
>>> ax.set_title('NonUniformImage: interpolated')
>>> im = mpl.image.NonUniformImage(ax, interpolation='bilinear')
>>> xcenters = xedges[:-1] + 0.5 * (xedges[1:] - xedges[:-1])
>>> ycenters = yedges[:-1] + 0.5 * (yedges[1:] - yedges[:-1])
>>> im.set_data(xcenters, ycenters, H)
>>> ax.images.append(im)
>>> ax.set_xlim(xedges[0], xedges[-1])
>>> ax.set_ylim(yedges[0], yedges[-1])
>>> ax.set_aspect('equal')
>>> plt.show()
```

```python
numpy.histogramdd(sample, bins=10, range=None, normed=False, weights=None)
```

Compute the multidimensional histogram of some data.

**Parameters**

- **sample** : array_like
  The data to be histogrammed. It must be an (N,D) array or data that can be converted to such. The rows of the resulting array are the coordinates of points in a D dimensional polytope.

- **bins** : sequence or int, optional
  The bin specification:
  - A sequence of arrays describing the bin edges along each dimension.
  - The number of bins for each dimension (nx, ny, ... =bins)
  - The number of bins for all dimensions (nx=ny=...=bins).
range : sequence, optional

A sequence of lower and upper bin edges to be used if the edges are not given explicitly in bins. Defaults to the minimum and maximum values along each dimension.

normed : bool, optional

If False, returns the number of samples in each bin. If True, returns the bin density \( \text{bin\_count} / \text{sample\_count} / \text{bin\_volume} \).

weights : array_like (N,), optional

An array of values \( w_i \) weighing each sample \((x_i, y_i, z_i, ...)\). Weights are normalized to 1 if normed is True. If normed is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

Returns

H : ndarray

The multidimensional histogram of sample x. See normed and weights for the different possible semantics.

edges : list

A list of D arrays describing the bin edges for each dimension.

See Also:

histogram
1-D histogram

histogram2d
2-D histogram

Examples

>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5, 8, 4), 6, 9, 5)

numpy.bincount (x, weights=None, minlength=None)

Count number of occurrences of each value in array of non-negative ints.

The number of bins (of size 1) is one larger than the largest value in x. If minlength is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of x). Each bin gives the number of occurrences of its index value in x. If weights is specified the input array is weighted by it, i.e. if a value \( n \) is found at position \( i \), \( \text{out}[n] += \text{weight}[i] \) instead of \( \text{out}[n] += 1 \).

Parameters

x : array_like, 1 dimension, nonnegative ints

Input array.

weights : array_like, optional

Weights, array of the same shape as x.

minlength : int, optional

New in version 1.6.0. A minimum number of bins for the output array.

Returns

out : ndarray of ints
The result of binning the input array. The length of \textit{out} is equal to \texttt{np.amax(x)+1}.

\textbf{Raises}

\textbf{ValueError}

If the input is not 1-dimensional, or contains elements with negative values, or if \textit{min-length} is non-positive.

\textbf{TypeError}

If the type of the input is float or complex.

\textbf{See Also:}

\texttt{histogram}, \texttt{digitize}, \texttt{unique}

\textbf{Examples}

\begin{verbatim}
>>> np.bincount(np.arange(5))
array([1, 1, 1, 1, 1])
>>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
array([1, 3, 1, 1, 0, 0, 0, 1])
>>>
x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
>>> np.bincount(x).size == np.amax(x)+1
True
\end{verbatim}

The input array needs to be of integer dtype, otherwise a TypeError is raised:

\begin{verbatim}
>>> np.bincount(np.arange(5, dtype=np.float))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    TypeError: array cannot be safely cast to required type
\end{verbatim}

A possible use of \texttt{bincount} is to perform sums over variable-size chunks of an array, using the \texttt{weights} keyword.

\begin{verbatim}
>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> np.bincount(x, weights=w)
array([ 0.3, 0.7, 1.1])
\end{verbatim}

\texttt{numpy.digitize(x, bins, right=False)}

Return the indices of the bins to which each value in input array belongs.

Each index \texttt{i} returned is such that \texttt{bins[i-1] <= x < bins[i]} if \texttt{bins} is monotonically increasing, or \texttt{bins[i-1] > x >= bins[i]} if \texttt{bins} is monotonically decreasing. If values in \texttt{x} are beyond the bounds of \texttt{bins}, 0 or \texttt{len(bins)} is returned as appropriate. If \texttt{right} is True, then the right bin is closed so that the index \texttt{i} is such that \texttt{bins[i-1] < x <= bins[i]} or \texttt{bins[i-1] >= x > bins[i]} if \texttt{bins} is monotonically increasing or decreasing, respectively.

**Parameters**

\texttt{x : array_like}

Input array to be binned. It has to be 1-dimensional.

\texttt{bins : array_like}

Array of bins. It has to be 1-dimensional and monotonic.

\texttt{right : bool, optional}
Indicating whether the intervals include the right or the left bin edge. Default behavior is \( (\text{right}==\text{False}) \) indicating that the interval does not include the right edge. The left bin and is open in this case. I.e., \( \text{bins}[i-1] \leq x < \text{bins}[i] \) is the default behavior for monotonically increasing bins.

**Returns**

- **out**: ndarray of ints
  
  Output array of indices, of same shape as \( x \).

**Raises**

- **ValueError**
  
  If the input is not 1-dimensional, or if \( \text{bins} \) is not monotonic.

- **TypeError**
  
  If the type of the input is complex.

**See Also:**

- `bincount`, `histogram`, `unique`

**Notes**

If values in \( x \) are such that they fall outside the bin range, attempting to index \( \text{bins} \) with the indices that \( \text{digitize} \) returns will result in an `IndexError`.

**Examples**

```python
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
...     print bins[inds[n]-1], '<=', x[n], '<', bins[inds[n]]
...     ...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5

>>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = np.array([0,5,10,15,20])
>>> np.digitize(x,bins,right=True)
array([1, 2, 3, 4, 4])
>>> np.digitize(x,bins,right=False)
array([1, 3, 3, 4, 5])
```

### 3.31 Test Support (numpy.testing)

Common test support for all numpy test scripts.

This single module should provide all the common functionality for numpy tests in a single location, so that test scripts can just import it and work right away.
## 3.32 Asserts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert_almost_equal</code></td>
<td>Raise an assertion if two items are not equal up to desired precision.</td>
</tr>
<tr>
<td><code>assert_approx_equal</code></td>
<td>Raise an assertion if two items are not equal up to significant digits.</td>
</tr>
<tr>
<td><code>assert_allclose</code></td>
<td>Raise an assertion if two objects are not equal up to desired precision.</td>
</tr>
<tr>
<td><code>assert_array_almost_equal</code></td>
<td>Raise an assertion if two objects are not equal up to desired precision.</td>
</tr>
<tr>
<td><code>assert_array_almost_equal_nulp</code></td>
<td>Raise an assertion if two objects are not equal up to desired tolerance.</td>
</tr>
<tr>
<td><code>assert_array_max_ulp</code></td>
<td>Check that all items of arrays differ in at most N Units in the Last Place.</td>
</tr>
<tr>
<td><code>assert_array_equal</code></td>
<td>Raise an assertion if two array_like objects are not equal.</td>
</tr>
<tr>
<td><code>assert_array_less</code></td>
<td>Raise an assertion if two array_like objects are not ordered by less than.</td>
</tr>
<tr>
<td><code>assert_equal</code></td>
<td>Raise an assertion if two objects are not equal.</td>
</tr>
<tr>
<td><code>assert_raises</code></td>
<td>Fail unless an exception of class exception_class is thrown by callable when</td>
</tr>
<tr>
<td><code>assert_warns</code></td>
<td>Fail unless the given callable throws the specified warning.</td>
</tr>
<tr>
<td><code>assert_string_equal</code></td>
<td>Test if two strings are equal.</td>
</tr>
</tbody>
</table>

```python
numpy.testing.assert_almost_equal(actual, desired[, decimal=7, err_msg='', verbose=True])
```

Raise an assertion if two items are not equal up to desired precision.

**Note:** It is recommended to use one of `assert_allclose`, `assert_array_almost_equal_nulp` or `assert_array_max_ulp` instead of this function for more consistent floating point comparisons.

The test is equivalent to \( \text{abs}(\text{desired} - \text{actual}) < 0.5 \times 10^{*(-\text{decimal})} \).

Given two objects (numbers or ndarrays), check that all elements of these objects are almost equal. An exception is raised at conflicting values. For ndarrays this delegates to `assert_array_almost_equal`.

**Parameters**
- `actual` : array_like
  The object to check.
- `desired` : array_like
  The expected object.
- `decimal` : int, optional
  Desired precision, default is 7.
- `err_msg` : str, optional
  The error message to be printed in case of failure.
- `verbose` : bool, optional
  If True, the conflicting values are appended to the error message.

**Raises**
- `AssertionError` if actual and desired are not equal up to specified precision.

**See Also:**
- `assert_allclose`
  Compare two array_like objects for equality with desired relative and/or absolute precision.
- `assert_array_almost_equal_nulp`, `assert_array_max_ulp`, `assert_equal`
Examples

```python
>>> import numpy.testing as npt
test
```
Examples

```python
>>> np.testing.assert_approx_equal(0.12345677777777e-20, 0.1234567e-20)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345671e-20, significant=8)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345672e-20, significant=8)
```

... <type 'exceptions.AssertionError'>:
Items are not equal to 8 significant digits:
ACTUAL: 1.234567e-021
DESIRED: 1.2345672000000001e-021

the evaluated condition that raises the exception is
```python
>>> abs(0.12345670e-20/1e-21 - 0.12345672e-20/1e-21) >= 10**-(8-1)
```
True

`numpy.testing.assert_array_almost_equal(x, y, decimal=6, err_msg='', verbose=True)`
Raise an assertion if two objects are not equal up to desired precision.

**Note:** It is recommended to use one of `assert_allclose`, `assert_array_almost_equal_nulp` or `assert_array_max_ulp` instead of this function for more consistent floating point comparisons.

The test verifies identical shapes and verifies values with `abs(desired-actual) < 0.5 * 10**(-decimal)`.

Given two array_like objects, check that the shape is equal and all elements of these objects are almost equal. An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

**Parameters**

- `x`: array_like
  The actual object to check.

- `y`: array_like
  The desired, expected object.

- `decimal`: int, optional
  Desired precision, default is 6.

- `err_msg`: str, optional
  The error message to be printed in case of failure.

- `verbose`: bool, optional
  If True, the conflicting values are appended to the error message.

**Raises**

- `AssertionError`
  If actual and desired are not equal up to specified precision.

**See Also:**

- `assert_allclose`
  Compare two array_like objects for equality with desired relative and/or absolute precision.

- `assert_array_almost_equal_nulp`, `assert_array_max_ulp`, `assert_equal`
Examples

the first assert does not raise an exception

```python
>>> np.testing.assert_array_almost_equal([1.0, 2.333, np.nan],
    [1.0, 2.333, np.nan])

>>> np.testing.assert_array_almost_equal([1.0, 2.33333, np.nan],
    [1.0, 2.33339, np.nan], decimal=5)
  <type 'exceptions.AssertionError'>:
  AssertionError:
  Arrays are not almost equal
  (mismatch 50.0%)
  x: array([ 1.      ,  2.33333,   NaN])
  y: array([ 1.      ,  2.33339,   NaN])

>>> np.testing.assert_array_almost_equal([1.0, 2.33333, np.nan],
    [1.0, 2.33333, 5.], decimal=5)
  <type 'exceptions.ValueError'>:
  ValueError:
  Arrays are not almost equal
  x: array([ 1.      ,  2.33333,   NaN])
  y: array([ 1.      ,  2.33333,   5.0])
```

numpy.testing.assert_allclose(actual, desired, rtol=1e-07, atol=0, err_msg='', verbose=True)

Raise an assertion if two objects are not equal up to desired tolerance.

The test is equivalent to allclose(actual, desired, rtol, atol). It compares the difference between actual and desired to atol + rtol * abs(desired). New in version 1.5.0.

**Parameters**

- **actual**: array_like
  - Array obtained.
- **desired**: array_like
  - Array desired.
- **rtol**: float, optional
  - Relative tolerance.
- **atol**: float, optional
  - Absolute tolerance.
- **err_msg**: str, optional
  - The error message to be printed in case of failure.
- **verbose**: bool, optional
  - If True, the conflicting values are appended to the error message.

**Raises**

- **AssertionError**
  - If actual and desired are not equal up to specified precision.

**See Also:**

assert_array_almost_equal_nulp, assert_array_max_ulp
Examples

```python
>>> x = [1e-5, 1e-3, 1e-1]
>>> y = np.arccos(np.cos(x))
>>> assert_allclose(x, y, rtol=1e-5, atol=0)
```

`numpy.testing.assert_almost_equal_nulp(x, y, nulp=1)`

Compare two arrays relatively to their spacing.

This is a relatively robust method to compare two arrays whose amplitude is variable.

Parameters

- `x, y` : array_like
  Input arrays.
- `nulp` : int, optional
  The maximum number of unit in the last place for tolerance (see Notes). Default is 1.

Returns

None

Raises

- `AssertionError`
  If the spacing between `x` and `y` for one or more elements is larger than `nulp`.

See Also:

- `assert_array_max_ulp`
  Check that all items of arrays differ in at most N Units in the Last Place.
- `spacing`
  Return the distance between `x` and the nearest adjacent number.

Notes

An assertion is raised if the following condition is not met:

```
abs(x - y) <= nulps * spacing(max(abs(x), abs(y)))
```

Examples

```python
>>> x = np.array([1., 1e-10, 1e-20])
>>> eps = np.finfo(x.dtype).eps
>>> np.testing.assert_array_almost_equal_nulp(x, x*eps/2 + x)
```

`numpy.testing.assert_array_max_ulp(a, b, maxulp=1, dtype=None)`

Check that all items of arrays differ in at most N Units in the Last Place.

Parameters

- `a, b` : array_like
  Input arrays to be compared.
- `maxulp` : int, optional
  The maximum number of units in the last place that elements of `a` and `b` can differ. Default is 1.
- `dtype` : dtype, optional
  Data-type to convert `a` and `b` to if given. Default is None.
Returns

ret : ndarray
    Array containing number of representable floating point numbers between items in \( a \)
    and \( b \).

Raises

AssertionError
    If one or more elements differ by more than \( \text{maxulp} \).

See Also:

assert_array_almost_equal_nulp
    Compare two arrays relatively to their spacing.

Examples

>>> a = np.linspace(0., 1., 100)
>>> res = np.testing.assert_array_max_ulp(a, np.arcsin(np.sin(a)))

numpy.testing.assert_array_equal(x, y, err_msg='', verbose=True)
    Raise an assertion if two array_like objects are not equal.

    Given two array_like objects, check that the shape is equal and all elements of these objects are equal. An
    exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs
    are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

    The usual caution for verifying equality with floating point numbers is advised.

Parameters

x : array_like
    The actual object to check.

y : array_like
    The desired, expected object.

err_msg : str, optional
    The error message to be printed in case of failure.

verbose : bool, optional
    If True, the conflicting values are appended to the error message.

Raises

AssertionError
    If actual and desired objects are not equal.

See Also:

assert_allclose
    Compare two array_like objects for equality with desired relative and/or absolute precision.

assert_array_almost_equal_nulp, assert_array_max_ulp, assert_equal
Examples

The first assert does not raise an exception:

```python
>>> np.testing.assert_array_equal([1.0,2.33333,np.nan],
... [np.exp(0),2.33333, np.nan])
```

Assert fails with numerical inprecision with floats:

```python
>>> np.testing.assert_array_equal([1.0,np.pi,np.nan],
... [1, np.sqrt(np.pi)**2, np.nan])
...<type 'exceptions.ValueError'>:
AssertionError:
Arrays are not equal
```

(50.0%)
```
x: array([ 1. , 3.14159265, NaN])
y: array([ 1. , 3.14159265, NaN])
```

Use `assert_allclose` or one of the nulp (number of floating point values) functions for these cases instead:

```python
>>> np.testing.assert_allclose([1.0,np.pi,np.nan],
... [1, np.sqrt(np.pi)**2, np.nan],
... rtol=1e-10, atol=0)
```

`numpy.testing.assert_array_less(x, y, err_msg='', verbose=True)`

Raise an assertion if two array_like objects are not ordered by less than.

Given two array_like objects, check that the shape is equal and all elements of the first object are strictly smaller than those of the second object. An exception is raised at shape mismatch or incorrectly ordered values. Shape mismatch does not raise if an object has zero dimension. In contrast to the standard usage in numpy, NaNs are compared, no assertion is raised if both objects have NaNs in the same positions.

Parameters

- **x**: array_like
  The smaller object to check.
- **y**: array_like
  The larger object to compare.
- **err_msg**: string
  The error message to be printed in case of failure.
- **verbose**: bool
  If True, the conflicting values are appended to the error message.

Raises

- **AssertionError**
  If actual and desired objects are not equal.

See Also:

- `assert_array_equal`
  tests objects for equality
- `assert_array_almost_equal`
  test objects for equality up to precision
Examples

```python
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1.1, 2.0, np.nan])
...
<type 'exceptions.ValueError'>:
Arrays are not less-ordered
(mismatch 50.0%)  
x: array([ 1.,  1., NaN])
y: array([ 1.,  2., NaN])

>>> np.testing.assert_array_less([1.0, 4.0], 3)
...
<type 'exceptions.ValueError'>:
Arrays are not less-ordered
(mismatch 50.0%)  
x: array([ 1.,  4.])
y: array(3)

>>> np.testing.assert_array_less([1.0, 2.0, 3.0], [4])
...
<type 'exceptions.ValueError'>:
Arrays are not less-ordered
(shapes (3,), (1,) mismatch)  
x: array([ 1.,  2.,  3.])
y: array([4])
```

`numpy.testing.assert_equal(actual, desired, err_msg='', verbose=True)`
Raise an assertion if two objects are not equal.

Given two objects (scalars, lists, tuples, dictionaries or NumPy arrays), check that all elements of these objects are equal. An exception is raised at the first conflicting values.

**Parameters**
- `actual`: array_like
  The object to check.
- `desired`: array_like
  The expected object.
- `err_msg`: str, optional
  The error message to be printed in case of failure.
- `verbose`: bool, optional
  If True, the conflicting values are appended to the error message.

**Raises**
- `AssertionError`
  If actual and desired are not equal.

**Examples**

```python
>>> np.testing.assert_equal([4,5], [4,6])
...
<type 'exceptions.AssertionError'>:
Items are not equal:
item=1
```

3.32. Asserts
### numpy.testing.assert_raises

Fail unless an exception of class `exception_class` is thrown by callable when invoked with arguments `args` and keyword arguments `kwargs`. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

### numpy.testing.assert_warns

Fail unless the given callable throws the specified warning.

A warning of class `warning_class` should be thrown by the callable when invoked with arguments `args` and keyword arguments `kwargs`. If a different type of warning is thrown, it will not be caught, and the test case will be deemed to have suffered an error. New in version 1.4.0.

**Parameters**

- `warning_class`: class
  
  The class defining the warning that `func` is expected to throw.

- `func`: callable
  
  The callable to test.

- `*args`: Arguments
  
  Arguments passed to `func`.

- `**kwargs`: Kwargs
  
  Keyword arguments passed to `func`.

**Returns**

The value returned by `func`.

### numpy.testing.assert_string_equal

Test if two strings are equal.

If the given strings are equal, `assert_string_equal` does nothing. If they are not equal, an `AssertionError` is raised, and the diff between the strings is shown.

**Parameters**

- `actual`: str
  
  The string to test for equality against the expected string.

- `desired`: str
  
  The expected string.

**Examples**

```python
>>> np.testing.assert_string_equal('abc', 'abc')
```

```python
>>> np.testing.assert_string_equal('abc', 'abcd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
AssertionError: Differences in strings:
  - abc+ abcd?
```

### 3.32.1 Decorators
NumPy Reference, Release 1.8.1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deprecated(conditional)</code></td>
<td>Filter deprecation warnings while running the test suite.</td>
</tr>
<tr>
<td><code>knownfailureif(fail_condition[, msg])</code></td>
<td>Make function raise KnownFailureTest exception if given condition is true.</td>
</tr>
<tr>
<td><code>skipif(skip_condition[, msg])</code></td>
<td>Make function raise SkipTest exception if a given condition is true.</td>
</tr>
<tr>
<td><code>slow(t)</code></td>
<td>Label a test as ‘slow’.</td>
</tr>
<tr>
<td><code>decorate_methods(cls, decorator[, testmatch])</code></td>
<td>Apply a decorator to all methods in a class matching a regular expression.</td>
</tr>
</tbody>
</table>

**numpy.testing.decorators.deprecated(conditional=True)**

Filter deprecation warnings while running the test suite.

This decorator can be used to filter DeprecationWarning’s, to avoid printing them during the test suite run, while checking that the test actually raises a DeprecationWarning.

**Parameters**

- `conditional`: bool or callable, optional
  - Flag to determine whether to mark test as deprecated or not. If the condition is a callable, it is used at runtime to dynamically make the decision. Default is True.

**Returns**

- `decorator`: function
  - The deprecated decorator itself.

**Notes**

New in version 1.4.0.

**numpy.testing.decorators.knownfailureif(fail_condition, msg=None)**

Make function raise KnownFailureTest exception if given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

- `fail_condition`: bool or callable
  - Flag to determine whether to mark the decorated test as a known failure (if True) or not (if False).

- `msg`: str, optional
  - Message to give on raising a KnownFailureTest exception. Default is None.

**Returns**

- `decorator`: function
  - Decorator, which, when applied to a function, causes SkipTest to be raised when `skip_condition` is True, and the function to be called normally otherwise.

**Notes**

The decorator itself is decorated with the nose.tools.make_decorator function in order to transmit function name, and various other metadata.

**numpy.testing.decorators.setastest(tf=True)**

Signals to nose that this function is or is not a test.

**Parameters**

- `tf`: bool
If True, specifies that the decorated callable is a test. If False, specifies that the decorated callable is not a test. Default is True.

**Notes**

This decorator can’t use the nose namespace, because it can be called from a non-test module. See also `istest` and `nottest` in `nose.tools`.

**Examples**

`setastest` can be used in the following way:

```python
from numpy.testing.decorators import setastest

@setastest(False)
def func_with_test_in_name(arg1, arg2):
    pass
```

**numpy.testing.decorators.skipif** *(skip_condition, msg=None)*

Make function raise SkipTest exception if a given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

- `skip_condition` : bool or callable
  - Flag to determine whether to skip the decorated test.

- `msg` : str, optional
  - Message to give on raising a SkipTest exception. Default is None.

**Returns**

- `skip_condition` : function
  - Decorator which, when applied to a function, causes SkipTest to be raised when `skip_condition` is True, and the function to be called normally otherwise.

**Notes**

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

**numpy.testing.decorators.slow** *(t)*

Label a test as ‘slow’.

The exact definition of a slow test is obviously both subjective and hardware-dependent, but in general any individual test that requires more than a second or two should be labeled as slow (the whole suite consists of thousands of tests, so even a second is significant).

**Parameters**

- `t` : callable
  - The test to label as slow.

**Returns**

- `t` : callable
  - The decorated test `t`.
Examples

The `numpy.testing` module includes import decorators as dec. A test can be decorated as slow like this:

```python
from numpy.testing import *

@dec.slow
def test_big(self):
    print 'Big, slow test'
```

`numpy.testing.decorate_methods(cls, decorator, testmatch=None)`

Apply a decorator to all methods in a class matching a regular expression.

The given decorator is applied to all public methods of `cls` that are matched by the regular expression `testmatch` (`testmatch.search(methodname)`). Methods that are private, i.e. start with an underscore, are ignored.

Parameters

- **cls**: class
  - Class whose methods to decorate.
- **decorator**: function
  - Decorator to apply to methods
- **testmatch**: compiled regexp or str, optional
  - The regular expression. Default value is None, in which case the nose default (`re.compile(r’(?:^(?:[^\b_\%^#]{1}[\b_\%^#]{1}[Tt]est' % os.sep))`) is used. If `testmatch` is a string, it is compiled to a regular expression first.

3.32.2 Test Running

<table>
<thead>
<tr>
<th><code>Tester</code></th>
<th>Nose test runner.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run_module_suite(file_to_run)</code></td>
<td></td>
</tr>
<tr>
<td><code>rundocs(filename, raise_on_error)</code></td>
<td>Run doctests found in the given file.</td>
</tr>
</tbody>
</table>

`numpy.testing.Tester`

alias of NoseTester

`numpy.testing.run_module_suite(file_to_run=None)`

`numpy.testing.rundocs(filename=None, raise_on_error=True)`

Run doctests found in the given file.

By default `rundocs` raises an `AssertionError` on failure.

Parameters

- **filename**: str
  - The path to the file for which the doctests are run.
- **raise_on_error**: bool
  - Whether to raise an `AssertionError` when a doctest fails. Default is True.
Notes

The doctests can be run by the user/developer by adding the doctests argument to the test() call. For example, to run all tests (including doctests) for numpy.lib:

```python
>>> np.lib.test(doctests=True)
```

3.33 Window functions

3.33.1 Various windows

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bartlett(M)</td>
<td>Return the Bartlett window.</td>
</tr>
<tr>
<td>blackman(M)</td>
<td>Return the Blackman window.</td>
</tr>
<tr>
<td>hamming(M)</td>
<td>Return the Hamming window.</td>
</tr>
<tr>
<td>hanning(M)</td>
<td>Return the Hanning window.</td>
</tr>
<tr>
<td>kaiser(M, beta)</td>
<td>Return the Kaiser window.</td>
</tr>
</tbody>
</table>

**numpy.bartlett(M)**

Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

**Parameters**

- **M**: int
  
  Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

- **out**: array
  
  The triangular window, with the maximum value normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

See Also:

blackman, hamming, hanning, kaiser

**Notes**

The Bartlett window is defined as

\[
w(n) = \frac{2}{M-1} \left( \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right)
\]

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The fourier transform of the Bartlett is the product of two sinc functions. Note the excellent discussion in Kanasewich.

**References**

[R11], [R12], [R13], [R14], [R15]
Examples

```python
>>> np.bartlett(12)
array([ 0.        ,  0.18181818,  0.36363636,  0.54545455,  0.72727273,
       0.90909091,  0.90909091,  0.72727273,  0.54545455,  0.36363636,
       0.18181818,  0.        ])
```

Plot the window and its frequency response (requires SciPy and matplotlib):

```python
>>> from numpy.fft import fft, fftshift
>>> window = np.bartlett(51)
```

```python
>>> plt.plot(window)
```

```python
>>> plt.title("Bartlett window")
```

```python
>>> plt.ylabel("Amplitude")
```

```python
>>> plt.xlabel("Sample")
```

```python
>>> plt.show()
```

```python
>>> plt.figure()
```

```python
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
```

```python
>>> plt.title("Frequency response of Bartlett window")
```

```python
>>> plt.ylabel("Magnitude [dB]")
```

```python
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

```python
>>> plt.axis('tight')
```

```python
>>> plt.show()
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
Notes

The Blackman window is defined as

\[ w(n) = 0.42 - 0.5 \cos(2\pi n / M) + 0.08 \cos(4\pi n / M) \]

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a “near optimal” tapering function, almost as good (by some measures) as the kaiser window.

References


Examples

```python
>>> np.blackman(12)
array([-1.38777878e-17, 3.26064346e-02, 1.59903635e-01,
       4.14397981e-01, 7.36045180e-01, 9.67046769e-01,
       9.67046769e-01, 7.36045180e-01, 4.14397981e-01,
       1.59903635e-01, 3.26064346e-02, -1.38777878e-17])
```

Plot the window and the frequency response:

```python
>>> from numpy.fft import fft, fftshift

>>> window = np.blackman(51)

>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]

>>> plt.title("Blackman window")
<matplotlib.text.Text object at 0x...>

>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>

>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>

>>> plt.show()
```

```python
>>> A = fft(window, 2048) / 25.5

>>> mag = np.abs(fftshift(A))

>>> freq = np.linspace(-0.5, 0.5, len(A))

>>> response = 20 * np.log10(mag)

>>> response = np.clip(response, -100, 100)

>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]

>>> plt.title("Frequency response of Blackman window")
<matplotlib.text.Text object at 0x...>

>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>

>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>

>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)

>>> plt.show()
```
numpy.hanning(M)

Return the Hamming window.

The Hamming window is a taper formed by using a weighted cosine.

Parameters
M : int
   Number of points in the output window. If zero or less, an empty array is returned.

Returns
out : ndarray
   The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

See Also:
bartlett, blackman, hanning, kaiser

Notes
The Hamming window is defined as

\[ w(n) = 0.54 - 0.46 \cos \left( \frac{2\pi n}{M - 1} \right) \quad 0 \leq n \leq M - 1 \]

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References
[R21], [R22], [R23], [R24]

Examples
>>> np.hanning(12)
array([ 0.08 , 0.15302337, 0.34890909, 0.60546483, 0.84123594,
       0.98136677, 0.98136677, 0.84123594, 0.60546483, 0.34890909,
       0.15302337, 0.08 ])

Plot the window and the frequency response:
>>> from numpy.fft import fft, fftshift
>>> window = np.hanning(51)
>>> plt.plot(window)
<matplotlib.lines.Line2D object at 0x...>
>>> plt.title("Hamming window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()
```python
>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Hamming window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()
```

numpy.hanning(M)

Return the Hanning window.

The Hanning window is a taper formed by using a weighted cosine.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

Returns

out : ndarray, shape(M,)

The window, with the maximum value normalized to one (the value one appears only if
M is odd).

See Also:

bartlett, blackman, hamming, kaiser

Notes

The Hanning window is defined as

\[
w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) \quad 0 \leq n \leq M - 1
\]

The Hanning was named for Julius van Hann, an Austrian meteorologist. It is also known as the Cosine Bell. Some authors prefer that it be called a Hann window, to help avoid confusion with the very similar Hamming window.

Most references to the Hanning window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R25], [R26], [R27], [R28]
Examples

```python
>>> np.hanning(12)
array([ 0. , 0.07937323, 0.29229249, 0.57115742, 0.82743037, 0.97974649, 0.97974649, 0.82743037, 0.57115742, 0.29229249, 0.07937323, 0. ])
```

Plot the window and its frequency response:

```python
>>> from numpy.fft import fft, fftshift
>>> window = np.hanning(51)
>>> plt.plot(window)
<matplotlib.lines.Line2D object at 0x...>
>>> plt.title("Hann window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Sample")
<matplotlib.text.Text object at 0x...>
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
<matplotlib.lines.Line2D object at 0x...>
>>> plt.title("Frequency response of the Hann window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()
```

numpy.kaiser(M, beta)

Return the Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

Parameters

- **M**: int
  Number of points in the output window. If zero or less, an empty array is returned.

- **beta**: float
  Shape parameter for window.

Returns

- **out**: array
  The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

See Also:
The Kaiser window is defined as

\[ w(n) = I_0 \left( \beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta) \]

with

\[-\frac{M-1}{2} \leq n \leq \frac{M-1}{2},\]

where \( I_0 \) is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple approximation to the DPSS window based on Bessel functions. The Kaiser window is a very good approximation to the Digital Prolate Spheroidal Sequence, or Slepian window, which is the transform which maximizes the energy in the main lobe of the window relative to total energy.

The Kaiser can approximate many other windows by varying the beta parameter.

<table>
<thead>
<tr>
<th>beta</th>
<th>Window shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Rectangular</td>
</tr>
<tr>
<td>5</td>
<td>Similar to a Hamming</td>
</tr>
<tr>
<td>6</td>
<td>Similar to a Hanning</td>
</tr>
<tr>
<td>8.6</td>
<td>Similar to a Blackman</td>
</tr>
</tbody>
</table>

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise NaNs will get returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means “removing the foot”, i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R34], [R35], [R36]

Examples

```python
>>> np.kaiser(12, 14)
array([ 7.72686684e-06, 3.46009194e-03, 4.65200189e-02,
       2.29737120e-01, 5.99885316e-01, 9.45674898e-01,
       9.45674898e-01, 5.99885316e-01, 2.29737120e-01,
       4.65200189e-02, 3.46009194e-03, 7.72686684e-06])
```

Plot the window and the frequency response:

```python
>>> from numpy.fft import fft, fftshift
>>> window = np.kaiser(51, 14)
>>> plt.plot(window)
<matplotlib.lines.Line2D object at 0x...>
>>> plt.title("Kaiser window")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
```
```python
>>> plt.xlabel("Sample")
>>> plt.show()

>>> plt.figure()
<matplotlib.figure.Figure object at 0x...>

>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Kaiser window")
<matplotlib.text.Text object at 0x...>

>>> plt.ylabel("Magnitude [dB]")
<matplotlib.text.Text object at 0x...>

>>> plt.xlabel("Normalized frequency [cycles per sample]")
<matplotlib.text.Text object at 0x...>

>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...)
>>> plt.show()
```
NumPy provides enhanced distutils functionality to make it easier to build and install sub-packages, auto-generate code, and extension modules that use Fortran-compiled libraries. To use features of NumPy distutils, use the setup command from numpy.distutils.core. A useful Configuration class is also provided in numpy.distutils.misc_util that can make it easier to construct keyword arguments to pass to the setup function (by passing the dictionary obtained from the todict() method of the class). More information is available in the NumPy Distutils Users Guide in <site-packages>/numpy/doc/DISTUTILS.txt.

### 4.1 Modules in numpy.distutils

#### 4.1.1 misc_util

```python
get_numpy_include_dirs()
dict_append(d, **kws)
appendpath(prefix, path)
allpath(name)  # Convert a /-separated pathname to one using the OS’s path separator.
dot_join(*args)
genrate_config_py(target)  # Generate config.py file containing system_info information used during building the package.
get_cmd(cmdname[, .cache])
terminal_has_colors()
red_text(s)
green_text(s)
yellow_text(s)
blue_text(s)
cyan_text(s)
cyg2win32(path)
all_strings(lst)  # Return True if all items in lst are string objects.
has_f_sources(sources)  # Return True if sources contains Fortran files
has_cxx_sources(sources)  # Return True if sources contains C++ files
filter_sources(sources)  # Return four lists of filenames containing
get_dependencies(sources)
is_local_src_dir(directory)  # Return true if directory is local directory.
get_ext_source_files(ext)
get_script_files(scripts)
numpy.distutils.misc_util.get_numpy_include_dirs()
```
numpy.distutils.misc_util.dict_append(d, **kws)

numpy.distutils.misc_util.appendpath(prefix, path)

numpy.distutils.misc_util.allpath(name)
   Convert a /-separated pathname to one using the OS’s path separator.

numpy.distutils.misc_util.dot_join(*args)

numpy.distutils.misc_util.generate_config_py(target)
   Generate config.py file containing system_info information used during building the package.

   Usage:
   config['py_modules'].append((packagename, '__config__', generate_config_py))

numpy.distutils.misc_util.get_cmd(cmdname, _cache={})

numpy.distutils.misc_util.terminal_has_colors()

numpy.distutils.misc_util.red_text(s)

numpy.distutils.misc_util.green_text(s)

numpy.distutils.misc_util.yellow_text(s)

numpy.distutils.misc_util.blue_text(s)

numpy.distutils.misc_util.cyan_text(s)

numpy.distutils.misc_util.cyg2win32(path)

numpy.distutils.misc_util.all_strings(lst)
   Return True if all items in lst are string objects.

numpy.distutils.misc_util.has_f_sources(sources)
   Return True if sources contains Fortran files.

numpy.distutils.misc_util.has_cxx_sources(sources)
   Return True if sources contains C++ files.

numpy.distutils.misc_util.filter_sources(sources)
   Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.

numpy.distutils.misc_util.get_dependencies(sources)

numpy.distutils.misc_util.is_local_src_dir(directory)
   Return true if directory is local directory.

numpy.distutils.misc_util.get_ext_source_files(ext)
numpy.distutils.misc_util.get_script_files(scripts)

class numpy.distutils.misc_util.Configuration(package_name=None, parent_name=None, top_path=None, package_path=None, **attrs)
Construct a configuration instance for the given package name. If parent_name is not None, then construct the package as a sub-package of the parent_name package. If top_path and package_path are None then they are assumed equal to the path of the file this instance was created in. The setup.py files in the numpy distribution are good examples of how to use the Configuration instance.

todict()
Return a dictionary compatible with the keyword arguments of distutils setup function.

Examples
>>> setup(**config.todict())

get_distribution()
Return the distutils distribution object for self.

get_subpackage(subpackage_name=None, package_path=None, parent_name=None, caller_level=1)
Return list of subpackage configurations.

Parameters
subpackage_name : str or None
Name of the subpackage to get the configuration. '*' in subpackage_name is handled as a wildcard.

subpackage_path : str
If None, then the path is assumed to be the local path plus the subpackage_name. If a setup.py file is not found in the subpackage_path, then a default configuration is used.

parent_name : str
Parent name.

add_subpackage(subpackage_name, package_path=None, standalone=False)
Add a sub-package to the current Configuration instance.

This is useful in a setup.py script for adding sub-packages to a package.

Parameters
subpackage_name : str
name of the subpackage

subpackage_path : str
if given, the subpackage path such as the subpackage is in subpackage_path / subpackage_name. If None, the subpackage is assumed to be located in the local path / subpackage_name.

standalone : bool

add_data_files(*files)
Add data files to configuration data_files.

Parameters
files : sequence
Argument(s) can be either
• 2-sequence (<datadir prefix>,<path to data file(s)>)
• paths to data files where python datadir prefix defaults to package dir.

Notes
The form of each element of the files sequence is very flexible allowing many combinations of where to get the files from the package and where they should ultimately be installed on the system. The most basic usage is for an element of the files argument sequence to be a simple filename. This will cause that file from the local path to be installed to the installation path of the self.name package (package path). The file argument can also be a relative path in which case the entire relative path will be installed into the package directory. Finally, the file can be an absolute path name in which case the file will be found at the absolute path name but installed to the package path.

This basic behavior can be augmented by passing a 2-tuple in as the file argument. The first element of the tuple should specify the relative path (under the package install directory) where the remaining sequence of files should be installed to (it has nothing to do with the file-names in the source distribution). The second element of the tuple is the sequence of files that should be installed. The files in this sequence can be filenames, relative paths, or absolute paths. For absolute paths the file will be installed in the top-level package installation directory (regardless of the first argument). Filenames and relative path names will be installed in the package install directory under the path name given as the first element of the tuple.

Rules for installation paths:
1. file.txt -> (., file.txt) -> parent/file.txt
2. foo/file.txt -> (foo, foo/file.txt) -> parent/foo/file.txt
3. /foo/bar/file.txt -> (./foo/bar/file.txt) -> parent/file.txt
4. *.txt -> parent/a.txt, parent/b.txt
5. foo/* .txt -> parent/foo/a.txt, parent/foo/b.txt
6. /txt -> (, /*txt) -> parent/c/a.txt, parent/d/b.txt
7. (sun, file.txt) -> parent/sun/file.txt
8. (sun, bar/file.txt) -> parent/sun/file.txt
9. (sun, /foo/bar/file.txt) -> parent/sun/file.txt
10. (sun, * .txt) -> parent/sun/a.txt, parent/sun/b.txt
11. (sun, bar/* .txt) -> parent/sun/a.txt, parent/sun/b.txt
12. (sun, */txt) -> parent/sun/c/a.txt, parent/d/b.txt

An additional feature is that the path to a data-file can actually be a function that takes no arguments and returns the actual path(s) to the data-files. This is useful when the data files are generated while building the package.

Examples
Add files to the list of data_files to be included with the package.

```python
>>> self.add_data_files('foo.dat',
...                    ('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
...                    'bar/cat.dat',
...                    '/full/path/to/can.dat')
```

will install these data files to:
where <package install directory> is the package (or sub-package) directory such as
’/usr/lib/python2.4/site-packages/mypackage’ (‘C: Python2.4 Lib site-packages mypackage’) or
’/usr/lib/python2.4/site- packages/mypackage/mysubpackage’ (‘C: Python2.4 Lib site-packages my-
package mysubpackage’).

add_data_dir(data_path)

Recursively add files under data_path to data_files list.

Recursively add files under data_path to the list of data_files to be installed (and distributed). The data_path
can be either a relative path-name, or an absolute path-name, or a 2-tuple where the first argument shows
where in the install directory the data directory should be installed to.

Parameters
data_path : seq or str

Argument can be either
• 2-sequence (<datadir suffix>, <path to data directory>)
• path to data directory where python datadir suffix defaults to package dir.

Notes

Rules for installation paths:
foo/bar -> (foo/bar, foo/bar) -> parent/foo/bar (gun, foo/bar) -> parent/gun foo/* -> (foo/a, foo/a),
(foo/b, foo/b) -> parent/foo/a, parent/foo/b (gun, foo/) -> (gun, foo/a), (gun, foo/b) -> gun (gun/, foo/)
-> parent/gun/a, parent/gun/b /foo/bar -> (bar, /foo/bar) -> parent/bar (gun, /foo/bar) -> parent/gun
(fun//gun/*, sun/foo/bar) -> parent/fun/foo/gun/bar

Examples

For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat:

```python
>>> self.add_data_dir('fun')
>>> self.add_data_dir(('sun', 'fun'))
>>> self.add_data_dir(('gun', '/full/path/to/fun'))
```

Will install data-files to the locations:

```plaintext
<package install directory>/
 fun/
   foo.dat
 bar/
   car.dat
 sun/
   foo.dat
 bar/
   car.dat
 gun/
```
add_include_dirs(*paths)
Add paths to configuration include directories.
Add the given sequence of paths to the beginning of the include_dirs list. This list will be visible to all extension modules of the current package.

add_headers(*files)
Add installable headers to configuration.
Add the given sequence of files to the beginning of the headers list. By default, headers will be installed under <python-includeself.name.replace('.', '/')>/ directory. If an item of files is a tuple, then its first argument specifies the actual installation location relative to the <python-includeself.name.replace('.', '/')>/ path.

Parameters
files : str or seq
Argument(s) can be either:
• 2-sequence (<includer suffix>,<path to header file(s)>)
• path(s) to header file(s) where python includedir suffix will default to package name.

add_extension(name, sources, **kw)
Add extension to configuration.
Create and add an Extension instance to the ext_modules list. This method also takes the following optional keyword arguments that are passed on to the Extension constructor.

Parameters
name : str
name of the extension
sources : seq
list of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.
include_dirs :
define_macros :
define_macros :
library_dirs :
libraries :
runtime_library_dirs :
extra_objects :
extra_objects :
extra_compile_args :
extra_compile_args :
extra_link_args :
extra_link_args :
extra_f77_compile_args :
extra_f77_compile_args :
extra_f90_compile_args :
extra_f90_compile_args :
export_symbols :
swig_opts :
depends :
The depends list contains paths to files or directories that the sources of the extension module depend on. If any path in the depends list is newer than the extension module, then the module will be rebuilt.
language :
f2py_options :
module_dirs :
extra_info : dict or list
dict or list of dict of keywords to be appended to keywords.

Notes
The self.paths(...) method is applied to all lists that may contain paths.

add_library (name, sources, **build_info)
Add library to configuration.

Parameters
name : str
Name of the extension.
sources : sequence
List of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.
build_info : dict, optional
The following keys are allowed:
• depends
• macros
• include_dirs
• extra_compiler_args
• extra_f77_compiler_args
• extra_f90_compiler_args
• f2py_options
• language

add_scripts (*files)
Add scripts to configuration.

Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/directory.
add_installed_library \((name, sources, install\_dir, build\_info=None)\)

Similar to add_library, but the specified library is installed.

Most C libraries used with distutils are only used to build python extensions, but libraries built through this method will be installed so that they can be reused by third-party packages.

**Parameters**

- name : str
  
  Name of the installed library.

- sources : sequence
  
  List of the library’s source files. See add_library for details.

- install_dir : str
  
  Path to install the library, relative to the current sub-package.

- build_info : dict, optional
  
  The following keys are allowed:
  * depends
  * macros
  * include\_dirs
  * extra\_compiler\_args
  * extra\_f77\_compiler\_args
  * extra\_f90\_compiler\_args
  * f2py\_options
  * language

**Returns**

None

**See Also:**

add_library, add_npy_pkg_config, get_info

**Notes**

The best way to encode the options required to link against the specified C libraries is to use a “libname.ini” file, and use get_info to retrieve the required options (see add_npy_pkg_config for more information).

add_npy_pkg_config \((template, install\_dir, subst\_dict=None)\)

Generate and install a npy-pkg config file from a template.

The config file generated from template is installed in the given install directory, using subst_dict for variable substitution.

**Parameters**

- template : str
  
  The path of the template, relatively to the current package path.

- install_dir : str
  
  Where to install the npy-pkg config file, relatively to the current package path.

- subst_dict : dict, optional
If given, any string of the form @key@ will be replaced by subst_dict[key] in the template file when installed. The install prefix is always available through the variable @prefix@, since the install prefix is not easy to get reliably from setup.py.

**See Also:**
add_installed_library, get_info

**Notes**
This works for both standard installs and in-place builds, i.e. the @prefix@ refer to the source directory for in-place builds.

**Examples**

```python
config.add_npy_pkg_config('foo.ini.in', 'lib', {'foo': bar})
```

Assuming the foo.ini.in file has the following content:

```ini
[meta]
Name=foo
Version=1.0
Description=dummy description

[default]
Cflags=-I@prefix@/include
Libs=
```

The generated file will have the following content:

```ini
[meta]
Name=bar
Version=1.0
Description=dummy description

[default]
Cflags=-Iprefix_dir/include
Libs=
```

and will be installed as foo.ini in the ‘lib’ subpath.

**paths (**paths, **kws)**
Apply glob to paths and prepend local_path if needed.

Applies glob.glob(...) to each path in the sequence (if needed) and pre-pends the local_path if needed. Because this is called on all source lists, this allows wildcard characters to be specified in lists of sources for extension modules and libraries and scripts and allows path-names be relative to the source directory.

**get_config_cmd()**
Returns the numpy.distutils config command instance.

**get_build_temp_dir()**
Return a path to a temporary directory where temporary files should be placed.

**have_f77c()**
Check for availability of Fortran 77 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

**Notes**
True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).
have_f90c()
    Check for availability of Fortran 90 compiler.
    Use it inside source generating function to ensure that setup distribution instance has been initialized.

Notes
    True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

get_version(version_file=None, version_variable=None)
    Try to get version string of a package.
    Return a version string of the current package or None if the version information could not be detected.

Notes
    This method scans files named __version__.py, <packagename>_version.py, version.py, and
    __svn_version__.py for string variables version, __version__, and <packagename>_version, until a ver-
    sion number is found.

make_svn_version_py(delete=True)
    Appends a data function to the data_files list that will generate __svn_version__.py file to the current
    package directory.
    Generate package __svn_version__.py file from SVN revision number, it will be removed after python
    exits but will be available when sdist, etc commands are executed.

Notes
    If __svn_version__.py existed before, nothing is done.
    This is intended for working with source directories that are in an SVN repository.

make_config_py(name='__config__')
    Generate package __config__.py file containing system_info information used during building the pack-
    age.
    This file is installed to the package installation directory.

get_info(*names)
    Get resources information.
    Return information (from system_info.get_info) for all of the names in the argument list in a single dictio-

4.1.2 Other modules

    system_info.get_info(name[, notfound_action])    notfound_action:
    system_info.get_standard_file(fname)      Returns a list of files named ‘fname’ from
    cpuinfo.cpu
    log.set_verbosity(v[, force])
    exec_command

numpy.distutils.system_info.get_info(name, notfound_action=0)
    notfound_action: 0 - do nothing 1 - display warning message 2 - raise error

numpy.distutils.system_info.get_standard_file(fname)
    Returns a list of files named ‘fname’ from 1) System-wide directory (directory-location of this module) 2)
Users HOME directory (os.environ['HOME'])

numpy.distutils.cpuinfo.cpu = <numpy.distutils.cpuinfo.LinuxCPUInfo object at 0x471cf10>

numpy.distutils.log.set_verbosity(v, force=False)

exec_command

Implements exec_command function that is (almost) equivalent to commands.getstatusoutput function but on NT, DOS systems the returned status is actually correct (though, the returned status values may be different by a factor). In addition, exec_command takes keyword arguments for (re-)defining environment variables.

Provides functions:

exec_command — execute command in a specified directory and in the modified environment.

find_executable — locate a command using info from environment variable PATH. Equivalent to posix which command.

Author: Pearu Peterson <pearu@cens.ioc.ee> Created: 11 January 2003

Requires: Python 2.x

Successfully tested on:

<table>
<thead>
<tr>
<th>os.name</th>
<th>sys.platform</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix</td>
<td>linux2</td>
<td>Debian (sid) Linux, Python 2.1.3+, 2.2.3+, 2.3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PyCrust 0.9.3, Idle 1.0.2</td>
</tr>
<tr>
<td>posix</td>
<td>linux2</td>
<td>Red Hat 9 Linux, Python 2.1.3, 2.2.2, 2.3.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>posix5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>darwin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nt</td>
</tr>
</tbody>
</table>

Known bugs: - Tests, that send messages to stderr, fail when executed from MSYS prompt because the messages are lost at some point.

Functions

exec_command(command[, execute_in, ...])  Return (status,output) of executed command.
find_executable(exe[, path, _cache])  Return full path of a executable or None.
get_exception()  
get_pythonexe()  
is_sequence(seq)  
make_temp_file([suffix, prefix, text])  
open_latin1(filename[, mode])  
quote_arg(arg)  
splitcmdline(line)  
temp_file_name()  

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test(<strong>kws</strong>)</td>
<td></td>
</tr>
<tr>
<td>test_cl(<strong>kws</strong>)</td>
<td></td>
</tr>
<tr>
<td>test_execute_in(<strong>kws</strong>)</td>
<td></td>
</tr>
<tr>
<td>test_nt(<strong>kws</strong>)</td>
<td></td>
</tr>
<tr>
<td>test_posix(<strong>kws</strong>)</td>
<td></td>
</tr>
<tr>
<td>test_svn(<strong>kws</strong>)</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2 Building Installable C libraries

Conventional C libraries (installed through `add_library`) are not installed, and are just used during the build (they are statically linked). An installable C library is a pure C library, which does not depend on the python C runtime, and is installed such that it may be used by third-party packages. To build and install the C library, you just use the method `add_installed_library` instead of `add_library`, which takes the same arguments except for an additional `install_dir` argument:

```python
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
```

#### 4.2.1 npy-pkg-config files

To make the necessary build options available to third parties, you could use the `npy-pkg-config` mechanism implemented in `numpy.distutils`. This mechanism is based on a `.ini` file which contains all the options. A `.ini` file is very similar to `.pc` files as used by the pkg-config unix utility:

```
[meta]
Name: foo
Version: 1.0
Description: foo library

[variables]
prefix = /home/user/local
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

Generally, the file needs to be generated during the build, since it needs some information known at build time only (e.g. `prefix`). This is mostly automatic if one uses the `Configuration` method `add_npy_pkg_config`. Assuming we have a template file `foo.ini.in` as follows:

```
[meta]
Name: foo
Version: @version@
Description: foo library

[variables]
prefix = @prefix@
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

This is mostly automatic if one uses the `Configuration` method `add_npy_pkg_config`. Assuming we have a template file `foo.ini.in` as follows:

```
[meta]
Name: foo
Version: @version@
Description: foo library

[variables]
prefix = @prefix@
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```
and the following code in setup.py:

```python
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
>>> subst = {'version': '1.0'}
>>> config.add_npy_pkg_config('foo.ini.in', 'lib', subst_dict=subst)
```

This will install the file foo.ini into the directory package_dir/lib, and the foo.ini file will be generated from foo.ini.in, where each @version@ will be replaced by subst_dict['version']. The dictionary has an additional prefix substitution rule automatically added, which contains the install prefix (since this is not easy to get from setup.py). npy-pkg-config files can also be installed at the same location as used for numpy, using the path returned from get_npy_pkg_dir function.

4.2.2 Reusing a C library from another package

Info are easily retrieved from the get_info function in numpy.distutils.misc_util:

```python
>>> info = get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=**info)
```

An additional list of paths to look for .ini files can be given to get_info.

4.3 Conversion of .src files

NumPy distutils supports automatic conversion of source files named <somefile>.src. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named <somefile>.src is encountered, a new file named <somefile> is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named named <file>.ext.src where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases.

4.3.1 Fortran files

This template converter will replicate all function and subroutine blocks in the file with names that contain ‘<...>’ according to the rules in ‘<...>’. The number of comma-separated words in ‘<...>’ determines the number of times the block is repeated. What these words are indicates what that repeat rule, ‘<...>’, should be replaced with in each block. All of the repeat rules in a block must contain the same number of comma-separated words indicating the number of times that block should be repeated. If the word in the repeat rule needs a comma, leftarrow, or rightarrow, then prepend it with a backslash \ . If a word in the repeat rule matches \ <index> then it will be replaced with the <index>-th word in the same repeat specification. There are two forms for the repeat rule: named and short.

Named repeat rule

A named repeat rule is useful when the same set of repeats must be used several times in a block. It is specified using <rule1=item1, item2, item3,..., itemN>, where N is the number of times the block should be repeated. On each repeat of the block, the entire expression, ‘<...>’ will be replaced first with item1, and then with item2, and so forth until N repeats are accomplished. Once a named repeat specification has been introduced, the same repeat rule may be used in the current block by referring only to the name (i.e. <rule1>).
Short repeat rule

A short repeat rule looks like <item1, item2, item3, ..., itemN>. The rule specifies that the entire expression, ‘<...>’ should be replaced first with item1, and then with item2, and so forth until N repeats are accomplished.

Pre-defined names

The following predefined named repeat rules are available:

- `<prefix=s,d,c,z>`
- `<c=s,d,c,z>`
- `<_t=real, double precision, complex, double complex>`
- `<ftype=real, double precision, complex, double complex>`
- `<ctype=float, double, complex_float, complex_double>`
- `<ftypereal=float, double precision, \0, \1>`
- `<ctypereal=float, double, \0, \1>`

4.3.2 Other files

Non-Fortran files use a separate syntax for defining template blocks that should be repeated using a variable expansion similar to the named repeat rules of the Fortran-specific repeats. The template rules for these files are:

1. “/*begin repeat “on a line by itself marks the beginning of a segment that should be repeated.

2. Named variable expansions are defined using #name=item1, item2, item3, ..., itemN# and placed on successive lines. These variables are replaced in each repeat block with corresponding word. All named variables in the same repeat block must define the same number of words.

3. In specifying the repeat rule for a named variable, item*N is short-hand for item, item, ..., item repeated N times. In addition, parenthesis in combination with *N can be used for grouping several items that should be repeated. Thus, #name=(item1, item2)*4# is equivalent to #name=item1, item2, item1, item2, item1, item2, item1, item2#

4. “*/ “on a line by itself marks the end of the the variable expansion naming. The next line is the first line that will be repeated using the named rules.

5. Inside the block to be repeated, the variables that should be expanded are specified as @name@.

6. “/**end repeat**/ “on a line by itself marks the previous line as the last line of the block to be repeated.
NumPy provides a C-API to enable users to extend the system and get access to the array object for use in other routines. The best way to truly understand the C-API is to read the source code. If you are unfamiliar with (C) source code, however, this can be a daunting experience at first. Be assured that the task becomes easier with practice, and you may be surprised at how simple the C-code can be to understand. Even if you don’t think you can write C-code from scratch, it is much easier to understand and modify already-written source code than create it de novo.

Python extensions are especially straightforward to understand because they all have a very similar structure. Admittedly, NumPy is not a trivial extension to Python, and may take a little more snooping to grasp. This is especially true because of the code-generation techniques, which simplify maintenance of very similar code, but can make the code a little less readable to beginners. Still, with a little persistence, the code can be opened to your understanding. It is my hope, that this guide to the C-API can assist in the process of becoming familiar with the compiled-level work that can be done with NumPy in order to squeeze that last bit of necessary speed out of your code.

5.1 Python Types and C-Structures

Several new types are defined in the C-code. Most of these are accessible from Python, but a few are not exposed due to their limited use. Every new Python type has an associated PyObject * with an internal structure that includes a pointer to a “method table” that defines how the new object behaves in Python. When you receive a Python object into C code, you always get a pointer to a PyObject structure. Because a PyObject structure is very generic and defines only PyObject_HEAD, by itself it is not very interesting. However, different objects contain more details after the PyObject_HEAD (but you have to cast to the correct type to access them — or use accessor functions or macros).

5.1.1 New Python Types Defined

Python types are the functional equivalent in C of classes in Python. By constructing a new Python type you make available a new object for Python. The ndarray object is an example of a new type defined in C. New types are defined in C by two basic steps:
1. creating a C-structure (usually named Py{Name}Object) that is binary-compatible with the PyObject structure itself but holds the additional information needed for that particular object;

2. populating the PyTypeObject table (pointed to by the ob_type member of the PyObject structure) with pointers to functions that implement the desired behavior for the type.

Instead of special method names which define behavior for Python classes, there are “function tables” which point to functions that implement the desired results. Since Python 2.2, the PyTypeObject itself has become dynamic which allows C types that can be “sub-typed” from other C-types in C, and sub-classed in Python. The children types inherit the attributes and methods from their parent(s).

There are two major new types: the ndarray (PyArray_Type) and the ufunc (PyUFunc_Type). Additional types play a supportive role: the PyArrayIter_Type, the PyArrayMultiIter_Type, and the PyArrayDescr_Type. The PyArrayIter_Type is the type for a flat iterator for an ndarray (the object that is returned when getting the flat attribute). The PyArrayMultiIter_Type is the type of the object returned when calling broadcast(). It handles iteration and broadcasting over a collection of nested sequences. Also, the PyArrayDescr_Type is the data-type-descriptor type whose instances describe the data. Finally, there are 21 new scalar-array types which are new Python scalars corresponding to each of the fundamental data types available for arrays. An additional 10 other types are place holders that allow the array scalars to fit into a hierarchy of actual Python types.

**PyArray_Type**

The Python type of the ndarray is PyArray_Type. In C, every ndarray is a pointer to a PyArrayObject structure. The ob_type member of this structure contains a pointer to the PyArray_Type typeobject.

**PyArrayObject**

The PyArrayObject C-structure contains all of the required information for an array. All instances of an ndarray (and its subclasses) will have this structure. For future compatibility, these structure members should normally be accessed using the provided macros. If you need a shorter name, then you can make use of NPY_AO which is defined to be equivalent to PyArrayObject.

```c
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;
```

**PyArrayObject.PyObject_HEAD**

This is needed by all Python objects. It consists of (at least) a reference count member (ob_refcnt) and a pointer to the typeobject (ob_type). (Other elements may also be present if Python was compiled with special options see Include/object.h in the Python source tree for more information). The ob_type member points to a Python type object.

char *PyArrayObject.data

A pointer to the first element of the array. This pointer can (and normally should) be recast to the data type of the array.

int PyArrayObject.nd

An integer providing the number of dimensions for this array. When nd is 0, the array is sometimes called a
rank-0 array. Such arrays have undefined dimensions and strides and cannot be accessed. NPY_MAXDIMS is the largest number of dimensions for any array.

np array object dimensions
An array of integers providing the shape in each dimension as long as nd ≥ 1. The integer is always large enough to hold a pointer on the platform, so the dimension size is only limited by memory.

np array object strides
An array of integers providing for each dimension the number of bytes that must be skipped to get to the next element in that dimension.

PyObject *py array object base
This member is used to hold a pointer to another Python object that is related to this array. There are two use cases: 1) If this array does not own its own memory, then base points to the Python object that owns it (perhaps another array object), 2) If this array has the NPY_ARRAY_UPDATEIFCOPY flag set, then this array is a working copy of a “misbehaved” array. As soon as this array is deleted, the array pointed to by base will be updated with the contents of this array.

PyArray_Descr *py array object descr
A pointer to a data-type descriptor object (see below). The data-type descriptor object is an instance of a new built-in type which allows a generic description of memory. There is a descriptor structure for each data type supported. This descriptor structure contains useful information about the type as well as a pointer to a table of function pointers to implement specific functionality.

int py array object flags
Flags indicating how the memory pointed to by data is to be interpreted. Possible flags are NPY_ARRAY_C_CONTIGUOUS, NPY_ARRAY_F_CONTIGUOUS, NPY_ARRAY_OWNDATA, NPY_ARRAY_ALIGNED, NPY_ARRAY_WRITEABLE, and NPY_ARRAY_UPDATEIFCOPY.

PyObject *py array object weakreflist
This member allows array objects to have weak references (using the weakref module).

PyArrayDescr_Type

PyArrayDescr_Type
The PyArrayDescr_Type is the built-in type of the data-type-descriptor objects used to describe how the bytes comprising the array are to be interpreted. There are 21 statically-defined PyArray_Descr objects for the built-in data-types. While these participate in reference counting, their reference count should never reach zero. There is also a dynamic table of user-defined PyArray_Descr objects that is also maintained. Once a data-type-descriptor object is “registered” it should never be deallocated either. The function PyArray_DescrFromType (...) can be used to retrieve a PyArray_Descr object from an enumerated type-number (either built-in or user-defined).

PyArray_Descr
The format of the PyArray_Descr structure that lies at the heart of the PyArrayDescr_Type is

typedef struct {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char unused;
    int flags;
    int type_num;
    int elsize;
    int alignment;
    PyArray_ArrayDescr *subarray;
} PyArray_Descr;
PyObject *fields;
PyArray_ArrFuncs *f;
} PyArray_Descr;

PyTypeObject *PyArray_Descr.typeobj

Pointer to a typeobject that is the corresponding Python type for the elements of this array. For the builtin types, this points to the corresponding array scalar. For user-defined types, this should point to a user-defined typeobject. This typeobject can either inherit from array scalars or not. If it does not inherit from array scalars, then the NPY_USE_GETITEM and NPY_USE_SETITEM flags should be set in the flags member.

char PyArray_Descr.kind

A character code indicating the kind of array (using the array interface typestring notation). A ‘b’ represents Boolean, a ‘i’ represents signed integer, a ‘u’ represents unsigned integer, ‘f’ represents floating point, ‘c’ represents complex floating point, ‘S’ represents 8-bit character string, ‘U’ represents 32-bit/character unicode string, and ‘V’ represents arbitrary.

char PyArray_Descr.type

A traditional character code indicating the data type.

char PyArray_Descr.byteorder

A character indicating the byte-order: ‘>’ (big-endian), ‘<’ (little-endian), ‘=’ (native), ‘!’ (irrelevant, ignore). All builtin data-types have byteorder ‘=’.

int PyArray_Descr.flags

A data-type bit-flag that determines if the data-type exhibits object-array like behavior. Each bit in this member is a flag which are named as:

NPY_ITEM_REFCOUNT

NPY_ITEM_HASOBJECT

Indicates that items of this data-type must be reference counted (using Py_INCREF and Py_DECREF).

NPY_ITEM_LISTPICKLE

Indicates arrays of this data-type must be converted to a list before pickling.

NPY_ITEM_IS_POINTER

Indicates the item is a pointer to some other data-type

NPY_NEEDS_INIT

Indicates memory for this data-type must be initialized (set to 0) on creation.

NPY_NEEDS_PYAPI

Indicates this data-type requires the Python C-API during access (so don’t give up the GIL if array access is going to be needed).

NPY_USE_GETITEM

On array access use the f->getitem function pointer instead of the standard conversion to an array scalar. Must use if you don’t define an array scalar to go along with the data-type.

NPY_USE_SETITEM

When creating a 0-d array from an array scalar use f->setitem instead of the standard copy from an array scalar. Must use if you don’t define an array scalar to go along with the data-type.

NPY_FROM_FIELDS

The bits that are inherited for the parent data-type if these bits are set in any field of the data-type. Currently (NPY_NEEDS_INIT | NPY_LIST_PICKLE | NPY_ITEM_REFCOUNT | NPY_NEEDS_PYAPI).
NPY_OBJECT_DTYPE_FLAGS
  Bits set for the object data-type: (NPY_LIST_PICKLE | NPY_USE_GETITEM | NPY_ITEM_IS_POINTER | NPY_REFCOUNT | NPY_NEEDS_INIT | NPY_NEEDS_PYAPI).

PyDataType_FLAGCHK(PyArray_Descr *dtype, int flags)
  Return true if all the given flags are set for the data-type object.

PyDataType_REFCHK(PyArray_Descr *dtype)
  Equivalent to PyDataType_FLAGCHK(dtype, NPY_ITEM_REFCOUNT).

int PyArray_Descr.type_num
  A number that uniquely identifies the data type. For new data-types, this number is assigned when the data-type is registered.

int PyArray_Descr.elsize
  For data types that are always the same size (such as long), this holds the size of the data type. For flexible data types where different arrays can have a different elementsize, this should be 0.

int PyArray_Descr.alignment
  A number providing alignment information for this data type. Specifically, it shows how far from the start of a 2-element structure (whose first element is a char), the compiler places an item of this type: 
  offsetof(struct {char c; type v;}, v)

PyArray_ArrayDescr *PyArray_Descr.subarray
  If this is non-NULL, then this data-type descriptor is a C-style contiguous array of another data-type descriptor. In other-words, each element that this descriptor describes is actually an array of some other base descriptor. This is most useful as the data-type descriptor for a field in another data-type descriptor. The fields member should be NULL if this is non-NULL (the fields member of the base descriptor can be non-NULL however).

The PyArray_ArrayDescr structure is defined using
  typedef struct {
    PyArray_Descr *base;
    PyObject *shape;
  } PyArray_ArrayDescr;

The elements of this structure are:

PyArray_Descr *PyArray_ArrayDescr.base
  The data-type-descriptor object of the base-type.

PyObject *PyArray_ArrayDescr.shape
  The shape (always C-style contiguous) of the sub-array as a Python tuple.

PyObject *PyArray_Descr.fields
  If this is non-NULL, then this data-type descriptor has fields described by a Python dictionary whose keys are names (and also titles if given) and whose values are tuples that describe the fields. Recall that a data-type-descriptor always describes a fixed-length set of bytes. A field is a named sub-region of that total, fixed-length collection. A field is described by a tuple composed of another data-type descriptor and a byte offset. Optionally, the tuple may contain a title which is normally a Python string. These tuples are placed in this dictionary keyed by name (and also title if given).

PyArray_ArrFuncs *PyArray_Descr.f
  A pointer to a structure containing functions that the type needs to implement internal features. These functions are not the same thing as the universal functions (ufuncs) described later. Their signatures can vary arbitrarily.

PyArray_ArrFuncs
  Functions implementing internal features. Not all of these function pointers must be defined for a given type. The required members are nonzero, copyswap, copyswapn, setitem, getitem, and cast. These are assumed to be non-NULL and NULL entries will cause a program crash. The other functions may be NULL.
which will just mean reduced functionality for that data-type. (Also, the nonzero function will be filled in with a default function if it is NULL when you register a user-defined data-type).

```c
typedef struct {
    PyArray_VectorUnaryFunc *cast[NPY_NTYPES];
    PyArray_GetItemFunc *getitem;
    PyArray_SetItemFunc *setitem;
    PyArray_CopySwapNFunc *copyswapn;
    PyArray_CopySwapFunc *copyswap;
    PyArray_CompareFunc *compare;
    PyArray_ArgFunc *argmax;
    PyArray_DotFunc *dotfunc;
    PyArray_ScanFunc *scanfunc;
    PyArray_FromStrFunc *fromstr;
    PyArray_NonzeroFunc * nonzero;
    PyArray_FillFunc *fill;
    PyArray_FillWithScalarFunc *fillwithscalar;
    PyArray_SortFunc *sort[NPY_NSORTS];
    PyArray_ArgSortFunc *argsort[NPY_NSORTS];
    PyObject *castdict;
    PyArray_ScalarKindFunc *scalarkind;
    int **cancastscalarkindto;
    int *cancastto;
    int listpickle
} PyArray_ArrFuncs;
```

The concept of a behaved segment is used in the description of the function pointers. A behaved segment is one that is aligned and in native machine byte-order for the data-type. The nonzero, copyswap, copyswapn, getitem, and setitem functions can (and must) deal with mis-behaved arrays. The other functions require behaved memory segments.

```c
void cast(void *from, void *to, npy_intp n, void *fromarr,
                  void *toarr)

void *toarr)

An array of function pointers to cast from the current type to all of the other built-in types. Each function casts a contiguous, aligned, and not swapped buffer pointed at by from to a contiguous, aligned, and not swapped buffer pointed at by to. The number of items to cast is given by n, and the arguments fromarr and toarr are interpreted as PyArrayObjects for flexible arrays to get itemsize information.

PyObject *getitem(void *data, void *arr)

A pointer to a function that returns a standard Python object from a single element of the array object arr pointed to by data. This function must be able to deal with “misbehaved” (misaligned and/or swapped) arrays correctly.

```c
int setitem(PyObject *item, void *data, void *arr)
```

A pointer to a function that sets the Python object item into the array, arr, at the position pointed to by data. This function deals with “misbehaved” arrays. If successful, a zero is returned, otherwise, a negative one is returned (and a Python error set).

```c
void copyswapn(void *dest, npy_intp dstride, void *src,
                  npy_intp sstride, npy_intp n, int swap, void *arr)
```

void copyswap(void *dest, void *src, int swap, void *arr)

These members are both pointers to functions to copy data from src to dest and swap if indicated. The value of arr is only used for flexible (NPY_STRING, NPY_UNICODE, and NPY_VOID) arrays (and is obtained from arr->descr->elsize). The second function copies a single value, while the first loops over n values with the provided strides. These functions can deal with misbehaved src data. If src is
NULL then no copy is performed. If swap is 0, then no byteswapping occurs. It is assumed that dest and src do not overlap. If they overlap, then use memmove(...) first followed by copyswap(n) with NULL valued src.

```c
int compare (const void* d1, const void* d2, void* arr)
A pointer to a function that compares two elements of the array, arr, pointed to by d1 and d2. This function requires behaved arrays. The return value is 1 if *d1 > *d2, 0 if *d1 == *d2, and -1 if *d1 < *d2. The array object arr is used to retrieve itemsize and field information for flexible arrays.
```

```c
int argmax(void* data, npy_intp n, npy_intp* max_ind,
void* arr)
A pointer to a function that retrieves the index of the largest of n elements in arr beginning at the element pointed to by data. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the largest element is returned in max_ind.
```

```c
void dotfunc(void* ip1, npy_intp is1, void* ip2, npy_intp is2,
void* op, npy_intp n, void* arr)
A pointer to a function that multiplies two n-length sequences together, adds them, and places the result in element pointed to by op of arr. The start of the two sequences are pointed to by ip1 and ip2. To get to the next element in each sequence requires a jump of is1 and is2 bytes, respectively. This function requires behaved (though not necessarily contiguous) memory.
```

```c
int scanfunc(FILE* fd, void* ip, void* sep, void* arr)
A pointer to a function that scans (scanf style) one element of the corresponding type from the file descriptor fd into the array memory pointed to by ip. The array is assumed to be behaved. If sep is not NULL, then a separator string is also scanned from the file before returning. The last argument arr is the array to be scanned into. A 0 is returned if the scan is successful. A negative number indicates something went wrong: -1 means the end of file was reached before the separator string could be scanned, -4 means that the end of file was reached before the element could be scanned, and -3 means that the element could not be interpreted from the format string. Requires a behaved array.
```

```c
int fromstr(char* str, void* ip, char** endptr, void* arr)
A pointer to a function that converts the string pointed to by str to one element of the corresponding type and places it in the memory location pointed to by ip. After the conversion is completed, *endptr points to the rest of the string. The last argument arr is the array into which ip points (needed for variable-size data-types). Returns 0 on success or -1 on failure. Requires a behaved array.
```

```c
Bool nonzero(void* data, void* arr)
A pointer to a function that returns TRUE if the item of arr pointed to by data is nonzero. This function can deal with misbehaved arrays.
```

```c
void fill(void* data, npy_intp length, void* arr)
A pointer to a function that fills a contiguous array of given length with data. The first two elements of the array must already be filled-in. From these two values, a delta will be computed and the values from item 3 to the end will be computed by repeatedly adding this computed delta. The data buffer must be well-behaved.
```

```c
void fillwithscalar(void* buffer, npy_intp length,
void* value, void* arr)
A pointer to a function that fills a contiguous buffer of the given length with a single scalar value whose address is given. The final argument is the array which is needed to get the itemsize for variable-length arrays.
```

```c
int sort(void* start, npy_intp length, void* arr)
An array of function pointers to a particular sorting algorithms. A particular sorting algorithm is obtained
using a key (so far NPY_QUICKSORT, :data`NPY_HEAPSORT`, and NPY_MERGESORT are defined).
These sorts are done in-place assuming contiguous and aligned data.

```c
int argsort(void* start, npy_intp* result, npy_intp length,
             void* arr)
```

An array of function pointers to sorting algorithms for this data type. The same sorting algorithms as for
sort are available. The indices producing the sort are returned in result (which must be initialized with
indices 0 to length-1 inclusive).

```c
PyObject* castdict
```

Either NULL or a dictionary containing low-level casting functions for user-defined data-types. Each
function is wrapped in a PyCObject * and keyed by the data-type number.

```c
NPY_SCALARKIND scalarkind(PyArrayObject* arr)
```

A function to determine how scalars of this type should be interpreted. The argument is NULL or a
0-dimensional array containing the data (if that is needed to determine the kind of scalar). The return
value must be of type NPY_SCALARKIND.

```c
int **cancastscalarkindto
```

Either NULL or an array of NPY_NSCALARKINDS pointers. These pointers should each be either NULL
or a pointer to an array of integers (terminated by NPY_NOTYPE) indicating data-types that a scalar of
this data-type of the specified kind can be cast to safely (this usually means without losing precision).

```c
int *cancastto
```

Either NULL or an array of integers (terminated by NPY_NOTYPE) indicated data-types that this data-type
can be cast to safely (this usually means without losing precision).

```c
int listpickle
```

Unused.

The PyArray_Type typeobject implements many of the features of Python objects including the tp_as_number,
 tp_as_sequence, tp_as_mapping, and tp_as_buffer interfaces. The rich comparison (tp_richcompare) is also used
along with new-style attribute lookup for methods (tp_methods) and properties (tp_getset). The PyArray_Type can
also be sub-typed.

Tip: The tp_as_number methods use a generic approach to call whatever function has been registered for handling
the operation. The function PyNumeric_SetOps(..) can be used to register functions to handle particular mathematical
operations (for all arrays). When the umath module is imported, it sets the numeric operations for all arrays to the
corresponding ufuncs. The tp_str and tp_repr methods can also be altered using PyString_SetStringFunction(...).

**PyUFunc_Type**

**PyUFunc_Type**

The ufunc object is implemented by creation of the PyUFunc_Type. It is a very simple type that implements
only basic getattribute behavior, printing behavior, and has call behavior which allows these objects to act
like functions. The basic idea behind the ufunc is to hold a reference to fast 1-dimensional (vector) loops for
each data type that supports the operation. These one-dimensional loops all have the same signature and are
the key to creating a new ufunc. They are called by the generic looping code as appropriate to implement the
N-dimensional function. There are also some generic 1-d loops defined for floating and complex floating arrays
that allow you to define a ufunc using a single scalar function (e.g. atanh).

**PyUFuncObject**

The core of the ufunc is the PyUFuncObject which contains all the information needed to call the underlying
C-code loops that perform the actual work. It has the following structure:
typedef struct {
    PyObject_HEAD
    int nin;
    int nout;
    int nargs;
    int identity;
    PyUFuncGenericFunction *functions;
    void **data;
    int ntypes;
    int check_return;
    char *name;
    char *types;
    char *doc;
    void *ptr;
    PyObject *obj;
    PyObject *userloops;
    npy_uint32 *op_flags;
    npy_uint32 *iter_flags;
} PyUFuncObject;

PyUFuncObject(PyObject_HEAD
required for all Python objects.

int PyUFuncObject.nin
    The number of input arguments.

int PyUFuncObject.nout
    The number of output arguments.

int PyUFuncObject.nargs
    The total number of arguments (nin + nout). This must be less than NPY_MAXARGS.

int PyUFuncObject.identity
    Either PyUFunc_One, PyUFunc_Zero, or PyUFunc_None to indicate the identity for this operation.
    It is only used for a reduce-like call on an empty array.

void PyUFuncObject.functions(char** args, npy_intp* dims,
    npy_intp* steps, void* extradata)
    An array of function pointers — one for each data type supported by the ufunc. This is the vector loop
    that is called to implement the underlying function dims [0] times. The first argument, args, is an array of
    nargs pointers to behaved memory. Pointers to the data for the input arguments are first, followed by the
    pointers to the data for the output arguments. How many bytes must be skipped to get to the next element
    in the sequence is specified by the corresponding entry in the steps array. The last argument allows the
    loop to receive extra information. This is commonly used so that a single, generic vector loop can be used
    for multiple functions. In this case, the actual scalar function to call is passed in as extradata. The size of
    this function pointer array is ntypes.

void **PyUFuncObject.data
    Extra data to be passed to the 1-d vector loops or NULL if no extra-data is needed. This C-array must be
    the same size (i.e. ntypes) as the functions array. NULL is used if extra_data is not needed. Several C-API
    calls for UFuncs are just 1-d vector loops that make use of this extra data to receive a pointer to the actual
    function to call.

int PyUFuncObject.ntypes
    The number of supported data types for the ufunc. This number specifies how many different 1-d loops
    (of the builtin data types) are available.

int PyUFuncObject.check_return

5.1. Python Types and C-Structures
Obsolete and unused. However, it is set by the corresponding entry in the main ufunc creation routine: PyUFunc_FromFuncAndData(...).

char *PyUFuncObject.name
A string name for the ufunc. This is used dynamically to build the __doc__ attribute of ufuncs.

char *PyUFuncObject.types
An array of nargs × ntypes 8-bit type_numbers which contains the type signature for the function for each of the supported (builtin) data types. For each of the ntypes functions, the corresponding set of type numbers in this array shows how the args argument should be interpreted in the 1-d vector loop. These type numbers do not have to be the same type and mixed-type ufuncs are supported.

char *PyUFuncObject.doc
Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when __doc__ is retrieved.

void *PyUFuncObject.ptr
Any dynamically allocated memory. Currently, this is used for dynamic ufuncs created from a python function to store room for the types, data, and name members.

PyObject *PyUFuncObject.obj
For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.

PyObject *PyUFuncObject.userloops
A dictionary of user-defined 1-d vector loops (stored as CObject ptrs) for user-defined types. A loop may be registered by the user for any user-defined type. It is retrieved by type number. User defined type numbers are always larger than NPY_USERDEF.

npy_uint32 PyUFuncObject.op_flags
Override the default operand flags for each ufunc operand.

npy_uint32 PyUFuncObject.iter_flags
Override the default nditer flags for the ufunc.

PyArrayIter_Type

This is an iterator object that makes it easy to loop over an N-dimensional array. It is the object returned from the flat attribute of an ndarray. It is also used extensively throughout the implementation internals to loop over an N-dimensional array. The tp_as_mapping interface is implemented so that the iterator object can be indexed (using 1-d indexing), and a few methods are implemented through the tp_methods table. This object implements the next method and can be used anywhere an iterator can be used in Python.

PyArrayIterObject

The C-structure corresponding to an object of PyArrayIter_Type is the PyArrayIterObject. The PyArrayIterObject is used to keep track of a pointer into an N-dimensional array. It contains associated information used to quickly march through the array. The pointer can be adjusted in three basic ways: 1) advance to the “next” position in the array in a C-style contiguous fashion, 2) advance to an arbitrary N-dimensional coordinate in the array, and 3) advance to an arbitrary one-dimensional index into the array. The members of the PyArrayIterObject structure are used in these calculations. Iterator objects keep their own dimension and strides information about an array. This can be adjusted as needed for “broadcasting,” or to loop over only specific dimensions.

typedef struct {
    PyObject_HEAD
    int nd_m1;
    npy_intp index;
} PyArrayIterObject;
npy_intp size;
npy_intp coordinates[NPY_MAXDIMS];
npy_intp dims_m1[NPY_MAXDIMS];
npy_intp strides[NPY_MAXDIMS];
npy_intp backstrides[NPY_MAXDIMS];
npy_intp factors[NPY_MAXDIMS];
PyArrayObject *ao;
char *dataptr;
Bool contiguous;
}

PyArrayIterObject;
each iterator represents the broadcasted shape and size, but has its strides adjusted so that the correct element from the array is used at each iteration.

**PyArrayMultiIterObject**

```c
typedef struct {
    PyObject_HEAD
    int numiter;
    npy_intp size;
    npy_intp index;
    int nd;
    npy_intp dimensions[NPY_MAXDIMS];
    PyArrayIterObject *iters[NPY_MAXDIMS];
} PyArrayMultiIterObject;
```

**PyArrayMultiIterObject.PyObject_HEAD**
Needed at the start of every Python object (holds reference count and type identification).

**int PyArrayMultiIterObject.numiter**
The number of arrays that need to be broadcast to the same shape.

**npy_intp PyArrayMultiIterObject.size**
The total broadcasted size.

**npy_intp PyArrayMultiIterObject.index**
The current (1-d) index into the broadcasted result.

**int PyArrayMultiIterObject.nd**
The number of dimensions in the broadcasted result.

**npy_intp *PyArrayMultiIterObject.dimensions**
The shape of the broadcasted result (only nd slots are used).

**PyArrayIterObject **PyArrayMultiIterObject.iters**
An array of iterator objects that holds the iterators for the arrays to be broadcast together. On return, the iterators are adjusted for broadcasting.

**PyArrayNeighborhoodIter_Type**

**PyArrayNeighborhoodIter_Type**
This is an iterator object that makes it easy to loop over an N-dimensional neighborhood.

**PyArrayNeighborhoodIterObject**
The C-structure corresponding to an object of PyArrayNeighborhoodIter_Type is the PyArrayNeighborhoodIterObject.

**PyArrayFlags_Type**

**PyArrayFlags_Type**
When the flags attribute is retrieved from Python, a special builtin object of this type is constructed. This special type makes it easier to work with the different flags by accessing them as attributes or by accessing them as if the object were a dictionary with the flag names as entries.
ScalarArrayTypes

There is a Python type for each of the different built-in data types that can be present in the array. Most of these are simple wrappers around the corresponding data type in C. The C-names for these types are `Py{TYPE}ArrType_Type` where `{TYPE}` can be

- `Bool`
- `Byte`
- `Short`
- `Int`
- `Long`
- `LongLong`
- `UByte`
- `UShort`
- `UInt`
- `ULong`
- `ULongLong`
- `Half`
- `Float`
- `Double`
- `LongDouble`
- `CFloat`
- `CDouble`
- `CLongDouble`
- `String`
- `Unicode`
- `Void`
- `Object`

These type names are part of the C-API and can therefore be created in extension C-code. There is also a `PyIntpArrType_Type` and a `PyUIntpArrType_Type` that are simple substitutes for one of the integer types that can hold a pointer on the platform. The structure of these scalar objects is not exposed to C-code. The function `PyArray_ScalarAsCtype (...)` can be used to extract the C-type value from the array scalar and the function `PyArray_Scalar (...)` can be used to construct an array scalar from a C-value.

5.1.2 Other C-Structures

A few new C-structures were found to be useful in the development of NumPy. These C-structures are used in at least one C-API call and are therefore documented here. The main reason these structures were defined is to make it easy to use the Python ParseTuple C-API to convert from Python objects to a useful C-Object.

`PyArray_Dims`

This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is:

```c
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```

The members of this structure are

- `npy_intp *PyArray_Dims.ptr` A pointer to a list of (npy_intp) integers which usually represent array shape or array strides.
- `int PyArray_Dims.len` The length of the list of integers. It is assumed safe to access `ptr [0]` to `ptr [len-1].`

`PyArray_Chunk`

This is equivalent to the buffer object structure in Python up to the `ptr` member. On 32-bit platforms (i.e. if `NPY_SIZEOF_INT == NPY_SIZEOF_INTP`) or in Python 2.5, the `len` member also matches an equivalent member of the buffer object. It is useful to represent a generic single-segment chunk of memory.

```c
typedef struct {
    PyObject_HEAD
    PyObject *base;
    void *ptr;
    npy_intp len;
    int flags;
} PyArray_Chunk;
```

The members are
NumPy Reference, Release 1.8.1

PyArray_Chunk.PyObject_HEAD
Necessary for all Python objects. Included here so that the PyArray_Chunk structure matches that of the buffer object (at least to the len member).

PyObject *PyArray_Chunk.base
The Python object this chunk of memory comes from. Needed so that memory can be accounted for properly.

void *PyArray_Chunk.ptr
A pointer to the start of the single-segment chunk of memory.

npy_intp PyArray_Chunk.len
The length of the segment in bytes.

int PyArray_Chunk.flags
Any data flags (e.g. NPY_ARRAY_WRITEABLE) that should be used to interpret the memory.

PyArrayInterface

See Also:
The Array Interface

PyArrayInterface
The PyArrayInterface structure is defined so that NumPy and other extension modules can use the rapid array interface protocol. The __array_struct__ method of an object that supports the rapid array interface protocol should return a PyCObject that contains a pointer to a PyArrayInterface structure with the relevant details of the array. After the new array is created, the attribute should be DECREP’d which will free the PyArrayInterface structure. Remember to INCREF the object (whose __array_struct__ attribute was retrieved) and point the base member of the new PyArrayObject to this same object. In this way the memory for the array will be managed correctly.

typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    npy_intp *shape;
    npy_intp *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;

int PyArrayInterface.two
the integer 2 as a sanity check.

int PyArrayInterface.nd
the number of dimensions in the array.

char PyArrayInterface.typekind

int PyArrayInterface.itemsize
The number of bytes each item in the array requires.

int PyArrayInterface.flags
Any of the bits NPY_ARRAY_C_CONTIGUOUS (1), NPY_ARRAY_F_CONTIGUOUS (2),
NPY_ARRAY_aligned (0x100), NPY_ARRAY_NOTSWAPPED (0x200), or NPY_ARRAY_WRITEABLE (0x400) to indicate something about the data. The NPY_ARRAY_ALIGNED, NPY_ARRAY_C_CONTIGUOUS, and NPY_ARRAY_F_CONTIGUOUS flags can actually be determined from the other parameters. The flag NPY ARR_HAS_DESCR (0x800) can also be set to indicate to objects consuming the version 3 array interface that the descr member of the structure is present (it will be ignored by objects consuming version 2 of the array interface).

npy_intp *PyArrayInterface.shape
An array containing the size of the array in each dimension.

npy_intp *PyArrayInterface.strides
An array containing the number of bytes to jump to get to the next element in each dimension.

void *PyArrayInterface.data
A pointer to the first element of the array.

PyObject *PyArrayInterface.descr
A Python object describing the data-type in more detail (same as the descr key in __array_interface__). This can be NULL if typekind and itemsize provide enough information. This field is also ignored unless ARR_HAS_DESCR flag is on in flags.

Internally used structures

Internally, the code uses some additional Python objects primarily for memory management. These types are not accessible directly from Python, and are not exposed to the C-API. They are included here only for completeness and assistance in understanding the code.

PyUFuncLoopObject
A loose wrapper for a C-structure that contains the information needed for looping. This is useful if you are trying to understand the ufunc looping code. The PyUFuncLoopObject is the associated C-structure. It is defined in the ufuncobject.h header.

PyUFuncReduceObject
A loose wrapper for the C-structure that contains the information needed for reduce-like methods of ufuncs. This is useful if you are trying to understand the reduce, accumulate, and reduce-at code. The PyUFuncReduceObject is the associated C-structure. It is defined in the ufuncobject.h header.

PyUFunc_Loop1d
A simple linked-list of C-structures containing the information needed to define a 1-d loop for a ufunc for every defined signature of a user-defined data-type.

PyArrayMapIter_type
Advanced indexing is handled with this Python type. It is simply a loose wrapper around the C-structure containing the variables needed for advanced array indexing. The associated C-structure, PyArrayMap_iterObject, is useful if you are trying to understand the advanced-index mapping code. It is defined in the arrayobject.h header. This type is not exposed to Python and could be replaced with a C-structure. As a Python type it takes advantage of reference-counted memory management.

5.2 System configuration

When NumPy is built, information about system configuration is recorded, and is made available for extension modules using NumPy’s C API. These are mostly defined in numpyconfig.h (included in ndarrayobject.h). The public symbols are prefixed by NPY_*. NumPy also offers some functions for querying information about the platform in use.
For private use, Numpy also constructs a `config.h` in the NumPy include directory, which is not exported by Numpy (that is a python extension which use the numpy C API will not see those symbols), to avoid namespace pollution.

### 5.2.1 Data type sizes

The `NPY_SIZEOF_{CTYPE}` constants are defined so that `sizeof` information is available to the pre-processor.

- `NPY_SIZEOF_SHORT`
- `NPY_SIZEOF_INT`
- `NPY_SIZEOF_LONG`
- `NPY_SIZEOF_LONGLONG` (sizeof(longlong) where longlong is defined appropriately on the platform.)
- `NPY_SIZEOF_PY_LONG_LONG`
- `NPY_SIZEOF_FLOAT`
- `NPY_SIZEOF_DOUBLE`
- `NPY_SIZEOF_LONG_DOUBLE`
- `NPY_SIZEOF_PY_INTPTR_T` (Size of a pointer on this platform (sizeof(void *)) (A macro defines NPY_SIZEOF_INTP as well.)

### 5.2.2 Platform information

- `NPY_CPU_X86`
- `NPY_CPU_AMD64`
- `NPY_CPU_IA64`
- `NPY_CPU_PPC`
- `NPY_CPU_PPC64`
- `NPY_CPU_SPARC`
- `NPY_CPU_SPARC64`
- `NPY_CPU_S390`
**NPY_CPU_PARISC**

New in version 1.3.0. CPU architecture of the platform; only one of the above is defined.

Defined in `numpy/npy_cpu.h`

**NPY_LITTLE_ENDIAN**

**NPY_BIG_ENDIAN**

**NPY_BYTE_ORDER**

New in version 1.3.0. Portable alternatives to the `endian.h` macros of GNU Libc. If big endian, `NPY_BYTE_ORDER == NPY_BIG_ENDIAN`, and similarly for little endian architectures.

Defined in `numpy/npy_endian.h`.

**PyArray_GetEndianness()**

New in version 1.3.0. Returns the endianness of the current platform. One of `NPY_CPU_BIG`, `NPY_CPU_LITTLE`, or `NPY_CPU_UNKNOWN_ENDIAN`.

### 5.3 Data Type API

The standard array can have 24 different data types (and has some support for adding your own types). These data types all have an enumerated type, an enumerated type-character, and a corresponding array scalar Python type object (placed in a hierarchy). There are also standard C typedefs to make it easier to manipulate elements of the given data type. For the numeric types, there are also bit-width equivalent C typedefs and named typenumbers that make it easier to select the precision desired.

**Warning:** The names for the types in c code follows c naming conventions more closely. The Python names for these types follow Python conventions. Thus, `NPY_FLOAT` picks up a 32-bit float in C, but `numpy.float_` in Python corresponds to a 64-bit double. The bit-width names can be used in both Python and C for clarity.

### 5.3.1 Enumerated Types

There is a list of enumerated types defined providing the basic 24 data types plus some useful generic names. Whenever the code requires a type number, one of these enumerated types is requested. The types are all called `NPY_{NAME}`:

**NPY_BOOL**

The enumeration value for the boolean type, stored as one byte. It may only be set to the values 0 and 1.

**NPY_BYTE**

**NPY_INT8**

The enumeration value for an 8-bit/1-byte signed integer.

**NPY_SHORT**

**NPY_INT16**

The enumeration value for a 16-bit/2-byte signed integer.

**NPY_INT**
NPY_INT32
The enumeration value for a 32-bit/4-byte signed integer.

NPY_LONG
Equivalent to either NPY_INT or NPY_LONGLONG, depending on the platform.

NPY_LONGLONG

NPY_INT64
The enumeration value for a 64-bit/8-byte signed integer.

NPY_UBYTE

NPY_UINT8
The enumeration value for an 8-bit/1-byte unsigned integer.

NPY_USHORT

NPY_UINT16
The enumeration value for a 16-bit/2-byte unsigned integer.

NPY_UINT

NPY_UINT32
The enumeration value for a 32-bit/4-byte unsigned integer.

NPY_ULONGLONG
Equivalent to either NPY_UINT or NPY_ULONGLONG, depending on the platform.

NPY_ULONGLONG

NPY_UINT64
The enumeration value for a 64-bit/8-byte unsigned integer.

NPY_HALF

NPY_FLOAT16
The enumeration value for a 16-bit/2-byte IEEE 754-2008 compatible floating point type.

NPY_FLOAT

NPY_FLOAT32
The enumeration value for a 32-bit/4-byte IEEE 754 compatible floating point type.

NPY_DOUBLE

NPY_FLOAT64
The enumeration value for a 64-bit/8-byte IEEE 754 compatible floating point type.

NPY_LONGDOUBLE
The enumeration value for a platform-specific floating point type which is at least as large as NPY_DOUBLE, but larger on many platforms.

NPY_CFLOAT
NPY_COMPLEX64
The enumeration value for a 64-bit/8-byte complex type made up of two NPY_FLOAT values.

NPY_CDOUBLE

NPY_COMPLEX128
The enumeration value for a 128-bit/16-byte complex type made up of two NPY_DOUBLE values.

NPY_CLONGDOUBLE
The enumeration value for a platform-specific complex floating point type which is made up of two NPY_LONGDOUBLE values.

NPY_DATETIME
The enumeration value for a data type which holds dates or datetimes with a precision based on selectable date or time units.

NPY_TIMedelta
The enumeration value for a data type which holds lengths of times in integers of selectable date or time units.

NPY_STRING
The enumeration value for ASCII strings of a selectable size. The strings have a fixed maximum size within a given array.

NPY_UNICODE
The enumeration value for UCS4 strings of a selectable size. The strings have a fixed maximum size within a given array.

NPY_OBJECT
The enumeration value for references to arbitrary Python objects.

NPY_VOID
Primarily used to hold struct dtypes, but can contain arbitrary binary data.

Some useful aliases of the above types are

NPY_INTP
The enumeration value for a signed integer type which is the same size as a (void *) pointer. This is the type used by all arrays of indices.

NPY_UINTP
The enumeration value for an unsigned integer type which is the same size as a (void *) pointer.

NPY_MASK
The enumeration value of the type used for masks, such as with the NPY_ITER_ARRAYMASK iterator flag. This is equivalent to NPY_UINT8.

NPY_DEFAULT_TYPE
The default type to use when no dtype is explicitly specified, for example when calling np.zeros(shape). This is equivalent to NPY_DOUBLE.

Other useful related constants are

NPY_NTYPES
The total number of built-in NumPy types. The enumeration covers the range from 0 to NPY_NTYPES-1.

NPY_NOTYPE
A signal value guaranteed not to be a valid type enumeration number.

NPY_USERDEF
The start of type numbers used for Custom Data types.
The various character codes indicating certain types are also part of an enumerated list. References to type characters (should they be needed at all) should always use these enumerations. The form of them is \texttt{NPY_{NAME}LTR} where \texttt{\{NAME\}} can be

\begin{verbatim}
BOOL, BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, LONGLONG, ULONGLONG, HALF, FLOAT, DOUBLE, LONGDOUBLE, CFLOAT, CDOUBLE, CLONGDOUBLE, DATETIME, TIMEDELTA, OBJECT, STRING, VOID
INTP, UINTP
GENBOOL, SIGNED, UNSIGNED, FLOATING, COMPLEX
\end{verbatim}

The latter group of \texttt{\{NAME\}s} corresponds to letters used in the array interface typestring specification.

\subsection*{5.3.2 Defines}

\subsubsection*{Max and min values for integers}

\begin{verbatim}
NPY_MAX_INT\{bits\}
NPY_MAX_UINT\{bits\}
NPY_MIN_INT\{bits\}
These are defined for \texttt{\{bits\}} = 8, 16, 32, 64, 128, and 256 and provide the maximum (minimum) value of the corresponding (unsigned) integer type. Note: the actual integer type may not be available on all platforms (i.e. 128-bit and 256-bit integers are rare).
NPY_MIN_{type}
This is defined for \texttt{\{type\}} = BYTE, SHORT, INT, LONG, LONGLONG, INTP
NPY_MAX_{type}
This is defined for all defined for \texttt{\{type\}} = BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG, LONGLONG, ULONGLONG, INTP, UINTP
\end{verbatim}

\subsubsection*{Number of bits in data types}

All \texttt{NPY_SIZEOF\{CTYPE\}} constants have corresponding \texttt{NPY_BITSOF\{CTYPE\}} constants defined. The \texttt{NPY_BITSOF\{CTYPE\}} constants provide the number of bits in the data type. Specifically, the available \texttt{\{CTYPE\}s} are

\begin{verbatim}
BOOL, CHAR, SHORT, INT, LONG, LONGLONG, FLOAT, DOUBLE, LONGDOUBLE
\end{verbatim}

\subsubsection*{Bit-width references to enumerated typenums}

All of the numeric data types (integer, floating point, and complex) have constants that are defined to be a specific enumerated type number. Exactly which enumerated type a bit-width type refers to is platform dependent. In particular, the constants available are \texttt{PyArray\_{NAME}\{BITS\}} where \texttt{\{NAME\}} is INT, UINT, FLOAT, COMPLEX and \texttt{\{BITS\}} can be 8, 16, 32, 64, 80, 96, 128, 160, 192, 256, and 512. Obviously not all bit-widths are available on all platforms for all the kinds of numeric types. Commonly 8-, 16-, 32-, 64-bit integers; 32-, 64-bit floats; and 64-, 128-bit complex types are available.
Integer that can hold a pointer

The constants NPY_INTP and NPY_UINTP refer to an enumerated integer type that is large enough to hold a pointer on the platform. Index arrays should always be converted to NPY_INTP, because the dimension of the array is of type npy_intp.

5.3.3 C-type names

There are standard variable types for each of the numeric data types and the bool data type. Some of these are already available in the C-specification. You can create variables in extension code with these types.

Boolean

npy_bool
  unsigned char; The constants NPY_FALSE and NPY_TRUE are also defined.

(Un)Signed Integer

Unsigned versions of the integers can be defined by pre-pending a ‘u’ to the front of the integer name.

npy_(u)byte
  (unsigned) char

npy_(u)short
  (unsigned) short

npy_(u)int
  (unsigned) int

npy_(u)long
  (unsigned) long int

npy_(u)longlong
  (unsigned long long int)

npy_(u)intptr
  (unsigned) Py_intptr_t (an integer that is the size of a pointer on the platform).

(Complex) Floating point

npy_(c)float
  float

npy_(c)double
  double

npy_(c)longdouble
  long double

complex types are structures with .real and .imag members (in that order).
Bit-width names

There are also typedefs for signed integers, unsigned integers, floating point, and complex floating point types of specific bit-widths. The available type names are

\[
\text{npy_int\{bits\}, numpy_uint\{bits\}, npy_float\{bits\}, and npy_complex\{bits\}}
\]

where \(\{bits\}\) is the number of bits in the type and can be \(8, 16, 32, 64, 128,\) and \(256\) for integer types; \(16, 32, 64, 80, 96, 128,\) and \(256\) for floating-point types; and \(32, 64, 128, 160, 192,\) and \(512\) for complex-valued types. Which bit-widths are available is platform dependent. The bolded bit-widths are usually available on all platforms.

5.3.4 Printf Formatting

For help in printing, the following strings are defined as the correct format specifier in printf and related commands.

\[
\text{NPY_LONGLONG_FMT, NPY_ULONGLONG_FMT, NPY_INTP_FMT, NPY_UINTP_FMT, NPY_LONGDOUBLE_FMT}
\]

5.4 Array API

The test of a first-rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function.

— F. Scott Fitzgerald

For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.

— Richard P. Feynman

5.4.1 Array structure and data access

These macros all access the \text{PyArrayObject} structure members. The input argument, \(\text{arr}\), can be any \text{PyObject} that is directly interpretable as a \text{PyArrayObject} (any instance of the \text{PyArray_Type} and its sub-types).

\[
\text{int PyArray_NDIM (PyArrayObject *arr)}
\]

The number of dimensions in the array.

\[
\text{npy_intp *PyArray_DIMS (PyArrayObject *arr)}
\]

Returns a pointer to the dimensions/shape of the array. The number of elements matches the number of dimensions of the array.

\[
\text{npy_intp *PyArray_SHAPE (PyArrayObject *arr)}
\]

New in version 1.7. A synonym for \text{PyArray_DIMS}, named to be consistent with the ‘shape’ usage within Python.

\[
\text{void *PyArray_DATA (PyArrayObject *arr)}
\]

\[
\text{char *PyArray_BYTES (PyArrayObject *arr)}
\]

These two macros are similar and obtain the pointer to the data-buffer for the array. The first macro can (and should be) assigned to a particular pointer where the second is for generic processing. If you have not
guaranteed a contiguous and/or aligned array then be sure you understand how to access the data in the array to avoid memory and/or alignment problems.

\texttt{npy\_intp \*PyArray\_STRIDES (PyArrayObject* arr)}

Returns a pointer to the strides of the array. The number of elements matches the number of dimensions of the array.

\texttt{npy\_intp PyArray\_DIM (PyArrayObject* arr, int n)}

Returns the shape in the \textit{n}th dimension.

\texttt{npy\_intp PyArray\_STRIDE (PyArrayObject* arr, int n)}

Returns the stride in the \textit{n}th dimension.

\texttt{PyObject \*PyArray\_BASE (PyArrayObject* arr)}

This returns the base object of the array. In most cases, this means the object which owns the memory the array is pointing at.

If you are constructing an array using the C API, and specifying your own memory, you should use the function \texttt{PyArray\_SetBaseObject} to set the base to an object which owns the memory.

If the \texttt{NPY\_ARRAY\_UPDATEIFCOPY} flag is set, it has a different meaning, namely base is the array into which the current array will be copied upon destruction. This overloading of the base property for two functions is likely to change in a future version of NumPy.

\texttt{PyArray\_Descr \*PyArray\_DESCR (PyArrayObject* arr)}

Returns a borrowed reference to the dtype property of the array.

\texttt{PyArray\_Descr \*PyArray\_DTYPE (PyArrayObject* arr)}

New in version 1.7. A synonym for \texttt{PyArray\_DESCR}, named to be consistent with the ‘dtype’ usage within Python.

\texttt{void PyArray\_ENABLEFLAGS (PyArrayObject* arr, int flags)}

New in version 1.7. Enables the specified array flags. This function does no validation, and assumes that you know what you’re doing.

\texttt{void PyArray\_CLEARFLAGS (PyArrayObject* arr, int flags)}

New in version 1.7. Clears the specified array flags. This function does no validation, and assumes that you know what you’re doing.

\texttt{int PyArray\_FLAGS (PyArrayObject* arr)}

\texttt{int PyArray\_ITEMSIZE (PyArrayObject* arr)}

Return the itemsize for the elements of this array.

\texttt{int PyArray\_TYPE (PyArrayObject* arr)}

Return the (builtin) typenumber for the elements of this array.

\texttt{PyObject \*PyArray\_GETITEM (PyArrayObject* arr, void* itemptr)}

Get a Python object from the ndarray, \textit{arr}, at the location pointed to by itemptr. Return NULL on failure.

\texttt{int PyArray\_SETITEM (PyArrayObject* arr, void* itemptr, PyObject* obj)}

Convert \textit{obj} and place it in the ndarray, \textit{arr}, at the place pointed to by itemptr. Return -1 if an error occurs or 0 on success.

\texttt{npy\_intp PyArray\_SIZE (PyArrayObject* arr)}

Returns the total size (in number of elements) of the array.

\texttt{npy\_intp PyArray\_Size (PyArrayObject* obj)}

Returns 0 if \textit{obj} is not a sub-class of bigndarray. Otherwise, returns the total number of elements in the array. Safer version of \texttt{PyArray\_SIZE (obj)}.

5.4. Array API
npy_intp PyArray_NBYTES (PyArrayObject* arr)

Returns the total number of bytes consumed by the array.

**Data access**

These functions and macros provide easy access to elements of the ndarray from C. These work for all arrays. You may need to take care when accessing the data in the array, however, if it is not in machine byte-order, misaligned, or not writeable. In other words, be sure to respect the state of the flags unless you know what you are doing, or have previously guaranteed an array that is writeable, aligned, and in machine byte-order using PyArray_FromAny. If you wish to handle all types of arrays, the copyswap function for each type is useful for handling misbehaved arrays. Some platforms (e.g. Solaris) do not like misaligned data and will crash if you de-reference a misaligned pointer. Other platforms (e.g. x86 Linux) will just work more slowly with misaligned data.

void* PyArray_GetPtr (PyArrayObject* aobj, npy_intp* ind)

Return a pointer to the data of the ndarray, `aobj`, at the N-dimensional index given by the c-array, `ind`, (which must be at least `aobj`->nd in size). You may want to typecast the returned pointer to the data type of the ndarray.

void* PyArray_GETPTR1 (PyArrayObject* obj, npy_intp i)

void* PyArray_GETPTR2 (PyArrayObject* obj, npy_intp i, npy_intp j)

void* PyArray_GETPTR3 (PyArrayObject* obj, npy_intp i, npy_intp j, npy_intp k)

void* PyArray_GETPTR4 (PyArrayObject* obj, npy_intp i, npy_intp j, npy_intp k, npy_intp l)

Quick, inline access to the element at the given coordinates in the ndarray, `obj`, which must have respectively 1, 2, 3, or 4 dimensions (this is not checked). The corresponding `i`, `j`, `k`, and `l` coordinates can be any integer but will be interpreted as `npy_intp`. You may want to typecast the returned pointer to the data type of the ndarray.

### 5.4.2 Creating arrays

**From scratch**

PyObject* PyArray_NewFromDescr (PyTypeObject* subtype, PyArray_Descr* descr, int nd, npy_intp* dims, npy_intp* strides, void* data, int flags, PyObject* obj)

This function steals a reference to `descr` if it is not NULL. This is the main array creation function. Most new arrays are created with this flexible function.

The returned object is an object of Python-type `subtype`, which must be a subtype of `PyArray_Type`. The array has `nd` dimensions, described by `dims`. The data-type descriptor of the new array is `descr`.

If `subtype` is of an array subclass instead of the base `&PyArray_Type`, then `obj` is the object to pass to the `__array_finalize__` method of the subclass. If `data` is NULL, then new memory will be allocated and `flags` can be non-zero to indicate a Fortran-style contiguous array. If `data` is not NULL, then it is assumed to point to the memory to be used for the array and the `flags` argument is used as the new flags for the array (except the state of `NPY_OWNDATA` and `NPY_ARRAY_UPDATEIFCOPY` flags of the new array will be reset).

In addition, if `data` is non-NULL, then `strides` can also be provided. If `strides` is NULL, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (`flags` is nonzero for `data` = NULL or `flags` & `NPY_ARRAY_F_CONTIGUOUS` is nonzero non-NULL `data`). Any provided `dims` and `strides` are copied into newly allocated dimension and strides arrays for the new array object.
PyObject* PyArray_NewLikeArray (PyArrayObject* prototype, NPY_ORDER order, PyArray_Descr* descr, int subok)
New in version 1.6. This function steals a reference to descr if it is not NULL.

This array creation routine allows for the convenient creation of a new array matching an existing array’s shapes and memory layout, possibly changing the layout and/or data type.

When order is NPY_ANYORDER, the result order is NPY_FORTRANORDER if prototype is a fortran array, NPY_CORDER otherwise. When order is NPY_KEEPORDER, the result order matches that of prototype, even when the axes of prototype aren’t in C or Fortran order.

If descr is NULL, the data type of prototype is used.

If subok is 1, the newly created array will use the sub-type of prototype to create the new array, otherwise it will create a base-class array.

PyObject* PyArray_New (PyTypeObject* subtype, int nd, npy_intp* dims, int type_num, npy_intp* strides, void* data, int itemsize, int flags, PyObject* obj)
This is similar to PyArray_DescrNew (...) except you specify the data-type descriptor with type_num and itemsize, where type_num corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then itemsize is ignored. Otherwise, itemsize specifies the particular size of this array.

**Warning:** If data is passed to PyArray_NewFromDescr or PyArray_New, this memory must not be deallocated until the new array is deleted. If this data came from another Python object, this can be accomplished using Py_INCREF on that object and setting the base member of the new array to point to that object. If strides are passed in they must be consistent with the dimensions, the itemsize, and the data of the array.

PyObject* PyArray_SimpleNew (int nd, npy_intp* dims, int typenum)
Create a new unitialized array of type, typenum, whose size in each of nd dimensions is given by the integer array, dims. This function cannot be used to create a flexible-type array (no itemsize given).

PyObject* PyArray_SimpleNewFromData (int nd, npy_intp* dims, int typenum, void* data)
Create an array wrapper around data pointed to by the given pointer. The array flags will have a default that the data area is well-behaved and C-style contiguous. The shape of the array is given by the dims c-array of length nd. The data-type of the array is indicated by typenum.

PyObject* PyArray_SimpleNewFromDescr (int nd, npy_intp* dims, PyArray_Descr* descr)
This function steals a reference to descr if it is not NULL.

Create a new array with the provided data-type descriptor, descr, of the shape determined by nd and dims.

PyObject_FILLWBYTE (PyObject* obj, int val)
Fill the array pointed to by obj—which must be a (subclass of) bigndarray—with the contents of val (evaluated as a byte). This macro calls memset, so obj must be contiguous.

PyObject* PyArray_Zeros (int nd, npy_intp* dims, PyArray_Descr* dtype, int fortran)
Construct a new nd-dimensional array with shape given by dims and data type given by dtype. If fortran is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. Fill the memory with zeros (or the 0 object if dtype corresponds to NPY_OBJECT).

PyObject* PyArray_ZEROS (int nd, npy_intp* dims, int type_num, int fortran)
Macro form of PyArray_Zeros which takes a type-number instead of a data-type object.

PyObject* PyArray_Empty (int nd, npy_intp* dims, PyArray_Descr* dtype, int fortran)
Construct a new nd-dimensional array with shape given by dims and data type given by dtype. If fortran is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. The array is uninitialized unless the data type corresponds to NPY_OBJECT in which case the array is filled with Py_None.

PyObject* PyArray_EMPTY (int nd, npy_intp* dims, int typenum, int fortran)
Macro form of PyArray_Empty which takes a type-number, typenum, instead of a data-type object.
PyObject\* PyArray_Arange (double start, double stop, double step, int typenum)

Construct a new 1-dimensional array of data-type, typenum, that ranges from start to stop (exclusive) in increments of step. Equivalent to arange (start, stop, step, dtype).

PyObject\* PyArray_ArangeObj (PyObject* start, PyObject* stop, PyObject* step, PyArray_Descr* descr)

Construct a new 1-dimensional array of data-type determined by descr, that ranges from start to stop (exclusive) in increments of step. Equivalent to arange (start, stop, step, typenum).

int PyArray_SetBaseObject (PyArrayObject* arr, PyObject* obj)

New in version 1.7. This function steals a reference to obj and sets it as the base property of arr.

If you construct an array by passing in your own memory buffer as a parameter, you need to set the array’s base property to ensure the lifetime of the memory buffer is appropriate.

The return value is 0 on success, -1 on failure.

If the object provided is an array, this function traverses the chain of base pointers so that each array points to the owner of the memory directly. Once the base is set, it may not be changed to another value.

From other objects

PyObject\* PyArray_FromAny (PyObject* op, PyArray_Descr* dtype, int min_depth, int max_depth, int requirements, PyObject* context)

This is the main function used to obtain an array from any nested sequence, or object that exposes the array interface, op. The parameters allow specification of the required dtype, the minimum (min_depth) and maximum (max_depth) number of dimensions acceptable, and other requirements for the array. The dtype argument needs to be a PyArray_Descr structure indicating the desired data-type (including required byteorder). The dtype argument may be NULL, indicating that any data-type (and byteorder) is acceptable. Unless FORCECAST is present in flags, this call will generate an error if the data type cannot be safely obtained from the object. If you want to use NULL for the dtype and ensure the array is not swapped then use PyArray_CheckFromAny. A value of 0 for either of the depth parameters causes the parameter to be ignored. Any of the following array flags can be added (e.g. using |) to get the requirements argument. If your code can handle general (e.g. strided, byte-swapped, or unaigned arrays) then requirements may be 0. Also, if op is not already an array (or does not expose the array interface), then a new array will be created (and filled from op using the sequence protocol). The new array will have NPY_DEFAULT as its flags member. The context argument is passed to the __array__ method of op and is only used if the array is constructed that way. Almost always this parameter is NULL.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

NPY_ARRAY_C_CONTIGUOUS
Make sure the returned array is C-style contiguous

NPY_ARRAY_F_CONTIGUOUS
Make sure the returned array is Fortran-style contiguous.

NPY_ARRAY_ALIGNED
Make sure the returned array is aligned on proper boundaries for its data type. An aligned array has the data pointer and every strides factor as a multiple of the alignment factor for the data-type descriptor.

NPY_ARRAY_WRITEABLE
Make sure the returned array can be written to.

NPY_ARRAY_ENSURECOPY
Make sure a copy is made of op. If this flag is not present, data is not copied if it can be avoided.

NPY_ARRAY_ENSUREARRAY
Make sure the result is a base-class ndarray or bigndarray. By default, if op is an instance of a subclass of
the bigndarray, an instance of that same subclass is returned. If this flag is set, an ndarray object will be returned instead.

**NPY_ARRAY_FORCECAST**

Force a cast to the output type even if it cannot be done safely. Without this flag, a data cast will occur only if it can be done safely, otherwise an error is raised.

**NPY_ARRAY_UPDATEIFCOPY**

If \( op \) is already an array, but does not satisfy the requirements, then a copy is made (which will satisfy the requirements). If this flag is present and a copy (of an object that is already an array) must be made, then the corresponding \texttt{NPY_ARRAY_UPDATEIFCOPY} flag is set in the returned copy and \( op \) is made to be read-only. When the returned copy is deleted (presumably after your calculations are complete), its contents will be copied back into \( op \) and the \( op \) array will be made writeable again. If \( op \) is not writeable to begin with, then an error is raised. If \( op \) is not already an array, then this flag has no effect.

**NPY_ARRAY_BEHAVED**

\texttt{NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE}

**NPY_ARRAY_CARRAY**

\texttt{NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_BEHAVED}

**NPY_ARRAY_CARRAY_RO**

\texttt{NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_ALIGNED}

**NPY_ARRAY_FARRAY**

\texttt{NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_BEHAVED}

**NPY_ARRAY_FARRAY_RO**

\texttt{NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_ALIGNED}

**NPY_ARRAY_DEFAULT**

\texttt{NPY_ARRAY_CARRAY}

**NPY_ARRAY_IN_ARRAY**

\texttt{NPY_ARRAY_ALIGNED}

**NPY_ARRAY_IN_FARRAY**

\texttt{NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_ALIGNED}

**NPY_OUT_ARRAY**

\texttt{NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_WRITEABLE | NPY_ARRAY_ALIGNED}

**NPY_ARRAY_OUT_FARRAY**

\texttt{NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_WRITEABLE | NPY_ARRAY_ALIGNED}

**NPY_ARRAY_INOUT_ARRAY**

\texttt{NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_WRITEABLE | NPY_ARRAY_ALIGNED | NPY_ARRAY_UPDATEIFCOPY}

**NPY_ARRAY_INOUT_FARRAY**

\texttt{NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_WRITEABLE | NPY_ARRAY_ALIGNED | NPY_ARRAY_UPDATEIFCOPY}

```c
int PyArray_GetArrayParamsFromObject (PyObject* \( op \), PyArray_Descr* \( \text{requested\_dtype} \), npy_bool \( \text{writeable} \), PyArray_Descr** \( \text{out\_dtype} \), int* \( \text{out\_ndim} \), npy_intp* \( \text{out\_dims} \), PyArrayObject** \( \text{out\_arr} \), PyObject* \( \text{context} \))
```

New in version 1.6. Retrieves the array parameters for viewing/converting an arbitrary PyObject* to a NumPy array. This allows the “innate type and shape” of Python list-of-lists to be discovered without actually converting to an array. PyArray_FromAny calls this function to analyze its input.

5.4. Array API 1345
In some cases, such as structured arrays and the __array__ interface, a data type needs to be used to make sense of the object. When this is needed, provide a Descr for 'requested_dtype', otherwise provide NULL. This reference is not stolen. Also, if the requested dtype doesn't modify the interpretation of the input, out_dtype will still get the “innate” dtype of the object, not the dtype passed in ‘requested_dtype’.

If writing to the value in ‘op’ is desired, set the boolean ‘writeable’ to 1. This raises an error when ‘op’ is a scalar, list of lists, or other non-writeable ‘op’. This differs from passing NPY_ARRAY_WRITEABLE to PyArray_FromAny, where the writeable array may be a copy of the input.

When success (0 return value) is returned, either out_arr is filled with a non-NULL PyArrayObject and the rest of the parameters are untouched, or out_arr is filled with NULL, and the rest of the parameters are filled.

Typical usage:

```python
PyArrayObject *arr = NULL;
PyArray_Descr *dtype = NULL;
int ndim = 0;
.npy_intp dims[NPY_MAXDIMS];

if (PyArray_GetArrayParamsFromObject(op, NULL, 1, &dtype,
    &ndim, &dims, &arr, NULL) < 0) {
    return NULL;
}
if (arr == NULL) {
    ... validate/change dtype, validate flags, ndim, etc ...
    // Could make custom strides here too
    arr = PyArray_NewFromDescr(PyArray_Type, dtype, ndim,
        dims, NULL,
        fortran ? NPY_ARRAY_F_CONTIGUOUS : 0,
        NULL);
    if (arr == NULL) {
        return NULL;
    }
    if (PyArray_CopyObject(arr, op) < 0) {
        Py_DECREF(arr);
        return NULL;
    }
} else {
    ... in this case the other parameters weren’t filled, just
    validate and possibly copy arr itself ...
}
... use arr ...
```

PyObject* PyArray_CheckFromAny (PyObject* op, PyArray_Descr* dtype, int min_depth, int max_depth,
    int requirements, PyObject* context)

Nearly identical to PyArray_FromAny (...) except requirements can contain NPY_ARRAY_NOTSWAPPED (over-riding the specification in dtype) and NPY_ARRAY_ELEMENTSTRIDES which indicates that the array should be aligned in the sense that the strides are multiples of the element size.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

**NPY_ARRAY_NOTSWAPPED**

Make sure the returned array has a data-type descriptor that is in machine byte-order, over-riding any specification in the dtype argument. Normally, the byte-order requirement is determined by the dtype argument. If this flag is set and the dtype argument does not indicate a machine byte-order descriptor (or is NULL and the object is already an array with a data-type descriptor that is not in machine byte- order), then a new data-type descriptor is created and used with its byte-order field set to native.
NPY_ARRAY_BEHAVED_NS
    NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE | NPY_ARRAY_NOTSWAPPED

NPY_ARRAY_ELEMENTSTRIDES
    Make sure the returned array has strides that are multiples of the element size.

PyObject* PyArray_FromArray (PyArrayObject* op, PyArray_Descr* newtype, int requirements)
    Special case of PyArray_FromAny for when op is already an array but it needs to be of a specific newtype
    (including byte-order) or has certain requirements.

PyObject* PyArray_FromStructInterface (PyObject* op)
    Returns an ndarray object from a Python object that exposes the __array_struct__ method and follows
    the array interface protocol. If the object does not contain this method then a borrowed reference to
    Py_NotImplemented is returned.

PyObject* PyArray_FromInterface (PyObject* op)
    Returns an ndarray object from a Python object that exposes the __array_shape__ and
    __array_typestr__ methods following the array interface protocol. If the object does not contain
    one of these method then a borrowed reference to Py_NotImplemented is returned.

PyObject* PyArray_FromArrayAttr (PyObject* op, PyArray_Descr* dtype, PyObject* context)
    Return an ndarray object from a Python object that exposes the __array__ method. The __array__
    method can take 0, 1, or 2 arguments ([dtype, context]) where context is used to pass information about where
    the __array__ method is being called from (currently only used in ufuncs).

PyObject* PyArray_ContiguousFromAny (PyObject* op, int typenum, int min_depth, int max_depth)
    This function returns a (C-style) contiguous and behaved function array from any nested sequence or array
    interface exporting object, op, of (non-flexible) type given by the enumerated typenum, of minimum depth
    min_depth, and of maximum depth max_depth. Equivalent to a call to PyArray_FromAny with requirements
    set to NPY_DEFAULT and the type_num member of the type argument set to typenum.

PyObject* PyArray_FromObject (PyObject *op, int typenum, int min_depth, int max_depth)
    Return an aligned and in native-byteorder array from any nested sequence or array-interface exporting object,
    op, of a type given by the enumerated typenum. The minimum number of dimensions the array can have is
    given by min_depth while the maximum is max_depth. This is equivalent to a call to PyArray_FromAny
    with requirements set to BEHAVED.

PyObject* PyArray_EnsureArray (PyObject* op)
    This function steals a reference to op and makes sure that op is a base-class ndarray. It special cases array
    scalars, but otherwise calls PyArray_FromAny (op, NULL, 0, 0, NPY_ARRAY_ENSUREARRAY).

PyObject* PyArray_FromString (char* string, npy_intp slen, PyArray_Descr* dtype, npy_intp num,
    char* sep)
    Construct a one-dimensional ndarray of a single type from a binary or (ASCII) text string of length slen.
    The data-type of the array to-be-created is given by dtype. If num is -1, then copy the entire string and return
    an appropriately sized array, otherwise, num is the number of items to copy from the string. If sep is NULL
    (or ""), then interpret the string as bytes of binary data, otherwise convert the sub-strings separated by sep to
    items of data-type dtype. Some data-types may not be readable in text mode and an error will be raised if that
    occurs. All errors return NULL.

PyObject* PyArray_FromFile (FILE* fp, PyArray_Descr* dtype, npy_intp num, char* sep)
    Construct a one-dimensional ndarray of a single type from a binary or text file. The open file pointer is fp, the
    data-type of the array to be created is given by dtype. This must match the data in the file. If num is -1, then
    read until the end of the file and return an appropriately sized array, otherwise, num is the number of items to
    read. If sep is NULL (or ""), then read from the file in binary mode, otherwise read from the file in text mode
    with sep providing the item separator. Some array types cannot be read in text mode in which case an error is
    raised.

PyObject* PyArray_FromBuffer (PyObject* buf, PyArray_Descr* dtype, npy_intp count, npy_intp offset)
    Construct a one-dimensional ndarray of a single type from an object, buf, that exports the (single-segment)
buffer protocol (or has an attribute __buffer__ that returns an object that exports the buffer protocol). A writeable buffer will be tried first followed by a read-only buffer. The NPY_ARRAY_WRITEABLE flag of the returned array will reflect which one was successful. The data is assumed to start at offset bytes from the start of the memory location for the object. The type of the data in the buffer will be interpreted depending on the data-type descriptor, dtype. If count is negative then it will be determined from the size of the buffer and the requested itemsize, otherwise, count represents how many elements should be converted from the buffer.

```c
int PyArray_CopyInto (PyArrayObject* dest, PyArrayObject* src)
Copy from the source array, src, into the destination array, dest, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of src must be broadcastable to the shape of dest. The data areas of dest and src must not overlap.
```

```c
int PyArray_MoveInto (PyArrayObject* dest, PyArrayObject* src)
Move data from the source array, src, into the destination array, dest, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of src must be broadcastable to the shape of dest. The data areas of dest and src may overlap.
```

```c
PyArrayObject* PyArray_GETCONTIGUOUS (PyObject* op)
If op is already (C-style) contiguous and well-behaved then just return a reference, otherwise return a (contiguous and well-behaved) copy of the array. The parameter op must be a (sub-class of an) ndarray and no checking for that is done.
```

```c
PyObject* PyArray_FROM_O (PyObject* obj)
Convert obj to an ndarray. The argument can be any nested sequence or object that exports the array interface. This is a macro form of PyArray_FromAny using NULL, 0, 0, 0 for the other arguments. Your code must be able to handle any data-type descriptor and any combination of data-flags to use this macro.
```

```c
PyObject* PyArray_FROM_OF (PyObject* obj, int requirements)
Similar to PyArray_FROM_O except it can take an argument of requirements indicating properties the resulting array must have. Available requirements that can be enforced are NPY_ARRAY_C_CONTIGUOUS, NPY_ARRAY_F_CONTIGUOUS, NPY_ARRAY_ALIGNED, NPY_ARRAY_WRITEABLE, NPY_ARRAY_NOTSWAPPED, NPY_ARRAY_ENSURECOPY, NPY_ARRAY_UPDATEIFCOPY, NPY_ARRAY_FORCECAST, and NPY_ARRAY_ENSUREARRAY. Standard combinations of flags can also be used:
```

```c
PyObject* PyArray_FROM_OT (PyObject* obj, int typenum)
Similar to PyArray_FROM_O except it can take an argument of typenum specifying the type-number the returned array.
```

```c
PyObject* PyArray_FROM_OTF (PyObject* obj, int typenum, int requirements)
Combination of PyArray_FROM_OF and PyArray_FROM_OT allowing both a typenum and a flags argument to be provided.
```

```c
PyObject* PyArray_FROMANY (PyObject* obj, int typenum, int min, int max, int requirements)
Similar to PyArray_FromAny except the data-type is specified using a typenumber. PyArray_DescrFromType (typenum) is passed directly to PyArray_FromAny. This macro also adds NPY_DEFAULT to requirements if NPY_ARRAY_ENSURECOPY is passed in as requirements.
```

```c
PyObject *PyArray_CheckAxis (PyObject* obj, int* axis, int requirements)
Encapsulate the functionality of functions and methods that take the axis= keyword and work properly with None as the axis argument. The input array is obj, while *axis is a converted integer (so that >=MAXDIMS is the None value), and requirements gives the needed properties of obj. The output is a converted version of the input so that requirements are met and if needed a flattening has occurred. On output negative values of *axis are converted and the new value is checked to ensure consistency with the shape of obj.
```
5.4.3 Dealing with types

General check of Python Type

**PyArray_Check** (op)
Evaluates true if `op` is a Python object whose type is a sub-type of `PyArray_Type`.

**PyArray_CheckExact** (op)
Evaluates true if `op` is a Python object with type `PyArray_Type`.

**PyArray_HasArrayInterface** (op, out)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created `ndarray` using the interface or `out` will contain **NULL** if an error during conversion occurs. Otherwise, `out` will contain a borrowed reference to `Py_NotImplemented` and no error condition is set.

**PyArray_HasArrayInterfaceType** (op, type, context, out)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created `ndarray` using the interface or `out` will contain **NULL** if an error during conversion occurs. Otherwise, `out` will contain a borrowed reference to `Py_NotImplemented` and no error condition is set. This version allows setting of the type and context in the part of the array interface that looks for the `__array__` attribute.

**PyArray_IsZeroDim** (op)
Evaluates true if `op` is an instance of (a subclass of) `PyArray_Type` and has 0 dimensions.

**PyArray_IsScalar** (op, cls)
Evaluates true if `op` is an instance of `Py{cls}ArrType_Type`.

**PyArray_CheckScalar** (op)
Evaluates true if `op` is either an array scalar (an instance of a sub-type of `PyGenericArr_Type`), or an instance of (a sub-class of) `PyArray_Type` whose dimensionality is 0.

**PyArray_IsPythonScalar** (op)
Evaluates true if `op` is a builtin Python “scalar” object (int, float, complex, str, unicode, long, bool).

**PyArray_IsAnyScalar** (op)
Evaluates true if `op` is either a Python scalar or an array scalar (an instance of a sub-type of `PyGenericArr_Type`).

Data-type checking

For the typenum macros, the argument is an integer representing an enumerated array data type. For the array type checking macros the argument must be a `PyObject *` that can be directly interpreted as a `PyArrayObject *`.

**PyTypeNum_ISUNSIGNED** (num)

**PyDataType_ISUNSIGNED** (descr)

**PyArray_ISUNSIGNED** (obj)
Type represents an unsigned integer.

**PyTypeNum_ISSIGNED** (num)

**PyDataType_ISSIGNED** (descr)

**PyArray_ISSIGNED** (obj)
Type represents a signed integer.

5.4. Array API
```python
PyTypeNum_ISINTEGER (num)

PyDataType_ISINTEGER (descr)

PyArray_ISINTEGER (obj)
    Type represents any integer.

PyTypeNum_ISFLOAT (num)

PyDataType_ISFLOAT (descr)

PyArray_ISFLOAT (obj)
    Type represents any floating point number.

PyTypeNum_ISCOMPLEX (num)

PyDataType_ISCOMPLEX (descr)

PyArray_ISCOMPLEX (obj)
    Type represents any complex floating point number.

PyTypeNum_ISNUMBER (num)

PyDataType_ISNUMBER (descr)

PyArray_ISNUMBER (obj)
    Type represents any integer, floating point, or complex floating point number.

PyTypeNum_ISSTRING (num)

PyDataType_ISSTRING (descr)

PyArray_ISSTRING (obj)
    Type represents a string data type.

PyTypeNum_ISPYTHON (num)

PyDataType_ISPYTHON (descr)

PyArray_ISPYTHON (obj)
    Type represents an enumerated type corresponding to one of the standard Python scalar (bool, int, float, or complex).

PyTypeNum_ISFLEXIBLE (num)

PyDataType_ISFLEXIBLE (descr)

PyArray_ISFLEXIBLE (obj)
    Type represents one of the flexible array types (NPY_STRING, NPY_UNICODE, or NPY_VOID).
```
PyTypeNum_ISUSERDEF (num)

PyDataType_ISUSERDEF (descr)

PyArray_ISUSERDEF (obj)
Type represents a user-defined type.

PyTypeNum_ISEXTENDED (num)

PyDataType_ISEXTENDED (descr)

PyArray_ISEXTENDED (obj)
Type is either flexible or user-defined.

PyTypeNum_ISOBJECT (num)

PyDataType_ISOBJECT (descr)

PyArray_ISOBJECT (obj)
Type represents object data type.

PyTypeNum_ISBOOL (num)

PyDataType_ISBOOL (descr)

PyArray_ISBOOL (obj)
Type represents Boolean data type.

PyDataType_HASFIELDS (descr)

PyArray_HASFIELDS (obj)
Type has fields associated with it.

PyArray_ISNOTSWAPPED (m)
Evaluates true if the data area of the ndarray m is in machine byte-order according to the array’s data-type descriptor.

PyArray_ISBYTESWAPPED (m)
Evaluates true if the data area of the ndarray m is not in machine byte-order according to the array’s data-type descriptor.

Bool PyArray_EquivTypes (PyArray_Descr* type1, PyArray_Descr* type2)
Return NPY_TRUE if type1 and type2 actually represent equivalent types for this platform (the fortran member of each type is ignored). For example, on 32-bit platforms, NPY_LONG and NPY_INT are equivalent. Otherwise return NPY_FALSE.

Bool PyArray_EquivArrTypes (PyArrayObject* a1, PyArrayObject * a2)
Return NPY_TRUE if a1 and a2 are arrays with equivalent types for this platform.

Bool PyArray_EquivTypenums (int typenum1, int typenum2)
Special case of PyArray_EquivTypes (...) that does not accept flexible data types but may be easier to call.

int PyArray_EquivByteorders ({byteorder} bl, {byteorder} bh)
True if byteorder characters ( NPY_LITTLE, NPY_BIG, NPY_NATIVE, NPY_IGNORE ) are either equal or
equivalent as to their specification of a native byte order. Thus, on a little-endian machine NPY_LITTLE and NPY_NATIVE are equivalent where they are not equivalent on a big-endian machine.

Converting data types

PyObject* PyArray_Cast (PyArrayObject* arr, int typenum)
Mainly for backwards compatibility to the Numeric C-API and for simple casts to non-flexible types. Return a new array object with the elements of arr cast to the data-type typenum which must be one of the enumerated types and not a flexible type.

PyObject* PyArray_CastToType (PyArrayObject* arr, PyArray_Descr* type, int fortran)
Return a new array of the type specified, casting the elements of arr as appropriate. The fortran argument specifies the ordering of the output array.

int PyArray_CastTo (PyArrayObject* out, PyArrayObject* in)
As of 1.6, this function simply calls PyArray_CopyInto, which handles the casting.

Cast the elements of the array in into the array out. The output array should be writeable, have an integer-multiple of the number of elements in the input array (more than one copy can be placed in out), and have a data type that is one of the builtin types. Returns 0 on success and -1 if an error occurs.

PyArray_VectorUnaryFunc* PyArray_GetCastFunc (PyArray_Descr* from, int totype)
Return the low-level casting function to cast from the given descriptor to the builtin type number. If no casting function exists return NULL and set an error. Using this function instead of direct access to from ->f->cast will allow support of any user-defined casting functions added to a descriptors casting dictionary.

int PyArray_CanCastSafely (int fromtype, int totype)
Returns non-zero if an array of data type fromtype can be cast to an array of data type totype without losing information. An exception is that 64-bit integers are allowed to be cast to 64-bit floating point values even though this can lose precision on large integers so as not to proliferate the use of long doubles without explicit requests. Flexible array types are not checked according to their lengths with this function.

int PyArray_CanCastTo (PyArray_Descr* fromtype, PyArray_Descr* totype)
PyArray_CanCastTypeTo supercedes this function in NumPy 1.6 and later.
Equivalent to PyArray_CanCastTypeTo(fromtype, totype, NPY_SAFE_CASTING).

int PyArray_CanCastTypeTo (PyArray_Descr* fromtype, PyArray_Descr* totype, NPY_CASTING casting)
New in version 1.6. Returns non-zero if an array of data type fromtype (which can include flexible types) can be cast safely to an array of data type totype (which can include flexible types) according to the casting rule casting. For simple types with NPY_SAFE_CASTING, this is basically a wrapper around PyArray_CanCastSafely, but for flexible types such as strings or unicode, it produces results taking into account their sizes.

int PyArray_CanCastArrayTo (PyArrayObject* arr, PyArray_Descr* totype, NPY_CASTING casting)
New in version 1.6. Returns non-zero if arr can be cast to totype according to the casting rule given in casting. If arr is an array scalar, its value is taken into account, and non-zero is also returned when the value will not overflow or be truncated to an integer when converting to a smaller type.

This is almost the same as the result of PyArray_CanCastTypeTo(PyArray_MinScalarType(arr), totype, casting), but it also handles a special case arising because the set of uint values is not a subset of the int values for types with the same number of bits.

PyArray_Descr* PyArray_MinScalarType (PyArrayObject* arr)
New in version 1.6. If arr is an array, returns its data type descriptor, but if arr is an array scalar (has 0 dimensions), it finds the data type of smallest size to which the value may be converted without overflow or truncation to an integer.
This function will not demote complex to float or anything to boolean, but will demote a signed integer to an
unsigned integer when the scalar value is positive.

PyArray_Descr* PyArray_PromoteTypes (PyArray_Descr* type1, PyArray_Descr* type2)
New in version 1.6. Finds the data type of smallest size and kind to which type1 and type2 may be safely
converted. This function is symmetric and associative.

PyArray_Descr* PyArray_ResultType (npy_intp narrs, PyArrayObject**arrs, npy_intp ndtypes,
PyArray_Descr**dtypes)
New in version 1.6. This applies type promotion to all the inputs, using the NumPy rules for combining scalars
and arrays, to determine the output type of a set of operands. This is the same result type that ufuncs produce.
The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex)
the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the
arrays, the data types are combined with PyArray_PromoteTypes to produce the return value.

Otherwise, PyArray_MinScalarType is called on each array, and the resulting data types are all combined with
PyArray_PromoteTypes to produce the return value.

The set of int values is not a subset of the uint values for types with the same number of bits, something not
reflected in PyArray_MinScalarType, but handled as a special case in PyArray_ResultType.

int PyArray_ObjectType (PyObject* op, int mintype)
This function is superceded by PyArray_MinScalarType and/or PyArray_ResultType.

This function is useful for determining a common type that two or more arrays can be converted to. It only
works for non-flexible array types as no itemsize information is passed. The mintype argument represents the
minimum type acceptable, and op represents the object that will be converted to an array. The return value is
the enumerated typenumber that represents the data-type that op should have.

void PyArray_ArrayType (PyObject* op, PyArray_Descr* mintype, PyArray_Descr* outtype)
This function is superceded by PyArray_ResultType.

This function works similarly to PyArray_ObjectType (...) except it handles flexible arrays. The mintype
argument can have an itemsize member and the outtype argument will have an itemsize member at least as big
but perhaps bigger depending on the object op.

PyArrayObject** PyArray_ConvertToCommonType (PyObject* op, int* n)
The functionality this provides is largely superceded by iterator NpyIter introduced in 1.6, with flag
NPY_ITER_COMMON_DTYPE or with the same dtype parameter for all operands.

Convert a sequence of Python objects contained in op to an array of ndarrays each having the same data type.
The type is selected based on the typenumber (larger type number is chosen over a smaller one) ignoring objects
that are only scalars. The length of the sequence is returned in n, and an n-length array of PyArrayObject
pointers is the return value (or NULL if an error occurs). The returned array must be freed by the caller of this
type (using PyDataMem_FREE ) and all the array objects in it DECREF ’d or a memory-leak will occur.
The example template-code below shows a typically usage:

mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
{code}
before return
for (i=0; i<n; i++) Py_DECREF(mps[i]);
PyDataMem_FREE(mps);
{return}

char* PyArray_Zero (PyArrayObject* arr)
A pointer to newly created memory of size arr ->itemsize that holds the representation of 0 for that type. The

5.4. Array API 1353
returned pointer, `ret`, must be freed using `PyDataMem_FREE` (ret) when it is not needed anymore.

char* `PyArray_One` (PyArrayObject* `arr`)
A pointer to newly created memory of size `arr`->itemsize that holds the representation of 1 for that type. The returned pointer, `ret`, must be freed using `PyDataMem_FREE` (ret) when it is not needed anymore.

int `PyArray_ValidType` (int `typenum`)
Returns `NPY_TRUE` if `typenum` represents a valid type-number (builtin or user-defined or character code). Otherwise, this function returns `NPY_FALSE`.

New data types

void `PyArray_InitArrFuncs` (PyArray_ArrFuncs* `f`)
Initialize all function pointers and members to NULL.

int `PyArray_RegisterDataType` (PyArray_Descr* `dtype`)
Register a data-type as a new user-defined data type for arrays. The type must have most of its entries filled in. This is not always checked and errors can produce segfaults. In particular, the typeobj member of the `dtype` structure must be filled with a Python type that has a fixed-size element-size that corresponds to the `elsize` member of `dtype`. Also the `f` member must have the required functions: nonzero, copyswap, copyswapn, getitem, setitem, and cast (some of the cast functions may be NULL if no support is desired). To avoid confusion, you should choose a unique character typecode but this is not enforced and not relied on internally.

A user-defined type number is returned that uniquely identifies the type. A pointer to the new structure can then be obtained from `PyArray_DescrFromType` using the returned type number. A -1 is returned if an error occurs. If this `dtype` has already been registered (checked only by the address of the pointer), then return the previously-assigned type-number.

int `PyArray_RegisterCastFunc` (PyArray_Descr* `descr`, int `totype`, PyArray_VectorUnaryFunc* `castfunc`)
Register a low-level casting function, `castfunc`, to convert from the data-type, `descr`, to the given data-type number, `totype`. Any old casting function is over-written. A 0 is returned on success or a -1 on failure.

int `PyArray_RegisterCanCast` (PyArray_Descr* `descr`, int `totype`, NPY_SCALARKIND `scalar`)
Register the data-type number, `totype`, as castable from data-type object, `descr`, of the given `scalar` kind. Use `scalar = NPY_NOSCALAR` to register that an array of data-type `descr` can be cast safely to a data-type whose type_number is `totype`.

Special functions for NPY_OBJECT

int `PyArray_INCREF` (PyArrayObject* `op`)
Used for an array, `op`, that contains any Python objects. It increments the reference count of every object in the array according to the data-type of `op`. A -1 is returned if an error occurs, otherwise 0 is returned.

void `PyArray_Item_INCREF` (char* `ptr`, PyArray_Descr* `dtype`)
A function to INCREF all the objects at the location `ptr` according to the data-type `dtype`. If `ptr` is the start of a record with an object at any offset, then this will (recursively) increment the reference count of all object-like items in the record.

int `PyArray_XDECREF` (PyArrayObject* `op`)
Used for an array, `op`, that contains any Python objects. It decrements the reference count of every object in the array according to the data-type of `op`. Normal return value is 0. A -1 is returned if an error occurs.

void `PyArray_Item_XDECREF` (char* `ptr`, PyArray_Descr* `dtype`)
A function to XDECREF all the object-like items at the location `ptr` as recorded in the data-type, `dtype`. This works recursively so that if `dtype` itself has fields with data-types that contain object-like items, all the object-like fields will be XDECREF’d.
void **PyArray_FillObjectArray** (PyArrayObject* \textit{arr}, PyObject* \textit{obj})

Fill a newly created array with a single value \textit{obj} at all locations in the structure with object data-types. No checking is performed but \textit{arr} must be of data-type NPY_OBJECT and be single-segment and uninitialized (no previous objects in position). Use \textbf{PyArray_DECREF} (\textit{arr}) if you need to decrement all the items in the object array prior to calling this function.

### 5.4.4 Array flags

The \texttt{flags} attribute of the \texttt{PyArrayObject} structure contains important information about the memory used by the array (pointed to by the data member). This flag information must be kept accurate or strange results and even segfaults may result.

There are 6 (binary) flags that describe the memory area used by the data buffer. These constants are defined in \texttt{arrayobject.h} and determine the bit-position of the flag. Python exposes a nice attribute-based interface as well as a dictionary-like interface for getting (and, if appropriate, setting) these flags.

Memory areas of all kinds can be pointed to by a ndarray, necessitating these flags. If you get an arbitrary \texttt{PyArrayObject} in C-code, you need to be aware of the flags that are set. If you need to guarantee a certain kind of array (like NPY_ARRAY_C_CONTIGUOUS and NPY_ARRAY_BEHAVED), then pass these requirements into the \texttt{PyArray_FromAny} function.

#### Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manipulate. It may not be aligned with word boundaries (very important on some platforms). It might have its data in a different byte-order than the machine recognizes. It might not be writeable. It might be in Fortran-contiguous order. The array flags are used to indicate what can be said about data associated with an array.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

**NPY\_ARRAY\_C\_CONTIGUOUS**

The data area is in C-style contiguous order (last index varies the fastest).

**NPY\_ARRAY\_F\_CONTIGUOUS**

The data area is in Fortran-style contiguous order (first index varies the fastest).

**Note:** Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension \texttt{arr.strides[dim]} may be \texttt{arbitrary} if \texttt{arr.shape[dim]} == 1 or the array has no elements. It does \texttt{not} generally hold that \texttt{self.strides[-1]} == \texttt{self.itemsize} for C-style contiguous arrays or \texttt{self.strides[0]} == \texttt{self.itemsize} for Fortran-style contiguous arrays is true. The correct way to access the \texttt{itemsize} of an array from the C API is \texttt{PyArray_ITEMSIZE(arr)}.

**See Also:**

\textit{Internal memory layout of an ndarray}

**NPY\_ARRAY\_OWNDATA**

The data area is owned by this array.

**NPY\_ARRAY\_ALIGNED**

The data area and all array elements are aligned appropriately.

5.4. Array API
**NPY_ARRAY_WRITEABLE**

The data area can be written to.

Notice that the above 3 flags are are defined so that a new, well-behaved array has these flags defined as true.

**NPY_ARRAY_UPDATEIFCOPY**

The data area represents a (well-behaved) copy whose information should be transferred back to the original when this array is deleted.

This is a special flag that is set if this array represents a copy made because a user required certain flags in `PyArray_FromAny` and a copy had to be made of some other array (and the user asked for this flag to be set in such a situation). The base attribute then points to the “misbehaved” array (which is set read_only). When the array with this flag set is deallocated, it will copy its contents back to the “misbehaved” array (casting if necessary) and will reset the “misbehaved” array to `NPY_ARRAY_WRITEABLE`. If the “misbehaved” array was not `NPY_ARRAY_WRITEABLE` to begin with then `PyArray_FromAny` would have returned an error because `NPY_ARRAY_UPDATEIFCOPY` would not have been possible.

`PyArray_UpdateFlags(obj, flags)` will update the `obj->flags` for `flags` which can be any of `NPY_ARRAY_C_CONTIGUOUS`, `NPY_ARRAY_F_CONTIGUOUS`, `NPY_ARRAY_ALIGNED`, or `NPY_ARRAY_WRITEABLE`.

**Combinations of array flags**

- **NPY_ARRAY_BEHAVED**
  
  `NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE`

- **NPY_ARRAY_CARRAY**
  
  `NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_BEHAVED`

- **NPY_ARRAY_CARRAY_RO**
  
  `NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_ALIGNED`

- **NPY_ARRAY_FARRAY**
  
  `NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_BEHAVED`

- **NPY_ARRAY_FARRAY_RO**
  
  `NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_ALIGNED`

- **NPY_ARRAY_DEFAULT**
  
  `NPY_ARRAY_CARRAY`

- **NPY_ARRAY_UPDATE_ALL**
  
  `NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_F_CONTIGUOUS | NPY_ARRAY_ALIGNED`

**Flag-like constants**

These constants are used in `PyArray_FromAny` (and its macro forms) to specify desired properties of the new array.

- **NPY_ARRAY_FORCECAST**
  
  Cast to the desired type, even if it can’t be done without losing information.

- **NPY_ARRAY_ENSURECOPY**
  
  Make sure the resulting array is a copy of the original.

- **NPY_ARRAY_ENSUREARRAY**
  
  Make sure the resulting object is an actual ndarray (or bigndarray), and not a sub-class.

- **NPY_ARRAY_NOTSWAPPED**
  
  Only used in `PyArray_CheckFromAny` to over-ride the byteorder of the data-type object passed in.
NPY_ARRAY_BEAHVED_NS
NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE | NPY_ARRAY_NOTSWAPPED

Flag checking

For all of these macros arr must be an instance of a (subclass of) PyArray_Type, but no checking is done.

PyArray_CHKFLAGS (arr, flags)
The first parameter, arr, must be an ndarray or subclass. The parameter, flags, should be an integer consisting of bitwise combinations of the possible flags an array can have: NPY_ARRAY_C_CONTIGUOUS, NPY_ARRAY_F_CONTIGUOUS, NPY_ARRAY_OWNDATA, NPY_ARRAY_ALIGNED, NPY_ARRAY_WRITEABLE, NPY_ARRAY_UPDATEIFCOPY.

PyArray_IS_C_CONTIGUOUS (arr)
Evaluates true if arr is C-style contiguous.

PyArray_IS_F_CONTIGUOUS (arr)
Evaluates true if arr is Fortran-style contiguous.

PyArray_ISFORTRAN (arr)
Evaluates true if arr is Fortran-style contiguous and not C-style contiguous. PyArray_IS_F_CONTIGUOUS is the correct way to test for Fortran-style contiguity.

PyArray_ISWRITEABLE (arr)
Evaluates true if the data area of arr can be written to

PyArray_ISALIGNED (arr)
Evaluates true if the data area of arr is properly aligned on the machine.

PyArray_ISBEHAVED (arr)
Evaluates true if the data area of arr is aligned and writeable and in machine byte-order according to its descriptor.

PyArray_ISBEHAVED_RO (arr)
Evaluates true if the data area of arr is aligned and in machine byte-order.

PyArray_ISCARRAY (arr)
Evaluates true if the data area of arr is C-style contiguous, and PyArray_ISBEHAVED (arr) is true.

PyArray_ISFARRAY (arr)
Evaluates true if the data area of arr is Fortran-style contiguous and PyArray_ISBEHAVED (arr) is true.

PyArray_ISCARRAY_RO (arr)
Evaluates true if the data area of arr is C-style contiguous, aligned, and in machine byte-order.

PyArray_ISFARRAY_RO (arr)
Evaluates true if the data area of arr is Fortran-style contiguous, aligned, and in machine byte-order.

PyArray_ISONESEGMENT (arr)
Evaluates true if the data area of arr consists of a single (C-style or Fortran-style) contiguous segment.

void PyArray_UpdateFlags (PyArrayObject* arr, int flagmask)
The NPY_ARRAY_C_CONTIGUOUS, NPY_ARRAY_ALIGNED, and NPY_ARRAY_F_CONTIGUOUS array flags can be “calculated” from the array object itself. This routine updates one or more of these flags of arr as specified in flagmask by performing the required calculation.

Warning: It is important to keep the flags updated (using PyArray_UpdateFlags can help) whenever a manipulation with an array is performed that might cause them to change. Later calculations in NumPy that rely on the state of these flags do not repeat the calculation to update them.
5.4.5 Array method alternative API

Conversion

PyObject* PyArray_GetField(PyArrayObject* self, PyArray_Descr* dtype, int offset)
   Equivalent to ndarray.getfield (self, dtype, offset). Return a new array of the given dtype using the data in the current array at a specified offset in bytes. The offset plus the itemsize of the new array type must be less than self->descr->elsize or an error is raised. The same shape and strides as the original array are used. Therefore, this function has the effect of returning a field from a record array. But, it can also be used to select specific bytes or groups of bytes from any array type.

int PyArray_SetField(PyArrayObject* self, PyArray_Descr* dtype, int offset, PyObject* val)
   Equivalent to ndarray.setfield (self, val, dtype, offset). Set the field starting at offset in bytes and of the given dtype to val. The offset plus dtype->elsize must be less than self->descr->elsize or an error is raised. Otherwise, the val argument is converted to an array and copied into the field pointed to. If necessary, the elements of val are repeated to fill the destination array. But, the number of elements in the destination must be an integer multiple of the number of elements in val.

PyObject* PyArray_Byteswap (PyArrayObject* self, Bool inplace)
   Equivalent to ndarray.byteswap (self, inplace). Return an array whose data area is byteswapped. If inplace is non-zero, then do the byteswap inplace and return a reference to self. Otherwise, create a byteswapped copy and leave self unchanged.

PyObject* PyArray_NewCopy (PyArrayObject* old, NPY_ORDER order)
   Equivalent to ndarray.copy (self, fortran). Make a copy of the old array. The returned array is always aligned and writeable with data interpreted as the same as the old array. If order is NPY_CORDER, then a C-style contiguous array is returned. If order is NPY_FORTRANORDER, then a Fortran-style contiguous array is returned. If order is NPY_ANYORDER, then the array returned is Fortran-style contiguous only if the old one is; otherwise, it is C-style contiguous.

PyObject* PyArray_ToList (PyArrayObject* self)
   Equivalent to ndarray.tolist (self). Return a nested Python list from self.

PyObject* PyArray_ToString (PyArrayObject* self, NPY_ORDER order)
   Equivalent to ndarray tostring (self, order). Return the bytes of this array in a Python string.

PyObject* PyArray_ToFile (PyArrayObject* self, FILE* fp, char* sep, char* format)
   Write the contents of self to the file pointer fp in C-style contiguous fashion. Write the data as binary bytes if sep is the string "\"or NULL. Otherwise, write the contents of self as text using the sep string as the item separator. Each item will be printed to the file. If the format string is not NULL or "\", then it is a Python print statement format string showing how the items are to be written.

int PyArray_Dump (PyObject* self, PyObject* file, int protocol)
   Pickle the object in self to the given file (either a string or a Python file object). If file is a Python string it is considered to be the name of a file which is then opened in binary mode. The given protocol is used (if protocol is negative, or the highest available is used). This is a simple wrapper around cPickle.dump (self, file, protocol).

PyObject* PyArray_Dumps (PyObject* self, int protocol)
   Pickle the object in self to a Python string and return it. Use the Pickle protocol provided (or the highest available if protocol is negative).

int PyArray_FillWithScalar (PyArrayObject* arr, PyObject* obj)
   Fill the array, arr, with the given scalar object, obj. The object is first converted to the data type of arr, and then copied into every location. A -1 is returned if an error occurs, otherwise 0 is returned.

PyObject* PyArray_View (PyArrayObject* self, PyArray_Descr* dtype, PyTypeObject *ptype)
   Equivalent to ndarray.view (self, dtype). Return a new view of the array self as possibly a different data-type, dtype, and different array subclass ptype.
If *dtype* is NULL, then the returned array will have the same data type as *self*. The new data-type must be consistent with the size of *self*. Either the itemsizes must be identical, or *self* must be single-segment and the total number of bytes must be the same. In the latter case the dimensions of the returned array will be altered in the last (or first for Fortran-style contiguous arrays) dimension. The data area of the returned array and *self* is exactly the same.

**Shape Manipulation**

PyObject* **PyArray_Newshape** (PyArrayObject* *self*, PyArray_Dims* *newshape*)  
Result will be a new array (pointing to the same memory location as *self* if possible), but having a shape given by *newshape*. If the new shape is not compatible with the strides of *self*, then a copy of the array with the new specified shape will be returned.

PyObject* **PyArray_Reshape** (PyArrayObject* *self*, PyObject* *shape*)  
Equivalent to *ndarray.reshape* (*self*, *shape*) where *shape* is a sequence. Converts *shape* to a PyArray_Dims structure and calls PyArray_Newshape internally.

PyObject* **PyArray_Squeeze** (PyArrayObject* *self*)  
Equivalent to *ndarray.squeeze* (*self*). Return a new view of *self* with all of the dimensions of length 1 removed from the shape.

**Warning**: matrix objects are always 2-dimensional. Therefore, **PyArray_Squeeze** has no effect on arrays of matrix sub-class.

PyObject* **PyArray_SwapAxes** (PyArrayObject* *self*, int *a1*, int *a2*)  
Equivalent to *ndarray.swapaxes* (*self*, *a1*, *a2*). The returned array is a new view of the data in *self* with the given axes, *a1* and *a2*, swapped.

PyObject* **PyArray_Resize** (PyArrayObject* *self*, PyArray_Dims* *newshape*, int *refcheck*, NPY_ORDER *fortran*)  
Equivalent to *ndarray.resize* (*self*, *newshape*, *refcheck* = *refcheck*, *order* = *fortran*). This function only works on single-segment arrays. It changes the shape of *self* inplace and will reallocate the memory for *self* if *newshape* has a different total number of elements then the old shape. If reallocation is necessary, then *self* must own its data, have *self* ->*base* = NULL, have *self* ->*weakrefs* = NULL, and (unless refcheck is 0) not be referenced by any other array. A reference to the new array is returned. The fortran argument can be NPY_ANYORDER, NPY_CORDER, or NPY_FORTRANORDER. It currently has no effect. Eventually it could be used to determine how the resize operation should view the data when constructing a differently-dimensioned array.

PyObject* **PyArray_Transpose** (PyArrayObject* *self*, PyArray_Dims* *permute*)  
Equivalent to *ndarray.transpose* (*self*, *permute*). Permute the axes of the ndarray object *self* according to the data structure *permute* and return the result. If *permute* is NULL, then the resulting array has its axes reversed. For example if *self* has shape 10 × 20 × 30, and *permute* .ptr is (0,2,1) the shape of the result is 10 × 30 × 20. If *permute* is NULL, the shape of the result is 30 × 20 × 10.

PyObject* **PyArray_Flatten** (PyArrayObject* *self*, NPY_ORDER *order*)  
Equivalent to *ndarray.flatten* (*self*, *order*). Return a 1-d copy of the array. If *order* is NPY_FORTRANORDER the elements are scanned out in Fortran order (first-dimension varies the fastest). If *order* is NPY_CORDER, the elements of *self* are scanned in C-order (last dimension varies the fastest). If *order* NPY_ANYORDER, then the result of **PyArray_ISFORTRAN** (*self*) is used to determine which order to flatten.

PyObject* **PyArray_Ravel** (PyArrayObject* *self*, NPY_ORDER *order*)  
Equivalent to *self.ravel(order)*. Same basic functionality as **PyArray_Flatten** (*self*, *order*) except if *order* is 0 and *self* is C-style contiguous, the shape is altered but no copy is performed.
**Item selection and manipulation**

PyObject* **PyArray_TakeFrom**(PyArrayObject* self, PyObject* indices, int axis, PyArrayObject* ret, NPY_CLIPMODE clipmode)

Equivalent to ndarray.take (self, indices, axis, ret, clipmode) except axis = None in Python is obtained by setting axis = NPY_MAXDIMS in C. Extract the items from self indicated by the integer-valued indices along the given axis. The clipmode argument can be NPY_RAISE, NPY_WRAP, or NPY_CLIP to indicate what to do with out-of-bound indices. The ret argument can specify an output array rather than having one created internally.

PyObject* **PyArray_PutTo**(PyArrayObject* self, PyObject* values, PyObject* indices, NPY_CLIPMODE clipmode)

Equivalent to self.put(values, indices, clipmode). Put values into self at the corresponding (flattened) indices. If values is too small it will be repeated as necessary.

PyObject* **PyArray_PutMask**(PyArrayObject* self, PyObject* values, PyObject* mask)

Place the values in self wherever corresponding positions (using a flattened context) in mask are true. The mask and self arrays must have the same total number of elements. If values is too small, it will be repeated as necessary.

PyObject* **PyArray_Repeat**(PyArrayObject* self, PyObject* op, int axis)

Equivalent to ndarray.repeat (self, op, axis). Copy the elements of self, op times along the given axis. Either op is a scalar integer or a sequence of length self ->dimensions[ axis ] indicating how many times to repeat each item along the axis.

PyObject* **PyArray_Choose**(PyArrayObject* self, PyObject* op, PyArrayObject* ret, NPY_CLIPMODE clipmode)

Equivalent to ndarray.choose (self, op, ret, clipmode). Create a new array by selecting elements from the sequence of arrays in op based on the integer values in self. The arrays must all be broadcastable to the same shape and the entries in self should be between 0 and len(op). The output is placed in ret unless it is NULL in which case a new output is created. The clipmode argument determines behavior for when entries in self are not between 0 and len(op).

NPY_RAISE
raise a ValueError;

NPY_WRAP
wrap values < 0 by adding len(op) and values >= len(op) by subtracting len(op) until they are in range;

NPY_CLIP
all values are clipped to the region [0, len(op) ).

PyObject* **PyArray_Sort**(PyArrayObject* self, int axis)

Equivalent to ndarray.sort (self, axis). Return an array with the items of self sorted along axis.

PyObject* **PyArray_ArgSort**(PyArrayObject* self, int axis)

Equivalent to ndarray.argsort (self, axis). Return an array of indices such that selection of these indices along the given axis would return a sorted version of self. If self ->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a record array, create a new data-type with a different order of names and construct a view of the array with that new data-type.

PyObject* **PyArray_LexSort**(PyObject* sort_keys, int axis)

Given a sequence of arrays (sort_keys) of the same shape, return an array of indices (similar to PyArray_ArgSort (...)) that would sort the arrays lexicographically. A lexicographic sort specifies that when two keys are found to be equal, the order is based on comparison of subsequent keys. A merge sort (which leaves equal entries unmoved) is required to be defined for the types. The sort is accomplished by sorting the indices first using the first sort_key and then using the second sort_key and so forth. This is equivalent to the lexsort(sort_keys, axis) Python command. Because of the way the merge-sort works, be sure
to understand the order the sort keys must be in (reversed from the order you would use when comparing two elements).

If these arrays are all collected in a record array, then PyArray_Sort (...) can also be used to sort the array directly.

PyObject* PyArray_SearchSorted (PyArrayObject* self, PyObject* values)
Equivalent to ndarray.searchsorted (self, values). Assuming self is a 1-d array in ascending order representing bin boundaries then the output is an array the same shape as values of bin numbers, giving the bin into which each item in values would be placed. No checking is done on whether or not self is in ascending order.

int PyArray_Partition (PyArrayObject *self, PyArrayObject *ktharray, int axis, NPY_SELECTKIND which)
Equivalent to ndarray.partition (self, ktharray, axis, kind). Partitions the array so that the values of the element indexed by ktharray are in the positions they would be if the array is fully sorted and places all elements smaller than the kth before and all elements equal or greater after the kth element. The ordering of all elements within the partitions is undefined. If self->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a record array, create a new data-type with a different order of names and construct a view of the array with that new data-type. Returns zero on success and -1 on failure.

PyObject* PyArray_ArgPartition (PyArrayObject *op, PyArrayObject *ktharray, int axis, NPY_SELECTKIND which)
Equivalent to ndarray.argpartition (self, ktharray, axis, kind). Return an array of indices such that selection of these indices along the given axis would return a partitioned version of self.

PyArray_Diagonal (PyArrayObject* self, int offset, int axis1, int axis2)
Equivalent to ndarray.diagonal (self, offset, axis1, axis2). Return the offset diagonals of the 2-d arrays defined by axis1 and axis2.

npy_intp PyArray_CountNonzero (PyArrayObject* self)
New in version 1.6. Counts the number of non-zero elements in the array object self.

PyObject* PyArray_Nonzero (PyArrayObject* self)
Equivalent to ndarray.nonzero (self). Returns a tuple of index arrays that select elements of self that are nonzero. If (nd=PyArray_NDIM (self))==1, then a single index array is returned. The index arrays have data type NPY_INTP. If a tuple is returned (nd \neq 1), then its length is nd.

PyObject* PyArray_Compress (PyArrayObject* self, PyObject* condition, int axis, PyArrayObject* out)
Equivalent to ndarray.compress (self, condition, axis). Return the elements along axis corresponding to elements of condition that are true.

Calculation

Tip: Pass in NPY_MAXDIMS for axis in order to achieve the same effect that is obtained by passing in axis = None in Python (treating the array as a 1-d array).

PyObject* PyArray_ArgMax (PyArrayObject* self, int axis)
Equivalent to ndarray.argmax (self, axis). Return the index of the largest element of self along axis.

PyObject* PyArray_ArgMin (PyArrayObject* self, int axis)
Equivalent to ndarray.argmin (self, axis). Return the index of the smallest element of self along axis.

PyObject* PyArray_Max (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.max (self, axis). Return the largest element of self along the given axis.
**Note:** The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not “large” enough to handle the output. By default, all integer data-types are made at least as large as NPY_LONG for the “add” and “multiply” ufuncs (which form the basis for mean, sum, cumsum, prod, and cumprod functions).

PyObject* PyArray_Min (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.min (self, axis). Return the smallest element of self along the given axis.

PyObject* PyArray_Ptp (PyArrayObject* self, int axis, PyArrayObject* out)
Equivalent to ndarray.ptp (self, axis). Return the difference between the largest element of self along axis and the smallest element of self along axis.

PyObject* PyArray_Mean (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.mean (self, axis, rtype). The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not “large” enough to handle the output. By default, all integer data-types are made at least as large as NPY_LONG for the “add” and “multiply” ufuncs (which form the basis for mean, sum, cumsum, prod, and cumprod functions).

PyObject* PyArray_Trace (PyArrayObject* self, int offset, int axis1, int axis2, int rtype, PyArrayObject* out)
Equivalent to ndarray.trace (self, offset, axis1, axis2, rtype). Return the sum (using rtype as the data type of summation) over the offset diagonal elements of the 2-d arrays defined by axis1 and axis2 variables. A positive offset chooses diagonals above the main diagonal. A negative offset selects diagonals below the main diagonal.

PyObject* PyArray_Clip (PyArrayObject* self, PyObject* min, PyObject* max)
Equivalent to ndarray.clip (self, min, max). Clip an array, self, so that values larger than max are fixed to max and values less than min are fixed to min.

PyObject* PyArray_Conjugate (PyArrayObject* self)
Equivalent to ndarray.conjugate (self). Return the complex conjugate of self. If self is not of complex data type, then return self with an reference.

PyObject* PyArray_Round (PyArrayObject* self, int decimals, PyArrayObject* out)
Equivalent to ndarray.round (self, decimals, out). Returns the array with elements rounded to the nearest decimal place. The decimal place is defined as the 10^(-decimal) digit so that negative decimals cause rounding to the nearest 10's, 100's, etc. If out is NULL, then the output array is created, otherwise the output is placed in out which must be the correct size and type.

PyObject* PyArray_Std (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.std (self, axis, rtype). Return the standard deviation using data along axis converted to data type rtype.

PyObject* PyArray_Sum (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.sum (self, axis, rtype). Return 1-d vector sums of elements in self along axis. Perform the sum after converting data to data type rtype.

PyObject* PyArray_CumSum (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.cumsum (self, axis, rtype). Return cumulative 1-d sums of elements in self along axis. Perform the sum after converting data to data type rtype.

PyObject* PyArray_Prod (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.prod (self, axis, rtype). Return 1-d products of elements in self along axis. Perform the product after converting data to data type rtype.

PyObject* PyArray_CumProd (PyArrayObject* self, int axis, int rtype, PyArrayObject* out)
Equivalent to ndarray.cumprod (self, axis, rtype). Return 1-d cumulative products of elements in self along axis. Perform the product after converting data to data type rtype.
PyObject* PyArray_All (PyArrayObject* self, int axis, PyArrayObject* out)

Equivalent to ndarray.all (self, axis). Return an array with True elements for every 1-d sub-array of self defined by axis in which all the elements are True.

PyObject* PyArray_Any (PyArrayObject* self, int axis, PyArrayObject* out)

Equivalent to ndarray.any (self, axis). Return an array with True elements for every 1-d sub-array of self defined by axis in which any of the elements are True.

5.4.6 Functions

Array Functions

int PyArray_AsCArray (PyObject** op, void* ptr, npy_intp* dims, int nd, int typenum, int itemsize)

Sometimes it is useful to access a multidimensional array as a C-style multi-dimensional array so that algorithms can be implemented using C’s a[i][j][k] syntax. This routine returns a pointer, ptr, that simulates this kind of C-style array, for 1-, 2-, and 3-d ndarrays.

Parameters

- **op** – The address to any Python object. This Python object will be replaced with an equivalent well-behaved, C-style contiguous, ndarray of the given data type specified by the last two arguments. Be sure that stealing a reference in this way to the input object is justified.
- **ptr** – The address to a (ctype* for 1-d, ctype** for 2-d or ctype*** for 3-d) variable where ctype is the equivalent C-type for the data type. On return, ptr will be addressable as a 1-d, 2-d, or 3-d array.
- **dims** – An output array that contains the shape of the array object. This array gives boundaries on any looping that will take place.
- **nd** – The dimensionality of the array (1, 2, or 3).
- **typenum** – The expected data type of the array.
- **itemsize** – This argument is only needed when typenum represents a flexible array. Otherwise it should be 0.

Note: The simulation of a C-style array is not complete for 2-d and 3-d arrays. For example, the simulated arrays of pointers cannot be passed to subroutines expecting specific, statically-defined 2-d and 3-d arrays. To pass to functions requiring those kind of inputs, you must statically define the required array and copy data.

int PyArray_Free (PyObject* op, void* ptr)

Must be called with the same objects and memory locations returned from PyArray_AsCArray (...). This function cleans up memory that otherwise would get leaked.

PyObject* PyArray_Concatenate (PyObject* obj, int axis)

Join the sequence of objects in obj together along axis into a single array. If the dimensions or types are not compatible an error is raised.

PyObject* PyArray_InnerProduct (PyObject* obj1, PyObject* obj2)

Compute a product-sum over the last dimensions of obj1 and obj2. Neither array is conjugated.

PyObject* PyArray_MatrixProduct (PyObject* obj1, PyObject* obj)

Compute a product-sum over the last dimension of obj1 and the second-to-last dimension of obj2. For 2-d arrays this is a matrix-product. Neither array is conjugated.
PyObject* **PyArray_MatrixProduct2** (PyObject* obj1, PyObject* obj, PyObject* out)

New in version 1.6. Same as PyArray_MatrixProduct, but store the result in `out`. The output array must have the correct shape, type, and be C-contiguous, or an exception is raised.

PyObject* **PyArray_EinsteinSum** (char* subscripts, npy_intp nop, PyArrayObject** op_in, PyArray_Descr* dtype, NPY_ORDER order, NPY_CASTING casting, PyArrayObject* out)

New in version 1.6. Applies the einstein summation convention to the array operands provided, returning a new array or placing the result in `out`. The string in `subscripts` is a comma separated list of index letters. The number of operands is in `nop`, and `op_in` is an array containing those operands. The data type of the output can be forced with `dtype`, the output order can be forced with `order` (NPY_KEEPORDER is recommended), and when `dtype` is specified, `casting` indicates how permissive the data conversion should be.

See the einsum function for more details.

PyObject* **PyArray_CopyAndTranspose** (PyObject *op)

A specialized copy and transpose function that works only for 2-d arrays. The returned array is a transposed copy of `op`.

PyObject* **PyArray_Correlate** (PyObject* op1, PyObject* op2, int mode)

Compute the 1-d correlation of the 1-d arrays `op1` and `op2`. The correlation is computed at each output point by multiplying `op1` by a shifted version of `op2` and summing the result. As a result of the shift, needed values outside of the defined range of `op1` and `op2` are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as `op1`, 2 - return all possible shifts (any overlap at all is accepted).

**Notes**

This does not compute the usual correlation: if `op2` is larger than `op1`, the arguments are swapped, and the conjugate is never taken for complex arrays. See PyArray_Correlate2 for the usual signal processing correlation.

PyObject* **PyArray_Correlate2** (PyObject* op1, PyObject* op2, int mode)

Updated version of PyArray_Correlate, which uses the usual definition of correlation for 1d arrays. The correlation is computed at each output point by multiplying `op1` by a shifted version of `op2` and summing the result. As a result of the shift, needed values outside of the defined range of `op1` and `op2` are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as `op1`, 2 - return all possible shifts (any overlap at all is accepted).

**Notes**

Compute z as follows:

\[ z[k] = \sum_n \text{op1}[n] \ast \text{conj(op2}[n+k]) \]

PyObject* **PyArray_Where** (PyObject* condition, PyObject* x, PyObject* y)

If both `x` and `y` are NULL, then return PyArray_Nonzero (condition). Otherwise, both `x` and `y` must be given and the object returned is shaped like `condition` and has elements of `x` and `y` where `condition` is respectively True or False.

**Other functions**

Bool **PyArray_CheckStrides** (int elsize, int nd, npy_intp numbytes, npy_intp* dims, npy_intp* newstrides)

Determine if `newstrides` is a strides array consistent with the memory of an `nd` -dimensional array with shape `dims` and element-size, `elsize`. The `newstrides` array is checked to see if jumping by the provided number of bytes in each direction will ever mean jumping more than `numbytes` which is the assumed size of the available memory segment. If `numbytes` is 0, then an equivalent `numbytes` is computed assuming `nd`, `dims`, and `elsize` refer to a single-segment array. Return NPY_TRUE if `newstrides` is acceptable, otherwise return NPY_FALSE.
npy_intp PyArray_MultiplyList (npy_intp* seq, int n)

int PyArray_MultiplyIntList (int* seq, int n)

Both of these routines multiply an \( n \)-length array, \( seq \), of integers and return the result. No overflow checking is performed.

int PyArray_CompareLists (npy_intp* l1, npy_intp* l2, int n)

Given two \( n \)-length arrays of integers, \( l1 \) and \( l2 \), return 1 if the lists are identical; otherwise, return 0.

## 5.4.7 Auxiliary Data With Object Semantics

New in version 1.7.0.

**NpyAuxData**

When working with more complex dtypes which are composed of other dtypes, such as the struct dtype, creating inner loops that manipulate the dtypes requires carrying along additional data. NumPy supports this idea through a struct `NpyAuxData`, mandating a few conventions so that it is possible to do this.

Defining an `NpyAuxData` is similar to defining a class in C++, but the object semantics have to be tracked manually since the API is in C. Here’s an example for a function which doubles up an element using an element copier function as a primitive:

```c
typedef struct {
    NpyAuxData base;
    ElementCopier_Func *func;
    NpyAuxData *funcdata;
) eldoubler_aux_data;

void free_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *d = (eldoubler_aux_data *)data;
    /* Free the memory owned by this auxdata */
    NPY_AUXDATA_FREE(d->funcdata);
    PyArray_free(d);
}

NpyAuxData *clone_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        return NULL;
    }
    /* Raw copy of all data */
    memcpy(ret, data, sizeof(eldoubler_aux_data));
    /* Fix up the owned auxdata so we have our own copy */
    ret->funcdata = NPY_AUXDATA_CLONE(ret->funcdata);
    if (ret->funcdata == NULL) {
        PyArray_free(ret);
        return NULL;
    }
    return (NpyAuxData *)ret;
}
```

5.4. Array API
NpyAuxData *create_element_doubler_aux_data(
    ElementCopier_Func *func,
    NpyAuxData *funcdata)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        PyErr_NoMemory();
        return NULL;
    }
    memset(&ret, 0, sizeof(eldoubler_aux_data));
    ret->base->free = &free_element_doubler_aux_data;
    ret->base->clone = &clone_element_doubler_aux_data;
    ret->func = func;
    ret->funcdata = funcdata;

    return (NpyAuxData *)ret;
}

NpyAuxData_FreeFunc
The function pointer type for NpyAuxData free functions.

NpyAuxData_CloneFunc
The function pointer type for NpyAuxData clone functions. These functions should never set the Python exception on error, because they may be called from a multi-threaded context.

NPY_AUXDATA_FREE (auxdata)
A macro which calls the auxdata’s free function appropriately, does nothing if auxdata is NULL.

NPY_AUXDATA_CLONE (auxdata)
A macro which calls the auxdata’s clone function appropriately, returning a deep copy of the auxiliary data.

5.4.8 Array Iterators
As of Numpy 1.6, these array iterators are superceded by the new array iterator, NpyIter.

An array iterator is a simple way to access the elements of an N-dimensional array quickly and efficiently. Section 2 provides more description and examples of this useful approach to looping over an array.

PyObject* PyArray_IterNew (PyObject* arr)
Return an array iterator object from the array, arr. This is equivalent to arr. flat. The array iterator object makes it easy to loop over an N-dimensional non-contiguous array in C-style contiguous fashion.

PyObject* PyArray_IterAllButAxis (PyObject* arr, int *axis)
Return an array iterator that will iterate over all axes but the one provided in *axis. The returned iterator cannot be used with PyArray_ITER_GOTO1D. This iterator could be used to write something similar to what ufuncs do wherein the loop over the largest axis is done by a separate sub-routine. If *axis is negative then *axis will be set to the axis having the smallest stride and that axis will be used.

PyObject *PyArray_BroadcastToShape (PyObject* arr, npy_intp *dimensions, int nd)
Return an array iterator that is broadcast to iterate as an array of the shape provided by dimensions and nd.

int PyArrayIter_Check (PyObject* op)
Evaluates true if op is an array iterator (or instance of a subclass of the array iterator type).

void PyArray_ITER_RESET (PyObject* iterator)
Reset an iterator to the beginning of the array.
void \texttt{PyArray\_ITER\_NEXT} (PyObject* \textit{iterator})

Increment the index and the dataptr members of the \textit{iterator} to point to the next element of the array. If the array is not (C-style) contiguous, also increment the N-dimensional coordinates array.

void \texttt{PyArray\_ITER\_DATA} (PyObject* \textit{iterator})

A pointer to the current element of the array.

void \texttt{PyArray\_ITER\_GOTO} (PyObject* \textit{iterator}, npy_intp* \textit{destination})

Set the \textit{iterator} index, dataptr, and coordinates members to the location in the array indicated by the N-dimensional c-array, \textit{destination}, which must have size at least \textit{iterator} ->nd_m1+1.

\texttt{PyArray\_ITER\_GOTO1D} (PyObject* \textit{iterator}, npy_intp \textit{index})

Set the \textit{iterator} index and dataptr to the location in the array indicated by the integer \textit{index} which points to an element in the C-styled flattened array.

int \texttt{PyArray\_ITER\_NOTDONE} (PyObject* \textit{iterator})

Evaluates TRUE as long as the iterator has not looped through all of the elements, otherwise it evaluates FALSE.

\subsection{5.4.9 Broadcasting (multi-iterators)}

\texttt{PyObject* \texttt{PyArray\_MultiIterNew}} (int \textit{num}, ...)

A simplified interface to broadcasting. This function takes the number of arrays to broadcast and then \textit{num} extra (PyObject *) arguments. These arguments are converted to arrays and iterators are created. \texttt{PyArray\_Broadcast} is then called on the resulting multi-iterator object. The resulting, broadcasted multi-iterator object is then returned. A broadcasted operation can then be performed using a single loop and using \texttt{PyArray\_MultiIter\_NEXT} (..)

void \texttt{PyArray\_MultiIter\_RESET} (PyObject* \textit{multi})

Reset all the iterators to the beginning in a multi-iterator object, \textit{multi}.

void \texttt{PyArray\_MultiIter\_NEXT} (PyObject* \textit{multi})

Advance each iterator in a multi-iterator object, \textit{multi}, to its next (broadcasted) element.

void \texttt{*PyArray\_MultiIter\_DATA} (PyObject* \textit{multi}, int \textit{i})

Return the data-pointer of the \textit{i}th iterator in a multi-iterator object.

void \texttt{PyArray\_MultiIter\_NEXTi} (PyObject* \textit{multi}, int \textit{i})

Advance the pointer of only the \textit{i}th iterator.

void \texttt{PyArray\_MultiIter\_GOTO} (PyObject* \textit{multi}, npy_intp* \textit{destination})

Advance each iterator in a multi-iterator object, \textit{multi}, to the given \textit{N} -dimensional \textit{destination} where \textit{N} is the number of dimensions in the broadcasted array.

void \texttt{PyArray\_MultiIter\_GOTO1D} (PyObject* \textit{multi}, npy_intp \textit{index})

Advance each iterator in a multi-iterator object, \textit{multi}, to the corresponding location of the \textit{index} into the flattened broadcasted array.

int \texttt{PyArray\_MultiIter\_NOTDONE} (PyObject* \textit{multi})

Evaluates TRUE as long as the multi-iterator has not looped through all of the elements (of the broadcasted result), otherwise it evaluates FALSE.

int \texttt{PyArray\_Broadcast} (PyArrayMultiIterObject* \textit{mit})

This function encapsulates the broadcasting rules. The \textit{mit} container should already contain iterators for all the arrays that need to be broadcast. On return, these iterators will be adjusted so that iteration over each simultaneously will accomplish the broadcasting. A negative number is returned if an error occurs.

int \texttt{PyArray\_RemoveSmallest} (PyArrayMultiIterObject* \textit{mit})

This function takes a multi-iterator object that has been previously “broadcasted,” finds the dimension with the smallest “sum of strides” in the broadcasted result and adapts all the iterators so as not to iterate over that dimension (by effectively making them of length-1 in that dimension). The corresponding dimension is
returned unless \texttt{mit} ->nd is 0, then -1 is returned. This function is useful for constructing ufunc-like routines that broadcast their inputs correctly and then call a strided 1-d version of the routine as the inner-loop. This 1-d version is usually optimized for speed and for this reason the loop should be performed over the axis that won’t require large stride jumps.

5.4.10 Neighborhood iterator

New in version 1.4.0. Neighborhood iterators are subclasses of the iterator object, and can be used to iter over a neighborhood of a point. For example, you may want to iterate over every voxel of a 3d image, and for every such voxel, iterate over an hypercube. Neighborhood iterator automatically handle boundaries, thus making this kind of code much easier to write than manual boundaries handling, at the cost of a slight overhead.

\begin{verbatim}
PyObject* PyArray_NeighborhoodIterNew(PyArrayIterObject* iter, npy_intp bounds, int mode, PyArrayObject* fill_value)

This function creates a new neighborhood iterator from an existing iterator. The neighborhood will be computed relatively to the position currently pointed by \texttt{iter}, the bounds define the shape of the neighborhood iterator, and the mode argument the boundaries handling mode.

The \texttt{bounds} argument is expected to be a (2 * iter->ao->nd) arrays, such as the range bound[2*i]->bounds[2*i+1] defines the range where to walk for dimension i (both bounds are included in the walked coordinates). The bounds should be ordered for each dimension (bounds[2*i] <= bounds[2*i+1]).

The mode should be one of:
\begin{itemize}
  \item \texttt{NPY\_NEIGHBORHOOD\_ITER\_ZERO\_PADDING}: zero padding. Outside bounds values will be 0.
  \item \texttt{NPY\_NEIGHBORHOOD\_ITER\_ONE\_PADDING}: one padding. Outside bounds values will be 1.
  \item \texttt{NPY\_NEIGHBORHOOD\_ITER\_CONSTANT\_PADDING}: constant padding. Outside bounds values will be the same as the first item in \texttt{fill\_value}.
  \item \texttt{NPY\_NEIGHBORHOOD\_ITER\_MIRROR\_PADDING}: mirror padding. Outside bounds values will be as if the array items were mirrored. For example, for the array \([1, 2, 3, 4]\), \(x[-2]\) will be 2, \(x[-2]\) will be 1, \(x[4]\) will be 4, \(x[5]\) will be 1, etc...
  \item \texttt{NPY\_NEIGHBORHOOD\_ITER\_CIRCULAR\_PADDING}: circular padding. Outside bounds values will be as if the array was repeated. For example, for the array \([1, 2, 3, 4]\), \(x[-2]\) will be 3, \(x[-2]\) will be 4, \(x[4]\) will be 1, \(x[5]\) will be 2, etc...
\end{itemize}

If the mode is constant filling (\texttt{NPY\_NEIGHBORHOOD\_ITER\_CONSTANT\_PADDING}), \texttt{fill\_value} should point to an array object which holds the filling value (the first item will be the filling value if the array contains more than one item). For other cases, \texttt{fill\_value} may be NULL.

\begin{itemize}
  \item The iterator holds a reference to \texttt{iter}
  \item Return NULL on failure (in which case the reference count of \texttt{iter} is not changed)
  \item \texttt{iter} itself can be a Neighborhood iterator: this can be useful for .e.g automatic boundaries handling
  \item the object returned by this function should be safe to use as a normal iterator
  \item If the position of \texttt{iter} is changed, any subsequent call to \texttt{PyArrayNeighborhoodIter\_Next} is undefined behavior, and \texttt{PyArrayNeighborhoodIter\_Reset} must be called.
\end{itemize}
\end{verbatim}
iter, bounds, NPY_NEIGHBORHOOD_ITER_ZERO_PADDING, NULL);

for(i = 0; i < iter->size; ++i) {
    for (j = 0; j < neigh_iter->size; ++j) {
        // Walk around the item currently pointed by iter->dataptr
        PyArrayNeighborhoodIter_Next(neigh_iter);
    }
    // Move to the next point of iter
    PyArrayIter_Next(iter);
    PyArrayNeighborhoodIter_Reset(neigh_iter);
}

int PyArrayNeighborhoodIter_Reset (PyArrayNeighborhoodIterObject* iter)
Reset the iterator position to the first point of the neighborhood. This should be called whenever the iter argument given at PyArray_NeighborhoodIterObject is changed (see example)

int PyArrayNeighborhoodIter_Next (PyArrayNeighborhoodIterObject* iter)
After this call, iter->dataptr points to the next point of the neighborhood. Calling this function after every point of the neighborhood has been visited is undefined.

5.4.11 Array Scalars

PyObject* PyArray_Return (PyArrayObject* arr)
This function checks to see if arr is a 0-dimensional array and, if so, returns the appropriate array scalar. It should be used whenever 0-dimensional arrays could be returned to Python.

PyObject* PyArray_Scalar (void* data, PyArray_Descr* dtype, PyObject* itemsize)
Return an array scalar object of the given enumerated typenum and itemsize by copying from memory pointed to by data. If swap is nonzero then this function will byteswap the data if appropriate to the data-type because array scalars are always in correct machine-byte order.

PyObject* PyArray_ToScalar (void* data, PyArrayObject* arr)
Return an array scalar object of the type and itemsize indicated by the array object arr copied from the memory pointed to by data and swapping if the data in arr is not in machine byte-order.

PyObject* PyArray_FromScalar (PyObject* scalar, PyArray_Descr* outcode)
Return a 0-dimensional array of type determined by outcode from scalar which should be an array-scalar object. If outcode is NULL, then the type is determined from scalar.

void PyArray_ScalarAsCtype (PyObject* scalar, void* ctypeptr)
Return in ctypeptr a pointer to the actual value in an array scalar. There is no error checking so scalar must be an array-scalar object, and ctypeptr must have enough space to hold the correct type. For flexible-sized types, a pointer to the data is copied into the memory of ctypeptr, for all other types, the actual data is copied into the address pointed to by ctypeptr.

void PyArray_CastScalarToCtype (PyObject* scalar, void* ctypeptr, PyArray_Descr* outcode)
Return the data (cast to the data type indicated by outcode) from the array-scalar, scalar, into the memory pointed to by ctypeptr (which must be large enough to handle the incoming memory).

PyObject* PyArray_TypeObjectFromType (int type)
Returns a scalar type-object from a type-number, type. Equivalent to PyArray_DescrFromType (type)->typeobj except for reference counting and error-checking. Returns a new reference to the type-object on success or NULL on failure.

NPY_SCALARKIND PyArray_ScalarKind (int typenum, PyArrayObject** arr)
See the function PyArray_MinScalarType for an alternative mechanism introduced in NumPy 1.6.0.
Return the kind of scalar represented by typenum and the array in *arr (if arr is not NULL). The array is assumed to be rank-0 and only used if typenum represents a signed integer. If arr is not NULL and the first element is negative then NPY_INTNEG_SCALAR is returned, otherwise NPY_INTPOS_SCALAR is returned. The possible return values are NPY_{kind}_SCALAR where (kind) can be INTPOS, INTNEG, FLOAT, COMPLEX, BOOL, or OBJECT. NPY_NOSCALAR is also an enumerated value NPY_SCALARKIND variables can take on.

int PyArray_CanCoerceScalar (char thistype, char neededtype, NPY_SCALARKIND scalar)

See the function PyArray_ResultType for details of NumPy type promotion, updated in NumPy 1.6.0.

Implements the rules for scalar coercion. Scalars are only silently coerced from thistype to needed-type if this function returns nonzero. If scalar is NPY_NOSCALAR, then this function is equivalent to PyArray_CanCastSafely. The rule is that scalars of the same KIND can be coerced into arrays of the same KIND. This rule means that high-precision scalars will never cause low-precision arrays of the same KIND to be upcast.

5.4.12 Data-type descriptors

<table>
<thead>
<tr>
<th>Warning: Data-type objects must be reference counted so be aware of the action on the data-type reference of different C-API calls. The standard rule is that when a data-type object is returned it is a new reference. Functions that take PyArray_Descr * objects and return arrays steal references to the data-type their inputs unless otherwise noted. Therefore, you must own a reference to any data-type object used as input to such a function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int PyArray_Descr_Check (PyObject* obj)</td>
</tr>
<tr>
<td>Evaluates as true if obj is a data-type object (PyArray_Descr *).</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrNew (PyArray_Descr* obj)</td>
</tr>
<tr>
<td>Return a new data-type object copied from obj (the fields reference is just updated so that the new object points to the same fields dictionary if any).</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrNewFromType (int typenum)</td>
</tr>
<tr>
<td>Create a new data-type object from the built-in (or user-registered) data-type indicated by typenum. All builtin types should not have any of their fields changed. This creates a new copy of the PyArray_Descr structure so that you can fill it in as appropriate. This function is especially needed for flexible data-types which need to have a new elsize member in order to be meaningful in array construction.</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrNewByteorder (PyArray_Descr* obj, char newendian)</td>
</tr>
<tr>
<td>Create a new data-type object with the byteorder set according to newendian. All referenced data-type objects (in subdescr and fields members of the data-type object) are also changed (recursively). If a byteorder of NPY_IGNORE is encountered it is left alone. If newendian is NPY_SWAP, then all byte-orders are swapped. Other valid newendian values are NPY_NATIVE, NPY_LITTLE, and NPY_BIG which all cause the returned data-typed descriptor (and all it’s referenced data-type descriptors) to have the corresponding byte-order.</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrFromObject (PyObject* op, PyArray_Descr* mintype)</td>
</tr>
<tr>
<td>Determine an appropriate data-type object from the object op (which should be a “nested” sequence object) and the minimum data-type descriptor mintype (which can be NULL). Similar in behavior to array(op).dtype. Don’t confuse this function with PyArray_DescrConverter. This function essentially looks at all the objects in the (nested) sequence and determines the data-type from the elements it finds.</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrFromScalar (PyObject* scalar)</td>
</tr>
<tr>
<td>Return a data-type object from an array-scalar object. No checking is done to be sure that scalar is an array scalar. If no suitable data-type can be determined, then a data-type of NPY_OBJECT is returned by default.</td>
</tr>
<tr>
<td>PyArray_Descr* PyArray_DescrFromType (int typenum)</td>
</tr>
<tr>
<td>Returns a data-type object corresponding to typenum. The typenum can be one of the enumerated types, a</td>
</tr>
</tbody>
</table>
character code for one of the enumerated types, or a user-defined type.

```c
int PyArray_DescrConverter (PyObject* obj, PyArray_Descr** dtype)
```

Convert any compatible Python object, `obj`, to a data-type object in `dtype`. A large number of Python objects can be converted to data-type objects. See *Data type objects* (*dtype*) for a complete description. This version of the converter converts None objects to a `NPY_DEFAULT_TYPE` data-type object. This function can be used with the “O&” character code in `PyArg_ParseTuple` processing.

```c
int PyArray_DescrConverter2 (PyObject* obj, PyArray_Descr** dtype)
```

Convert any compatible Python object, `obj`, to a data-type object in `dtype`. This version of the converter converts None objects so that the returned data-type is `NULL`. This function can also be used with the “O&” character in `PyArg_ParseTuple` processing.

```c
int PyArray_DescrAlignConverter (PyObject* obj, PyArray_Descr** dtype)
```

Like `PyArray_DescrConverter` except it aligns C-struct-like objects on word-boundaries as the compiler would.

```c
int PyArray_DescrAlignConverter2 (PyObject* obj, PyArray_Descr** dtype)
```

Like `PyArray_DescrConverter2` except it aligns C-struct-like objects on word-boundaries as the compiler would.

```c
PyObject *PyArray_FieldNames (PyObject* dict)
```

Take the fields dictionary, `dict`, such as the one attached to a data-type object and construct an ordered-list of field names such as is stored in the names field of the `PyArray_Descr` object.

### 5.4.13 Conversion Utilities

#### For use with `PyArg_ParseTuple`

All of these functions can be used in `PyArg_ParseTuple (...)` with the “O&” format specifier to automatically convert any Python object to the required C-object. All of these functions return `NPY_SUCCEED` if successful and `NPY_FAIL` if not. The first argument to all of these function is a Python object. The second argument is the `address` of the C-type to convert the Python object to.

**Warning:** Be sure to understand what steps you should take to manage the memory when using these conversion functions. These functions can require freeing memory, and/or altering the reference counts of specific objects based on your use.

```c
int PyArray_Converter (PyObject* obj, PyObject** address)
```

Convert any Python object to a `PyArrayObject`. If `PyArray_Check (obj)` is TRUE then its reference count is incremented and a reference placed in `address`. If `obj` is not an array, then convert it to an array using `PyArray_FromAny`. No matter what is returned, you must `DECREF` the object returned by this routine in `address` when you are done with it.

```c
int PyArray_OutputConverter (PyObject* obj, PyArrayObject** address)
```

This is a default converter for output arrays given to functions. If `obj` is `Py_None` or `NULL`, then `*address` will be `NULL` but the call will succeed. If `PyArray_Check (obj)` is TRUE then it is returned in `*address` without incrementing its reference count.

```c
int PyArray_IntpConverter (PyObject* obj, PyArray_Dims* seq)
```

Convert any Python sequence, `obj`, smaller than `NPY_MAXDIMS` to a C-array of `npy_intp`. The Python object could also be a single number. The `seq` variable is a pointer to a structure with members `ptr` and `len`. On successful return, `seq ->ptr` contains a pointer to memory that must be freed to avoid a memory leak. The restriction on memory size allows this converter to be conveniently used for sequences intended to be interpreted as array shapes.

5.4. Array API
int PyArray_BufferConverter (PyObject* obj, PyArray_Chunk* buf)
  Convert any Python object, obj, with a (single-segment) buffer interface to a variable with members that detail
  the object’s use of its chunk of memory. The buf variable is a pointer to a structure with base, ptr, len, and flags
  members. The PyArray_Chunk structure is binary compatible with the Python’s buffer object (through
  its len member on 32-bit platforms and its ptr member on 64-bit platforms or in Python 2.5). On return, the
  base member is set to obj (or its base if obj is already a buffer object pointing to another object). If you
  need to hold on to the memory be sure to INCREF the base member. The chunk of memory is pointed to
  by buf ->ptr member and has length buf ->len. The flags member of buf is NPY_BEHAVED_RO with the
  NPY_ARRAY_WRITEABLE flag set if obj has a writeable buffer interface.

int PyArray_AxisConverter (PyObject * obj, int* axis)
  Convert a Python object, obj, representing an axis argument to the proper value for passing to the functions that
  take an integer axis. Specifically, if obj is None, axis is set to NPY_MAXDIMS which is interpreted correctly by
  the C-API functions that take axis arguments.

int PyArray_BoolConverter (PyObject* obj, Bool* value)
  Convert any Python object, obj, to NPY_TRUE or NPY_FALSE, and place the result in value.

int PyArray_BytorderConverter (PyObject* obj, char* endian)

int PyArray_SortkindConverter (PyObject* obj, NPY_SORTKIND* sort)
  Convert Python strings into one of NPY_QUICKSORT (starts with ‘q’ or ‘Q’), NPY_HEAPSORT (starts with
  ‘h’ or ‘H’), or NPY_MERGESORT (starts with ‘m’ or ‘M’).

int PyArray_SearchsideConverter (PyObject* obj, NPY_SEARCHSIDE* side)
  Convert Python strings into one of NPY_SEARCHLEFT (starts with ‘l’ or ‘L’) or NPY_SEARCHRIGHT (starts
  with ‘r’ or ‘R’).

int PyArray_OrderConverter (PyObject* obj, NPY_ORDER* order)
  Convert the Python strings ‘C’, ‘F’, ‘A’, and ‘K’ into the NPY_ORDER enumeration NPY_CORDER,
  NPY_FORTRANORDER, NPY_ANYORDER, and NPY_KEEPORDER.

int PyArray_CastingConverter (PyObject* obj, NPY_CASTING* casting)
  Convert the Python strings ‘no’, ‘equiv’, ‘safe’, ‘same_kind’, and ‘unsafe’ into the NPY_CASTING enumera-
  tion NPY_NO_CASTING, NPY_EQUIV_CASTING, NPY_SAFE_CASTING, NPY_SAME_KIND_CASTING,
  and NPY_UNSAFE_CASTING.

int PyArray_ClipmodeConverter (PyObject* object, NPY_CLIPMODE* val)
  Convert the Python strings ‘clip’, ‘wrap’, and ‘raise’ into the NPY_CLIPMODE enumeration NPY_CLIP,
  NPY_WRAP, and NPY_RAISE.

int PyArray_ConvertClipmodeSequence (PyObject* object, NPY_CLIPMODE* modes, int n)
  Converts either a sequence of clipmodes or a single clipmode into a C array of NPY_CLIPMODE values. The
  number of clipmodes n must be known before calling this function. This function is provided to help functions
  allow a different clipmode for each dimension.

Other conversions

int PyArray_PyIntAsInt (PyObject* op)
  Convert all kinds of Python objects (including arrays and array scalars) to a standard integer. On error, -1 is
  returned and an exception set. You may find useful the macro:

  #define error_converting(x) (((x) == -1) && PyErr_Occurred())

npy_intp PyArray_PyIntAsIntp (PyObject* op)
  Convert all kinds of Python objects (including arrays and array scalars) to a (platform-pointer-sized) integer.
  On error, -1 is returned and an exception set.
int `PyArray_IntpFromSequence` (PyObject* `seq`, npy_intp* `vals`, int `maxvals`)

Convert any Python sequence (or single Python number) passed in as `seq` to (up to) `maxvals` pointer-sized integers and place them in the `vals` array. The sequence can be smaller then `maxvals` as the number of converted objects is returned.

int `PyArray_TypestrConvert` (int `itemsize`, int `gentype`)

Convert typestring characters (with `itemsize`) to basic enumerated data types. The typestring character corresponding to signed and unsigned integers, floating point numbers, and complex-floating point numbers are recognized and converted. Other values of `gentype` are returned. This function can be used to convert, for example, the string ‘f4’ to `NPY_FLOAT32`.

5.4.14 Miscellaneous

Importing the API

In order to make use of the C-API from another extension module, the `import_array()` command must be used. If the extension module is self-contained in a single .c file, then that is all that needs to be done. If, however, the extension module involves multiple files where the C-API is needed then some additional steps must be taken.

```c
void import_array (void)
```

This function must be called in the initialization section of a module that will make use of the C-API. It imports the module where the function-pointer table is stored and points the correct variable to it.

```c
PY_ARRAY_UNIQUE_SYMBOL
```

Using these #defines you can use the C-API in multiple files for a single extension module. In each file you must define `PY_ARRAY_UNIQUE_SYMBOL` to some name that will hold the C-API (e.g. myextension:Array_API). This must be done before including the numpy/arrayobject.h file. In the module initialization routine you call `import_array()`. In addition, in the files that do not have the module initialization sub_routine define `NO_IMPORT_ARRAY` prior to including numpy/arrayobject.h.

Suppose I have two files coolmodule.c and coolhelper.c which need to be compiled and linked into a single extension module. Suppose coolmodule.c contains the required initcool module initialization function (with the import_array() function called). Then, coolmodule.c would have at the top:

```c
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```c
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#define NO_IMPORT_ARRAY
#include numpy/arrayobject.h
```

Checking the API Version

Because python extensions are not used in the same way as usual libraries on most platforms, some errors cannot be automatically detected at build time or even runtime. For example, if you build an extension using a function available only for numpy >= 1.3.0, and you import the extension later with numpy 1.2, you will not get an import error (but almost certainly a segmentation fault when calling the function). That’s why several functions are provided to check for numpy versions. The macros `NPY_VERSION` and `NPY_FEATURE_VERSION` corresponds to the numpy version used to build the extension, whereas the versions returned by the functions `PyArray_GetNDArrayCVersion` and `PyArray_GetNDArrayCFeatureVersion` corresponds to the runtime numpy’s version.

5.4. Array API
The rules for ABI and API compatibilities can be summarized as follows:

- Whenever `NPY_VERSION` != `PyArray_GetNDArrayCVersion`, the extension has to be recompiled (ABI incompatibility).
- `NPY_VERSION` == `PyArray_GetNDArrayCVersion` and `NPY_FEATURE_VERSION` <= `PyArray_GetNDArrayCFeatureVersion` means backward compatible changes.

ABI incompatibility is automatically detected in every numpy’s version. API incompatibility detection was added in numpy 1.4.0. If you want to supported many different numpy versions with one extension binary, you have to build your extension with the lowest `NPY_FEATURE_VERSION` as possible.

```c
unsigned int PyArray_GetNDArrayCVersion (void)
```

This just returns the value `NPY_VERSION`. `NPY_VERSION` changes whenever a backward incompatible change at the ABI level. Because it is in the C-API, however, comparing the output of this function from the value defined in the current header gives a way to test if the C-API has changed thus requiring a re-compilation of extension modules that use the C-API. This is automatically checked in the function `import_array`.

```c
unsigned int PyArray_GetNDArrayCFeatureVersion (void)
```

New in version 1.4.0. This just returns the value `NPY_FEATURE_VERSION`. `NPY_FEATURE_VERSION` changes whenever the API changes (e.g. a function is added). A changed value does not always require a recompile.

### Internal Flexibility

```c
int PyArray_SetNumericOps (PyObject* dict)
```

NumPy stores an internal table of Python callable objects that are used to implement arithmetic operations for arrays as well as certain array calculation methods. This function allows the user to replace any or all of these Python objects with their own versions. The keys of the dictionary, `dict`, are the named functions to replace and the paired value is the Python callable object to use. Care should be taken that the function used to replace an internal array operation does not itself call back to that internal array operation (unless you have designed the function to handle that), or an unchecked infinite recursion can result (possibly causing program crash). The key names that represent operations that can be replaced are:

- `add`, `subtract`, `multiply`, `divide`, `remainder`, `power`, `square`, `reciprocal`, `ones_like`, `sqrt`, `negative`, `absolute`, `invert`, `left_shift`, `right_shift`, `bitwise_and`, `bitwise_xor`, `bitwise_or`, `less`, `less_equal`, `equal`, `not_equal`, `greater`, `greater_equal`, `floor_divide`, `true_divide`, `logical_or`, `logical_and`, `floor`, `ceil`, `maximum`, `minimum`, `rint`.

These functions are included here because they are used at least once in the array object’s methods. The function returns -1 (without setting a Python Error) if one of the objects being assigned is not callable.

```c
PyObject* PyArray_GetNumericOps (void)
```

Return a Python dictionary containing the callable Python objects stored in the the internal arithmetic operation table. The keys of this dictionary are given in the explanation for `PyArray_SetNumericOps`.

```c
void PyArray_SetStringFunction (PyObject* op, int repr)
```

This function allows you to alter the `tp_str` and `tp_repr` methods of the array object to any Python function. Thus you can alter what happens for all arrays when `str(arr)` or `repr(arr)` is called from Python. The function to be called is passed in as `op`. If `repr` is non-zero, then this function will be called in response to `repr(arr)`, otherwise the function will be called in response to `str(arr)`. No check on whether or not `op` is callable is performed. The callable passed in to `op` should expect an array argument and should return a string to be printed.

### Memory management

```c
char* PyDataMem_NEW (size_t nbytes)
```
**PyDataMem_FREE** (char* *ptr*)

char* **PyDataMem_RENEW** (void *ptr, size_t *newbytes)

Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.

**npy_intp** *PyDimMem_NEW* (nd)

**PyDimMem_FREE** (npy_intp* *ptr*)

npy_intp* **PyDimMem_RENEW** (npy_intp* *ptr*, npy_intp *newnd*)

Macros to allocate, free, and reallocate dimension and strides memory.

**PyArray_malloc** (nbytes)

**PyArray_free** (ptr)

**PyArray_realloc** (ptr, nbytes)

These macros use different memory allocators, depending on the constant NPY_USE_PYMEM. The system malloc is used when NPY_USE_PYMEM is 0, if NPY_USE_PYMEM is 1, then the Python memory allocator is used.

**Threading support**

These macros are only meaningful if NPY_ALLOW_THREADS evaluates True during compilation of the extension module. Otherwise, these macros are equivalent to whitespace. Python uses a single Global Interpreter Lock (GIL) for each Python process so that only a single thread may execute at a time (even on multi-cpu machines). When calling out to a compiled function that may take time to compute (and does not have side-effects for other threads like updated global variables), the GIL should be released so that other Python threads can run while the time-consuming calculations are performed. This can be accomplished using two groups of macros. Typically, if one macro in a group is used in a code block, all of them must be used in the same code block. Currently, NPY_ALLOW_THREADS is defined to the python-defined WITH_THREADS constant unless the environment variable NPY_NOSMP is set in which case NPY_ALLOW_THREADS is defined to be 0.

**Group 1**

This group is used to call code that may take some time but does not use any Python C-API calls. Thus, the GIL should be released during its calculation.

**NPY_BEGIN_ALLOW_THREADS**

Equivalent to Py_BEGIN_ALLOW_THREADS except it uses NPY_ALLOW_THREADS to determine if the macro if replaced with white-space or not.

**NPY_END_ALLOW_THREADS**

Equivalent to Py_END_ALLOW_THREADS except it uses NPY_ALLOW_THREADS to determine if the macro if replaced with white-space or not.

**NPY_BEGIN_THREADS_DEF**

Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

**NPY_BEGIN_THREADS**

Place right before code that does not need the Python interpreter (no Python C-API calls). This macro saves the Python state and releases the GIL.
**NPY_END_THREADS**
Place right after code that does not need the Python interpreter. This macro acquires the GIL and restores the Python state from the saved variable.

**NPY_BEGIN_THREADS_DESCR** *(PyArray_Descr *dtype)*
Useful to release the GIL only if *dtype* does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop. Equivalent to

**NPY_END_THREADS_DESCR** *(PyArray_Descr *dtype)*
Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

**Group 2**
This group is used to re-acquire the Python GIL after it has been released. For example, suppose the GIL has been released (using the previous calls), and then some path in the code (perhaps in a different subroutine) requires use of the Python C-API, then these macros are useful to acquire the GIL. These macros accomplish essentially a reverse of the previous three (acquire the LOCK saving what state it had) and then re-release it with the saved state.

**NPY_ALLOW_C_API_DEF**
Place in the variable declaration area to set up the necessary variable.

**NPY_ALLOW_C_API**
Place before code that needs to call the Python C-API (when it is known that the GIL has already been released).

**NPY_DISABLE_C_API**
Place after code that needs to call the Python C-API (to re-release the GIL).

---

**Tip:** Never use semicolons after the threading support macros.

---

**Priority**

**NPY_PRIORITY**
Default priority for arrays.

**NPY_SUBTYPE.Priority**
Default subtype priority.

**NPY_SCALAR_PRIORITY**
Default scalar priority (very small)

double **PyArray_GetPriority** *(PyObject* *obj*, double def)*
Return the __array_priority__ attribute (converted to a double) of *obj* or *def* if no attribute of that name exists. Fast returns that avoid the attribute lookup are provided for objects of type PyArray_Type.

---

**Default buffers**

**NPY_BUFSIZE**
Default size of the user-settable internal buffers.

**NPY_MIN_BUFSIZE**
Smallest size of user-settable internal buffers.

**NPY_MAX_BUFSIZE**
Largest size allowed for the user-settable buffers.
Other constants

NPY_NUM_FLOATTYPE
The number of floating-point types

NPY_MAXDIMS
The maximum number of dimensions allowed in arrays.

NPY_VERSION
The current version of the ndarray object (check to see if this variable is defined to guarantee the numpy/arrayobject.h header is being used).

NPY_FALSE
Defined as 0 for use with Bool.

NPY_TRUE
Defined as 1 for use with Bool.

NPY_FAIL
The return value of failed converter functions which are called using the “O&” syntax in PyArg_ParseTuple-like functions.

NPY_SUCCEED
The return value of successful converter functions which are called using the “O&” syntax in PyArg_ParseTuple-like functions.

Miscellaneous Macros

PyArray_SAMESHAPE (a1, a2)
Evaluates as True if arrays a1 and a2 have the same shape.

PyArray_MAX (a, b)
Returns the maximum of a and b. If (a) or (b) are expressions they are evaluated twice.

PyArray_MIN (a, b)
Returns the minimum of a and b. If (a) or (b) are expressions they are evaluated twice.

PyArray_CLT (a, b)

PyArray_CGT (a, b)

PyArray_CLE (a, b)

PyArray_CGE (a, b)

PyArray_CEQ (a, b)

PyArray_CNE (a, b)
Implements the complex comparisons between two complex numbers (structures with a real and imag member) using NumPy’s definition of the ordering which is lexicographic: comparing the real parts first and then the complex parts if the real parts are equal.

PyArray_REFCOUNT (PyObject* op)
Returns the reference count of any Python object.
PyArray_XDECREF_ERR (PyObject *obj)

DECREASE’s an array object which may have the NPY_ARRAY_UPDATEIFCOPY flag set without causing the contents to be copied back into the original array. Resets the NPY_ARRAY_WRITEABLE flag on the base object. This is useful for recovering from an error condition when NPY_ARRAY_UPDATEIFCOPY is used.

Enumerated Types

NPY_SORTKIND
A special variable-type which can take on the values NPY_{KIND} where {KIND} is

QUICKSORT, HEAPSORT, MERGESORT

NPY_NSORTS
Defined to be the number of sorts.

NPY_SCALARKIND
A special variable type indicating the number of “kinds” of scalars distinguished in determining scalar-coercion rules. This variable can take on the values NPY_{KIND} where {KIND} can be

NOSCALAR, BOOL_SCALAR, INTPOS_SCALAR, INTNEG_SCALAR, FLOAT_SCALAR, COMPLEX_SCALAR, OBJECT_SCALAR

NPY_NSCALARKINDS
Defined to be the number of scalar kinds (not including NPY_NOSCALAR).

NPY_ORDER
An enumeration type indicating the element order that an array should be interpreted in. When a brand new array is created, generally only NPY_CORDER and NPY_FORTRANORDER are used, whereas when one or more inputs are provided, the order can be based on them.

NPY_ANYORDER
Fortran order if all the inputs are Fortran, C otherwise.

NPY_CORDER
C order.

NPY_FORTRANORDER
Fortran order.

NPY_KEEPORDER
An order as close to the order of the inputs as possible, even if the input is in neither C nor Fortran order.

NPY_CLIPMODE
A variable type indicating the kind of clipping that should be applied in certain functions.

NPY.Raise
The default for most operations, raises an exception if an index is out of bounds.

NPY_CLIP
Clips an index to the valid range if it is out of bounds.

NPY_WRAP
Wraps an index to the valid range if it is out of bounds.

NPY_CASTING
New in version 1.6. An enumeration type indicating how permissive data conversions should be. This is used by the iterator added in NumPy 1.6, and is intended to be used more broadly in a future version.

NPY_NO_CASTING
Only allow identical types.
NPY_EQUIV_CASTING
Allow identical and casts involving byte swapping.

NPY_SAFE_CASTING
Only allow casts which will not cause values to be rounded, truncated, or otherwise changed.

NPYSAME_KIND_CASTING
Allow any safe casts, and casts between types of the same kind. For example, float64 -> float32 is permitted with this rule.

NPY_UNSAFE_CASTING
Allow any cast, no matter what kind of data loss may occur.

5.5 Array Iterator API

New in version 1.6.

5.5.1 Array Iterator

The array iterator encapsulates many of the key features in ufuncs, allowing user code to support features like output parameters, preservation of memory layouts, and buffering of data with the wrong alignment or type, without requiring difficult coding.

This page documents the API for the iterator. The C-API naming convention chosen is based on the one in the numpy-refactor branch, so will integrate naturally into the refactored code base. The iterator is named NpyIter and functions are named NpyIter_*. 

There is an introductory guide to array iteration which may be of interest for those using this C API. In many instances, testing out ideas by creating the iterator in Python is a good idea before writing the C iteration code.

5.5.2 Converting from Previous NumPy Iterators

The existing iterator API includes functions like PyArrayIter_Check, PyArray_Iter* and PyArray_ITER_*. The multi-iterator array includes PyArray_MultiIter*, PyArray_Broadcast, and PyArray_RemoveSmallest. The new iterator design replaces all of this functionality with a single object and associated API. One goal of the new API is that all uses of the existing iterator should be replaceable with the new iterator without significant effort. In 1.6, the major exception to this is the neighborhood iterator, which does not have corresponding features in this iterator.

Here is a conversion table for which functions to use with the new iterator:
**Iterator Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyArray_IterNew</td>
<td>NpyIter_New</td>
</tr>
<tr>
<td>PyArray_IterAllButAxis</td>
<td>NpyIter_New</td>
</tr>
<tr>
<td>PyArray_BroadcastToShape</td>
<td><strong>NOT SUPPORTED</strong> (Use the support for multiple operands instead.)</td>
</tr>
<tr>
<td>PyArray_IterCheck</td>
<td>Will need to add this in Python exposure</td>
</tr>
<tr>
<td>PyArray_Iter_RESET</td>
<td>NpyIter_Reset</td>
</tr>
<tr>
<td>PyArray_Iter_NEXT</td>
<td>Function pointer from NpyIter_GetIterNext</td>
</tr>
<tr>
<td>PyArray_Iter_DATA</td>
<td>NpyIter_GetDataPtrArray</td>
</tr>
<tr>
<td>PyArray_Iter_GOTO</td>
<td>NpyIter_GotoMultiIndex</td>
</tr>
<tr>
<td>PyArray_Iter_GOTO1D</td>
<td>NpyIter_GotoIndex or NpyIter_GotoIterIndex</td>
</tr>
<tr>
<td>PyArray_Iter_NOTDONE</td>
<td>Return value of iternext function pointer</td>
</tr>
</tbody>
</table>

**Multi-iterator Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyArray_MultiIterNew</td>
<td>NpyIter_MultiNew</td>
</tr>
<tr>
<td>PyArray_MultiIter_RESET</td>
<td>NpyIter_Reset</td>
</tr>
<tr>
<td>PyArray_MultiIter_NEXT</td>
<td>Function pointer from NpyIter_GetIterNext</td>
</tr>
<tr>
<td>PyArray_MultiIter_DATA</td>
<td>NpyIter_GetDataPtrArray</td>
</tr>
<tr>
<td>PyArray_MultiIter_NEXT1D</td>
<td><strong>NOT SUPPORTED</strong> (always lock-step iteration)</td>
</tr>
<tr>
<td>PyArray_MultiIter_GOTO</td>
<td>NpyIter_GotoMultiIndex</td>
</tr>
<tr>
<td>PyArray_MultiIter_GOTO1D</td>
<td>NpyIter_GotoIndex or NpyIter_GotoIterIndex</td>
</tr>
<tr>
<td>PyArray_MultiIter_NOTDONE</td>
<td>Return value of iternext function pointer</td>
</tr>
<tr>
<td>PyArray_Broadcast</td>
<td>Handled by NpyIter_MultiNew</td>
</tr>
<tr>
<td>PyArray_RemoveSmallest</td>
<td>Iterator flag NPY_ITER_EXTERNAL_LOOP</td>
</tr>
</tbody>
</table>

**Other Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyArray_ConvertToCommonType</td>
<td>Iterator flag NPY_ITERCOMMON_DTYPE</td>
</tr>
</tbody>
</table>

### 5.5.3 Simple Iteration Example

The best way to become familiar with the iterator is to look at its usage within the NumPy codebase itself. For example, here is a slightly tweaked version of the code for `PyArray_CountNonzero`, which counts the number of non-zero elements in an array.

```c
npy_intp PyArray_CountNonzero(PyArrayObject* self) {
    /* Nonzero boolean function */
    PyArray_NonzeroFunc* nonzero = PyArray_DESCR(self)->f->nonzero;

    NpyIter* iter;
    NpyIter_IterNextFunc *iternext;
    char** dataptr;
    npy_intp* strideptr,* innersizeptr;

    /* Handle zero-sized arrays specially */
    if (PyArray_SIZE(self) == 0) {
        return 0;
    }

    /* Create and use an iterator to count the nonzeros.
     * flag NPY_ITER_READONLY
     * - The array is never written to.
     * flag NPY_ITER_EXTERNAL_LOOP
     * - Inner loop is done outside the iterator for efficiency.
     * flag NPY_ITER_NPY_ITER_REFS_OK */
```
* Reference types are acceptable.
* order NPY_KEEPORDER
* Visit elements in memory order, regardless of strides.
* This is good for performance when the specific order
  elements are visited is unimportant.
* casting NPY_NO_CASTING
* No casting is required for this operation.

```c
iter = NpyIter_New(self, NPY_ITER_READONLY|
                   NPY_ITER_EXTERNAL_LOOP|
                   NPY_ITER_REFS_OK,
                   NPY_KEEPORDER, NPY_NO_CASTING,
                   NULL);
if (iter == NULL) {
    return -1;
}
```

```c
/*
 * The iternext function gets stored in a local variable
 * so it can be called repeatedly in an efficient manner.
 */
iternext = NpyIter_GetIterNext(iter, NULL);
if (iternext == NULL) {
    NpyIter_Deallocate(iter);
    return -1;
}
```

```c
/* The location of the data pointer which the iterator may update */
dataptr = NpyIter_GetDataPtrArray(iter);
/* The location of the stride which the iterator may update */
strideptr = NpyIter_GetInnerStrideArray(iter);
/* The location of the inner loop size which the iterator may update */
innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);
```

```c
/* The iteration loop */
do {
    /* Get the inner loop data/stride/count values */
    char* data = *dataptr;
    npy_intp stride = *strideptr;
    npy_intp count = *innersizeptr;

    /* This is a typical inner loop for NPY_ITER_EXTERNAL_LOOP */
    while (count--) {
        if (nonzero(data, self)) {
            ++nonzero_count;
        }
        data += stride;
    }

    /* Increment the iterator to the next inner loop */
} while(iternext(iter));
NpyIter_Deallocate(iter);
```

```c
return nonzero_count;
```
5.5.4 Simple Multi-Iteration Example

Here is a simple copy function using the iterator. The `order` parameter is used to control the memory layout of the allocated result, typically `NPY_KEEPORDER` is desired.

```c
PyObject *CopyArray(PyObject *arr, NPY_ORDER order)
{
    NpyIter *iter;
    NpyIter_IterNextFunc *iternext;
    PyObject *op[2], *ret;
    npy_uint32 flags;
    npy_uint32 op_flags[2];
    npy_intp itemsize, *innersizeptr, innerstride;
    char **dataptrarray;

    flags = NPY_ITER_EXTERNAL_LOOP;
    op[0] = arr;
    op[1] = NULL;
    op_flags[0] = NPY_ITER_READONLY;
    op_flags[1] = NPY_ITER_WRITEONLY | NPY_ITER_ALLOCATE;

    iter = NpyIter_MultiNew(2, op, flags, order, NPY_NO_CASTING,
                            op_flags, NULL);  
    if (iter == NULL) {
        return NULL;
    }

    iternext = NpyIter_GetIterNext(iter, NULL);
    innerstride = NpyIter_GetInnerStrideArray(iter)[0];
    itemsize = NpyIter_GetDescrArray(iter)[0]->elsize;

    innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);
    dataptrarray = NpyIter_GetDataPtrArray(iter);

    if (innerstride == itemsize) {
        do {
            memcpy(dataptrarray[1], dataptrarray[0],
                   itemsize * (*innersizeptr));
        } while (iternext(iter));
    }
}```
} else {  
    /* For efficiency, should specialize this based on item size... */  
    npy_intp i;
    do {
        npy_intp size = *innersizeptr;
        char *src = dataaddr[0], *dst = dataaddr[1];
        for (i = 0; i < size; i++, src += innerstride, dst += itemsize) {
            memcpy(dst, src, itemsize);
        }
    } while (iternext(iter));
}

/* Get the result from the iterator object array */
ret = NpyIter_GetOperandArray(iter)[1];
Py_INCREF(ret);

if (NpyIter_Deallocate(iter) != NPY_SUCCEED) {
    Py_DECREF(ret);
    return NULL;
}

return ret;

5.5.5 Iterator Data Types
The iterator layout is an internal detail, and user code only sees an incomplete struct.

NpyIter
This is an opaque pointer type for the iterator. Access to its contents can only be done through the iterator API.

NpyIter_Type
This is the type which exposes the iterator to Python. Currently, no API is exposed which provides access to the values of a Python-created iterator. If an iterator is created in Python, it must be used in Python and vice versa. Such an API will likely be created in a future version.

NpyIter_IterNextFunc
This is a function pointer for the iteration loop, returned by NpyIter_GetIterNext.

NpyIter_GetMultiIndexFunc
This is a function pointer for getting the current iterator multi-index, returned by NpyIter_GetGetMultiIndex.

5.5.6 Construction and Destruction

NpyIter* NpyIter_New(PyArrayObject* op, npy_uint32 flags, NPY_ORDER order, NPY_CASTING casting, PyArray_Descr* dtype)
Creates an iterator for the given numpy array object op.

Flags that may be passed in flags are any combination of the global and per-operand flags documented in NpyIter_MultiNew, except for NPY_ITER_ALLOCATE.

Any of the NPY_ORDER enum values may be passed to order. For efficient iteration, NPY_KEEPORDER is the best option, and the other orders enforce the particular iteration pattern.

Any of the NPY_CASTING enum values may be passed to casting. The values include NPY_NO_CASTING, NPY_EQUIV_CASTING, NPY_SAFE_CASTING, NPY_SAFE_CASTING, and
NPY_UNSAFE_CASTING. To allow the casts to occur, copying or buffering must also be enabled.

If \texttt{dtype} isn’t \texttt{NULL}, then it requires that data type. If copying is allowed, it will make a temporary copy if the data is castable. If \texttt{NPY_ITER_UPDATEIFCOPY} is enabled, it will also copy the data back with another cast upon iterator destruction.

Returns NULL if there is an error, otherwise returns the allocated iterator.

To make an iterator similar to the old iterator, this should work.

\begin{verbatim}
iter = NpyIter_New(op, NPY_ITER_READWRITE,
                  NPY_CORDER, NPY_NO_CASTING, NULL);
\end{verbatim}

If you want to edit an array with aligned double code, but the order doesn’t matter, you would use this.

\begin{verbatim}
dtype = PyArray_DescrFromType(NPY_DOUBLE);
iter = NpyIter_New(op, NPY_ITER_READWRITE|
                  NPY_ITER_BUFFERED| NPY_ITER_NBO| NPY_ITERAligned,
                  NPY_KEEPORDER,
                  NPY_SAME_KIND_CASTING,
                  dtype);
Py_DECREF(dtype);
\end{verbatim}

\begin{verbatim}
NpyIter* NpyIter_MultiNew(npy_intp nop, PyArrayObject** op,
                          npy_uint32 flags, 
                          NPY_ORDER order, NPY_CASTING casting, npy_uint32* op_flags, 
                          PyArray_Desc** op_dtypes)
\end{verbatim}

Creates an iterator for broadcasting the \texttt{nop} array objects provided in \texttt{op}, using regular NumPy broadcasting rules.

Any of the \texttt{NPY_ORDER} enum values may be passed to \texttt{order}. For efficient iteration, \texttt{NPY_KEEPORDER} is the best option, and the other orders enforce the particular iteration pattern. When using \texttt{NPY_KEEPORDER}, if you also want to ensure that the iteration is not reversed along an axis, you should pass the flag \texttt{NPY_ITER_DONT_NEGATE_STRIDES}.

Any of the \texttt{NPY_CASTING} enum values may be passed to \texttt{casting}. The values include \texttt{NPY_NO_CASTING}, \texttt{NPY_EQUIV_CASTING}, \texttt{NPY_SAFE_CASTING}, \texttt{NPY_SAME_KIND_CASTING}, and \texttt{NPY_UNSAFE_CASTING}. To allow the casts to occur, copying or buffering must also be enabled.

If \texttt{op_dtypes} isn’t \texttt{NULL}, it specifies a data type or \texttt{NULL} for each \texttt{op[i]}.

Returns NULL if there is an error, otherwise returns the allocated iterator.

Flags that may be passed in \texttt{flags}, applying to the whole iterator, are:

\begin{itemize}
  \item \texttt{NPY_ITER_C_INDEX}  
    Causes the iterator to track a raveled flat index matching C order. This option cannot be used with \texttt{NPY_ITER_F_INDEX}.
  \item \texttt{NPY_ITER_F_INDEX}  
    Causes the iterator to track a raveled flat index matching Fortran order. This option cannot be used with \texttt{NPY_ITER_C_INDEX}.
  \item \texttt{NPY_ITER_MULTI_INDEX}  
    Causes the iterator to track a multi-index. This prevents the iterator from coalescing axes to produce bigger inner loops.
  \item \texttt{NPY_ITER_EXTERNAL_LOOP}  
    Causes the iterator to skip iteration of the innermost loop, requiring the user of the iterator to handle it.
\end{itemize}
This flag is incompatible with NPY_ITER_C_INDEX, NPY_ITER_F_INDEX, and NPY_ITER_MULTI_INDEX.

NPY_ITER_DONT_NEGATE_STRIDES
This only affects the iterator when NPY_KEEPORDER is specified for the order parameter. By default with NPY_KEEPORDER, the iterator reverses axes which have negative strides, so that memory is traversed in a forward direction. This disables this step. Use this flag if you want to use the underlying memory-ordering of the axes, but don’t want an axis reversed. This is the behavior of numpy.ravel(a, order='K'), for instance.

NPY_ITER_COMMON_DTYPE
Causes the iterator to convert all the operands to a common data type, calculated based on the ufunc type promotion rules. Copying or buffering must be enabled.

If the common data type is known ahead of time, don’t use this flag. Instead, set the requested dtype for all the operands.

NPY_ITER_REFS_OK
Indicates that arrays with reference types (object arrays or structured arrays containing an object type) may be accepted and used in the iterator. If this flag is enabled, the caller must be sure to check whether :cfunc:`NpyIter_IterationNeedsAPI` is true, in which case it may not release the GIL during iteration.

NPY_ITER_ZEROSIZE_OK
Indicates that arrays with a size of zero should be permitted. Since the typical iteration loop does not naturally work with zero-sized arrays, you must check that the IterSize is non-zero before entering the iteration loop.

NPY_ITER_REDUCE_OK
Permits writeable operands with a dimension with zero stride and size greater than one. Note that such operands must be read/write.

When buffering is enabled, this also switches to a special buffering mode which reduces the loop length as necessary to not trample on values being reduced.

Note that if you want to do a reduction on an automatically allocated output, you must use NpyIter_GetOperandArray to get its reference, then set every value to the reduction unit before doing the iteration loop. In the case of a buffered reduction, this means you must also specify the flag NPY_ITER_DELAY_BUFALLOC, then reset the iterator after initializing the allocated operand to prepare the buffers.

NPY_ITER_RANGED
Enables support for iteration of sub-ranges of the full iterindex range [0, NpyIter_IterSize(iter)). Use the function NpyIter_ResetToIterIndexRange to specify a range for iteration.

This flag can only be used with NPY_ITER_EXTERNAL_LOOP when NPY_ITER_BUFFERED is enabled. This is because without buffering, the inner loop is always the size of the innermost iteration dimension, and allowing it to get cut up would require special handling, effectively making it more like the buffered version.

NPY_ITER_BUFFERED
Causes the iterator to store buffering data, and use buffering to satisfy data type, alignment, and byte-order requirements. To buffer an operand, do not specify the NPY_ITER_COPY or NPY_ITER_UPDATEIFCOPY flags, because they will override buffering. Buffering is especially useful for Python code using the iterator, allowing for larger chunks of data at once to amortize the Python interpreter overhead.

If used with NPY_ITER_EXTERNAL_LOOP, the inner loop for the caller may get larger chunks than would be possible without buffering, because of how the strides are laid out.
Note that if an operand is given the flag NPY_ITER_COPY or NPY_ITER_UPDATEIFCOPY, a copy will be made in preference to buffering. Buffering will still occur when the array was broadcast so elements need to be duplicated to get a constant stride.

In normal buffering, the size of each inner loop is equal to the buffer size, or possibly larger if NPY_ITER_GROWINNER is specified. If NPY_ITER_REDUCE_OK is enabled and a reduction occurs, the inner loops may become smaller depending on the structure of the reduction.

NPY_ITER_GROWINNER
When buffering is enabled, this allows the size of the inner loop to grow when buffering isn’t necessary. This option is best used if you’re doing a straight pass through all the data, rather than anything with small cache-friendly arrays of temporary values for each inner loop.

NPY_ITER_DELAY_BUFALLOC
When buffering is enabled, this delays allocation of the buffers until NpyIter_Reset or another reset function is called. This flag exists to avoid wasteful copying of buffer data when making multiple copies of a buffered iterator for multi-threaded iteration.

Another use of this flag is for setting up reduction operations. After the iterator is created, and a reduction output is allocated automatically by the iterator (be sure to use READWRITE access), its value may be initialized to the reduction unit. Use NpyIter_GetOperandArray to get the object. Then, call NpyIter_Reset to allocate and fill the buffers with their initial values.

Flags that may be passed in op_flags[i], where 0 <= i < nop:

NPY_ITER_READWRITE

NPY_ITER_READONLY

NPY_ITER_WRITEONLY
Indicate how the user of the iterator will read or write to op[i]. Exactly one of these flags must be specified per operand.

NPY_ITER_COPY
Allow a copy of op[i] to be made if it does not meet the data type or alignment requirements as specified by the constructor flags and parameters.

NPY_ITER_UPDATEIFCOPY
Triggers NPY_ITER_COPY, and when an array operand is flagged for writing and is copied, causes the data in a copy to be copied back to op[i] when the iterator is destroyed.

If the operand is flagged as write-only and a copy is needed, an uninitialized temporary array will be created and then copied to back to op[i] on destruction, instead of doing the unnecessary copy operation.

NPY_ITER_NBO

NPY_ITER_ALIGNED

NPY_ITER_CONTIG
Causes the iterator to provide data for op[i] that is in native byte order, aligned according to the dtype requirements, contiguous, or any combination.

By default, the iterator produces pointers into the arrays provided, which may be aligned or unaligned, and with any byte order. If copying or buffering is not enabled and the operand data doesn’t satisfy the constraints, an error will be raised.
The contiguous constraint applies only to the inner loop, successive inner loops may have arbitrary pointer changes.

If the requested data type is in non-native byte order, the NBO flag overrides it and the requested data type is converted to be in native byte order.

**NPY_ITER_ALLOCATE**

This is for output arrays, and requires that the flag **NPY_ITER_WRITEONLY** or **NPY_ITER_READWRITE** be set. If op[i] is NULL, creates a new array with the final broadcast dimensions, and a layout matching the iteration order of the iterator.

When op[i] is NULL, the requested data type op_dtypes[i] may be NULL as well, in which case it is automatically generated from the dtypes of the arrays which are flagged as readable. The rules for generating the dtype are the same is for UFuncs. Of special note is handling of byte order in the selected dtype. If there is exactly one input, the input’s dtype is used as is. Otherwise, if more than one input dtypes are combined together, the output will be in native byte order.

After being allocated with this flag, the caller may retrieve the new array by calling NpyIter_GetOperandArray and getting the i-th object in the returned C array. The caller must call Py_INCREF on it to claim a reference to the array.

**NPY_ITER_NO_SUBTYPE**

For use with **NPY_ITER_ALLOCATE**, this flag disables allocating an array subtype for the output, forcing it to be a straight ndarray.

TODO: Maybe it would be better to introduce a function NpyIter_GetWrappedOutput and remove this flag?

**NPY_ITER_NO_BROADCAST**

Ensures that the input or output matches the iteration dimensions exactly.

**NPY_ITER_ARRAYMASK**

New in version 1.7. Indicates that this operand is the mask to use for selecting elements when writing to operands which have the **NPY_ITER_WRITEMASKED** flag applied to them. Only one operand may have **NPY_ITER_ARRAYMASK** flag applied to it.

The data type of an operand with this flag should be either **NPY_BOOL**, **NPY_MASK**, or a struct dtype whose fields are all valid mask dtypes. In the latter case, it must match up with a struct operand being WRITEMASKED, as it is specifying a mask for each field of that array.

This flag only affects writing from the buffer back to the array. This means that if the operand is also **NPY_ITER_READWRITE** or **NPY_ITER_WRITEONLY**, code doing iteration can write to this operand to control which elements will be untouched and which ones will be modified. This is useful when the mask should be a combination of input masks, for example. Mask values can be created with the NpyMask_Create function.

**NPY_ITER_WRITEMASKED**

New in version 1.7. Indicates that only elements which the operand with the ARRAYMASK flag indicates are intended to be modified by the iteration. In general, the iterator does not enforce this, it is up to the code doing the iteration to follow that promise. Code can use the NpyMask_IsExposed inline function to test whether the mask at a particular element allows writing.

When this flag is used, and this operand is buffered, this changes how data is copied from the buffer into the array. A masked copying routine is used, which only copies the elements in the buffer for which NpyMask_IsExposed returns true from the corresponding element in the ARRAYMASK operand.
NpyIter* NpyIter_AdvancedNew(npy_intp *nop, PyArrayObject** op, npy_uint32 flags, NPY_ORDER order, NPY_CASTING casting, npy_uint32* op_flags, PyArray_Descr** op_dtypes, int oa_ndim, int** op_axes, npy_intp* itershape, npy_intp bufsize)

Extends NpyIter_MultiNew with several advanced options providing more control over broadcasting and buffering.

If '\-1/NULL' values are passed to 'oa_ndim, op_axes, itershape, and bufsize', it is equivalent to NpyIter_MultiNew.

The parameter 'oa_ndim', when not zero or '-1', specifies the number of dimensions that will be iterated with customized broadcasting. If it is provided, 'op_axes' must and 'itershape' can also be provided. The 'op_axes' parameter lets you control in detail how the axes of the operand arrays get matched together and iterated. In 'op_axes', you must provide an array of 'nop' pointers to 'oa_ndim'-sized arrays of type 'npy_intp'. If an entry in 'op_axes' is NULL, normal broadcasting rules will apply. In 'op_axes[j][i]' is stored either a valid axis of 'op[j]' or '-1' which means 'newaxis'. Within each 'op_axes[j]' array, axes may not be repeated. The following example is how normal broadcasting applies to a 3-D array, a 2-D array, a 1-D array and a scalar.

**Note:** Before NumPy 1.8 'oa_ndim == 0' was used for signalling that that '\'op_axes and itershape\' are unused. This is deprecated and should be replaced with '-1'. Better backward compatibility may be achieved by using NpyIter_MultiNew for this case.

```c
int oa_ndim = 3; /* # iteration axes */
int op0_axes[] = {0, 1, 2}; /* 3-D operand */
int op1_axes[] = {-1, 0, 1}; /* 2-D operand */
int op2_axes[] = {-1, -1, 0}; /* 1-D operand */
int op3_axes[] = {-1, -1, -1} /* 0-D (scalar) operand */
int* op_axes[] = {op0_axes, op1_axes, op2_axes, op3_axes};
```

The 'itershape' parameter allows you to force the iterator to have a specific iteration shape. It is an array of length 'oa_ndim'. When an entry is negative, its value is determined from the operands. This parameter allows automatically allocated outputs to get additional dimensions which don’t match up with any dimension of an input.

If 'buffersize' is zero, a default buffer size is used, otherwise it specifies how big of a buffer to use. Buffers which are powers of 2 such as 4096 or 8192 are recommended.

Returns NULL if there is an error, otherwise returns the allocated iterator.

NpyIter* NpyIter_Copy(NpyIter* iter)

Makes a copy of the given iterator. This function is provided primarily to enable multi-threaded iteration of the data.

**TODO:** Move this to a section about multithreaded iteration.

The recommended approach to multithreaded iteration is to first create an iterator with the flags NPY_ITER_EXTERNAL_LOOP, NPY_ITER_RANGED, NPY_ITER_BUFFERED, NPY_ITER_DELAY_BUFALLOC, and possibly NPY_ITER_GROWINNER. Create a copy of this iterator for each thread (minus one for the first iterator). Then, take the iteration index range [0, NpyIter_GetIterSize(iter)) and split it up into tasks, for example using a TBB parallel_for loop. When a thread gets a task to execute, it then uses its copy of the iterator by calling NpyIter_ResetToIterIndexRange and iterating over the full range.

When using the iterator in multi-threaded code or in code not holding the Python GIL, care must be taken to only call functions which are safe in that context. NpyIter.Copy cannot be safely called without the Python GIL, because it increments Python references. The Reset* and some other functions may be safely called by passing in the 'errmsg' parameter as non-NULL, so that the functions will pass back errors through it instead of setting a Python exception.
int NpyIter_RemoveAxis(NpyIter* iter, int axis)

Removes an axis from iteration. This requires that NPY_ITER_MULTI_INDEX was set for iterator creation, and does not work if buffering is enabled or an index is being tracked. This function also resets the iterator to its initial state.

This is useful for setting up an accumulation loop, for example. The iterator can first be created with all the dimensions, including the accumulation axis, so that the output gets created correctly. Then, the accumulation axis can be removed, and the calculation done in a nested fashion.

**WARNING:** This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

Returns NPY_SUCCEED or NPY_FAIL.

int NpyIter_RemoveMultiIndex(NpyIter* iter)

If the iterator is tracking a multi-index, this strips support for them, and does further iterator optimizations that are possible if multi-indices are not needed. This function also resets the iterator to its initial state.

**WARNING:** This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

After calling this function, :cfunc:`NpyIter_HasMultiIndex`(iter) will return false.

Returns NPY_SUCCEED or NPY_FAIL.

int NpyIter_EnableExternalLoop(NpyIter* iter)

If NpyIter_RemoveMultiIndex was called, you may want to enable the flag NPY_ITER_EXTERNAL_LOOP. This flag is not permitted together with NPY_ITER_MULTI_INDEX, so this function is provided to enable the feature after NpyIter_RemoveMultiIndex is called. This function also resets the iterator to its initial state.

**WARNING:** This function changes the internal logic of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

Returns NPY_SUCCEED or NPY_FAIL.

int NpyIter_Deallocate(NpyIter* iter)

Deallocates the iterator object. This additionally frees any copies made, triggering UPDATEIFCOPY behavior where necessary.

Returns NPY_SUCCEED or NPY_FAIL.

int NpyIter_Reset(NpyIter* iter, char** errmsg)

Resets the iterator back to its initial state, at the beginning of the iteration range.

Returns NPY_SUCCEED or NPY_FAIL. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

int NpyIter_ResetToIterIndexRange(NpyIter* iter, npy_intp istart, npy_intp iend, char** errmsg)

Resets the iterator and restricts it to the iterindex range [istart, iend). See NpyIter_Copy for an explanation of how to use this for multi-threaded iteration. This requires that the flag NPY_ITER_RANGED was passed to the iterator constructor.

If you want to reset both the iterindex range and the base pointers at the same time, you can do the following to avoid extra buffer copying (be sure to add the return code error checks when you copy this code).

```c
/* Set to a trivial empty range */
NpyIter_ResetToIterIndexRange(iter, 0, 0);
/* Set the base pointers */
NpyIter_ResetBasePointers(iter, baseptrs);
/* Set to the desired range */
NpyIter_ResetToIterIndexRange(iter, istart, iend);
```
Returns `NPY_SUCCEED` or `NPY_FAIL`. If `errmsg` is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, `*errmsg` is set to an error message. When `errmsg` is non-NULL, the function may be safely called without holding the Python GIL.

```c
int NpyIter_ResetBasePointers (NpyIter *iter, char** baseptrs, char** errmsg)
```

Resets the iterator back to its initial state, but using the values in `baseptrs` for the data instead of the pointers from the arrays being iterated. This function is intended to be used, together with the `op_axes` parameter, by nested iteration code with two or more iterators.

Returns `NPY_SUCCEED` or `NPY_FAIL`. If `errmsg` is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, `*errmsg` is set to an error message. When `errmsg` is non-NULL, the function may be safely called without holding the Python GIL.

**TODO:** Move the following into a special section on nested iterators.

Creating iterators for nested iteration requires some care. All the iterator operands must match exactly, or the calls to `NpyIter_ResetBasePointers` will be invalid. This means that automatic copies and output allocation should not be used haphazardly. It is possible to still use the automatic data conversion and casting features of the iterator by creating one of the iterators with all the conversion parameters enabled, then grabbing the allocated operands with the `NpyIter_GetOperandArray` function and passing them into the constructors for the rest of the iterators.

**WARNING:** When creating iterators for nested iteration, the code must not use a dimension more than once in the different iterators. If this is done, nested iteration will produce out-of-bounds pointers during iteration.

**WARNING:** When creating iterators for nested iteration, buffering can only be applied to the innermost iterator. If a buffered iterator is used as the source for `baseptrs`, it will point into a small buffer instead of the array and the inner iteration will be invalid.

The pattern for using nested iterators is as follows.

```c
NpyIter *iter1, *iter1;
NpyIter_IterNextFunc *iternext1, *iternext2;
char **dataptrs1;

/*
 * With the exact same operands, no copies allowed, and
 * no axis in op_axes used both in iter1 and iter2.
 * Buffering may be enabled for iter2, but not for iter1.
 */
iter1 = ...; iter2 = ...;

iternext1 = NpyIter_GetIterNext(iter1);
iternext2 = NpyIter_GetIterNext(iter2);
dataptrs1 = NpyIter_GetDataPtrArray(iter1);

do {
    NpyIter_ResetBasePointers(iter2, dataptrs1);
    do {
        /* Use the iter2 values */
    } while (iternext2(iter2));
} while (iternext1(iter1));
```

```c
int NpyIter_GotoMultiIndex (NpyIter* iter, npy_intp* multi_index)
```

Adjusts the iterator to point to the `ndim` indices pointed to by `multi_index`. Returns an error if a multi-index is not being tracked, the indices are out of bounds, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.
int **NpyIter_GotoIndex (NpyIter* iter, npy_intp index)
    Adjusts the iterator to point to the index specified. If the iterator was constructed with the flag
    NPY_ITER_C_INDEX, index is the C-order index, and if the iterator was constructed with the flag
    NPY_ITER_F_INDEX, index is the Fortran-order index. Returns an error if there is no index being tracked,
    the index is out of bounds, or inner loop iteration is disabled.
    Returns NPY_SUCCEED or NPY_FAIL.

npy_intp NpyIter_GetIterSize (NpyIter* iter)
    Returns the number of elements being iterated. This is the product of all the dimensions in the shape.

npy_intp NpyIter_GetIterIndex (NpyIter* iter)
    Gets the iterindex of the iterator, which is an index matching the iteration order of the iterator.

void NpyIter_GetIterIndexRange (NpyIter* iter, npy_intp* istart, npy_intp* iend)
    Gets the iterindex sub-range that is being iterated. If NPY_ITER_RANGED was not specified, this always
    returns the range [0, NpyIter_IterSize(iter)).

int NpyIter_GotoIterIndex (NpyIter* iter, npy_intp iterindex)
    Adjusts the iterator to point to the iterindex specified. The IterIndex is an index matching the iteration
    order of the iterator. Returns an error if the iterindex is out of bounds, buffering is enabled, or inner loop
    iteration is disabled.
    Returns NPY_SUCCEED or NPY_FAIL.

npy_bool NpyIter_HasDelayedBufAlloc (NpyIter* iter)
    Returns 1 if the flag NPY_ITER_DELAY_BUFALLOC was passed to the iterator constructor, and no call to one
    of the Reset functions has been done yet, 0 otherwise.

npy_bool NpyIter_HasExternalLoop (NpyIter* iter)
    Returns 1 if the caller needs to handle the inner-most 1-dimensional loop, or 0 if the iterator han-
    dles all looping. This is controlled by the constructor flag NPY_ITER_EXTERNAL_LOOP or
    NpyIter_EnableExternalLoop.

npy_bool NpyIter_HasMultiIndex (NpyIter* iter)
    Returns 1 if the iterator was created with the NPY_ITER_MULTI_INDEX flag, 0 otherwise.

npy_bool NpyIter_HasIndex (NpyIter* iter)
    Returns 1 if the iterator was created with the NPY_ITER_C_INDEX or NPY_ITER_F_INDEX flag, 0 other-
    wise.

npy_bool NpyIter_RequiresBuffering (NpyIter* iter)
    Returns 1 if the iterator requires buffering, which occurs when an operand needs conversion or alignment and
    so cannot be used directly.

npy_bool NpyIter_IsBuffered (NpyIter* iter)
    Returns 1 if the iterator was created with the NPY_ITER_BUFFERED flag, 0 otherwise.

npy_bool NpyIter_IsGrowInner (NpyIter* iter)
    Returns 1 if the iterator was created with the NPY_ITER_GROWINNER flag, 0 otherwise.

npy_intp NpyIter_GetBufferSize (NpyIter* iter)
    If the iterator is buffered, returns the size of the buffer being used, otherwise returns 0.

int NpyIter_GetNDim (NpyIter* iter)
    Returns the number of dimensions being iterated. If a multi-index was not requested in the iterator construc-
    tor, this value may be smaller than the number of dimensions in the original objects.

int NpyIter_GetNOp (NpyIter* iter)
    Returns the number of operands in the iterator.
When NPY_ITER_USE_MASKNA is used on an operand, a new operand is added to the end of the operand list in the iterator to track that operand’s NA mask. Thus, this equals the number of construction operands plus the number of operands for which the flag NPY_ITER_USE_MASKNA was specified.

```c
int NpyIter_GetFirstMaskNAOp (NpyIter* iter)
```

New in version 1.7. Returns the index of the first NA mask operand in the array. This value is equal to the number of operands passed into the constructor.

```c
npy_intp* NpyIter_GetAxisStrideArray (NpyIter* iter, int axis)
```

Gets the array of strides for the specified axis. Requires that the iterator be tracking a multi-index, and that buffering not be enabled.

This may be used when you want to match up operand axes in some fashion, then remove them with NpyIter_RemoveAxis to handle their processing manually. By calling this function before removing the axes, you can get the strides for the manual processing.

Returns NULL on error.

```c
int NpyIter_GetShape (NpyIter* iter, npy_intp* outshape)
```

Returns the broadcast shape of the iterator in `outshape`. This can only be called on an iterator which is tracking a multi-index.

Returns NPY_SUCCEED or NPY_FAIL.

```c
PyArray_Descr** NpyIter_GetDescrArray (NpyIter* iter)
```

This gives back a pointer to the nopt data type Descrs for the objects being iterated. The result points into `iter`, so the caller does not gain any references to the Descrs.

This pointer may be cached before the iteration loop, calling `iternext` will not change it.

```c
PyObject** NpyIter_GetOperandArray (NpyIter* iter)
```

This gives back a pointer to the nopt operand PyObjects that are being iterated. The result points into `iter`, so the caller does not gain any references to the PyObjects.

```c
npy_int8* NpyIter_GetMaskNAIndexArray (NpyIter* iter)
```

New in version 1.7. This gives back a pointer to the nopt indices which map construction operands with NPY_ITER_USE_MASKNA flagged to their corresponding NA mask operands and vice versa. For operands which were not flagged with NPY_ITER_USE_MASKNA, this array contains negative values.

```c
PyObject* NpyIter_GetIterView (NpyIter* iter, npy_intp i)
```

This gives back a reference to a new ndarray view, which is a view into the i-th object in the array as returned by `NpyIter_GetOperandArray()`, whose dimensions and strides match the internal optimized iteration pattern. A C-order iteration of this view is equivalent to the iterator’s iteration order.

For example, if an iterator was created with a single array as its input, and it was possible to rearrange all its axes and then collapse it into a single strided iteration, this would return a view that is a one-dimensional array.

```c
void NpyIter_GetReadFlags (NpyIter* iter, char* outreadflags)
```

Fills nopt flags. Sets `outreadflags[i]` to 1 if `op[i]` can be read from, and to 0 if not.

```c
void NpyIter_GetWriteFlags (NpyIter* iter, char* outwriteflags)
```

Fills nopt flags. Sets `outwriteflags[i]` to 1 if `op[i]` can be written to, and to 0 if not.

```c
int NpyIter_CreateCompatibleStrides (NpyIter* iter, npy_intp itemsize, npy_intp* outstrides)
```

Builds a set of strides which are the same as the strides of an output array created using the NPY_ITER_ALLOCATE flag, where NULL was passed for op_axes. This is for data packed contiguously, but not necessarily in C or Fortran order. This should be used together with NpyIter_GetShape and NpyIter_GetNDim with the flag NPY_ITER_MULTI_INDEX passed into the constructor.

A use case for this function is to match the shape and layout of the iterator and tack on one or more dimensions. For example, in order to generate a vector per input value for a numerical gradient, you pass in ndim*itemsize for `itemsize`, then add another dimension to the end with size ndim and stride itemsize. To do the Hessian matrix,
you do the same thing but add two dimensions, or take advantage of the symmetry and pack it into 1 dimension
with a particular encoding.

This function may only be called if the iterator is tracking a multi-index and if
NPY_ITER_DONT_NEGATE_STRIDES was used to prevent an axis from being iterated in reverse or-
der.

If an array is created with this method, simply adding ‘itemsize’ for each iteration will traverse the new array
matching the iterator.

Returns NPY_SUCCEED or NPY_FAIL.

\texttt{npy\\_bool NpyIter\\_IsFirstVisit (NpyIter\\* iter, int iop)}

New in version 1.7. Checks to see whether this is the first time the elements of the specified reduction operand
which the iterator points at are being seen for the first time. The function returns a reasonable answer for
reduction operands and when buffering is disabled. The answer may be incorrect for buffered non-reduction
operands.

This function is intended to be used in EXTERNAL\_LOOP mode only, and will produce some wrong answers
when that mode is not enabled.

If this function returns true, the caller should also check the inner loop stride of the operand, because if that
stride is 0, then only the first element of the innermost external loop is being visited for the first time.

\textbf{WARNING}: For performance reasons, ‘iop’ is not bounds-checked, it is not confirmed that ‘iop’ is actually a
reduction operand, and it is not confirmed that EXTERNAL\_LOOP mode is enabled. These checks are the
responsibility of the caller, and should be done outside of any inner loops.

5.5.7 Functions For Iteration

\texttt{NpyIter\\_IterNextFunc* NpyIter\\_GetIterNext (NpyIter\\* iter, char** errmsg)}

Returns a function pointer for iteration. A specialized version of the function pointer may be calculated by this
function instead of being stored in the iterator structure. Thus, to get good performance, it is required that the
function pointer be saved in a variable rather than retrieved for each loop iteration.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is
returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely
called without holding the Python GIL.

The typical looping construct is as follows.

\begin{verbatim}
NpyIter\_IterNextFunc *iternext = NpyIter\_GetIterNext(iter, NULL);
char** dataptr = NpyIter\_GetDataPtrArray(iter);

\texttt{do \{ /* use the addresses dataptr[0], ... dataptr[nop-1] */
\texttt{\} while(iternext(iter));
\end{verbatim}

When NPY\_ITER\_EXTERNAL\_LOOP is specified, the typical inner loop construct is as follows.

\begin{verbatim}
NpyIter\_IterNextFunc *iternext = NpyIter\_GetIterNext(iter, NULL);
char** dataptr = NpyIter\_GetDataPtrArray(iter);
npy_intp* stride = NpyIter\_GetInnerStrideArray(iter);
npy_intp* size_ptr = NpyIter\_GetInnerLoopSizePtr(iter), size;
npy_intp iop, nop = NpyIter\_GetNOp(iter);

\texttt{do \{
\texttt{size = *size_ptr;
\texttt{\} while (size--) { /* use the addresses dataptr[0], ... dataptr[nop-1] */
\end{verbatim}
for (iop = 0; iop < nop; ++iop) {
    dataptr[iop] += stride[iop];
}
} while (iternext());

Observe that we are using the dataptr array inside the iterator, not copying the values to a local temporary. This is possible because when iternext() is called, these pointers will be overwritten with fresh values, not incrementally updated.

If a compile-time fixed buffer is being used (both flags NPY_ITER_BUFFERED and NPY_ITER_EXTERNAL_LOOP), the inner size may be used as a signal as well. The size is guaranteed to become zero when iternext() returns false, enabling the following loop construct. Note that if you use this construct, you should not pass NPY_ITER_GROWINNER as a flag, because it will cause larger sizes under some circumstances.

/* The constructor should have buffersize passed as this value */
#define FIXED_BUFFER_SIZE 1024

NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char **dataptr = NpyIter_GetDataPtrArray(iter);
npy_intp *stride = NpyIter_GetInnerStrideArray(iter);
npy_intp *size_ptr = NpyIter_GetInnerLoopSizePtr(iter), size;
npy_intp i, iop, nop = NpyIter_GetNOp(iter);

/* One loop with a fixed inner size */
size = *size_ptr;
while (size == FIXED_BUFFER_SIZE) {
    /* This loop could be manually unrolled by a factor 
       which divides into FIXED_BUFFER_SIZE 
    */
    for (i = 0; i < FIXED_BUFFER_SIZE; ++i) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
    iternext();
    size = *size_ptr;
}

/* Finish-up loop with variable inner size */
if (size > 0) do {
    size = *size_ptr;
    while (size--) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
} while (iternext());

NpyIter_GetMultiIndexFunc *NpyIter_GetGetMultiIndex(NpyIter* iter, char** errmsg)
Returns a function pointer for getting the current multi-index of the iterator. Returns NULL if the iterator is not tracking a multi-index. It is recommended that this function pointer be cached in a local variable before the iteration loop.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when NPY_FAIL is
NumPy Reference, Release 1.8.1

returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely
called without holding the Python GIL.

char** NpyIter_GetDataPtrArray (NpyIter* iter)
This gives back a pointer to the nop data pointers. If NPY_ITEREXTERNAL_LOOP was not specified, each
data pointer points to the current data item of the iterator. If no inner iteration was specified, it points to the first
data item of the inner loop.

This pointer may be cached before the iteration loop, calling iternext will not change it. This function may
be safely called without holding the Python GIL.

char** NpyIter_GetInitialDataPtrArray (NpyIter* iter)
Gets the array of data pointers directly into the arrays (never into the buffers), corresponding to iteration index
0.

These pointers are different from the pointers accepted by NpyIter_ResetBasePointers, because the
direction along some axes may have been reversed.

This function may be safely called without holding the Python GIL.

npy_intp* NpyIter_GetIndexPtr (NpyIter* iter)
This gives back a pointer to the index being tracked, or NULL if no index is being tracked. It is only useable if
one of the flags NPY_ITER_C_INDEX or NPY_ITER_F_INDEX were specified during construction.

When the flag NPY_ITEREXTERNAL_LOOP is used, the code needs to know the parameters for doing the inner
loop. These functions provide that information.

npy_intp* NpyIter_GetInnerStrideArray (NpyIter* iter)
Returns a pointer to an array of the nop strides, one for each iterated object, to be used by the inner loop.

This pointer may be cached before the iteration loop, calling iternext will not change it. This function may
be safely called without holding the Python GIL.

npy_intp* NpyIter_GetInnerLoopSizePtr (NpyIter* iter)
Returns a pointer to the number of iterations the inner loop should execute.

This address may be cached before the iteration loop, calling iternext will not change it. The value itself
may change during iteration, in particular if buffering is enabled. This function may be safely called without
holding the Python GIL.

void NpyIter_GetInnerFixedStrideArray (NpyIter* iter, npy_intp* out_strides)
Gets an array of strides which are fixed, or will not change during the entire iteration. For strides that may
change, the value NPY_MAX_INTP is placed in the stride.

Once the iterator is prepared for iteration (after a reset if NPY_DELAY_BUFALLOC was used), call this to get
the strides which may be used to select a fast inner loop function. For example, if the stride is 0, that means
the inner loop can always load its value into a variable once, then use the variable throughout the loop, or if the
stride equals the itemsize, a contiguous version for that operand may be used.

This function may be safely called without holding the Python GIL.

5.6 UFunc API

5.6.1 Constants

UFUNC_ERR {HANDLER}
{HANDLER} can be IGNORE, WARN, RAISE, or CALL
UFUNC_{THING}_{ERR}  
{THING} can be MASK, SHIFT, or FPE, and {ERR} can be DIVIDEBYZERO, OVERFLOW, UNDFLOW, and INVALID.

PyUFunc_{VALUE}  
{VALUE} can be One (1), Zero (0), or None (-1)

5.6.2 Macros

NPy_LOOP_BEGIN_THREADS  
Used in universal function code to only release the Python GIL if loop->obj is not true (i.e. this is not an OBJECT array loop). Requires use of NPY_BEGIN_THREADS_DEF in variable declaration area.

NPy_LOOP_END_THREADS  
Used in universal function code to re-acquire the Python GIL if it was released (because loop->obj was not true).

UFUNC_CHECK_ERROR (loop)  
A macro used internally to check for errors and goto fail if found. This macro requires a fail label in the current code block. The loop variable must have at least members (obj, errormask, and errorobj). If loop ->obj is nonzero, then PyErr_Occurred () is called (meaning the GIL must be held). If loop ->obj is zero, then if loop ->errormask is nonzero, PyUFunc_checkfperr is called with arguments loop ->errormask and loop ->errobj. If the result of this check of the IEEE floating point registers is true then the code redirects to the fail label which must be defined.

UFUNC_CHECK_STATUS (ret)  
A macro that expands to platform-dependent code. The ret variable can can be any integer. The UFUNC_FPE_{ERR} bits are set in ret according to the status of the corresponding error flags of the floating point processor.

5.6.3 Functions

PyObject* PyUFunc_FromFuncAndData(PyUFuncGenericFunction* func,
void** data, char* types, int ntypes, int nin, int nout, int identity,
char* name, char* doc, int check_return)
Create a new broadcasting universal function from required variables. Each ufunc builds around the notion of an element-by-element operation. Each ufunc object contains pointers to 1-d loops implementing the basic functionality for each supported type.

Note: The func, data, types, name, and doc arguments are not copied by PyUFunc_FromFuncAndData. The caller must ensure that the memory used by these arrays is not freed as long as the ufunc object is alive.

Parameters

• func – Must to an array of length ntypes containing PyUFuncGenericFunction items. These items are pointers to functions that actually implement the underlying (element-by-element) function N times.

• data – Should be NULL or a pointer to an array of size ntypes . This array may contain arbitrary extra-data to be passed to the corresponding 1-d loop function in the func array.
• **types** – Must be of length \((nin + nout) \times ntypes\), and it contains the data-types (built-in only) that the corresponding function in the `func` array can deal with.

• **ntypes** – How many different data-type “signatures” the ufunc has implemented.

• **nin** – The number of inputs to this operation.

• **nout** – The number of outputs

• **name** – The name for the ufunc. Specifying a name of ‘add’ or ‘multiply’ enables a special behavior for integer-typed reductions when no dtype is given. If the input type is an integer (or boolean) data type smaller than the size of the int_ data type, it will be internally upcast to the int_ (or uint) data type.

• **doc** – Allows passing in a documentation string to be stored with the ufunc. The documentation string should not contain the name of the function or the calling signature as that will be dynamically determined from the object and available when accessing the `__doc__` attribute of the ufunc.

• **check_return** – Unused and present for backwards compatibility of the C-API. A corresponding `check_return` integer does exist in the ufunc structure and it does get set with this value when the ufunc object is created.

```c
int PyUFunc_RegisterLoopForType(PyUFuncObject* ufunc,

int usertype, PyUFuncGenericFunction function, int* arg_types, void* data)
```

This function allows the user to register a 1-d loop with an already-created ufunc to be used whenever the ufunc is called with any of its input arguments as the user-defined data-type. This is needed in order to make ufuncs work with built-in data-types. The data-type must have been previously registered with the numpy system. The loop is passed in as `function`. This loop can take arbitrary data which should be passed in as `data`. The data-types the loop requires are passed in as `arg_types` which must be a pointer to memory at least as large as `ufunc->args`.

```c
int PyUFunc_RegisterLoopForDescr(PyUFuncObject* ufunc,

PyArray_Descr* userdtype, PyUFuncGenericFunction function,

PyArray_Descr** arg_dtypes, void* data)
```

This function behaves like `PyUFunc_RegisterLoopForType` above, except that it allows the user to register a 1-d loop using PyArray_Descr objects instead of dtype type num values. This allows a 1-d loop to be registered for structured array data-types and custom data-types instead of scalar data-types.

```c
int PyUFunc_ReplacedLoopBySignature(PyUFuncObject* ufunc,

PyUFuncGenericFunction newfunc, int* signature,

PyUFuncGenericFunction* oldfunc)
```

Replace a 1-d loop matching the given `signature` in the already-created `ufunc` with the new 1-d loop `newfunc`. Return the old 1-d loop function in `oldfunc`. Return 0 on success and -1 on failure. This function works only with built-in types (use `PyUFunc_RegisterLoopForType` for user-defined types). A signature is an array of data-type numbers indicating the inputs followed by the outputs assumed by the 1-d loop.

```c
int PyUFunc_GenericFunction(PyUFuncObject* self,

PyObject* args, PyObject* kwds, PyArrayObject** mps)
```

A generic ufunc call. The ufunc is passed in as `self`, the arguments to the ufunc as `args` and `kwds`. The `mps` argument is an array of `PyArrayObject` pointers whose values are discarded and which receive the converted input arguments as well as the ufunc outputs when success is returned. The user is responsible for managing
this array and receives a new reference for each array in mps. The total number of arrays in mps is given by self
->min + self->nout.

Returns 0 on success, -1 on error.

int PyUFunc_checkfperr (int errmask, PyObject* errobj)
A simple interface to the IEEE error-flag checking support. The errmask argument is a mask of
UFUNC_MASK_{ERR} bitmasks indicating which errors to check for (and how to check for them). The
errobj must be a Python tuple with two elements: a string containing the name which will be used in any
communication of error and either a callable Python object (call-back function) or Py_None. The callable
object will only be used if UFUNC_ERR_CALL is set as the desired error checking method. This routine
manages the GIL and is safe to call even after releasing the GIL. If an error in the IEEE-compatibile hardware
is determined a -1 is returned, otherwise a 0 is returned.

void PyUFunc_clearfperr ()
Clear the IEEE error flags.

void PyUFunc_GetPyValues(char* name, int* bufsize,
int* errmask, PyObject** errobj)
Get the Python values used for ufunc processing from the thread-local storage area unless the defaults have
been set in which case the name lookup is bypassed. The name is placed as a string in the first element of
*errobj. The second element is the looked-up function to call on error callback. The value of the looked-up
buffer-size to use is passed into bufsize, and the value of the error mask is placed into errmask.

5.6.4 Generic functions

At the core of every ufunc is a collection of type-specific functions that defines the basic functionality for each of the
supported types. These functions must evaluate the underlying function \( N \geq 1 \) times. Extra-data may be passed in
that may be used during the calculation. This feature allows some general functions to be used as these basic looping
functions. The general function has all the code needed to point variables to the right place and set up a function call.
The general function assumes that the actual function to call is passed in as the extra data and calls it with the correct
values. All of these functions are suitable for placing directly in the array of functions stored in the functions member
of the PyUFuncObject structure.

void PyUFunc_f_f_As_d_d(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_d_d(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_f_f(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_g_g(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_F_F_As_D_D(char** args, npy_intp* dimensions,
Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking one input argument and returning one output. This function is passed in func. The letters correspond to dtypechar's of the supported data types (e - half, f - float, d - double, g - long double, F - cfloat, D - cdouble, G - clongdouble). The argument func must support the same signature. The _As_X_X variants assume ndarrays of one data type but cast the values to use an underlying function that takes a different data type. Thus, PyUFunc_f_f_As_d_d uses ndarrays of data type NPY_FLOAT but calls out to a C-function that takes double and returns double.
void PyUFunc_FF_F_As_DD_D(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_DD_D(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_FF_F(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_GG_G(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_ee_e(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_ee_e_As_ff_f(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_ee_e_As_dd_d(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)

void PyUFunc_O_O(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_OO_O(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)

void PyUFunc_O_O_method(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking two
input arguments and returning one output. The underlying function to call is passed in as func. The letters
correspond to dtypechar’s of the specific data type supported by the general-purpose function. The argument
func must support the corresponding signature. The _As_XX_X variants assume ndarrays of one data type
but cast the values at each iteration of the loop to use the underlying function that takes a different data type.

void PyUFunc_O_O(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_OO_O(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)
void PyUFunc_O_O_method(char** args, npy_intp* dimensions,
npy_intp* steps, void* func)

One-input, one-output, and two-input, one-output core 1-d functions for the NPY_OBJECT data type. These
functions handle reference count issues and return early on error. The actual function to call is func
and it must accept calls with the signature (PyObject*) (PyObject*) for PyUFunc_O_O or
(PyObject*) (PyObject *, PyObject *) for PyUFunc_OO_O.

This general purpose 1-d core function assumes that func is a string representing a method of the input object.
For each iteration of the loop, the Python object is extracted from the array and its func method is called
returning the result to the output array.
void PyUFunc_OO_O_method(char** args, npy_intp* dimensions,
                         npy_intp* steps, void* func)

This general purpose 1-d core function assumes that func is a string representing a method of the input object that takes one argument. The first argument in args is the method whose function is called, the second argument in args is the argument passed to the function. The output of the function is stored in the third entry of args.

void PyUFunc_On_Om(char** args, npy_intp* dimensions,
                    npy_intp* steps, void* func)

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case func is a pointer to a PyUFunc_PyFuncData structure which has definition

PyUFunc_PyFuncData

typedef struct
{
  int nin;
  int nout;
  PyObject *callable;
} PyUFunc_PyFuncData;

At each iteration of the loop, the nin input objects are extracted from their object arrays and placed into an argument tuple, the Python callable is called with the input arguments, and the nout outputs are placed into their object arrays.

5.6.5 Importing the API

PY_UFUNC_UNIQUE_SYMBOL

NO_IMPORT_UFUNC

void import_ufunc (void)

These are the constants and functions for accessing the ufunc C-API from extension modules in precisely the same way as the array C-API can be accessed. The import_ufunc() function must always be called (in the initialization subroutine of the extension module). If your extension module is in one file then that is all that is required. The other two constants are useful if your extension module makes use of multiple files. In that case, define PY_UFUNC_UNIQUE_SYMBOL to something unique to your code and then in source files that do not contain the module initialization function but still need access to the UFUNC API, define PY_UFUNC_UNIQUE_SYMBOL to the same name used previously and also define NO_IMPORT_UFUNC.

The C-API is actually an array of function pointers. This array is created (and pointed to by a global variable) by import_ufunc. The global variable is either statically defined or allowed to be seen by other files depending on the state of PY_UFUNC_UNIQUE_SYMBOL and NO_IMPORT_UFUNC.

5.7 Generalized Universal Function API

There is a general need for looping over not only functions on scalars but also over functions on vectors (or arrays), as explained on http://scipy.org/scipy/numpy/wiki/GeneralLoopingFunctions. We propose to realize this concept by generalizing the universal functions (ufuncs), and provide a C implementation that adds ~500 lines to the numpy code base. In current (specialized) ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version supports “sub-array” by “sub-array” operations. The Perl vector library PDL provides a similar functionality and its terms are re-used in the following.
Each generalized ufunc has information associated with it that states what the “core” dimensionality of the inputs is, as well as the corresponding dimensionality of the outputs (the element-wise ufuncs have zero core dimensions). The list of the core dimensions for all arguments is called the “signature” of a ufunc. For example, the ufunc numpy.add has signature ((), ()->()), defining two scalar inputs and one scalar output.

Another example is (see the GeneralLoopingFunctions page) the function inner1d(a, b) with a signature of (i), (i)->(). This applies the inner product along the last axis of each input, but keeps the remaining indices intact. For example, where a is of shape (3, 5, N) and b is of shape (5, N), this will return an output of shape (3, 5). The underlying elementary function is called 3*5 times. In the signature, we specify one core dimension (i) for each input and zero core dimensions () for the output, since it takes two 1-d arrays and returns a scalar. By using the same name i, we specify that the two corresponding dimensions should be of the same size (or one of them is of size 1 and will be broadcasted).

The dimensions beyond the core dimensions are called “loop” dimensions. In the above example, this corresponds to (3, 5).

The usual numpy “broadcasting” rules apply, where the signature determines how the dimensions of each input/output object are split into core and loop dimensions:

1. While an input array has a smaller dimensionality than the corresponding number of core dimensions, 1’s are prepended to its shape.
2. The core dimensions are removed from all inputs and the remaining dimensions are broadcasted; defining the loop dimensions.
3. The output is given by the loop dimensions plus the output core dimensions.

### 5.7.1 Definitions

**Elementary Function**

Each ufunc consists of an elementary function that performs the most basic operation on the smallest portion of array arguments (e.g. adding two numbers is the most basic operation in adding two arrays). The ufunc applies the elementary function multiple times on different parts of the arrays. The input/output of elementary functions can be vectors; e.g., the elementary function of inner1d takes two vectors as input.

**Signature**

A signature is a string describing the input/output dimensions of the elementary function of a ufunc. See section below for more details.

**Core Dimension**

The dimensionality of each input/output of an elementary function is defined by its core dimensions (zero core dimensions correspond to a scalar input/output). The core dimensions are mapped to the last dimensions of the input/output arrays.

**Dimension Name**

A dimension name represents a core dimension in the signature. Different dimensions may share a name, indicating that they are of the same size (or are broadcastable).

**Dimension Index**

A dimension index is an integer representing a dimension name. It enumerates the dimension names according to the order of the first occurrence of each name in the signature.

### 5.7.2 Details of Signature

The signature defines “core” dimensionality of input and output variables, and thereby also defines the contraction of the dimensions. The signature is represented by a string of the following format:
- Core dimensions of each input or output array are represented by a list of dimension names in parentheses, \((i_1, \ldots, i_N)\); a scalar input/output is denoted by \((\))\). Instead of \(i_1, i_2\), etc, one can use any valid Python variable name.

- Dimension lists for different arguments are separated by ",, "\). Input/output arguments are separated by "->".

- If one uses the same dimension name in multiple locations, this enforces the same size (or broadcastable size) of the corresponding dimensions.

The formal syntax of signatures is as follows:

\[
\text{<Signature>} ::= \text{<Input arguments> "->" <Output arguments>}
\]

\[
\text{<Input arguments>} ::= \text{<Argument list>}
\]

\[
\text{<Output arguments>} ::= \text{<Argument list>}
\]

\[
\text{<Argument list>} ::= \text{nil} | \text{<Argument> | <Argument> "," <Argument list>}
\]

\[
\text{<Argument>} ::= \text{"("<Core dimension list> ")+"}
\]

\[
\text{<Core dimension list>} ::= \text{nil} | \text{<Dimension name> | <Dimension name> "," <Core dimension list>}
\]

\[
\text{<Dimension name>} ::= \text{valid Python variable name}
\]

Notes:

1. All quotes are for clarity.

2. Core dimensions that share the same name must be broadcastable, as the two \(i\) in our example above. Each dimension name typically corresponding to one level of looping in the elementary function’s implementation.

3. White spaces are ignored.

Here are some examples of signatures:

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>() -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>inner1d</td>
<td>(i), (i) -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>sum1d</td>
<td>(i) -&gt; ()</td>
<td></td>
</tr>
<tr>
<td>dot2d</td>
<td>(m, n), (n, p) -&gt; (m, p)</td>
<td>matrix multiplication</td>
</tr>
<tr>
<td>outer_inner</td>
<td>(i, t), (j, t) -&gt; (i, j)</td>
<td>inner over the last dimension, outer over the second to last, and loop/broadcast over the rest.</td>
</tr>
</tbody>
</table>

### 5.7.3 C-API for implementing Elementary Functions

The current interface remains unchanged, and \text{PyUFunc\_FromFuncAndData} can still be used to implement (specialized) ufuncs, consisting of scalar elementary functions.

One can use \text{PyUFunc\_FromFuncAndDataAndSignature} to declare a more general ufunc. The argument list is the same as \text{PyUFunc\_FromFuncAndData}, with an additional argument specifying the signature as C string.

Furthermore, the callback function is of the same type as before, \text{void (*foo)(char **args, intp *dimensions, intp *steps, void *func)}\). When invoked, \text{args} is a list of length \text{nargs} containing the data of all input/output arguments. For a scalar elementary function, \text{steps} is also of length \text{nargs}, denoting the strides used for the arguments. \text{dimensions} is a pointer to a single integer defining the size of the axis to be looped over.

For a non-trivial signature, \text{dimensions} will also contain the sizes of the core dimensions as well, starting at the second entry. Only one size is provided for each unique dimension name and the sizes are given according to the first occurrence of a dimension name in the signature.

The first \text{nargs} elements of \text{steps} remain the same as for scalar ufuncs. The following elements contain the strides of all core dimensions for all arguments in order.

For example, consider a ufunc with signature \((i, j), (i) -> ()\). In this case, \text{args} will contain three pointers to the data of the input/output arrays \text{a, b, c}. Furthermore, \text{dimensions} will be \([N, I, J]\) to define the size of \(N\) of...
the loop and the sizes I and J for the core dimensions i and j. Finally, steps will be 
\([a_N, b_N, c_N, a_i, a_j, b_i]\), containing all necessary strides.

## 5.8 Numpy core libraries

New in version 1.3.0. Starting from numpy 1.3.0, we are working on separating the pure C, “computational” code from the python dependent code. The goal is twofolds: making the code cleaner, and enabling code reuse by other extensions outside numpy (scipy, etc...).

### 5.8.1 Numpy core math library

The numpy core math library (‘npymath’) is a first step in this direction. This library contains most math-related C99 functionality, which can be used on platforms where C99 is not well supported. The core math functions have the same API as the C99 ones, except for the npy_* prefix.

The available functions are defined in `<numpy/npy_math.h>` - please refer to this header when in doubt.

#### Floating point classification

**NPY_NAN**

This macro is defined to a NaN (Not a Number), and is guaranteed to have the signbit unset (‘positive’ NaN). The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_INFINITY**

This macro is defined to a positive inf. The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_PZERO**

This macro is defined to positive zero. The corresponding single and extension precision macro are available with the suffix F and L.

**NPY_NZERO**

This macro is defined to negative zero (that is with the sign bit set). The corresponding single and extension precision macro are available with the suffix F and L.

int \texttt{npy isnan}(x)

This is a macro, and is equivalent to C99 isnan: works for single, double and extended precision, and return a non 0 value is x is a NaN.

int \texttt{npy isfinite}(x)

This is a macro, and is equivalent to C99 isfinite: works for single, double and extended precision, and return a non 0 value is x is neither a NaN nor an infinity.

int \texttt{npy isinf}(x)

This is a macro, and is equivalent to C99 isinf: works for single, double and extended precision, and return a non 0 value is x is infinite (positive and negative).

int \texttt{npy signbit}(x)

This is a macro, and is equivalent to C99 signbit: works for single, double and extended precision, and return a non 0 value is x has the signbit set (that is the number is negative).

double \texttt{npy copysign}(\texttt{double} x, \texttt{double} y)

This is a function equivalent to C99 copysign: return x with the same sign as y. Works for any value, including inf and nan. Single and extended precisions are available with suffix f and l. New in version 1.4.0.
Useful math constants

The following math constants are available in npy_math.h. Single and extended precision are also available by adding the F and L suffixes respectively.

**NPY_E**
Base of natural logarithm (e)

**NPY_LOG2E**
Logarithm to base 2 of the Euler constant ($\ln(e)$)

**NPY_LOG10E**
Logarithm to base 10 of the Euler constant ($\ln(10)$)

**NPY_LOGE2**
Natural logarithm of 2 (ln(2))

**NPY_LOGE10**
Natural logarithm of 10 (ln(10))

**NPY_PI**
Pi ($\pi$)

**NPY_PI_2**
Pi divided by 2 ($\frac{\pi}{2}$)

**NPY_PI_4**
Pi divided by 4 ($\frac{\pi}{4}$)

**NPY_1_PI**
Reciprocal of pi ($\frac{1}{\pi}$)

**NPY_2_PI**
Two times the reciprocal of pi ($\frac{2}{\pi}$)

**NPY_EULER**

The Euler constant

$$\lim_{n \to \infty} \left( \sum_{k=1}^{n} \frac{1}{k} - \ln n \right)$$

Low-level floating point manipulation

Those can be useful for precise floating point comparison.

double **npy_nextafter**(double x, double y)

This is a function equivalent to C99 nextafter: return next representable floating point value from x in the direction of y. Single and extended precisions are available with suffix f and l. New in version 1.4.0.

double **npy_spacing**(double x)

This is a function equivalent to Fortran intrinsic. Return distance between x and next representable floating point value from x, e.g. spacing(1) == eps. spacing of nan and +/- inf return nan. Single and extended precisions are available with suffix f and l. New in version 1.4.0.

Complex functions

New in version 1.4.0. C99-like complex functions have been added. Those can be used if you wish to implement portable C extensions. Since we still support platforms without C99 complex type, you need to restrict to C90-compatible syntax, e.g.:
/* a = 1 + 2i */
npy_complex a = npy_cpack(1, 2);
npy_complex b;

b = npy_log(a);

### Linking against the core math library in an extension

New in version 1.4.0. To use the core math library in your own extension, you need to add the npymath compile and link options to your extension in your setup.py:

```python
>>> from numpy.distutils.misc_util import get_info
>>> info = get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=info)
```

In other words, the usage of info is exactly the same as when using blas_info and co.

### Half-precision functions

New in version 2.0.0. The header file `<numpy/halffloat.h>` provides functions to work with IEEE 754-2008 16-bit floating point values. While this format is not typically used for numerical computations, it is useful for storing values which require floating point but do not need much precision. It can also be used as an educational tool to understand the nature of floating point round-off error.

Like for other types, NumPy includes a typedef `npy_half` for the 16 bit float. Unlike for most of the other types, you cannot use this as a normal type in C, since is is a typedef for `npy_uint16`. For example, 1.0 looks like 0x3c00 to C, and if you do an equality comparison between the different signed zeros, you will get -0.0 != 0.0 (0x8000 != 0x0000), which is incorrect.

For these reasons, NumPy provides an API to work with `npy_half` values accessible by including `<numpy/halffloat.h>` and linking to ‘npymath’. For functions that are not provided directly, such as the arithmetic operations, the preferred method is to convert to float or double and back again, as in the following example.

```c
npy_half sum(int n, npy_half *array) {
    float ret = 0;
    while(n--) {
        ret += npy_half_to_float(*array);
    }
    return npy_float_to_half(ret);
}
```

External Links:

- 754-2008 IEEE Standard for Floating-Point Arithmetic
- OpenGL Half Float Pixel Support
- The OpenEXR image format.

**NPY_HALF_ZERO**

This macro is defined to positive zero.

**NPY_HALF_PZERO**

This macro is defined to positive zero.

**NPY_HALF_NZERO**

This macro is defined to negative zero.
The following macros are provided for half-precision floats:

**NPY_HALF_ONE**
This macro is defined to 1.0.

**NPY_HALF_NEGONE**
This macro is defined to -1.0.

**NPY_HALF_PINF**
This macro is defined to +inf.

**NPY_HALF_NINF**
This macro is defined to -inf.

**NPY_HALF_NAN**
This macro is defined to a NaN value, guaranteed to have its sign bit unset.

```c
float npy_half_to_float (npy_half h)
    Converts a half-precision float to a single-precision float.

double npy_half_to_double (npy_half h)
    Converts a half-precision float to a double-precision float.

npy_half npy_float_to_half (float f)
    Converts a single-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

npy_half npy_double_to_half (double d)
    Converts a double-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

int npy_half_eq (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 == h2).

int npy_half_ne (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 != h2).

int npy_half_le (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 <= h2).

int npy_half_lt (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 < h2).

int npy_half_ge (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 >= h2).

int npy_half_gt (npy_half h1, npy_half h2)
    Compares two half-precision floats (h1 > h2).

int npy_half_eq_nonan (npy_half h1, npy_half h2)
    Compares two half-precision floats that are known to not be NaN (h1 == h2). If a value is NaN, the result is undefined.

int npy_half_lt_nonan (npy_half h1, npy_half h2)
    Compares two half-precision floats that are known to not be NaN (h1 < h2). If a value is NaN, the result is undefined.

int npy_half_le_nonan (npy_half h1, npy_half h2)
    Compares two half-precision floats that are known to not be NaN (h1 <= h2). If a value is NaN, the result is undefined.
int npy_half_iszero (npy_half h)  
Tests whether the half-precision float has a value equal to zero. This may be slightly faster than calling 
npy_half_eq(h, NPY_ZERO).

int npy_half_isnan (npy_half h)  
Tests whether the half-precision float is a NaN.

int npy_half_isinf (npy_half h)  
Tests whether the half-precision float is plus or minus Inf.

int npy_half_isfinite (npy_half h)  
Tests whether the half-precision float is finite (not NaN or Inf).

int npy_half_signbit (npy_half h)  
Returns 1 if h is negative, 0 otherwise.

npy_half npy_half_copysign (npy_half x, npy_half y)  
Returns the value of x with the sign bit copied from y. Works for any value, including Inf and NaN.

npy_half npy_half_spacing (npy_half h)  
This is the same for half-precision float as npy_spacing and npy_spacingf described in the low-level floating 
point section.

npy_half npy_half_nextafter (npy_half x, npy_half y)  
This is the same for half-precision float as npy_nextafter and npy_nextafterf described in the low-level floating 
point section.

npy_uint16 npy_floatbits_to_halfbits (npy_uint32 f)  
Low-level function which converts a 32-bit single-precision float, stored as a uint32, into a 16-bit half-precision 
float.

npy_uint16 npy_doublebits_to_halfbits (npy_uint64 d)  
Low-level function which converts a 64-bit double-precision float, stored as a uint64, into a 16-bit half-precision 
float.

npy_uint32 npy_halfbits_to_floatbits (npy_uint16 h)  
Low-level function which converts a 16-bit half-precision float into a 32-bit single-precision float, stored as a 
uint32.

npy_uint64 npy_halfbits_to_doublebits (npy_uint16 h)  
Low-level function which converts a 16-bit half-precision float into a 64-bit double-precision float, stored as a 
uint64.

5.9 C API Deprecations

5.9.1 Background

The API exposed by NumPy for third-party extensions has grown over years of releases, and has allowed programmers 
to directly access NumPy functionality from C. This API can be best described as “organic”. It has emerged from 
multiple competing desires and from multiple points of view over the years, strongly influenced by the desire to make 
it easy for users to move to NumPy from Numeric and Numarray. The core API originated with Numeric in 1995 and 
there are patterns such as the heavy use of macros written to mimic Python’s C-API as well as account for compiler 
technology of the late 90’s. There is also only a small group of volunteers who have had very little time to spend on 
improving this API.

There is an ongoing effort to improve the API. It is important in this effort to ensure that code that compiles for 
NumPy 1.X continues to compile for NumPy 1.X. At the same time, certain API’s will be marked as deprecated so 
that future-looking code can avoid these API’s and follow better practices.
Another important role played by deprecation markings in the C API is to move towards hiding internal details of the NumPy implementation. For those needing direct, easy, access to the data of ndarrays, this will not remove this ability. Rather, there are many potential performance optimizations which require changing the implementation details, and NumPy developers have been unable to try them because of the high value of preserving ABI compatibility. By deprecating this direct access, we will in the future be able to improve NumPy’s performance in ways we cannot presently.

5.9.2 Deprecation Mechanism NPY_NO_DEPRECATED_API

In C, there is no equivalent to the deprecation warnings that Python supports. One way to do deprecations is to flag them in the documentation and release notes, then remove or change the deprecated features in a future major version (NumPy 2.0 and beyond). Minor versions of NumPy should not have major C-API changes, however, that prevent code that worked on a previous minor release. For example, we will do our best to ensure that code that compiled and worked on NumPy 1.4 should continue to work on NumPy 1.7 (but perhaps with compiler warnings).

To use the NPY_NO_DEPRECATED_API mechanism, you need to #define it to the target API version of NumPy before #including any NumPy headers. If you want to confirm that your code is clean against 1.7, use:

```
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
```

On compilers which support a #warning mechanism, NumPy issues a compiler warning if you do not define the symbol NPY_NO_DEPRECATED_API. This way, the fact that there are deprecations will be flagged for third-party developers who may not have read the release notes closely.
6.1 Numpy C Code Explanations

Fanaticism consists of redoubling your efforts when you have forgotten your aim. — George Santayana

An authority is a person who can tell you more about something than you really care to know. — Unknown

This Chapter attempts to explain the logic behind some of the new pieces of code. The purpose behind these explanations is to enable somebody to be able to understand the ideas behind the implementation somewhat more easily than just staring at the code. Perhaps in this way, the algorithms can be improved on, borrowed from, and/or optimized.

6.1.1 Memory model

One fundamental aspect of the ndarray is that an array is seen as a “chunk” of memory starting at some location. The interpretation of this memory depends on the stride information. For each dimension in an $N$-dimensional array, an integer (stride) dictates how many bytes must be skipped to get to the next element in that dimension. Unless you have a single-segment array, this stride information must be consulted when traversing through an array. It is not difficult to write code that accepts strides, you just have to use (char *) pointers because strides are in units of bytes. Keep in mind also that strides do not have to be unit-multiples of the element size. Also, remember that if the number of dimensions of the array is 0 (sometimes called a rank-0 array), then the strides and dimensions variables are NULL.

Besides the structural information contained in the strides and dimensions members of the PyArrayObject, the flags contain important information about how the data may be accessed. In particular, the NPY_ARRAY_ALIGNED flag is set when the memory is on a suitable boundary according to the data-type array. Even if you have a contiguous chunk of memory, you cannot just assume it is safe to dereference a data-type-specific pointer to an element. Only if the NPY_ARRAY_ALIGNED flag is set is this a safe operation (on some platforms it will work but on others, like Solaris, it will cause a bus error). The NPY_ARRAY_WRITEABLE should also be ensured if you plan on writing to the memory area of the array. It is also possible to obtain a pointer to an unwriteable memory area. Sometimes, writing to the memory area when the NPY_ARRAY_WRITEABLE flag is not set will just be rude. Other times it can cause program crashes (e.g. a data-area that is a read-only memory-mapped file).

6.1.2 Data-type encapsulation

The data-type is an important abstraction of the ndarray. Operations will look to the data-type to provide the key functionality that is needed to operate on the array. This functionality is provided in the list of function pointers pointed to by the ‘f’ member of the PyArray_Descr structure. In this way, the number of data-types can be extended simply by providing a PyArray_Descr structure with suitable function pointers in the ‘f’ member. For built-in types there are some optimizations that by-pass this mechanism, but the point of the data-type abstraction is to allow new data-types to be added.
One of the built-in data-types, the void data-type allows for arbitrary records containing 1 or more fields as elements of the array. A field is simply another data-type object along with an offset into the current record. In order to support arbitrarily nested fields, several recursive implementations of data-type access are implemented for the void type. A common idiom is to cycle through the elements of the dictionary and perform a specific operation based on the data-type object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering mis-aligned data must be recognized and taken into account if necessary.

6.1.3 N-D Iterators

A very common operation in much of NumPy code is the need to iterate over all the elements of a general, strided, N-dimensional array. This operation of a general-purpose N-dimensional loop is abstracted in the notion of an iterator object. To write an N-dimensional loop, you only have to create an iterator object from an ndarray, work with the dataptr member of the iterator object structure and call the macro PyArray_ITER_NEXT (it) on the iterator object to move to the next element. The “next” element is always in C-contiguous order. The macro works by first special casing the C-contiguous, 1-D, and 2-D cases which work very simply.

For the general case, the iteration works by keeping track of a list of coordinate counters in the iterator object. At each iteration, the last coordinate counter is increased (starting from 0). If this counter is smaller then one less than the size of the array in that dimension (a pre-computed and stored value), then the counter is increased and the dataptr member is increased by the strides in that dimension and the macro ends. If the end of a dimension is reached, the counter for the last dimension is reset to zero and the dataptr is moved back to the beginning of that dimension by subtracting the strides value times one less than the number of elements in that dimension (this is also pre-computed and stored in the backstrides member of the iterator object). In this case, the macro does not end, but a local dimension counter is decremented so that the next-to-last dimension replaces the role that the last dimension played and the previously-described tests are executed again on the next-to-last dimension. In this way, the dataptr is adjusted appropriately for arbitrary striding.

The coordinates member of the PyArrayIterObject structure maintains the current N-d counter unless the underlying array is C-contiguous in which case the coordinate counting is by-passed. The index member of the PyArrayIterObject keeps track of the current flat index of the iterator. It is updated by the PyArray_ITER_NEXT macro.

6.1.4 Broadcasting

In Numeric, broadcasting was implemented in several lines of code buried deep in ufuncobject.c. In NumPy, the notion of broadcasting has been abstracted so that it can be performed in multiple places. Broadcasting is handled by the function PyArray_Broadcast. This function requires a PyArrayMultiIterObject (or something that is a binary equivalent) to be passed in. The PyArrayMultiIterObject keeps track of the broadcasted number of dimensions and size in each dimension along with the total size of the broadcasted result. It also keeps track of the number of arrays being broadcast and a pointer to an iterator for each of the arrays being broadcasted.

The PyArray_Broadcast function takes the iterators that have already been defined and uses them to determine the broadcast shape in each dimension (to create the iterators at the same time that broadcasting occurs then use the PyMultiIter_New function). Then, the iterators are adjusted so that each iterator thinks it is iterating over an array with the broadcasted size. This is done by adjusting the iterators number of dimensions, and the shape in each dimension. This works because the iterator strides are also adjusted. Broadcasting only adjusts (or adds) length-1 dimensions. For these dimensions, the strides variable is simply set to 0 so that the data-pointer for the iterator over that array doesn’t move as the broadcasting operation operates over the extended dimension.

Broadcasting was always implemented in Numeric using 0-valued strides for the extended dimensions. It is done in exactly the same way in NumPy. The big difference is that now the array of strides is kept track of in a PyArrayIterObject, the iterators involved in a broadcasted result are kept track of in a PyArrayMultiIterObject, and the PyArray_Broadcast call implements the broad-casting rules.
6.1.5 Array Scalars

The array scalars offer a hierarchy of Python types that allow a one-to-one correspondence between the data-type stored in an array and the Python-type that is returned when an element is extracted from the array. An exception to this rule was made with object arrays. Object arrays are heterogeneous collections of arbitrary Python objects. When you select an item from an object array, you get back the original Python object (and not an object array scalar which does exist but is rarely used for practical purposes).

The array scalars also offer the same methods and attributes as arrays with the intent that the same code can be used to support arbitrary dimensions (including 0-dimensions). The array scalars are read-only (immutable) with the exception of the void scalar which can also be written to so that record-array field setting works more naturally (a[0]['f1'] = value).

6.1.6 Advanced (“Fancy”) Indexing

The implementation of advanced indexing represents some of the most difficult code to write and explain. In fact, there are two implementations of advanced indexing. The first works only with 1-D arrays and is implemented to handle expressions involving a.flat[obj]. The second is general-purpose that works for arrays of “arbitrary dimension” (up to a fixed maximum). The one-dimensional indexing approaches were implemented in a rather straightforward fashion, and so it is the general-purpose indexing code that will be the focus of this section.

There is a multi-layer approach to indexing because the indexing code can at times return an array scalar and at other times return an array. The functions with “_nice” appended to their name do this special handling while the function without the _nice appendage always return an array (perhaps a 0-dimensional array). Some special-case optimizations (the index being an integer scalar, and the index being a tuple with as many dimensions as the array) are handled in array_subscript_nice function which is what Python calls when presented with the code “a[obj].” These optimizations allow fast single-integer indexing, and also ensure that a 0-dimensional array is not created only to be discarded as the array scalar is returned instead. This provides significant speed-up for code that is selecting many scalars out of an array (such as in a loop). However, it is still not faster than simply using a list to store standard Python scalars, because that is optimized by the Python interpreter itself.

After these optimizations, the array_subscript function itself is called. This function first checks for field selection which occurs when a string is passed as the indexing object. Then, 0-D arrays are given special-case consideration. Finally, the code determines whether or not advanced, or fancy, indexing needs to be performed. If fancy indexing is not needed, then standard view-based indexing is performed using code borrowed from Numeric which parses the indexing object and returns the offset into the data-buffer and the dimensions necessary to create a new view of the array. The strides are also changed by multiplying each stride by the step-size requested along the corresponding dimension.

Fancy-indexing check

The fancy_indexing_check routine determines whether or not to use standard view-based indexing or new copy-based indexing. If the indexing object is a tuple, then view-based indexing is assumed by default. Only if the tuple contains an array object or a sequence object is fancy-indexing assumed. If the indexing object is an array, then fancy indexing is automatically assumed. If the indexing object is any other kind of sequence, then fancy-indexing is assumed by default. This is over-ridden to simple indexing if the sequence contains any slice, newaxis, or Ellipsis objects, and no arrays or additional sequences are also contained in the sequence. The purpose of this is to allow the construction of “slicing” sequences which is a common technique for building up code that works in arbitrary numbers of dimensions.

Fancy-indexing implementation

The concept of indexing was also abstracted using the idea of an iterator. If fancy indexing is performed, then a PyArrayMapIterObject is created. This internal object is not exposed to Python. It is created in order to handle...
the fancy-indexing at a high-level. Both get and set fancy-indexing operations are implemented using this object. Fancy indexing is abstracted into three separate operations: (1) creating the PyArrayMapIterObject from the indexing object, (2) binding the PyArrayMapIterObject to the array being indexed, and (3) getting (or setting) the items determined by the indexing object. There is an optimization implemented so that the PyArrayIterObject (which has its own less complicated fancy-indexing) is used for indexing when possible.

Creating the mapping object

The first step is to convert the indexing objects into a standard form where iterators are created for all of the index array inputs and all Boolean arrays are converted to equivalent integer index arrays (as if nonzero(arr) had been called). Finally, all integer arrays are replaced with the integer 0 in the indexing object and all of the index-array iterators are “broadcast” to the same shape.

Binding the mapping object

When the mapping object is created it does not know which array it will be used with so once the index iterators are constructed during mapping-object creation, the next step is to associate these iterators with a particular ndarray. This process interprets any ellipsis and slice objects so that the index arrays are associated with the appropriate axis (the axis indicated by the iteraxis entry corresponding to the iterator for the integer index array). This information is then used to check the indices to be sure they are within range of the shape of the array being indexed. The presence of ellipsis and/or slice objects implies a sub-space iteration that is accomplished by extracting a sub-space view of the array (using the index object resulting from replacing all the integer index arrays with 0) and storing the information about where this sub-space starts in the mapping object. This is used later during mapping-object iteration to select the correct elements from the underlying array.

Getting (or Setting)

After the mapping object is successfully bound to a particular array, the mapping object contains the shape of the resulting item as well as iterator objects that will walk through the currently-bound array and either get or set its elements as needed. The walk is implemented using the PyArray_MapIterNext function. This function sets the coordinates of an iterator object into the current array to be the next coordinate location indicated by all of the indexing-object iterators while adjusting, if necessary, for the presence of a sub-space. The result of this function is that the dataptr member of the mapping object structure is pointed to the next position in the array that needs to be copied out or set to some value.

When advanced indexing is used to extract an array, an iterator for the new array is constructed and advanced in phase with the mapping object iterator. When advanced indexing is used to place values in an array, a special “broadcasted” iterator is constructed from the object being placed into the array so that it will only work if the values used for setting have a shape that is “broadcastable” to the shape implied by the indexing object.

6.1.7 Universal Functions

Universal functions are callable objects that take \( N \) inputs and produce \( M \) outputs by wrapping basic 1-D loops that work element-by-element into full easy-to use functions that seamlessly implement broadcasting, type-checking and buffered coercion, and output-argument handling. New universal functions are normally created in C, although there is a mechanism for creating ufuncs from Python functions (frompyfunc). The user must supply a 1-D loop that implements the basic function taking the input scalar values and placing the resulting scalars into the appropriate output slots as explained implementation.

Setup

Every ufunc calculation involves some overhead related to setting up the calculation. The practical significance of this overhead is that even though the actual calculation of the ufunc is very fast, you will be able to write array and type-specific code that will work faster for small arrays than the ufunc. In particular, using ufuncs to perform many calculations on 0-D arrays will be slower than other Python-based solutions (the silently-imported scalarmath
module exists precisely to give array scalars the look-and-feel of ufunc-based calculations with significantly reduced overhead).

When a ufunc is called, many things must be done. The information collected from these setup operations is stored in a loop-object. This loop object is a C-structure (that could become a Python object but is not initialized as such because it is only used internally). This loop object has the layout needed to be used with PyArray_Broadcast so that the broadcasting can be handled in the same way as it is handled in other sections of code.

The first thing done is to look-up in the thread-specific global dictionary the current values for the buffer-size, the error mask, and the associated error object. The state of the error mask controls what happens when an error-condition is found. It should be noted that checking of the hardware error flags is only performed after each 1-D loop is executed. This means that if the input and output arrays are contiguous and of the correct type so that a single 1-D loop is performed, then the flags may not be checked until all elements of the array have been calculated. Looking up these values in a thread-specific dictionary takes time which is easily ignored for all but very small arrays.

After checking, the thread-specific global variables, the inputs are evaluated to determine how the ufunc should proceed and the input and output arrays are constructed if necessary. Any inputs which are not arrays are converted to arrays (using context if necessary). Which of the inputs are scalars (and therefore converted to 0-D arrays) is noted.

Next, an appropriate 1-D loop is selected from the 1-D loops available to the ufunc based on the input array types. This 1-D loop is selected by trying to match the signature of the data-types of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the types member of the ufunc structure. The signatures corresponding to user-defined types are stored in a linked-list of function-information with the head element stored as a CObject in the userloops dictionary keyed by the data-type number (the first user-defined type in the argument list is used as the key). The signatures are searched until a signature is found to which the input arrays can all be cast safely (ignoring any scalar arguments which are not allowed to determine the type of the result). The implication of this search procedure is that “lesser types” should be placed below “larger types” when the signatures are stored. If no 1-D loop is found, then an error is reported. Otherwise, the argument_list is updated with the stored signature — in case casting is necessary and to fix the output types assumed by the 1-D loop.

If the ufunc has 2 inputs and 1 output and the second input is an Object array then a special-case check is performed so that NotImplemented is returned if the second input is not an ndarray, has the __array_priority__ attribute, and has an __r{op}__ special method. In this way, Python is signaled to give the other object a chance to complete the operation instead of using generic object-array calculations. This allows (for example) sparse matrices to override the multiplication operator 1-D loop.

For input arrays that are smaller than the specified buffer size, copies are made of all non-contiguous, mis-aligned, or out-of-byteorder arrays to ensure that for small arrays, a single-loop is used. Then, array iterators are created for all the input arrays and the resulting collection of iterators is broadcast to a single shape.

The output arguments (if any) are then processed and any missing return arrays are constructed. If any provided output array doesn’t have the correct type (or is mis-aligned) and is smaller than the buffer size, then a new output array is constructed with the special UPDATEIFCOPY flag set so that when it is DECREASE’d on completion of the function, it’s contents will be copied back into the output array. Iterators for the output arguments are then processed.

Finally, the decision is made about how to execute the looping mechanism to ensure that all elements of the input arrays are combined to produce the output arrays of the correct type. The options for loop execution are one-loop (for contiguous, aligned, and correct data-type), strided-loop (for non-contiguous but still aligned and correct data-type), and a buffered loop (for mis-aligned or incorrect data-type situations). Depending on which execution method is called for, the loop is then setup and computed.

**Function call**

This section describes how the basic universal function computation loop is setup and executed for each of the three different kinds of execution possibilities. If NPY_ALLOW_THREADS is defined during compilation, then the Python Global Interpreter Lock (GIL) is released prior to calling all of these loops (as long as they don’t involve object arrays).
It is re-acquired if necessary to handle error conditions. The hardware error flags are checked only after the 1-D loop is calculated.

**One Loop**

This is the simplest case of all. The ufunc is executed by calling the underlying 1-D loop exactly once. This is possible only when we have aligned data of the correct type (including byte-order) for both input and output and all arrays have uniform strides (either contiguous, 0-D, or 1-D). In this case, the 1-D computational loop is called once to compute the calculation for the entire array. Note that the hardware error flags are only checked after the entire calculation is complete.

**Strided Loop**

When the input and output arrays are aligned and of the correct type, but the striding is not uniform (non-contiguous and 2-D or larger), then a second looping structure is employed for the calculation. This approach converts all of the iterators for the input and output arguments to iterate over all but the largest dimension. The inner loop is then handled by the underlying 1-D computational loop. The outer loop is a standard iterator loop on the converted iterators. The hardware error flags are checked after each 1-D loop is completed.

**Buffered Loop**

This is the code that handles the situation whenever the input and/or output arrays are either misaligned or of the wrong data-type (including being byte-swapped) from what the underlying 1-D loop expects. The arrays are also assumed to be non-contiguous. The code works very much like the strided loop except for the inner 1-D loop is modified so that pre-processing is performed on the inputs and post-processing is performed on the outputs in bufsize chunks (where bufsize is a user-settable parameter). The underlying 1-D computational loop is called on data that is copied over (if it needs to be). The setup code and the loop code is considerably more complicated in this case because it has to handle:

- memory allocation of the temporary buffers
- deciding whether or not to use buffers on the input and output data (mis-aligned and/or wrong data-type)
- copying and possibly casting data for any inputs or outputs for which buffers are necessary.
- special-casing Object arrays so that reference counts are properly handled when copies and/or casts are necessary.
- breaking up the inner 1-D loop into bufsize chunks (with a possible remainder).

Again, the hardware error flags are checked at the end of each 1-D loop.

**Final output manipulation**

Ufuncs allow other array-like classes to be passed seamlessly through the interface in that inputs of a particular class will induce the outputs to be of that same class. The mechanism by which this works is the following. If any of the inputs are not ndarrays and define the `__array_wrap__` method, then the class with the largest `__array_priority__` attribute determines the type of all the outputs (with the exception of any output arrays passed in). The `__array_wrap__` method of the input array will be called with the ndarray being returned from the ufunc as it’s input. There are two calling styles of the `__array_wrap__` function supported. The first takes the ndarray as the first argument and a tuple of “context” as the second argument. The context is (ufunc, arguments, output argument number). This is the first call tried. If a TypeError occurs, then the function is called with just the ndarray as the first argument.

**Methods**

There are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are reduce, accumulate, and reduceat. Each of these methods requires a setup command followed by a loop. There are four loop
styles possible for the methods corresponding to no-elements, one-element, strided-loop, and buffered-loop. These are the same basic loop styles as implemented for the general purpose function call except for the no-element and one-element cases which are special-cases occurring when the input array objects have 0 and 1 elements respectively.

**Setup**

The setup function for all three methods is `construct_reduce`. This function creates a reducing loop object and fills it with parameters needed to complete the loop. All of the methods only work on ufuncs that take 2-inputs and return 1 output. Therefore, the underlying 1-D loop is selected assuming a signature of `[ otype, otype, otype ]` where `otype` is the requested reduction data-type. The buffer size and error handling is then retrieved from (per-thread) global storage. For small arrays that are mis-aligned or have incorrect data-type, a copy is made so that the un-buffered section of code is used. Then, the looping strategy is selected. If there is 1 element or 0 elements in the array, then a simple looping method is selected. If the array is not mis-aligned and has the correct data-type, then strided looping is selected. Otherwise, buffered looping must be performed. Looping parameters are then established, and the return array is constructed. The output array is of a different shape depending on whether the method is reduce, accumulate, or reduceat. If an output array is already provided, then it's shape is checked. If the output array is not C-contiguous, aligned, and of the correct data-type, then a temporary copy is made with the `UPDATEIFCOPY` flag set. In this way, the methods will be able to work with a well-behaved output array but the result will be copied back into the true output array when the method computation is complete. Finally, iterators are set up to loop over the correct axis (depending on the value of axis provided to the method) and the setup routine returns to the actual computation routine.

**Reduce**

All of the ufunc methods use the same underlying 1-D computational loops with input and output arguments adjusted so that the appropriate reduction takes place. For example, the key to the functioning of reduce is that the 1-D loop is called with the output and the second input pointing to the same position in memory and both having a step-size of 0. The first input is pointing to the input array with a step-size given by the appropriate stride for the selected axis. In this way, the operation performed is

\[
o = \begin{cases} i[0] \\
o[k] = i[k]<\text{op}>o[k - 1] \quad k = 1 \ldots N
\end{cases}
\]

where \( N + 1 \) is the number of elements in the input, \( i, o \) is the output, and \( i[k] \) is the \( k \)th element of \( i \) along the selected axis. This basic operations is repeated for arrays with greater than 1 dimension so that the reduction takes place for every 1-D sub-array along the selected axis. An iterator with the selected dimension removed handles this looping.

For buffered loops, care must be taken to copy and cast data before the loop function is called because the underlying loop expects aligned data of the correct data-type (including byte-order). The buffered loop must handle this copying and casting prior to calling the loop function on chunks no greater than the user-specified bufsize.

**Accumulate**

The accumulate function is very similar to the reduce function in that the output and the second input both point to the output. The difference is that the second input points to memory one stride behind the current output pointer. Thus, the operation performed is

\[
o[0] = \begin{cases} i[0] \\
o[k] = i[k]<\text{op}>o[k] \quad k = 1 \ldots N
\end{cases}
\]

The output has the same shape as the input and each 1-D loop operates over \( N \) elements when the shape in the selected axis is \( N + 1 \). Again, buffered loops take care to copy and cast the data before calling the underlying 1-D computational loop.

**Reduceat**

The reduceat function is a generalization of both the reduce and accumulate functions. It implements a reduce over ranges of the input array specified by indices. The extra indices argument is checked to be sure that every input
is not too large for the input array along the selected dimension before the loop calculations take place. The loop implementation is handled using code that is very similar to the reduce code repeated as many times as there are elements in the indices input. In particular: the first input pointer passed to the underlying 1-D computational loop points to the input array at the correct location indicated by the index array. In addition, the output pointer and the second input pointer passed to the underlying 1-D loop point to the same position in memory. The size of the 1-D computational loop is fixed to be the difference between the current index and the next index (when the current index is the last index, then the next index is assumed to be the length of the array along the selected dimension). In this way, the 1-D loop will implement a reduce over the specified indices.

Mis-aligned or a loop data-type that does not match the input and/or output data-type is handled using buffered code where-in data is copied to a temporary buffer and cast to the correct data-type if necessary prior to calling the underlying 1-D function. The temporary buffers are created in (element) sizes no bigger than the user settable buffer-size value. Thus, the loop must be flexible enough to call the underlying 1-D computational loop enough times to complete the total calculation in chunks no bigger than the buffer-size.

6.2 Internal organization of numpy arrays

It helps to understand a bit about how numpy arrays are handled under the covers to help understand numpy better. This section will not go into great detail. Those wishing to understand the full details are referred to Travis Oliphant’s book “Guide to Numpy”.

Numpy arrays consist of two major components, the raw array data (from now on, referred to as the data buffer), and the information about the raw array data. The data buffer is typically what people think of as arrays in C or Fortran, a contiguous (and fixed) block of memory containing fixed sized data items. Numpy also contains a significant set of data that describes how to interpret the data in the data buffer. This extra information contains (among other things):

1. The basic data element’s size in bytes
2. The start of the data within the data buffer (an offset relative to the beginning of the data buffer).
3. The number of dimensions and the size of each dimension
4. The separation between elements for each dimension (the ‘stride’). This does not have to be a multiple of the element size
5. The byte order of the data (which may not be the native byte order)
6. Whether the buffer is read-only
7. Information (via the dtype object) about the interpretation of the basic data element. The basic data element may be as simple as a int or a float, or it may be a compound object (e.g., struct-like), a fixed character field, or Python object pointers.
8. Whether the array is to interpreted as C-order or Fortran-order.

This arrangement allow for very flexible use of arrays. One thing that it allows is simple changes of the metadata to change the interpretation of the array buffer. Changing the byteorder of the array is a simple change involving no rearrangement of the data. The shape of the array can be changed very easily without changing anything in the data buffer or any data copying at all.

Among other things that are made possible is one can create a new array metadata object that uses the same data buffer to create a new view of that data buffer that has a different interpretation of the buffer (e.g., different shape, offset, byte order, strides, etc) but shares the same data bytes. Many operations in numpy do just this such as slices. Other operations, such as transpose, don’t move data elements around in the array, but rather change the information about the shape and strides so that the indexing of the array changes, but the data in the doesn’t move.

Typically these new versions of the array metadata but the same data buffer are new ‘views’ into the data buffer. There is a different ndarray object, but it uses the same data buffer. This is why it is necessary to force copies through use of the .copy() method if one really wants to make a new and independent copy of the data buffer.
New views into arrays mean the object reference counts for the data buffer increase. Simply doing away with the original array object will not remove the data buffer if other views of it still exist.

6.3 Multidimensional Array Indexing Order Issues

What is the right way to index multi-dimensional arrays? Before you jump to conclusions about the one and true way to index multi-dimensional arrays, it pays to understand why this is a confusing issue. This section will try to explain in detail how numpy indexing works and why we adopt the convention we do for images, and when it may be appropriate to adopt other conventions.

The first thing to understand is that there are two conflicting conventions for indexing 2-dimensional arrays. Matrix notation uses the first index to indicate which row is being selected and the second index to indicate which column is selected. This is opposite the geometrically oriented-convention for images where people generally think the first index represents x position (i.e., column) and the second represents y position (i.e., row). This alone is the source of much confusion; matrix-oriented users and image-oriented users expect two different things with regard to indexing.

The second issue to understand is how indices correspond to the order the array is stored in memory. In Fortran the first index is the most rapidly varying index when moving through the elements of a two dimensional array as it is stored in memory. If you adopt the matrix convention for indexing, then this means the matrix is stored one column at a time (since the first index moves to the next row as it changes). Thus Fortran is considered a Column-major language. C has just the opposite convention. In C, the last index changes most rapidly as one moves through the array as stored in memory. Thus C is a Row-major language. The matrix is stored by rows. Note that in both cases it presumes that the matrix convention for indexing is being used, i.e., for both Fortran and C, the first index is the row. Note this convention implies that the indexing convention is invariant and that the data order changes to keep that so.

But that’s not the only way to look at it. Suppose one has large two-dimensional arrays (images or matrices) stored in data files. Suppose the data are stored by rows rather than by columns. If we are to preserve our index convention (whether matrix or image) that means that depending on the language we use, we may be forced to reorder the data if it is read into memory to preserve our indexing convention. For example if we read row-ordered data into memory without reordering, it will match the matrix indexing convention for C, but not for Fortran. Conversely, it will match the image indexing convention for Fortran, but not for C. For C, if one is using data stored in row order, and one wants to preserve the image index convention, the data must be reordered when reading into memory.

In the end, which you do for Fortran or C depends on which is more important, not reordering data or preserving the indexing convention. For large images, reordering data is potentially expensive, and often the indexing convention is inverted to avoid that.

The situation with numpy makes this issue yet more complicated. The internal machinery of numpy arrays is flexible enough to accept any ordering of indices. One can simply reorder indices by manipulating the internal stride information for arrays without reordering the data at all. Numpy will know how to map the new index order to the data without moving the data.

So if this is true, why not choose the index order that matches what you most expect? In particular, why not define row-ordered images to use the image convention? (This is sometimes referred to as the Fortran convention vs the C convention, thus the ‘C’ and ‘FORTRAN’ order options for array ordering in numpy.) The drawback of doing this is potential performance penalties. It’s common to access the data sequentially, either implicitly in array operations or explicitly by looping over rows of an image. When that is done, then the data will be accessed in non-optimal order. As the first index is incremented, what is actually happening is that elements spaced far apart in memory are being sequentially accessed, with usually poor memory access speeds. For example, for a two dimensional image ‘im’ defined so that im[0, 10] represents the value at x=0, y=10. To be consistent with usual Python behavior then im[0] would represent a column at x=0. Yet that data would be spread over the whole array since the data are stored in row order. Despite the flexibility of numpy’s indexing, it can’t really paper over the fact basic operations are rendered inefficient because of data order or that getting contiguous subarrays is still awkward (e.g., im[:,0] for the first row, vs im[0]), thus one can’t use an idiom such as for row in im; for col in im does work, but doesn’t yield contiguous column data.
As it turns out, numpy is smart enough when dealing with ufuncs to determine which index is the most rapidly varying one in memory and uses that for the innermost loop. Thus for ufuncs there is no large intrinsic advantage to either approach in most cases. On the other hand, use of .flat with a FORTRAN ordered array will lead to non-optimal memory access as adjacent elements in the flattened array (iterator, actually) are not contiguous in memory.

Indeed, the fact is that Python indexing on lists and other sequences naturally leads to an outside-to-inside ordering (the first index gets the largest grouping, the next the next largest, and the last gets the smallest element). Since image data are normally stored by rows, this corresponds to position within rows being the last item indexed.

If you do want to use Fortran ordering realize that there are two approaches to consider: 1) accept that the first index is just not the most rapidly changing in memory and have all your I/O routines reorder your data when going from memory to disk or visa versa, or use numpy’s mechanism for mapping the first index to the most rapidly varying data. We recommend the former if possible. The disadvantage of the latter is that many of numpy’s functions will yield arrays without Fortran ordering unless you are careful to use the ‘order’ keyword. Doing this would be highly inconvenient.

Otherwise we recommend simply learning to reverse the usual order of indices when accessing elements of an array. Granted, it goes against the grain, but it is more in line with Python semantics and the natural order of the data.
7.1 Numpy.i: a SWIG Interface File for NumPy

7.1.1 Introduction

The Simple Wrapper and Interface Generator (or SWIG) is a powerful tool for generating wrapper code for interfacing to a wide variety of scripting languages. SWIG can parse header files, and using only the code prototypes, create an interface to the target language. But SWIG is not omnipotent. For example, it cannot know from the prototype:

\[
double \text{rms}(\text{double}^* \ seq, \text{int} \ n);
\]

what exactly \text{seq} is. Is it a single value to be altered in-place? Is it an array, and if so what is its length? Is it input-only? Output-only? Input-output? SWIG cannot determine these details, and does not attempt to do so.

If we designed \text{rms}, we probably made it a routine that takes an input-only array of length \(n\) of \text{double} values called \text{seq} and returns the root mean square. The default behavior of SWIG, however, will be to create a wrapper function that compiles, but is nearly impossible to use from the scripting language in the way the C routine was intended.

For Python, the preferred way of handling contiguous (or technically, \textit{strided}) blocks of homogeneous data is with NumPy, which provides full object-oriented access to multidimensional arrays of data. Therefore, the most logical Python interface for the \text{rms} function would be (including doc string):

```python
def rms(seq):
    
    \text{rms}: \text{return the root mean square of a sequence}
    rms(\text{numpy.ndarray}) \rightarrow \text{double}
    rms(\text{list}) \rightarrow \text{double}
    rms(\text{tuple}) \rightarrow \text{double}


```

where \text{seq} would be a NumPy array of \text{double} values, and its length \(n\) would be extracted from \text{seq} internally before being passed to the C routine. Even better, since NumPy supports construction of arrays from arbitrary Python sequences, \text{seq} itself could be a nearly arbitrary sequence (so long as each element can be converted to a \text{double}) and the wrapper code would internally convert it to a NumPy array before extracting its data and length.

**SWIG** allows these types of conversions to be defined via a mechanism called \textit{typemaps}. This document provides information on how to use \texttt{numpy.i}, a SWIG interface file that defines a series of typemaps intended to make the type of array-related conversions described above relatively simple to implement. For example, suppose that the \texttt{rms} function prototype defined above was in a header file named \texttt{rms.h}. To obtain the Python interface discussed above, your SWIG interface file would need the following:

\%
#define SWIG_FILE_WITH_INIT


NumPy Reference, Release 1.8.1

#include "rms.h"
%}
%include "numpy.i"
%init %{
import_array();
%}
%apply (double* IN_ARRAY1, int DIM1) {(double* seq, int n)};
%include "rms.h"

Typemaps are keyed off a list of one or more function arguments, either by type or by type and name. We will
refer to such lists as signatures. One of the many typemaps defined by numpy.i is used above and has the signature
(double* IN_ARRAY1, int DIM1). The argument names are intended to suggest that the double* argument
is an input array of one dimension and that the int represents the size of that dimension. This is precisely the pattern
in the rms prototype.
Most likely, no actual prototypes to be wrapped will have the argument names IN_ARRAY1 and DIM1. We use
the SWIG %apply directive to apply the typemap for one-dimensional input arrays of type double to the actual
prototype used by rms. Using numpy.i effectively, therefore, requires knowing what typemaps are available and
what they do.
A SWIG interface file that includes the SWIG directives given above will produce wrapper code that looks something
like:
1 PyObject *_wrap_rms(PyObject *args) {
2
PyObject *resultobj = 0;
3
double *arg1 = (double *) 0 ;
4
int arg2 ;
5
double result;
6
PyArrayObject *array1 = NULL ;
7
int is_new_object1 = 0 ;
8
PyObject * obj0 = 0 ;
9
10
if (!PyArg_ParseTuple(args,(char *)"O:rms",&obj0)) SWIG_fail;
11
{
12
array1 = obj_to_array_contiguous_allow_conversion(
13
obj0, NPY_DOUBLE, &is_new_object1);
14
npy_intp size[1] = {
15
-1
16
};
17
if (!array1 || !require_dimensions(array1, 1) ||
18
!require_size(array1, size, 1)) SWIG_fail;
19
arg1 = (double*) array1->data;
20
arg2 = (int) array1->dimensions[0];
21
}
22
result = (double)rms(arg1,arg2);
23
resultobj = SWIG_From_double((double)(result));
24
{
25
if (is_new_object1 && array1) Py_DECREF(array1);
26
}
27
return resultobj;
28 fail:
29
{
30
if (is_new_object1 && array1) Py_DECREF(array1);
31
}
32
return NULL;
33 }

1422

Chapter 7. Numpy and SWIG


The typemaps from `numpy.i` are responsible for the following lines of code: 12–20, 25 and 30. Line 10 parses the input to the `rms` function. From the format string "O:rms", we can see that the argument list is expected to be a single Python object (specified by the O before the colon) and whose pointer is stored in `obj0`. A number of functions, supplied by `numpy.i`, are called to make and check the (possible) conversion from a generic Python object to a NumPy array. These functions are explained in the section Helper Functions, but hopefully their names are self-explanatory. At line 12 we use `obj0` to construct a NumPy array. At line 17, we check the validity of the result: that it is non-null and that it has a single dimension of arbitrary length. Once these states are verified, we extract the data buffer and length in lines 19 and 20 so that we can call the underlying C function at line 22. Line 25 performs memory management for the case where we have created a new array that is no longer needed.

This code has a significant amount of error handling. Note the `SWIG_fail` is a macro for `goto fail`, referring to the label at line 28. If the user provides the wrong number of arguments, this will be caught at line 10. If construction of the NumPy array fails or produces an array with the wrong number of dimensions, these errors are caught at line 17. And finally, if an error is detected, memory is still managed correctly at line 30.

Note that if the C function signature was in a different order:

```c
double rms(int n, double* seq);
```

that `SWIG` would not match the typemap signature given above with the argument list for `rms`. Fortunately, `numpy.i` has a set of typemaps with the data pointer given last:

```c
%apply (int DIM1, double* IN_ARRAY1) {((int n, double* seq))};
```

This simply has the effect of switching the definitions of `arg1` and `arg2` in lines 3 and 4 of the generated code above, and their assignments in lines 19 and 20.

### 7.1.2 Using numpy.i

The `numpy.i` file is currently located in the `numpy/docs/swig` sub-directory under the `numpy` installation directory. Typically, you will want to copy it to the directory where you are developing your wrappers.

A simple module that only uses a single `SWIG` interface file should include the following:

```c
{%
#define SWIG_FILE_WITH_INIT
%
#include "numpy.i"
%init {
import_array();
}%
```

Within a compiled Python module, `import_array()` should only get called once. This could be in a C/C++ file that you have written and is linked to the module. If this is the case, then none of your interface files should `#define SWIG_FILE_WITH_INIT` or call `import_array()`. Or, this initialization call could be in a wrapper file generated by `SWIG` from an interface file that has the `%init` block as above. If this is the case, and you have more than one `SWIG` interface file, then only one interface file should `#define SWIG_FILE_WITH_INIT` and call `import_array()`.

### 7.1.3 Available Typemaps

The typemap directives provided by `numpy.i` for arrays of different data types, say `double` and `int`, and dimensions of different types, say `int` or `long`, are identical to one another except for the C and NumPy type specifications. The typemaps are therefore implemented (typically behind the scenes) via a macro.
%numpy_typemaps (DATA_TYPE, DATA_TYPECODE, DIM_TYPE)

that can be invoked for appropriate (DATA_TYPE, DATA_TYPECODE, DIM_TYPE) triplets. For example:
%numpy_typemaps (double, NPY_DOUBLE, int)
%numpy_typemaps (int, NPY_INT, int)

The numpy.i interface file uses the %numpy_typemaps macro to implement typemaps for the following C data types and int dimension types:

- signed char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long
- unsigned long long
- float
- double

In the following descriptions, we reference a generic DATA_TYPE, which could be any of the C data types listed above, and DIM_TYPE which should be one of the many types of integers.

The typemap signatures are largely differentiated on the name given to the buffer pointer. Names with FARRAY are for Fortran-ordered arrays, and names with ARRAY are for C-ordered (or 1D arrays).

**Input Arrays**

Input arrays are defined as arrays of data that are passed into a routine but are not altered in-place or returned to the user. The Python input array is therefore allowed to be almost any Python sequence (such as a list) that can be converted to the requested type of array. The input array signatures are

1D:
- (DATA_TYPE IN_ARRAY1[ANY])
- (DATA_TYPE* IN_ARRAY1, int DIM1)
- (int DIM1, DATA_TYPE* IN_ARRAY1)

2D:
- (DATA_TYPE IN_ARRAY2[ANY][ANY])
- (DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2)
- (DATA_TYPE* IN_FARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* IN_FARRAY2)
3D:

- (DATA_TYPE IN_ARRAY3[ANY][ANY][ANY])
- (DATA_TYPE* IN_ARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_ARRAY3)
- (DATA_TYPE* IN_FARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_FARRAY3)

4D:

- (DATA_TYPE IN_ARRAY4[ANY][ANY][ANY][ANY])
- (DATA_TYPE* IN_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, , DIM_TYPE DIM4, DATA_TYPE* IN_ARRAY4)
- (DATA_TYPE* IN_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* IN_FARRAY4)

The first signature listed, (DATA_TYPE IN_ARRAY[ANY]) is for one-dimensional arrays with hard-coded dimensions. Likewise, (DATA_TYPE IN_ARRAY2[ANY][ANY]) is for two-dimensional arrays with hard-coded dimensions, and similarly for three-dimensional.

In-Place Arrays

In-place arrays are defined as arrays that are modified in-place. The input values may or may not be used, but the values at the time the function returns are significant. The provided Python argument must therefore be a NumPy array of the required type. The in-place signatures are

1D:

- (DATA_TYPE INPLACE_ARRAY1[ANY])
- (DATA_TYPE* INPLACE_ARRAY1, int DIM1)
- (int DIM1, DATA_TYPE* INPLACE_ARRAY1)

2D:

- (DATA_TYPE INPLACE_ARRAY2[ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* INPLACE_ARRAY2)
- (DATA_TYPE* INPLACE_FARRAY2, int DIM1, int DIM2)
- (int DIM1, int DIM2, DATA_TYPE* INPLACE_FARRAY2)

3D:

- (DATA_TYPE INPLACE_ARRAY3[ANY][ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY3, int DIM1, int DIM2, int DIM3)
- (int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_ARRAY3)
- (DATA_TYPE* INPLACE_FARRAY3, int DIM1, int DIM2, int DIM3)
4D:

- (DATA_TYPE INPLACE_ARRAY4[ANY][ANY][ANY][ANY])
- (DATA_TYPE* INPLACE_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* INPLACE_ARRAY4)
- (DATA_TYPE* INPLACE_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)
- (DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* INPLACE_FARRAY4)

These typemaps now check to make sure that the INPLACE_ARRAY arguments use native byte ordering. If not, an exception is raised.

**Argout Arrays**

Argout arrays are arrays that appear in the input arguments in C, but are in fact output arrays. This pattern occurs often when there is more than one output variable and the single return argument is therefore not sufficient. In Python, the conventional way to return multiple arguments is to pack them into a sequence (tuple, list, etc.) and return the sequence. This is what the argout typemaps do. If a wrapped function that uses these argout typemaps has more than one return argument, they are packed into a tuple or list, depending on the version of Python. The Python user does not pass these arrays in, they simply get returned. For the case where a dimension is specified, the python user must provide that dimension as an argument. The argout signatures are

1D:

- (DATA_TYPE ARGOUT_ARRAY1[ANY])
- (DATA_TYPE* ARGOUT_ARRAY1, int DIM1)
- (int DIM1, DATA_TYPE* ARGOUT_ARRAY1)

2D:

- (DATA_TYPE ARGOUT_ARRAY2[ANY][ANY])

3D:

- (DATA_TYPE ARGOUT_ARRAY3[ANY][ANY][ANY])

4D:

- (DATA_TYPE ARGOUT_ARRAY4[ANY][ANY][ANY][ANY])

These are typically used in situations where in C/C++, you would allocate a(n) array(s) on the heap, and call the function to fill the array(s) values. In Python, the arrays are allocated for you and returned as new array objects.

Note that we support DATA_TYPE* argout typemaps in 1D, but not 2D or 3D. This is because of a quirk with the SWIG typemap syntax and cannot be avoided. Note that for these types of 1D typemaps, the Python function will take a single argument representing DIM1.

**Argout View Arrays**

Argoutview arrays are for when your C code provides you with a view of its internal data and does not require any memory to be allocated by the user. This can be dangerous. There is almost no way to guarantee that the internal data
from the C code will remain in existence for the entire lifetime of the NumPy array that encapsulates it. If the user destroys the object that provides the view of the data before destroying the NumPy array, then using that array may result in bad memory references or segmentation faults. Nevertheless, there are situations, working with large data sets, where you simply have no other choice.

The C code to be wrapped for argoutview arrays are characterized by pointers: pointers to the dimensions and double pointers to the data, so that these values can be passed back to the user. The argoutview typemap signatures are therefore

1D:
- (DATA_TYPE** ARGOUTVIEW_ARRAY1, DIM_TYPE* DIM1 )
- (DIM_TYPE* DIM1, DATA_TYPE** ARGOUTVIEW_ARRAY1 )

2D:
- (DATA_TYPE** ARGOUTVIEW_ARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_ARRAY2 )
- (DATA_TYPE** ARGOUTVIEW_FARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_FARRAY2 )

3D:
- (DATA_TYPE** ARGOUTVIEW_ARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_ARRAY3)
- (DATA_TYPE** ARGOUTVIEW_FARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_FARRAY3)

4D:
- (DATA_TYPE** ARGOUTVIEW_ARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_ARRAY4)
- (DATA_TYPE** ARGOUTVIEW_FARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)
- (DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_FARRAY4)

Note that arrays with hard-coded dimensions are not supported. These cannot follow the double pointer signatures of these typemaps.

**Memory Managed Argout View Arrays**

A recent addition to numpy.i are typemaps that permit argout arrays with views into memory that is managed. See the discussion here.

1D:
- (DATA_TYPE** ARGOUTVIEWM_ARRAY1, DIM_TYPE* DIM1)
Output Arrays

The `numpy.i` interface file does not support typemaps for output arrays, for several reasons. First, C/C++ return arguments are limited to a single value. This prevents obtaining dimension information in a general way. Second, arrays with hard-coded lengths are not permitted as return arguments. In other words:

```c
double[3] newVector(double x, double y, double z);
```

is not legal C/C++ syntax. Therefore, we cannot provide typemaps of the form:

```c
%typemap(out) (TYPE[ANY]);
```

If you run into a situation where a function or method is returning a pointer to an array, your best bet is to write your own version of the function to be wrapped, either with `%extend` for the case of class methods or `%ignore` and `%rename` for the case of functions.

Other Common Types: `bool`

Note that C++ type `bool` is not supported in the list in the Available Typemaps section. NumPy bools are a single byte, while the C++ `bool` is four bytes (at least on my system). Therefore:
will result in typemaps that will produce code that reference improper data lengths. You can implement the following macro expansion:

%numpy_typemaps(bool, NPY_UINT, int)

to fix the data length problem, and Input Arrays will work fine, but In-Place Arrays might fail type-checking.

Other Common Types: complex

Typemap conversions for complex floating-point types is also not supported automatically. This is because Python and NumPy are written in C, which does not have native complex types. Both Python and NumPy implement their own (essentially equivalent) struct definitions for complex variables:

/* Python */
typedef struct {double real; double imag;} Py_complex;

/* NumPy */
typedef struct {float real, imag;} npy_cfloat;
typedef struct {double real, imag;} npy_cdouble;

We could have implemented:

%numpy_typemaps(Py_complex, NPY_CDOUBLE, int)
%numpy_typemaps(npy_cfloat, NPY_CFLOAT, int)
%numpy_typemaps(npy_cdouble, NPY_CDOUBLE, int)

which would have provided automatic type conversions for arrays of type Py_complex, npy_cfloat and npy_cdouble. However, it seemed unlikely that there would be any independent (non-Python, non-NumPy) application code that people would be using SWIG to generate a Python interface to, that also used these definitions for complex types. More likely, these application codes will define their own complex types, or in the case of C++, use std::complex. Assuming these data structures are compatible with Python and NumPy complex types, %numpy_typemap expansions as above (with the user’s complex type substituted for the first argument) should work.

7.1.4 NumPy Array Scalars and SWIG

SWIG has sophisticated type checking for numerical types. For example, if your C/C++ routine expects an integer as input, the code generated by SWIG will check for both Python integers and Python long integers, and raise an overflow error if the provided Python integer is too big to cast down to a C integer. With the introduction of NumPy scalar arrays into your Python code, you might conceivably extract an integer from a NumPy array and attempt to pass this to a SWIG-wrapped C/C++ function that expects an int, but the SWIG type checking will not recognize the NumPy array scalar as an integer. (Often, this does in fact work – it depends on whether NumPy recognizes the integer type you are using as inheriting from the Python integer type on the platform you are using. Sometimes, this means that code that works on a 32-bit machine will fail on a 64-bit machine.)

If you get a Python error that looks like the following:

TypeError: in method 'MyClass_MyMethod', argument 2 of type ‘int’

and the argument you are passing is an integer extracted from a NumPy array, then you have stumbled upon this problem. The solution is to modify the SWIG type conversion system to accept NumPy array scalars in addition to the standard integer types. Fortunately, this capability has been provided for you. Simply copy the file:

7.1. Numpy.i: a SWIG Interface File for NumPy
to the working build directory for your project, and this problem will be fixed. It is suggested that you do this anyway, as it only increases the capabilities of your Python interface.

Why is There a Second File?

The SWIG type checking and conversion system is a complicated combination of C macros, SWIG macros, SWIG typemaps and SWIG fragments. Fragments are a way to conditionally insert code into your wrapper file if it is needed, and not insert it if not needed. If multiple typemaps require the same fragment, the fragment only gets inserted into your wrapper code once.

There is a fragment for converting a Python integer to a C long. There is a different fragment that converts a Python integer to a C int, that calls the routine defined in the long fragment. We can make the changes we want here by changing the definition for the long fragment. SWIG determines the active definition for a fragment using a “first come, first served” system. That is, we need to define the fragment for long conversions prior to SWIG doing it internally. SWIG allows us to do this by putting our fragment definitions in the file pyfragments.swg. If we were to put the new fragment definitions in numpy.i, they would be ignored.

7.1.5 Helper Functions

The numpy.i file containes several macros and routines that it uses internally to build its typemaps. However, these functions may be useful elsewhere in your interface file. These macros and routines are implemented as fragments, which are described briefly in the previous section. If you try to use one or more of the following macros or functions, but your compiler complains that it does not recognize the symbol, then you need to force these fragments to appear in your code using:

```swig
%fragment("NumPy_Fragments");
```

in your SWIG interface file.

Macros

```swig
is_array(a)
```

Evaluates as true if `a` is non-NULL and can be cast to a PyArrayObject*.

```swig
array_type(a)
```

Evaluates to the integer data type code of `a`, assuming `a` can be cast to a PyArrayObject*.

```swig
array_numdims(a)
```

Evaluates to the integer number of dimensions of `a`, assuming `a` can be cast to a PyArrayObject*.

```swig
array_dimensions(a)
```

Evaluates to an array of type npy_intp and length array_numdims(a), giving the lengths of all of the dimensions of `a`, assuming `a` can be cast to a PyArrayObject*.

```swig
array_size(a,i)
```

Evaluates to the `i`-th dimension size of `a`, assuming `a` can be cast to a PyArrayObject*.

```swig
array_strides(a)
```

Evaluates to an array of type npy_intp and length array_numdims(a), giving the strides of all of the dimensions of `a`, assuming `a` can be cast to a PyArrayObject*. A stride is the distance in bytes between an element and its immediate neighbor along the same axis.
array_stride(a,i)
Evaluates to the i-th stride of a, assuming a can be cast to a PyArrayObject*.

array_data(a)
Evaluates to a pointer of type void* that points to the data buffer of a, assuming a can be cast to a PyArrayObject*.

array_descr(a)
Returns a borrowed reference to the dtype property (PyArray_Descr*) of a, assuming a can be cast to a PyArrayObject*.

array_flags(a)
Returns an integer representing the flags of a, assuming a can be cast to a PyArrayObject*.

array_enableflags(a,f)
Sets the flag represented by f of a, assuming a can be cast to a PyArrayObject*.

array_is_contiguous(a)
Evaluates as true if a is a contiguous array. Equivalent to (PyArray_ISCONTIGUOUS(a)).

array_is_native(a)
Evaluates as true if the data buffer of a uses native byte order. Equivalent to (PyArray_ISNOTSWAPPED(a)).

array_is_fortran(a)
Evaluates as true if a is FORTRAN ordered.

Routines

pytype_string()
Return type: const char*
Arguments:
• PyObject* py_obj, a general Python object.
Return a string describing the type of py_obj.

typecode_string()
Return type: const char*
Arguments:
• int typecode, a NumPy integer typecode.
Return a string describing the type corresponding to the NumPy typecode.

type_match()
Return type: int
Arguments:
• int actual_type, the NumPy typecode of a NumPy array.
• int desired_type, the desired NumPy typecode.
Make sure that actual_type is compatible with desired_type. For example, this allows character and byte types, or int and long types, to match. This is now equivalent to PyArray_EquivTypenums().

obj_to_array_no_conversion()
Return type: PyArrayObject*
Arguments:
  • PyObject* input, a general Python object.
  • int typecode, the desired NumPy typecode.

Cast input to a PyArrayObject* if legal, and ensure that it is of type typecode. If input cannot be cast, or the typecode is wrong, set a Python error and return NULL.

obj_to_array_allow_conversion()
Return type: PyArrayObject*
Arguments:
  • PyObject* input, a general Python object.
  • int typecode, the desired NumPy typecode of the resulting array.
  • int* is_new_object, returns a value of 0 if no conversion performed, else 1.

Convert input to a NumPy array with the given typecode. On success, return a valid PyArrayObject* with the correct type. On failure, the Python error string will be set and the routine returns NULL.

make_contiguous()
Return type: PyArrayObject*
Arguments:
  • PyArrayObject* ary, a NumPy array.
  • int* is_new_object, returns a value of 0 if no conversion performed, else 1.
  • int min_dims, minimum allowable dimensions.
  • int max_dims, maximum allowable dimensions.

Check to see if ary is contiguous. If so, return the input pointer and flag it as not a new object. If it is not contiguous, create a new PyArrayObject* using the original data, flag it as a new object and return the pointer.

make_fortran()
Return type: PyArrayObject*
Arguments
  • PyArrayObject* ary, a NumPy array.
  • int* is_new_object, returns a value of 0 if no conversion performed, else 1.

Check to see if ary is Fortran contiguous. If so, return the input pointer and flag it as not a new object. If it is not Fortran contiguous, create a new PyArrayObject* using the original data, flag it as a new object and return the pointer.

obj_to_array_contiguous_allow_conversion()
Return type: PyArrayObject*
Arguments:
  • PyObject* input, a general Python object.
  • int typecode, the desired NumPy typecode of the resulting array.
• int* is_new_object, returns a value of 0 if no conversion performed, else 1.

Convert input to a contiguous PyArrayObject* of the specified type. If the input object is not a contiguous PyArrayObject*, a new one will be created and the new object flag will be set.

**obj_to_array_fortran_allow_conversion()**

Return type: PyArrayObject*

Arguments:
• PyObject* input, a general Python object.
• int typecode, the desired NumPy typecode of the resulting array.
• int* is_new_object, returns a value of 0 if no conversion performed, else 1.

Convert input to a Fortran contiguous PyArrayObject* of the specified type. If the input object is not a Fortran contiguous PyArrayObject*, a new one will be created and the new object flag will be set.

**require_contiguous()**

Return type: int

Arguments:
• PyArrayObject* ary, a NumPy array.

Test whether ary is contiguous. If so, return 1. Otherwise, set a Python error and return 0.

**require_native()**

Return type: int

Arguments:
• PyArrayObject* ary, a NumPy array.

Require that ary is not byte-swapped. If the array is not byte-swapped, return 1. Otherwise, set a Python error and return 0.

**require_dimensions()**

Return type: int

Arguments:
• PyArrayObject* ary, a NumPy array.
• int exact_dimensions, the desired number of dimensions.

Require ary to have a specified number of dimensions. If the array has the specified number of dimensions, return 1. Otherwise, set a Python error and return 0.

**require_dimensions_n()**

Return type: int

Arguments:
• PyArrayObject* ary, a NumPy array.
• int* exact_dimensions, an array of integers representing acceptable numbers of dimensions.
• int n, the length of exact_dimensions.
Require \texttt{ary} to have one of a list of specified number of dimensions. If the array has one of the specified number of dimensions, return 1. Otherwise, set the Python error string and return 0.

\texttt{require\_size()}

Return type: \texttt{int}

Arguments:

- \texttt{PyArrayObject* ary}, a NumPy array.
- \texttt{npy\_int* size}, an array representing the desired lengths of each dimension.
- \texttt{int n}, the length of \texttt{size}.

Require \texttt{ary} to have a specified shape. If the array has the specified shape, return 1. Otherwise, set the Python error string and return 0.

\texttt{require\_fortran()}

Return type: \texttt{int}

Arguments:

- \texttt{PyArrayObject* ary}, a NumPy array.

Require the given \texttt{PyArrayObject} to be Fortran ordered. If the \texttt{PyArrayObject} is already Fortran ordered, do nothing. Else, set the Fortran ordering flag and recompute the strides.

### 7.1.6 Beyond the Provided Typemaps

There are many C or C++ array/NumPy array situations not covered by a simple \texttt{%include "numpy.i"} and subsequent \texttt{%apply} directives.

#### A Common Example

Consider a reasonable prototype for a dot product function:

\begin{verbatim}
double dot(int len, double* vec1, double* vec2);
\end{verbatim}

The Python interface that we want is:

\begin{verbatim}
def dot(vec1, vec2):
    """
    dot(PyObject,PyObject) -> double
    """
\end{verbatim}

The problem here is that there is one dimension argument and two array arguments, and our typemaps are set up for dimensions that apply to a single array (in fact, \texttt{SWIG} does not provide a mechanism for associating \texttt{len} with \texttt{vec2} that takes two Python input arguments). The recommended solution is the following:

\begin{verbatim}
%apply (int DIM1, double* IN\_ARRAY1) {(int len1, double* vec1),
                                        (int len2, double* vec2)}

%rename (dot) my\_dot;
%exception my\_dot {
    $action
    if (PyErr\_Occurred()) SWIG\_fail;
}
%inline %{
\end{verbatim}
double my_dot(int len1, double* vec1, int len2, double* vec2) {
    if (len1 != len2) {
        PyErr_Format(PyExc_ValueError,
                     "Arrays of lengths (%d,%d) given",
                     len1, len2);
        return 0.0;
    }
    return dot(len1, vec1, vec2);
}

If the header file that contains the prototype for `double dot()` also contains other prototypes that you want to wrap, so that you need to `%include` this header file, then you will also need a `%ignore dot;` directive, placed after the `%rename` and before the `%include` directives. Or, if the function in question is a class method, you will want to use `%extend` rather than `%inline` in addition to `%ignore`.

A note on error handling: Note that `my_dot` returns a `double` but that it can also raise a Python error. The resulting wrapper function will return a Python float representation of 0.0 when the vector lengths do not match. Since this is not NULL, the Python interpreter will not know to check for an error. For this reason, we add the `%exception` directive above for `my_dot` to get the behavior we want (note that `$action` is a macro that gets expanded to a valid call to `my_dot`). In general, you will probably want to write a SWIG macro to perform this task.

Other Situations

There are other wrapping situations in which `numpy.i` may be helpful when you encounter them.

- In some situations, it is possible that you could use the `%numpy_typemaps` macro to implement typemaps for your own types. See the Other Common Types: bool or Other Common Types: complex sections for examples. Another situation is if your dimensions are of a type other than `int` (say `long` for example):

  `%numpy_typemaps(double, NPY_DOUBLE, long)`

- You can use the code in `numpy.i` to write your own typemaps. For example, if you had a five-dimensional array as a function argument, you could cut-and-paste the appropriate four-dimensional typemaps into your interface file. The modifications for the fourth dimension would be trivial.

- Sometimes, the best approach is to use the `%extend` directive to define new methods for your classes (or overload existing ones) that take a `PyObject*` (that either is or can be converted to a `PyArrayObject*`) instead of a pointer to a buffer. In this case, the helper routines in `numpy.i` can be very useful.

- Writing typemaps can be a bit nonintuitive. If you have specific questions about writing SWIG typemaps for NumPy, the developers of `numpy.i` do monitor the Numpy-discussion and Swig-user mail lists.

A Final Note

When you use the `%apply` directive, as is usually necessary to use `numpy.i`, it will remain in effect until you tell SWIG that it shouldn’t be. If the arguments to the functions or methods that you are wrapping have common names, such as `length` or `vector`, these typemaps may get applied in situations you do not expect or want. Therefore, it is always a good idea to add a `%clear` directive after you are done with a specific typemap:

```swig
%apply (double* IN_ARRAY1, int DIM1) {((double* vector, int length))}
%include "my_header.h"
%clear (double* vector, int length);
```

In general, you should target these typemap signatures specifically where you want them, and then clear them after you are done.
7.1.7 Summary

Out of the box, numpy.i provides typemaps that support conversion between NumPy arrays and C arrays:

- That can be one of 12 different scalar types: signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float and double.
- That support 74 different argument signatures for each data type, including:
  - One-dimensional, two-dimensional, three-dimensional and four-dimensional arrays.
  - Input-only, in-place, argout, argoutview, and memory managed argoutview behavior.
  - Both C-ordering (“last dimension fastest”) or Fortran-ordering (“first dimension fastest”) support for 2D, 3D and 4D arrays.

The numpy.i interface file also provides additional tools for wrapper developers, including:

- A SWIG macro (%numpy_typemaps) with three arguments for implementing the 74 argument signatures for the user’s choice of (1) C data type, (2) NumPy data type (assuming they match), and (3) dimension type.
- Fourteen C macros and fifteen C functions that can be used to write specialized typemaps, extensions, or inlined functions that handle cases not covered by the provided typemaps. Note that the macros and functions are coded specifically to work with the NumPy C/API regardless of NumPy version number, both before and after the deprecation of some aspects of the API after version 1.6.

7.2 Testing the numpy.i Typemaps

7.2.1 Introduction

Writing tests for the numpy.i SWIG interface file is a combinatorial headache. At present, 12 different data types are supported, each with 74 different argument signatures, for a total of 888 typemaps supported “out of the box”. Each of these typemaps, in turn, might require several unit tests in order to verify expected behavior for both proper and improper inputs. Currently, this results in 1,427 individual unit tests that are performed when make test is run in the numpy/docs/swig subdirectory.

To facilitate this many similar unit tests, some high-level programming techniques are employed, including C and SWIG macros, as well as Python inheritance. The purpose of this document is to describe the testing infrastructure employed to verify that the numpy.i typemaps are working as expected.

7.2.2 Testing Organization

There are three independent testing frameworks supported, for one-, two-, and three-dimensional arrays respectively. For one-dimensional arrays, there are two C++ files, a header and a source, named:

Vector.h
Vector.cxx

that contain prototypes and code for a variety of functions that have one-dimensional arrays as function arguments.

The file:

Vector.i
is a SWIG interface file that defines a python module Vector that wraps the functions in Vector.h while utilizing the typemaps in numpy.i to correctly handle the C arrays.

The Makefile calls swig to generate Vector.py and Vector_wrap.cxx, and also executes the setup.py script that compiles Vector_wrap.cxx and links together the extension module _Vector.so or _Vector.dylib, depending on the platform. This extension module and the proxy file Vector.py are both placed in a subdirectory under the build directory.

The actual testing takes place with a Python script named:

testVector.py

that uses the standard Python library module unittest, which performs several tests of each function defined in Vector.h for each data type supported.

Two-dimensional arrays are tested in exactly the same manner. The above description applies, but with Matrix substituted for Vector. For three-dimensional tests, substitute Tensor for Vector. For four-dimensional tests, substitute SuperTensor for Vector. For the descriptions that follow, we will reference the Vector tests, but the same information applies to Matrix, Tensor and SuperTensor tests.

The command make test will ensure that all of the test software is built and then run all three test scripts.

### 7.2.3 Testing Header Files

Vector.h is a C++ header file that defines a C macro called TEST_FUNC_PROTOS that takes two arguments: TYPE, which is a data type name such as unsigned int; and SNAME, which is a short name for the same data type with no spaces, e.g. uint. This macro defines several function prototypes that have the prefix SNAME and have at least one argument that is an array of type TYPE. Those functions that have return arguments return a TYPE value.

TEST_FUNC_PROTOS is then implemented for all of the data types supported by numpy.i:

- signed char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long
- unsigned long long
- float
- double

### 7.2.4 Testing Source Files

Vector.cxx is a C++ source file that implements compilable code for each of the function prototypes specified in Vector.h. It defines a C macro TEST_FUNCS that has the same arguments and works in the same way as TEST_FUNC_PROTOS does in Vector.h. TEST_FUNCS is implemented for each of the 12 data types as above.
### 7.2.5 Testing SWIG Interface Files

Vector.i is a SWIG interface file that defines python module Vector. It follows the conventions for using numpy.i as described in this chapter. It defines a SWIG macro `%apply_numpy_typemaps` that has a single argument `TYPE`. It uses the SWIG directive `%apply` to apply the provided typemaps to the argument signatures found in Vector.h. This macro is then implemented for all of the data types supported by numpy.i. It then does a `%include "Vector.h"` to wrap all of the function prototypes in Vector.h using the typemaps in numpy.i.

### 7.2.6 Testing Python Scripts

After `make` is used to build the testing extension modules, testVector.py can be run to execute the tests. As with other scripts that use `unittest` to facilitate unit testing, testVector.py defines a class that inherits from `unittest.TestCase`:

```python
class VectorTestCase(unittest.TestCase):
    pass
```

However, this class is not run directly. Rather, it serves as a base class to several other python classes, each one specific to a particular data type. The `VectorTestCase` class stores two strings for typing information:

- `self.typeStr`: A string that matches one of the `SNAME` prefixes used in Vector.h and Vector.cxx. For example, "double".
- `self.typeCode`: A short (typically single-character) string that represents a data type in numpy and corresponds to `self.typeStr`. For example, if `self.typeStr` is "double", then `self.typeCode` should be "d".

Each test defined by the `VectorTestCase` class extracts the python function it is trying to test by accessing the Vector module’s dictionary:

```python
length = Vector.__dict__[self.typeStr + "Length"]
```

In the case of double precision tests, this will return the python function `Vector.doubleLength`.

We then define a new test case class for each supported data type with a short definition such as:

```python
class doubleTestCase(VectorTestCase):
    def __init__(self, methodName="runTest"): VectorTestCase.__init__(self, methodName)
        self.typeStr = "double"
        self.typeCode = "d"
```

Each of these 12 classes is collected into a `unittest.TestSuite`, which is then executed. Errors and failures are summed together and returned as the exit argument. Any non-zero result indicates that at least one test did not pass.
ACKNOWLEDGEMENTS

Large parts of this manual originate from Travis E. Oliphant’s book *Guide to Numpy* (which generously entered Public Domain in August 2008). The reference documentation for many of the functions are written by numerous contributors and developers of Numpy, both prior to and during the Numpy Documentation Marathon.

Please help to improve NumPy’s documentation! Instructions on how to join the ongoing documentation marathon can be found on the scipy.org website.
BIBLIOGRAPHY


    http://mathworld.wolfram.com/BinomialDistribution.html
[R128] David McKay, “Information Theory, Inference and Learning Algorithms,” chapter 23,
    http://www.inference.phy.cam.ac.uk/mackay/
    http://mathworld.wolfram.com/GammaDistribution.html
    http://mathworld.wolfram.com/HypergeometricDistribution.html
[R140] Abramowitz, M. and Stegun, I. A. (Eds.). Handbook of Mathematical Functions with Formulas, Graphs, and  
[R141] The Laplace distribution and generalizations By Samuel Kotz, Tomasz J. Kozubowski, Krzysztof Podgorski,  
    http://mathworld.wolfram.com/LaplaceDistribution.html
    and Other Fields, Birkhauser Verlag, Basel, pp 132-133.
    http://mathworld.wolfram.com/LogisticDistribution.html
[R147] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution  
    of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September  


INDEX

Symbols
__abs__ (numpy.ma.MaskedArray attribute), 290
__abs__ (numpy.ndarray attribute), 66
__add__ (numpy.ndarray attribute), 67
__add__() (numpy.ma.MaskedArray method), 290
__and__ (numpy.ma.MaskedArray attribute), 291
__and__ (numpy.ndarray attribute), 67
__array__() (in module numpy), 119
__array__() (numpy.generic method), 86
__array__() (numpy.ma.MaskedArray method), 294
__array__() (numpy.ndarray method), 69
__array_finalize__() (in module numpy), 119
__array_interface__ (built-in variable), 419
__array_interface__ (numpy.generic attribute), 75
__array_prepare__() (in module numpy), 119
__array_priority__ (in module numpy), 119
__array_priority__ (numpy.generic attribute), 75
__array_priority__ (numpy.ma.MaskedArray attribute), 258
__array_struct__ (C variable), 421
__array_struct__ (numpy.generic attribute), 75
__array_wrap__() (in module numpy), 119
__array_wrap__() (numpy.generic method), 76, 86
__array_wrap__() (numpy.ma.MaskedArray method), 294
__array_wrap__() (numpy.ndarray method), 70
__call__() (numpy.polynomial.polynomial.Polynomial attribute), 957
__call__() (numpy.polynomial.chebyshev.Chebyshev method), 982
__call__() (numpy.polynomial.hermite.Hermite method), 1064
__call__() (numpy.polynomial.laguerre.Laguerre method), 1037
__call__() (numpy.polynomial.legendre.Legendre method), 1010
__call__() (numpy.polynomial.polynomial.Polynomial method), 957
__call__() (numpy.vectorize method), 643
__contains__ (numpy.ma.MaskedArray attribute), 295
__contains__ (numpy.ndarray attribute), 70
__copy__() (numpy.ma.MaskedArray method), 293
__copy__() (numpy.ndarray method), 69
__deepcopy__() (numpy.ma.MaskedArray method), 293
__deepcopy__() (numpy.ndarray method), 69
__delitem__ (numpy.ma.MaskedArray attribute), 294
__div__ (numpy.ndarray attribute), 67
__div__() (numpy.ma.MaskedArray method), 290
__divmod__ (numpy.ma.MaskedArray attribute), 290
__divmod__ (numpy.ndarray attribute), 67
__eq__ (numpy.ndarray attribute), 66
__eq__() (numpy.ma.MaskedArray method), 289
__floordiv__ (numpy.ndarray attribute), 67
__floordiv__() (numpy.ma.MaskedArray method), 290
__getitem__ (numpy.ndarray attribute), 66
__getitem__() (numpy.ma.MaskedArray method), 289
__hex__ (numpy.ndarray attribute), 70
__hex__() (numpy.ma.MaskedArray method), 259
__iadd__ (numpy.ndarray attribute), 68
__iadd__() (numpy.ma.MaskedArray method), 292
__ifloordiv__ (numpy.ndarray attribute), 68
__ifloordiv__() (numpy.ma.MaskedArray method), 292
__imul__ (numpy.ndarray attribute), 68
__iadd__() (numpy.ma.MaskedArray method), 291
__iand__ (numpy.ndarray attribute), 68
__iand__() (numpy.ma.MaskedArray method), 291
__ilshift__ (numpy.ndarray attribute), 68
__ilshift__() (numpy.ma.MaskedArray method), 291
__imod__ (numpy.ndarray attribute), 68
__imod__() (numpy.ma.MaskedArray method), 291
__imul__ (numpy.ndarray attribute), 68
__int__ (numpy.ndarray attribute), 70
A (numpy.matrix attribute), 120, 121
A1 (numpy.matrix attribute), 121
absolute (in module numpy), 930
abspath() (numpy.DataSource method), 702
accumulate
  ufunc methods, 1417
accumulate() (numpy.ufunc method), 442
add (in module numpy), 917
add() (in module numpy.core.defchararray), 532
add_data_dir() (numpy.distutils.misc_util.Configuration method), 1309
add_data_files() (numpy.distutils.misc_util.Configuration method), 1307
add_extension() (numpy.distutils.misc_util.Configuration method), 1310
add_headers() (numpy.distutils.misc_util.Configuration method), 1310
add_include_dirs() (numpy.distutils.misc_util.Configuration method), 1307
add_installed_library() (numpy.distutils.misc_util.Configuration method), 1310
add_library() (numpy.distutils.misc_util.Configuration method), 1311
add_npy_pkg_config() (numpy.distutils.misc_util.Configuration method), 1312

\_int__ (numpy.ma.MaskedArray method), 259
\_invert__ (numpy.ndarray attribute), 66
\_ior__ (numpy.ma.MaskedArray attribute), 292
\_ior__ (numpy.ndarray attribute), 68
\_ipow__ (numpy.ndarray attribute), 68
\_ipow__ (numpy.ma.MaskedArray method), 292
\_irshift__ (numpy.ma.MaskedArray attribute), 292
\_irshift__ (numpy.ndarray attribute), 68
\_isub__ (numpy.ndarray attribute), 68
\_isub__ (numpy.ma.MaskedArray method), 291
\_itruediv__ (numpy.ma.MaskedArray attribute), 68
\_itruediv__ (numpy.ndarray attribute), 68
\_ixor__ (numpy.ma.MaskedArray attribute), 292
\_ixor__ (numpy.ndarray attribute), 68
\_le__ (numpy.ma.MaskedArray attribute), 289
\_le__ (numpy.ndarray attribute), 65
\_len__ (numpy.ma.MaskedArray attribute), 294
\_len__ (numpy.ndarray attribute), 70
\_long__ (numpy.ma.MaskedArray attribute), 259
\_long__ (numpy.ndarray attribute), 70
\_lshift__ (numpy.ma.MaskedArray attribute), 291
\_lshift__ (numpy.ndarray attribute), 67
\_lt__ (numpy.ma.MaskedArray attribute), 289
\_lt__ (numpy.ndarray attribute), 65
\_mod__ (numpy.ma.MaskedArray attribute), 290
\_mod__ (numpy.ndarray attribute), 67
\_mul__ (numpy.ndarray attribute), 67
\_mul__ (numpy.ma.MaskedArray attribute), 290
\_ne__ (numpy.ndarray attribute), 66
\_ne__ (numpy.ma.MaskedArray method), 289
\_neg__ (numpy.ndarray attribute), 66
\_new__ (numpy.ma.MaskedArray static method), 294
\_new__ (numpy.ndarray static method), 69
\_nonzero__ (numpy.ma.MaskedArray attribute), 289
\_nonzero__ (numpy.ndarray attribute), 66
\_oct__ (numpy.ma.MaskedArray attribute), 259
\_oct__ (numpy.ndarray attribute), 70
\_or__ (numpy.ma.MaskedArray attribute), 291
\_or__ (numpy.ndarray attribute), 67
\_pos__ (numpy.ndarray attribute), 66
\_pow__ (numpy.ndarray attribute), 67
\_pow__ (numpy.ma.MaskedArray method), 291
\_radd__ (numpy.ma.MaskedArray method), 290
\_rand__ (numpy.ma.MaskedArray attribute), 291
\_rdiv__ (numpy.ma.MaskedArray attribute), 290
\_rdivmod__ (numpy.ma.MaskedArray attribute), 290
\_reduce__ (numpy.dtype method), 102
\_reduce__ (numpy.generic method), 87
\_reduce__ (numpy.ma.MaskedArray attribute), 293
\_reduce__ (numpy.ndarray method), 69
\_repr__ (numpy.ndarray attribute), 71
\_repr__ (numpy.ma.MaskedArray attribute), 291
\_rfloordiv__ (numpy.ma.MaskedArray method), 292
\_rlshift__ (numpy.ma.MaskedArray attribute), 291
\_rmod__ (numpy.ma.MaskedArray attribute), 290
\_rmul__ (numpy.ma.MaskedArray method), 290
\_ror__ (numpy.ma.MaskedArray attribute), 291
\_rpow__ (numpy.ma.MaskedArray method), 291
\_rrshift__ (numpy.ma.MaskedArray attribute), 291
\_rshift__ (numpy.ma.MaskedArray attribute), 291
\_rshift__ (numpy.ndarray attribute), 67
\_rsub__ (numpy.ma.MaskedArray method), 290
\_rxor__ (numpy.ma.MaskedArray attribute), 291
\_setitem__ (numpy.ndarray attribute), 70
\_setitem__ (numpy.ma.MaskedArray attribute), 294
\_setmask__ (numpy.ma.MaskedArray method), 290
\_setslice__ (numpy.ndarray attribute), 70
\_setslice__ (numpy.ma.MaskedArray method), 295
\_str__ (numpy.ndarray attribute), 71
\_str__ (numpy.ma.MaskedArray method), 292
\_sub__ (numpy.ndarray attribute), 67
\_sub__ (numpy.ma.MaskedArray method), 290
\_truediv__ (numpy.ndarray attribute), 67
\_truediv__ (numpy.ma.MaskedArray method), 290
\_xor__ (numpy.ma.MaskedArray attribute), 291
\_xor__ (numpy.ndarray attribute), 67

A (numpy.matrix attribute), 120, 121
A1 (numpy.matrix attribute), 121
absolute (in module numpy), 930
abspath() (numpy.DataSource method), 702
accumulate
  ufunc methods, 1417
accumulate() (numpy.ufunc method), 442
add (in module numpy), 917
add() (in module numpy.core.defchararray), 532
add_data_dir() (numpy.distutils.misc_util.Configuration method), 1309
add_data_files() (numpy.distutils.misc_util.Configuration method), 1307
add_extension() (numpy.distutils.misc_util.Configuration method), 1310
add_headers() (numpy.distutils.misc_util.Configuration method), 1310
add_include_dirs() (numpy.distutils.misc_util.Configuration method), 1307
add_installed_library() (numpy.distutils.misc_util.Configuration method), 1310
add_library() (numpy.distutils.misc_util.Configuration method), 1311
add_npy_pkg_config() (numpy.distutils.misc_util.Configuration method), 1312
asmatrix() (in module numpy), 155, 462, 500
asscalar() (in module numpy), 502
assert_allclose() (in module numpy.testing), 1287
assert_almost_equal() (in module numpy.testing), 1284
assert_approx_equal() (in module numpy.testing), 1285
assert_array_almost_equal() (in module numpy.testing), 1286
assert_array_almost_equal_nulp() (in module numpy.testing), 1288
assert_array_equal() (in module numpy.testing), 1289
assert_array_less() (in module numpy.testing), 1290
assert_array_max_ulp() (in module numpy.testing), 1288
assert_equal() (in module numpy.testing), 1291
assert_raises() (in module numpy.testing), 1292
assert_string_equal() (in module numpy.testing), 1292
assert_warns() (in module numpy.testing), 1292
astype() (numpy.chararray method), 170
astype() (numpy.core.defchararray.chararray method), 559
astype() (numpy.generic method), 79
astype() (numpy.lib.user_array.container method), 227
astype() (numpy.ma.MaskedArray method), 261
astype() (numpy.matrix method), 131
astype() (numpy.ndarray method), 15, 48
astype() (numpy.recarray method), 199
astype() (numpy.record method), 220
at() (numpy.ufunc method), 445
atleast_1d (in module numpy.ma), 325, 781
atleast_1d() (in module numpy), 492
atleast_2d (in module numpy.ma), 325, 781
atleast_2d() (in module numpy), 492
atleast_3d (in module numpy.ma), 326, 782
atleast_3d() (in module numpy), 493
attributes
  ufunc, 438
average() (in module numpy), 1263
average() (in module numpy.ma), 373, 829
axis, 61
B
bartlett() (in module numpy), 1296
base, 3
base (numpy.chararray attribute), 162
base (numpy.core.defchararray.chararray attribute), 551
base (numpy.dtype attribute), 90, 589
base (numpy.generic attribute), 75, 77
base (numpy.ma.MaskedArray attribute), 253
base (numpy.matrix attribute), 121
base (numpy.ndarray attribute), 13, 39
base (numpy.recarray attribute), 190
base (numpy.record attribute), 217
base_repr() (in module numpy), 701
baseclass (numpy.ma.MaskedArray attribute), 252
basis() (numpy.polynomial.chebyshev.Chebyshev static method), 982
basis() (numpy.polynomial.hermite.Hermite static method), 1064
basis() (numpy.polynomial.hermite_e.HermiteE static method), 1092
basis() (numpy.polynomial.laguerre.Laguerre static method), 1037
basis() (numpy.polynomial.legendre.Legendre static method), 1010
basis() (numpy.polynomial.polynomial.Polynomial static method), 957
beta() (in module numpy.random), 1141
beta() (numpy.random.RandomState method), 1184
binary_repr() (in module numpy), 530, 700
bincount() (in module numpy), 1281
binomial() (in module numpy.random), 1141
binomial() (numpy.random.RandomState method), 1184
bitwise_and (in module numpy), 524
bitwise_or (in module numpy), 524
bitwise_xor (in module numpy), 525
blackman() (in module numpy), 1297
blue_text() (in module numpy.distutils.misc_util), 1306
bmint() (in module numpy), 155, 484
broadcast (class in numpy), 230, 494
broadcast_arrays() (in module numpy), 496
broadcastable, 431
broadcasting, 431, 1412
buffers, 432
busday_count() (in module numpy), 581
busday_offset() (in module numpy), 579
busdaycalendar (class in numpy), 577
byteorder (numpy.dtype attribute), 98
bytes() (in module numpy.random), 1139
bytes() (numpy.random.RandomState method), 1185
byteswap() (numpy.generic method), 79, 87
byteswap() (numpy.lib.user_array.container method), 227
byteswap() (numpy.ma.MaskedArray method), 261
byteswap() (numpy.matrix method), 132
byteswap() (numpy.ndarray method), 16, 49
byteswap() (numpy.recarray method), 200
byteswap() (numpy.record method), 220
C
C-API
  array, 1340
    iterator, 1379, 1395
    ndarray, 1340, 1379
    ufunc, 1395, 1401
C-order, 35
c_(in module numpy), 648
can_cast() (in module numpy), 582
capitalize() (in module numpy.core.defchararray), 532

cast() (numpy.polynomial.chebyshev.Chebyshev static method), 983

chebadd() (in module numpy.polynomial.chebyshev), 1001
chebaddcompanion() (in module numpy.polynomial.chebyshev), 1006
chebder() (in module numpy.polynomial.chebyshev), 999
chebderivative() (in module numpy.polynomial.chebyshev), 1003
chebdomain (in module numpy.polynomial.chebyshev), 1006
chebfit() (in module numpy.polynomial.chebyshev), 995
chebfroducts() (in module numpy.polynomial.chebyshev), 994
chebgauss() (in module numpy.polynomial.chebyshev), 1004
chebgrid2d() (in module numpy.polynomial.chebyshev), 992
chebgrid3d() (in module numpy.polynomial.chebyshev), 993
chebint() (in module numpy.polynomial.chebyshev), 1000
chebline() (in module numpy.polynomial.chebyshev), 1007
chebmul() (in module numpy.polynomial.chebyshev), 1002
chebmutk() (in module numpy.polynomial.chebyshev), 1003
chebpol() (in module numpy.polynomial.chebyshev), 1006
chebpower() (in module numpy.polynomial.chebyshev), 1004
chebroots() (in module numpy.polynomial.chebyshev), 994
chebsub() (in module numpy.polynomial.chebyshev), 1002
chebrtrim() (in module numpy.polynomial.chebyshev), 1006
chebval() (in module numpy.polynomial.chebyshev), 990
chebval2d() (in module numpy.polynomial.chebyshev), 991
chebval3d() (in module numpy.polynomial.chebyshev), 991
chebweight() (in module numpy.polynomial.chebyshev), 998
chebweight() (in module numpy.polynomial.chebyshev), 1005
chebxf (in module numpy.polynomial.chebyshev), 1006
Chebyshef (class in numpy.polynomial.chebyshev), 981
chebzero (in module numpy.polynomial.chebyshev), 1006

chisquare() (in module numpy.random), 1142
chisquare() (numpy.random.RandomState method), 1185
chisquare() (numpy.random.RandomState method), 1137
cholesky() (in module numpy.linalg), 715
choose() (in module numpy), 663
choose() (in module numpy.ma), 416, 872
choose() (numpy.generic method), 79
choose() (numpy.ma.MaskedArray method), 270
choose() (numpy.matrix method), 133
choose() (numpy.ndarray method), 17, 58
choose() (numpy.recarray method), 201
choose() (numpy.record method), 220
clip (numpvma.MaskedArray attribute), 279, 412, 868
clip() (in module numpy), 928
clip() (in module numpy.ma), 411, 867
clip() (numpy.generic method), 80
clip() (numpy.matrix method), 133
clip() (numpy.ndarray method), 17, 63
clip() (numpy.recarray method), 201
code generation, 1317
column-major, 35
column_stack (in module numpy.ma), 328, 334, 784, 790
column_stack() (in module numpy), 503
common_fill_value() (in module numpy.ma), 367, 823
common_type() (in module numpy), 586
comparison, 65, 288
compress() (in module numpy), 665
compress() (numpy.generic method), 80
compress() (numpy.ma.MaskedArray method), 270
compress() (numpy.matrix method), 133
compress() (numpy.ndarray method), 17, 61
compress() (numpy.recarray method), 201
Index
construction from string, 94
construction from tuple, 95
construction from type, 93
field, 88
record, 88
scalar, 88
sub-array, 88, 96
dtype (class in numpy), 89, 587
dtype (numpy.chararray attribute), 163
dtype (numpy.core.defchararray.chararray attribute), 553
dtype (numpy.generic attribute), 75, 77
dtype (numpy.ma.MaskedArray attribute), 254
dtype (numpy.matrix attribute), 123
dtype (numpy.ndarray attribute), 6, 40
dtype (numpy.recarray attribute), 192
dtype (numpy.record attribute), 217
dump() (in module numpy.ma), 366, 822
dump() (numpy.chararray method), 171
dump() (numpy.core.defchararray.chararray method), 560
dump() (numpy.generic method), 81
dump() (numpy.ma.MaskedArray method), 277
dump() (numpy.matrix method), 135
dump() (numpy.ndarray method), 19, 48
dump() (numpy.recarray method), 203
dump() (numpy.record method), 221
dumps() (in module numpy.ma), 366, 822
dumps() (numpy.chararray method), 172
dumps() (numpy.core.defchararray.chararray method), 561
dumps() (numpy.generic method), 81
dumps() (numpy.ma.MaskedArray method), 277
dumps() (numpy.matrix method), 135
dumps() (numpy.ndarray method), 19, 48
dumps() (numpy.recarray method), 203
dumps() (numpy.record method), 221
ediff1d() (in module numpy), 901
ediff1d() (in module numpy.ma), 417, 873
eig() (in module numpy.linalg), 719
eigh() (in module numpy.linalg), 721
eigval() (in module numpy.linalg), 722
eigvalsh() (in module numpy.linalg), 723
einsum() (in module numpy), 710
ellipsis, 103
empty (in module numpy.ma), 302, 758
empty() (in module numpy), 451
empty() (in module numpy.matlib), 939
empty_like (in module numpy.ma), 302, 758
empty_like() (in module numpy), 452
enable_external_loop() (numpy.nditer method), 677
encode() (in module numpy.core.defchararray), 534
encode() (numpy.chararray method), 172
find() (numpy.chararray method), 172
find() (numpy.core.defchararray.chararray method), 561
find_common_type() (in module numpy), 597
finfo (class in numpy), 592
fit() (numpy.polynomial.chebyshev.Chebyshev method), 985
fit() (numpy.polynomial.hermite.Hermite method), 1066
fit() (numpy.polynomial.hermite_e.HermiteE method), 1094
fit() (numpy.polynomial.laguerre.Laguerre method), 1039
fit() (numpy.polynomial.legendre.Legendre method), 1012
fit() (numpy.polynomial.polynomial.Polynomial method), 960
fix() (in module numpy), 892
fix_invalid() (in module numpy.ma), 237, 352, 808
flags (numpy.chararray attribute), 164
flags (numpy.core.defchararray.chararray attribute), 553
flags (numpy.dtype attribute), 100
flags (numpy.generic attribute), 75, 77
flags (numpy.ma.MaskedArray attribute), 255
flags (numpy.matrix attribute), 123
flags (numpy.ndarray attribute), 7, 36
flags (numpy.recarray attribute), 192
flags (numpy.record attribute), 217
flat (numpy.chararray attribute), 165
flat (numpy.core.defchararray.chararray attribute), 554
flat (numpy.dtype attribute), 75, 77
flat (numpy.generic attribute), 258
flat (numpy.matrix attribute), 124
flat (numpy.ndarray attribute), 8, 41, 228, 488
flat (numpy.recarray attribute), 193
flat (numpy.record attribute), 217
flatiter (class in numpy), 678
flatnonzero() (in module numpy), 1251
flatnotmasked() (in module numpy.ma), 343, 799
flatnotmasked_contiguous() (in module numpy.ma), 343, 799
flatnotmasked_edges() (in module numpy.ma), 344, 800
flatten (numpy.ma.MaskedArray attribute), 265, 321, 777
flattent() (numpy.chararray method), 172
flattent() (numpy.core.defchararray.chararray method), 561
flattent() (numpy.dtype attribute), 81
flattent() (numpy.matrix method), 136
flattent() (numpy.ndarray method), 20, 56, 489
flattent() (numpy.recarray method), 203
flattent() (numpy.record method), 222
fliplr() (in module numpy), 519
flipud() (in module numpy), 520
floor (in module numpy), 892
floor_divide (in module numpy), 922
flush() (numpy.memmap method), 159, 696
fmax (in module numpy), 934
fmin (in module numpy), 935
fmod (in module numpy), 922
format_parser (class in numpy), 591
Fortran-order, 35
frexp (in module numpy), 914
from dict
dtype construction, 96
dtype type, 96
from dtypes
dtype construction, 93
from list
dtype construction, 96
from None
dtype construction, 93
from string
dtype construction, 94
from tuple
dtype construction, 95
from type
dtype construction, 93
fromarrays() (in module numpy.core.records), 469
frombuffer (in module numpy.ma), 299, 755
frombuffer() (in module numpy), 463
fromfile() (in module numpy), 464
fromfile() (in module numpy.core.records), 470
fromfunction (in module numpy.ma), 300, 756
fromfunction() (in module numpy), 465
fromiter() (in module numpy), 466
frompyfunc() (in module numpy), 643
fromrecords() (in module numpy.core.records), 469
fromregex() (in module numpy.ma), 300, 756
fromroots() (numpy.polynomial.chebyshev.Chebyshev method), 986
fromroots() (numpy.polynomial.hermite.Hermite method), 1068
fromroots() (numpy.polynomial.hermite_e.HermiteE method), 1095
fromroots() (numpy.polynomial.laguerre.Laguerre method), 1040
fromroots() (numpy.polynomial.legendre.Legendre method), 1013
fromstring() (in module numpy), 466, 691
fromstring() (in module numpy.core.records), 470
fv() (in module numpy), 630
G
gamma() (in module numpy.random), 1146
gamma() (numpy.random.RandomState method), 1190
generate_config_py() (in module numpy.distutils.misc_util), 1306
generic (class in numpy), 76
genfromtxt() (in module numpy), 687
geometric() (in module numpy.random), 1147
geometric() (numpy.random.RandomState method), 1191
get_build_temp_dir() (numpy.distutils.misc_util.Configuration method), 1313
get_cmd() (in module numpy.distutils.misc_util), 1306
get_config_cmd() (numpy.distutils.misc_util.Configuration method), 1313
get_dependencies() (in module numpy.distutils.misc_util), 1306
get_distribution() (numpy.distutils.misc_util.Configuration method), 1307
get_ext_source_files() (in module numpy.distutils.misc_util), 1306
get_fill_value() (numpy.ma.MaskedArray method), 296, 370, 827
get_info() (in module numpy.distutils.system_info), 1314
get_info() (numpy.distutils.misc_util.Configuration method), 1314
get_numpy_include_dirs() (in module numpy.distutils.misc_util), 1305
get_printoptions() (in module numpy), 699
get_script_files() (in module numpy.distutils.misc_util), 1306
get_standard_file() (in module numpy.distutils.system_info), 1314
get_state() (in module numpy.random), 1233
get_state() (numpy.random.RandomState method), 1192
get_subpackage() (numpy.distutils.misc_util.Configuration method), 1307
get_version() (numpy.distutils.misc_util.Configuration method), 1314
getA() (numpy.matrix method), 136
getA1() (numpy.matrix method), 137
getbuffer() (in module numpy), 945
getbufsize() (in module numpy), 946
getdata() (in module numpy.ma), 311, 767
geterr() (in module numpy), 603
geterrcall() (in module numpy), 605
geterrobj() (in module numpy), 607
getfield() (numpy.characterarray method), 173
getfield() (numpy.core.defchararray.characterarray method), 562
getfield() (numpy.generic method), 81
getfield() (numpy.matrix method), 138
getfield() (numpy.ndarray method), 20, 52
getfield() (numpy.recarray method), 204
getfield() (numpy.record method), 222
getH() (numpy.matrix method), 137
getI() (numpy.matrix method), 138
getT() (numpy.matrix method), 138
gradient() (in module numpy), 901
greater() (in module numpy), 751
greater() (in module numpy.core.defchararray), 543
greater_equal() (in module numpy), 752
greater_equal() (in module numpy.core.defchararray), 543
green_text() (in module numpy.distutils.misc_util), 1306
gumbel() (in module numpy.random), 1148
gumbel() (numpy.random.RandomState method), 1192
H
H (numpy.matrix attribute), 120, 121
hamming() (in module numpy), 1298
hanning() (in module numpy), 1300
harden_mask (in module numpy.ma), 349, 805
harden_mask() (numpy.ma.MaskedArray method), 295, 350, 806
hardmask (numpy.ma.MaskedArray attribute), 252
has_cxx_sources() (in module numpy.distutils.misc_util), 1306
has_f_sources() (in module numpy.distutils.misc_util), 1306
has_samecoef() (numpy.polynomial.chebyshev.Chebyshev method), 986
has_samecoef() (numpy.polynomial.hermite.Hermite method), 1068
has_samecoef() (numpy.polynomial.hermite_e.HermiteE method), 1096
has_samecoef() (numpy.polynomial.laguerre.Laguerre method), 1041
has_samecoef() (numpy.polynomial.legendre.Legendre method), 1014
has_samecoef() (numpy.polynomial.polynomial.Polynomial method), 961
has_samedomain() (numpy.polynomial.chebyshev.Chebyshev method), 987
has_samedomain() (numpy.polynomial.hermite.Hermite method), 1068
has_samedomain() (numpy.polynomial.hermite_e.HermiteE method), 1096
has_samedomain() (numpy.polynomial.laguerre.Laguerre method), 1041
has_samedomain() (numpy.polynomial.legendre.Legendre method), 1014
has_samedomain() (numpy.polynomial.polynomial.Polynomial method), 961
has_samedomain() (numpy.polynomial.chebyshev.Chebyshev method), 987
has_samedomain() (numpy.polynomial.hermite.Hermite method), 1069
has_samedomain() (numpy.polynomial.hermite_e.HermiteE method), 1096
getslice
ndarray special methods, 103
hermsub() (in module numpy.polynomial.hermite), 1084
hermtrim() (in module numpy.polynomial.hermite), 1088
hermval() (in module numpy.polynomial.hermite), 1072
hermval2d() (in module numpy.polynomial.hermite), 1073
hermval3d() (in module numpy.polynomial.hermite), 1073
hermvander() (in module numpy.polynomial.hermite), 1079
hermvander2d() (in module numpy.polynomial.hermite), 1080
hermvander3d() (in module numpy.polynomial.hermite), 1080
hermweight() (in module numpy.polynomial.hermite), 1087
hermz (in module numpy.polynomial.hermite), 1088
ermz (module numpy.polynomial.hermite), 1088
hfft() (in module numpy.fft), 625
histogram() (in module numpy), 1277
histogram2d() (in module numpy), 1278
histogramdd() (in module numpy), 1280
holidays (numpy.busdaycalendar attribute), 578
hsplit (in module numpy.ma), 331, 787
hsplit() (in module numpy), 509
hstack (in module numpy.ma), 330, 336, 786, 792
hstack() (in module numpy), 506
hypergeometric() (in module numpy.random), 1150
hypergeometric() (numpy.random.RandomState method), 1194
hypot (in module numpy), 881
I (numpy.matrix attribute), 120, 121
i0() (in module numpy), 911
identity (in module numpy.ma), 400, 857
identity (numpy.ufunc attribute), 440
identity() (in module numpy), 454
identity() (in module numpy.ma, 942
identity() (in module numpy.polynomial.chebyshev.Chebyshev static method), 987
identity() (numpy.polynomial.hermite.Hermite static method), 1069
identity() (numpy.polynomial.hermite_e.HermiteE static method), 1097
identity() (numpy.polynomial.laguerre.Laguerre static method), 1042
identity() (numpy.polynomial.legendre.Legendre static method), 1015
identity() (numpy.polynomial.polynomial.Polynomial static method), 962
ids() (numpy.ma.MaskedArray method), 292
ifft() (in module numpy.fft), 610
ifft2() (in module numpy.fft), 613
ifftn() (in module numpy.fft), 616
iifftshift() (in module numpy.fft), 628
ihfft() (in module numpy.fft), 625
iinfo (class in numpy), 593
imag (numpy.chararray attribute), 166
imag (numpy.core.defchararray.chararray attribute), 555
imag (numpy.generic attribute), 75, 77
imag (numpy.ma.MaskedArray attribute), 258
imag (numpy.matrix attribute), 125
imag (numpy.ndarray attribute), 9, 41
imag (numpy.recarray attribute), 194
imag (numpy.record attribute), 217
imag() (in module numpy), 926
import_array (C function), 1373
import_ufunc (C function), 1401
in1d() (in module numpy), 1236
index (numpy.broadcast attribute), 231, 495
index() (in module numpy.core.defchararray), 545
index() (numpy.chararray method), 173
index() (numpy.core.defchararray.chararray method), 562
indexing, 102, 107, 1413
indices() (in module numpy), 653
indices() (in module numpy.ma), 417, 873
info() (in module numpy), 646
inner() (in module numpy), 706
inner() (in module numpy.ma), 401, 857
innerproduct() (in module numpy.ma), 402, 858
insert() (in module numpy), 514
integ() (numpy.polynomial.chebyshev.Chebyshev method), 988
integ() (numpy.polynomial.hermite.Hermite method), 1069
integ() (numpy.polynomial.hermite_e.HermiteE method), 1097
integ() (numpy.polynomial.laguerre.Laguerre method), 1042
integ() (numpy.polynomial.legendre.Legendre method), 1015
integ() (numpy.polynomial.polynomial.Polynomial method), 962
interface
array, 418
interp() (in module numpy), 938
intersect1d() (in module numpy), 1237
inv() (in module numpy.linalg), 733
invert (in module numpy), 526
ipmt() (in module numpy), 635
irfft() (in module numpy.fft), 620
irfft2() (in module numpy.fft), 621
irfftn() (in module numpy.fft), 623
irr() (in module numpy), 637
is_busday() (in module numpy), 578
is_local_src_dir() (in module numpy.distutils.mise_util), 1306
isalignedstruct (numpy.dtype attribute), 91, 589
isalnum() (numpy.chararray method), 174
isalnum() (numpy.core.defchararray.chararray method), 563
isalpha() (in module numpy.core.defchararray), 546
isalpha() (numpy.chararray method), 174
isalpha() (numpy.core.defchararray.chararray method), 563
isbuiltin (numpy.dtype attribute), 91, 100, 589
isclose() (in module numpy), 749
iscosmetic() (in module numpy), 743
iscosmeticobj() (in module numpy), 743
iscontiguous() (numpy.ma.MaskedArray method), 293
isdecimal() (module numpy.core.defchararray), 546
isdecimal() (numpy.chararray method), 174
isdecimal() (numpy.core.defchararray.chararray method), 563
isdigit() (in module numpy.core.defchararray), 546
isdigit() (numpy.chararray method), 174
isdigit() (numpy.core.defchararray.chararray method), 563
isfinite (in module numpy), 739
isfortran() (in module numpy), 744
isinf (in module numpy), 740
islower() (in module numpy.core.defchararray), 546
islower() (numpy.chararray method), 174
islower() (numpy.core.defchararray.chararray method), 563
isnan (in module numpy), 741
isnative (in module numpy), 741
isnumeric (numpy.dtype attribute), 91, 101, 590
isnumeric() (in module numpy.core.defchararray), 547
isnumeric() (numpy.chararray method), 174
isnumeric() (numpy.core.defchararray.chararray method), 563
isposinf() (in module numpy), 742
isreal() (in module numpy), 745
isrealobj() (in module numpy), 745
isscalar() (in module numpy), 746
issctype() (in module numpy), 595
isspace() (in module numpy.core.defchararray), 547
isspace() (numpy.chararray method), 174
isspace() (numpy.core.defchararray.chararray method), 563
issubclass_() (in module numpy), 597
issubdtype() (in module numpy), 596
issubtype() (in module numpy), 596
istitle() (in module numpy.core.defchararray), 547
istitle() (numpy.chararray method), 174
istitle() (numpy.core.defchararray.chararray method), 563
isupper() (in module numpy.core.defchararray.chararray method), 564
isupper() (numpy.chararray method), 175
isupper() (numpy.core.defchararray.chararray method), 564
isupper() (numpy.generic method), 81
isupper() (numpy.ma.MaskedArray method), 272
isupper() (numpy.matrix method), 139
isupper() (numpy.ndarray method), 21, 45
isupper() (numpy.recarray method), 205
isupper() (numpy.record method), 222
itemset() (numpy.generic method), 82
itemset() (numpy.matrix method), 140
itemset() (numpy.ndarray method), 205
itemset() (numpy.recarray method), 222
itemsize (numpy.chararray attribute), 166
itemsize (numpy.core.defchararray.chararray attribute), 555
itemsize (numpy.dtype attribute), 98
itemsize (numpy.generic attribute), 75, 77
itemsize (numpy.ma.MaskedArray attribute), 256
itemsize (numpy.matrix attribute), 125
itemsize (numpy.ndarray attribute), 9, 39
itemsize (numpy.recarray attribute), 194
itemsize (numpy.record attribute), 217
iterator
C-API, 1379, 1395
iternext() (numpy.nditer method), 677
iters (numpy.broadcast attribute), 231, 495
ix_() (in module numpy), 654

J
join() (in module numpy.core.defchararray), 534
join() (numpy.chararray method), 175
join() (numpy.core.defchararray.chararray method), 564

K
kaiser() (in module numpy), 1301
keyword arguments
ufunc, 437
kind (numpy.dtype attribute), 98
knownfailureif() (in module numpy.testing.decorators), 1293
kron() (in module numpy), 714

L
lag2poly() (module numpy.polynomial.laguerre), 1062
lagadd() (module numpy.polynomial.laguerre), 1056
lagcompanion() (module numpy.polynomial.laguerre), 1060
lagder() (module numpy.polynomial.laguerre), 1054
lagdiv() (module numpy.polynomial.laguerre), 1058
lagdomain (module numpy.polynomial.laguerre), 1061
lagfit() (module numpy.polynomial.laguerre), 1049
<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lagfromroots()</td>
<td>numpy.polynomial.laguerre</td>
<td>1049</td>
</tr>
<tr>
<td>laggauss()</td>
<td>numpy.polynomial.laguerre</td>
<td>1059</td>
</tr>
<tr>
<td>laggrid2d()</td>
<td>numpy.polynomial.laguerre</td>
<td>1047</td>
</tr>
<tr>
<td>laggrid3d()</td>
<td>numpy.polynomial.laguerre</td>
<td>1047</td>
</tr>
<tr>
<td>lagint()</td>
<td>numpy.polynomial.laguerre</td>
<td>1055</td>
</tr>
<tr>
<td>lagmul()</td>
<td>numpy.polynomial.laguerre</td>
<td>1061</td>
</tr>
<tr>
<td>lagmulx()</td>
<td>numpy.polynomial.laguerre</td>
<td>1058</td>
</tr>
<tr>
<td>lagone()</td>
<td>numpy.polynomial.laguerre</td>
<td>1061</td>
</tr>
<tr>
<td>lagpow()</td>
<td>numpy.polynomial.laguerre</td>
<td>1059</td>
</tr>
<tr>
<td>lagroots()</td>
<td>numpy.polynomial.laguerre</td>
<td>1057</td>
</tr>
<tr>
<td>lagtrim()</td>
<td>numpy.polynomial.laguerre</td>
<td>1061</td>
</tr>
<tr>
<td>lagval()</td>
<td>numpy.polynomial.laguerre</td>
<td>1052</td>
</tr>
<tr>
<td>lagvander2d()</td>
<td>numpy.polynomial.laguerre</td>
<td>1052</td>
</tr>
<tr>
<td>lagvander3d()</td>
<td>numpy.polynomial.laguerre</td>
<td>1053</td>
</tr>
<tr>
<td>lagx()</td>
<td>numpy.polynomial.laguerre</td>
<td>1060</td>
</tr>
<tr>
<td>lagzero()</td>
<td>numpy.polynomial.laguerre</td>
<td>1061</td>
</tr>
<tr>
<td>laplace()</td>
<td>numpy.random</td>
<td>1151</td>
</tr>
<tr>
<td>ldelta()</td>
<td>numpy.random.RandomState</td>
<td>1196</td>
</tr>
<tr>
<td>left_shift()</td>
<td>numpy.random.RandomState</td>
<td>915</td>
</tr>
<tr>
<td>leg2poly()</td>
<td>numpy.polynomial.legendre</td>
<td>1034</td>
</tr>
<tr>
<td>legadd()</td>
<td>numpy.polynomial.legendre</td>
<td>1028</td>
</tr>
<tr>
<td>legcompanion()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>legder()</td>
<td>numpy.polynomial.legendre</td>
<td>1026</td>
</tr>
<tr>
<td>legdiv()</td>
<td>numpy.polynomial.legendre</td>
<td>1031</td>
</tr>
<tr>
<td>legend()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>legfit()</td>
<td>numpy.polynomial.legendre</td>
<td>1008</td>
</tr>
<tr>
<td>legfromroots()</td>
<td>numpy.polynomial.legendre</td>
<td>1022</td>
</tr>
<tr>
<td>leggauss()</td>
<td>numpy.polynomial.legendre</td>
<td>1032</td>
</tr>
<tr>
<td>leggrid2d()</td>
<td>numpy.polynomial.legendre</td>
<td>1019</td>
</tr>
<tr>
<td>leggrid3d()</td>
<td>numpy.polynomial.legendre</td>
<td>1020</td>
</tr>
<tr>
<td>legint()</td>
<td>numpy.polynomial.legendre</td>
<td>1027</td>
</tr>
<tr>
<td>legline()</td>
<td>numpy.polynomial.legendre</td>
<td>1034</td>
</tr>
<tr>
<td>legmul()</td>
<td>numpy.polynomial.legendre</td>
<td>1030</td>
</tr>
<tr>
<td>legmulx()</td>
<td>numpy.polynomial.legendre</td>
<td>1030</td>
</tr>
<tr>
<td>legone()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>legpow()</td>
<td>numpy.polynomial.legendre</td>
<td>1031</td>
</tr>
<tr>
<td>legroots()</td>
<td>numpy.polynomial.legendre</td>
<td>1021</td>
</tr>
<tr>
<td>legsub()</td>
<td>numpy.polynomial.legendre</td>
<td>1029</td>
</tr>
<tr>
<td>legtrim()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>legval()</td>
<td>numpy.polynomial.legendre</td>
<td>1017</td>
</tr>
<tr>
<td>legval2d()</td>
<td>numpy.polynomial.legendre</td>
<td>1018</td>
</tr>
<tr>
<td>legval3d()</td>
<td>numpy.polynomial.legendre</td>
<td>1019</td>
</tr>
<tr>
<td>legvander()</td>
<td>numpy.polynomial.legendre</td>
<td>1024</td>
</tr>
<tr>
<td>legvander2d()</td>
<td>numpy.polynomial.legendre</td>
<td>1025</td>
</tr>
<tr>
<td>legvander3d()</td>
<td>numpy.polynomial.legendre</td>
<td>1025</td>
</tr>
<tr>
<td>legweight()</td>
<td>numpy.polynomial.legendre</td>
<td>1032</td>
</tr>
<tr>
<td>legx()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>legzero()</td>
<td>numpy.polynomial.legendre</td>
<td>1033</td>
</tr>
<tr>
<td>less()</td>
<td>numpy.core.defchararray</td>
<td>752</td>
</tr>
<tr>
<td>less_equal()</td>
<td>numpy.core.defchararray</td>
<td>544</td>
</tr>
<tr>
<td>less_equal()</td>
<td>numpy.core.defchararray</td>
<td>752</td>
</tr>
<tr>
<td>lexsort()</td>
<td>numpy.core.defchararray</td>
<td>543</td>
</tr>
<tr>
<td>linexp()</td>
<td>numpy.random</td>
<td>473</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.chebyshev.Chebyshev</td>
<td>988</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.hermite.Hermite</td>
<td>1070</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.hermite_e.HermiteE</td>
<td>1097</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.laguerre.Laguerre</td>
<td>1042</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.legendre.Legendre</td>
<td>1015</td>
</tr>
<tr>
<td>linspace()</td>
<td>numpy.polynomial.legendre.Polynomial</td>
<td>963</td>
</tr>
<tr>
<td>ljust()</td>
<td>numpy.core.defchararray</td>
<td>534</td>
</tr>
<tr>
<td>ljust()</td>
<td>numpy.chararray</td>
<td>175</td>
</tr>
<tr>
<td>ljust()</td>
<td>numpy.core.defchararray.chararray</td>
<td>564</td>
</tr>
<tr>
<td>load()</td>
<td>numpy.random</td>
<td>680</td>
</tr>
<tr>
<td>load()</td>
<td>numpy.random.ma</td>
<td>366, 822</td>
</tr>
<tr>
<td>load_library()</td>
<td>numpy.ctypeslib</td>
<td>576</td>
</tr>
<tr>
<td>loads()</td>
<td>numpy.random.ma</td>
<td>366, 823</td>
</tr>
<tr>
<td>loadtxt()</td>
<td>numpy.random.ma</td>
<td>467, 884</td>
</tr>
<tr>
<td>log()</td>
<td>numpy.random</td>
<td>907</td>
</tr>
<tr>
<td>log10()</td>
<td>numpy.random</td>
<td>908</td>
</tr>
<tr>
<td>log1p()</td>
<td>numpy.random</td>
<td>909</td>
</tr>
<tr>
<td>log2()</td>
<td>numpy.random</td>
<td>908</td>
</tr>
<tr>
<td>logaddexp()</td>
<td>numpy.random</td>
<td>910</td>
</tr>
<tr>
<td>logaddexp2()</td>
<td>numpy.random</td>
<td>910</td>
</tr>
<tr>
<td>logical_and()</td>
<td>numpy.random</td>
<td>746</td>
</tr>
<tr>
<td>logical_not()</td>
<td>numpy.random</td>
<td>747</td>
</tr>
<tr>
<td>logical_or()</td>
<td>numpy.random</td>
<td>747</td>
</tr>
</tbody>
</table>
logical_xor (in module numpy), 747
logistic() (in module numpy.random), 1152
logistic() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
logseries() (in module numpy.random), 1155
logseries() (numpy.random.RandomState method), 1197
lognormal() (in module numpy.random), 1153
lognormal() (numpy.random.RandomState method), 1198
M
MachAr (class in numpy), 594
make_config_py() (numpy.distutils.misc_util.Configuration
method), 1314
make_mask() (in module numpy.ma), 338, 794
make_mask_descr() (in module numpy.ma), 340, 797
make_mask_none() (in module numpy.ma), 339, 795
make_svn_version_py() (numpy.distutils.misc_util.Configuration
method), 1314
mapparms() (numpy.polynomial.chebyshev.Chebyshev
method), 989
mapparms() (numpy.polynomial.hermite.Hermite
method), 1070
mapparms() (numpy.polynomial.hermite_e.HermiteE
method), 1098
mapparms() (numpy.polynomial.laguerre.Laguerre
method), 1043
mapparms() (numpy.polynomial.legendre.Legendre
method), 1016
mapparms() (numpy.polynomial.polynomial.Polynomial
method), 963
mask (numpy.ma.masked_array attribute), 343, 799
mask (numpy.ma.MaskedArrayAttribute), 251, 315, 771
mask_cols() (in module numpy.ma), 346, 802
mask_indices() (in module numpy), 657
mask_or() (in module numpy.ma), 340, 347, 796, 803
mask_rowcols() (in module numpy.ma), 347, 803
mask_rows() (in module numpy.ma), 348, 805
masked (in module numpy.ma), 239
masked_arrays, 233
masked_all() (in module numpy.ma), 303, 759
masked_all_like() (in module numpy.ma), 304, 760
masked_array (in module numpy.ma), 235, 298, 754
masked_equal() (in module numpy.ma), 238, 353, 809
masked_greater() (in module numpy.ma), 238, 353, 810
masked_greater_equal() (in module numpy.ma), 238, 354, 810
masked_inside() (in module numpy.ma), 239, 354, 810
masked_invalid() (in module numpy.ma), 239, 355, 811
masked_less() (in module numpy.ma), 240, 355, 812
masked_not_equal() (in module numpy.ma), 240, 356, 812
masked_object() (in module numpy.ma), 241, 356, 812
masked_outside() (in module numpy.ma), 242, 357, 813
masked_values() (in module numpy.ma), 242, 358, 814
masked_where() (in module numpy.ma), 243, 359, 815
MaskedArray (class in numpy.ma), 251
MaskType (in module numpy.ma), 298, 356, 812
mat() (in module numpy), 483
matrix, 119
matrix (class in numpy), 120
matrix_power() (in module numpy.linalg), 713
max (numpy.iinfo attribute), 594
max() (in module numpy.ma), 391, 848
max() (numpy.generic method), 82
max() (numpy.ma.MaskedArray method), 281, 394, 850
max() (numpy.matrix method), 140
max() (numpy.ndarray method), 22
max() (numpy.ndarray method), 22
maximum (in module numpy), 933
maximum_fill_value() (in module numpy.ma), 368, 369, 824, 825
mean (in module numpy.ma), 377, 833
mean() (in module numpy), 1265
mean() (numpy.generic method), 82
mean() (numpy.ma.MaskedArray method), 281, 385, 841
mean() (numpy.matrix method), 141
mean() (numpy.ndarray method), 22, 64
mean() (numpy.ndarray method), 22, 64
mean() (numpy.recarray method), 222
median() (in module numpy), 1262
median() (in module numpy.ma), 378, 834
memmap (class in numpy), 156, 694
memory maps, 156
memory model
ndarray, 1411
meshgrid() (in module numpy), 476
metadata (numpy.dtype attribute), 91, 590
methods
accumulate, ufunc, 1417
reduce, ufunc, 1417
reduceat, ufunc, 1417
ufunc, 440
mgrid (in module numpy), 477
min (numpy.iinfo attribute), 594
mint() (in module numpy.ma), 392, 848
mint() (numpy.generic method), 82

Index 1463
min() (numpy.ma.MaskedArray method), 282, 394, 850
min() (numpy.matrix method), 141
min() (numpy.ndarray method), 22, 62
min() (numpy.recarray method), 206
min() (numpy.record method), 222
min_scalar_type() (in module numpy), 584
minimum (in module numpy), 934
mintypecode() (in module numpy), 599
mirr() (in module numpy.ma), 332, 788
mod (in module numpy), 923
mod() (in module numpy.core.defchararray), 532
modf (in module numpy), 924
mr_ (in module numpy.ma), 332, 788
msort() (in module numpy), 1244
multinomial() (in module numpy.random), 1156
multinomial() (numpy.random.RandomState method), 1201
multiply (in module numpy), 918
multiply() (in module numpy.core.defchararray), 532
multivariate_normal() (in module numpy.random), 1157
multivariate_normal() (numpy.random.RandomState method), 1201

N
name (numpy.dtype attribute), 91, 98, 590
names (numpy.dtype attribute), 92, 99, 590
nan_to_num() (in module numpy), 936
nanargmax() (in module numpy), 1248
nanargmin() (in module numpy), 1249
nanmean() (in module numpy), 1259
nanmax() (in module numpy), 1269
nanstd() (in module numpy), 1258
nansum() (in module numpy), 1270
nanvar() (in module numpy), 1271
nargs (numpy.ufunc attribute), 438
nbytes (numpy.chararray attribute), 166
nbytes (numpy.core.defchararray.chararray attribute), 555
nbytes (numpy.generic attribute), 77
nbytes (numpy.ma.MaskedArray attribute), 256
nbytes (numpy.matrix attribute), 126
nbytes (numpy.ndarray attribute), 10, 39
nbytes (numpy.recarray attribute), 195
nbytes (numpy.record attribute), 217
ndarray, 107
C-API, 1340, 1379
memory model, 1411
special methods getslice, 103
special methods setslice, 103
view, 104
ndarray (class in numpy), 4
ndenumerate (class in numpy), 229, 677
ndim (numpy.chararray attribute), 166
ndim (numpy.core.defchararray.chararray attribute), 555
ndim (numpy.generic attribute), 75, 77
ndim (numpy.ma.MaskedArray attribute), 256
ndim (numpy.matrix attribute), 126
ndim (numpy.ndarray attribute), 10, 38
ndim (numpy.recarray attribute), 195
ndim (numpy.record attribute), 218
ndincr() (numpy.ndindex method), 678
ndindex (class in numpy), 678
nditer (class in numpy), 672
ndpointer() (in module numpy.ctypeslib), 576
negative (in module numpy), 918
negative_binomial() (in module numpy.random), 1158
negative_binomial() (numpy.random.RandomState method), 1203
newaxis, 103
newaxis (in module numpy), 104
newbuffer() (in module numpy), 945
newbyteorder() (numpy.dtype method), 92, 101, 590
newbyteorder() (numpy.generic method), 82
newbyteorder() (numpy.matrix method), 142
newbyteorder() (numpy.ndarray method), 23
newbyteorder() (numpy.recarray method), 206
newbyteorder() (numpy.record method), 223
next (numpy.broadcast attribute), 232, 496
next (numpy.flatiter attribute), 680
next (numpy.nditer attribute), 677
next() (numpy.ndenumerate method), 230, 678
next() (numpy.ndindex method), 678
nin (numpy.ufunc attribute), 438
NO_IMPORT_ARRAY (C macro), 1373
NO_IMPORT_UFUNC (C variable), 1401
nomask (in module numpy.ma), 251
non-contiguous, 35
noncentral_chisquare() (in module numpy.random), 1159
noncentral_chisquare() (numpy.random.RandomState method), 1203
noncentral_f() (in module numpy.random), 1161
noncentral_f() (numpy.random.RandomState method), 1206
nonzero (in module numpy.ma), 312, 768
nonzero() (in module numpy), 650, 1250
nonzero() (numpy.chararray method), 176
nonzero() (numpy.core.defchararray.chararray method), 565
nonzero() (numpy.generic method), 83
nonzero() (numpy.ma.MaskedArray method), 272, 316, 772
nonzero() (numpy.matrix method), 142
nonzero() (numpy.ndarray method), 23, 61
nonzero() (numpy.recarray method), 207
nonzero() (numpy.record method), 223
norm() (in module numpy.linalg), 724
normal() (in module numpy.random), 1162
normal() (numpy.random.RandomState method), 1207

1464 Index
not_equal (in module numpy), 753
not_equal() (in module numpy.core.defchararray), 542
notmasked_contiguous() (in module numpy.ma), 344, 801
notmasked_edges() (in module numpy.ma), 345, 801
nout (numpy.ufunc attribute), 438
nper() (in module numpy), 638
npv() (in module numpy), 633
NPY_1_PI (C variable), 1405
NPY_2_PI (C variable), 1405
NPY_ALLOW_C_API (C macro), 1376
NPY_ALLOW_C_API_DEF (C macro), 1376
NPY_ANYORDER (C variable), 1378
NPY_ARRAY_ALIGNED (C variable), 1344, 1355
NPY_ARRAY_BEHAVED (C variable), 1345, 1356
NPY_ARRAY_BEHAVED_NS (C variable), 1346, 1357
NPY_ARRAY_C_CONTIGUOUS (C variable), 1344,
1355
NPY_ARRAY_CARRAY (C variable), 1345, 1356
NPY_ARRAY_CARRAY_RO (C variable), 1345, 1356
NPY_ARRAY_ELEMENTSSTRIDES (C variable), 1347
NPY_ARRAY_ENSUREARRAY (C variable), 1344,
1356
NPY_ARRAY_ENSURECOPY (C variable), 1344, 1356
NPY_ARRAY_F_CONTIGUOUS (C variable), 1344,
1355
NPY_ARRAY_FARRAY (C variable), 1345, 1356
NPY_ARRAY_FARRAY_RO (C variable), 1345, 1356
NPY_ARRAY_DEFAULT (C variable), 1345, 1356
NPY_ARRAY_FORCECAST (C variable), 1345, 1356
NPY_ARRAY_IN_ARRAY (C variable), 1345
NPY_ARRAY_IN_FARRAY (C variable), 1345
NPY_ARRAY_INOUT_ARRAY (C variable), 1345
NPY_ARRAY_INOUT_FARRAY (C variable), 1345
NPY_ARRAY_NOTSWAPPED (C variable), 1346, 1356
NPY_ARRAY_OWNDATA (C variable), 1355
NPY_ARRAY_UPDATE_ALL (C variable), 1356
NPY_ARRAY_UPDATEIFCOPY (C variable), 1345,
1356
NPY_ARRAY_WRITEABLE (C variable), 1344, 1355
NPY_AUXDATA_CLONE (C function), 1366
NPY_AUXDATA_FREE (C function), 1366
NPY_BEGIN_ALLOW_THREADS (C macro), 1375
NPY_BEGIN_THREADS (C macro), 1375
NPY_BEGIN_THREADS_DEF (C function), 1376
NPY_BIG_ENDIAN (C variable), 1334
npy_bool (C type), 1339
NPY_BOOL (C variable), 1335
NPY_BUFSIZE (C variable), 1335
NPY_BYTE (C variable), 1335
NPY_BYTE_ORDER (C variable), 1335
NPY_CASTING (C type), 1378
NPY_CDOUBLE (C variable), 1337
NPY_CFLOAT (C variable), 1336
NPY_CLIP (C variable), 1360, 1378
NPY_CLIPMODE (C type), 1378
NPY_CLONEDOUBLE (C variable), 1337
NPY_COMPLEX128 (C variable), 1337
NPY_COMPLEX64 (C variable), 1336
npy_copysign (C function), 1404
NPY_CORDER (C variable), 1378
NPY_CPU_AMD64 (C variable), 1334
NPY_CPU_IA64 (C variable), 1334
NPY_CPU_PARISC (C variable), 1334
NPY_CPU_PPC (C variable), 1334
NPY_CPU_PPC64 (C variable), 1334
NPY_CPU_S390 (C variable), 1334
NPY_CPU_SPARC (C variable), 1334
NPY_CPU_SPARC64 (C variable), 1334
NPY_CPU_X86 (C variable), 1334
NPY_DATETIME (C variable), 1337
NPY_DEFAULT_TYPE (C variable), 1337
NPY_DISABLE_C_API (C macro), 1376
NPY_DOUBLE (C variable), 1336
npy_double_to_half (C function), 1407
npy_doublebits_to_halfbits (C function), 1408
NPY_E (C variable), 1405
NPY_END_ALLOW_THREADS (C macro), 1375
NPY_END_THREADS (C macro), 1375
NPY_END_THREADS_DESCR (C function), 1376
NPY_EQUIV_CASTING (C variable), 1378
NPY_EULER (C variable), 1405
NPY_FALSE (C variable), 1377
NPYFLOAT (C variable), 1336
NPY_FLOAT (C variable), 1336
npy_float_to_half (C function), 1407
npy_floatbits_to_halfbits (C function), 1408
NPY_FLOAT16 (C variable), 1336
NPY_FLOAT32 (C variable), 1336
NPY_FLOAT64 (C variable), 1336
NPY_FROM_FIELDS (C variable), 1322
NPY_HALF (C variable), 1336
np_half_copysign (C function), 1408
np_half_eq (C function), 1407
np_half_eq_nonan (C function), 1407
np_half_ge (C function), 1407
np_half_gt (C function), 1407
np_half_isfinite (C function), 1408
np_half_isinf (C function), 1408
np_half_isnan (C function), 1408
np_half_iszero (C function), 1407
np_half_le (C function), 1407
np_half_le_nonan (C function), 1407
np_half_lt (C function), 1407
np_half_lt_nonan (C function), 1407
np_half_isfinite (C function), 1408
np_half_isinf (C function), 1408
np_half_isnan (C function), 1408
np_half_iszero (C function), 1407
np_half_le (C function), 1407
np_half_le_nonan (C function), 1407
np_half_gt (C function), 1407
np_half_l (C function), 1407
np_half_l_nonan (C function), 1407
NPY_HALF_NAN (C variable), 1407
Index 1465
Index
Index 1467
paretop() (in module numpy.random), 1163
paretop() (numpy.random.RandomState method), 1208
partition() (in module numpy), 1245
partition() (in module numpy.core.defchararray), 536
partition() (numpy.matrix method), 142
partition() (numpy.ndarray method), 23, 59
partition() (numpy.recarray method), 207
paths() (numpy.distutils.misc_util.Configuration method), 1313
percentile() (in module numpy), 1261
permutation() (in module numpy.random), 1139
permutation() (numpy.random.RandomState method), 1209
piecewise() (in module numpy), 644
pinv() (in module numpy.linalg), 734
place() (in module numpy), 669
pmt() (in module numpy), 634
poisson() (in module numpy.random), 1164
poisson() (numpy.random.RandomState method), 1210
poly() (in module numpy), 1121
poly1d (class in numpy), 1118
poly2cheb() (in module numpy.polynomial.chebyshev), 1008
poly2herm() (in module numpy.polynomial.hermite), 1090
poly2herme() (in module numpy.polynomial.hermite_e), 1117
poly2lag() (in module numpy.polynomial.laguerre), 1062
poly2leg() (in module numpy.polynomial.legendre), 1035
polyadd() (in moduleumpy), 1128
polyadd() (in module numpy.polynomial.polynomial), 977
polycompanion() (in module numpy.polynomial.polynomial), 980
polynherder() (in module numpy.polynomial.polynomial), 1113
polyder() (in module numpy), 1126
polyder() (in module numpy.polynomial.polynomial), 975
polydiv() (in module numpy), 1129
polydiv() (in module numpy.polynomial.polynomial), 978
polydomain (in module numpy.polynomial.polynomial), 980
polyfit() (in module numpy), 1123
polyfit() (in module numpy.ma), 408, 864
polyfit() (in module numpy.polynomial.polynomial), 970
polyfromroots() (in module numpy.polynomial.polynomial), 969
polygrid2d() (in module numpy.polynomial.polynomial), 967
polygrid3d() (in module numpy.polynomial.polynomial), 968
polyint() (in module numpy), 1127
polyint() (in module numpy.polynomial.polynomial), 975
polyline() (in module numpy.polynomial.polynomial), 981
polymul() (in module numpy.polynomial.polynomial), 1130
polymul() (in module numpy.polynomial.polynomial), 978
polymulx() (in module numpy.polynomial.polynomial), 978
Polynomial (class in numpy.polynomial.polynomial), 956
polyone (in module numpy.polynomial.polynomial), 980
polypow() (in module numpy.polynomial.polynomial), 979
polyroots() (in module numpy.polynomial.polynomial), 969
polysub() (in module numpy), 1131
polysub() (in module numpy.polynomial.polynomial), 977
polytrim() (in module numpy.polynomial.polynomial), 980
polyval() (in module numpy), 1120
polyval2d() (in module numpy.polynomial.polynomial), 965
polyval3d() (in module numpy.polynomial.polynomial), 966
polyval3d() (in module numpy.polynomial.polynomial), 967
polyvander() (in module numpy.polynomial.polynomial), 973
polyvander2d() (in module numpy.polynomial.polynomial), 973
polyvander2d() (in module numpy.polynomial.polynomial), 973
polyvander3d() (in module numpy.polynomial.polynomial), 974
polynx (in module numpy.polynomial.polynomial), 980
polyzero (in module numpy.polynomial.polynomial), 980
power (in module numpy), 920
power() (in module numpy.ma), 379, 835
power() (in module numpy.random), 1166
power() (numpy.random.RandomState method), 1211
ppmt() (in module numpy), 635
pprint() (numpy.record method), 223
prod (in module numpy.ma), 379, 835
prod() (in module numpy), 894
prod() (numpy.generic method), 83
prod() (numpy.ma.MaskedArray method), 282, 386, 842
prod() (numpy.matrix method), 143
prod() (numpy.ndarray method), 24, 64
prod() (numpy.recarray method), 208
prod() (numpy.record method), 223
product() (numpy.ma.MaskedArray method), 283
promote_types() (in module numpy), 584
protocol
array, 418
ptp() (in module numpy), 1260
ppp() (in module numpy.ma), 392, 848
ptt() (numpy.generic method), 83
ptt() (numpy.ma.MaskedArray method), 284, 395, 851
ptt() (numpy.matrix method), 144
ptt() (numpy.ndarray method), 24, 63
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyArray_ISUNSIGNED</td>
<td>(C function), 1349</td>
</tr>
<tr>
<td>PyArray_ISUSERDEF</td>
<td>(C function), 1351</td>
</tr>
<tr>
<td>PyArray_ISWRITEABLE</td>
<td>(C function), 1357</td>
</tr>
<tr>
<td>PyArray_IsZeroDim</td>
<td>(C function), 1349</td>
</tr>
<tr>
<td>PyArray_Item_INCREF</td>
<td>(C function), 1354</td>
</tr>
<tr>
<td>PyArray_Item_XDECREF</td>
<td>(C function), 1354</td>
</tr>
<tr>
<td>PyArray_ITEMSIZE</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray_ITER_DATA</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray_ITER_GOTO</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray_ITER_GOTO1D</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray_ITER_NEXT</td>
<td>(C function), 1366</td>
</tr>
<tr>
<td>PyArray_ITER_NOTDONE</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray_ITER_RESET</td>
<td>(C function), 1366</td>
</tr>
<tr>
<td>PyArray_ITER_AllButAxis</td>
<td>(C function), 1366</td>
</tr>
<tr>
<td>PyArray.IterNew</td>
<td>(C function), 1366</td>
</tr>
<tr>
<td>PyArray.LexSort</td>
<td>(C function), 1360</td>
</tr>
<tr>
<td>PyArray_malloc</td>
<td>(C function), 1375</td>
</tr>
<tr>
<td>PyArray_MatrixProduct</td>
<td>(C function), 1363</td>
</tr>
<tr>
<td>PyArray_MatrixProduct2</td>
<td>(C function), 1363</td>
</tr>
<tr>
<td>PyArray_MAX</td>
<td>(C function), 1377</td>
</tr>
<tr>
<td>PyArray.Min</td>
<td>(C function), 1361</td>
</tr>
<tr>
<td>PyArray.MinScalarType</td>
<td>(C function), 1352</td>
</tr>
<tr>
<td>PyArray.MoveInto</td>
<td>(C function), 1348</td>
</tr>
<tr>
<td>PyArray.MultiIter_DATA</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_GOTO</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_GOTO1D</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_NEXT</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_NEXTi</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_NOTDONE</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIter_RESET</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiIterNew</td>
<td>(C function), 1367</td>
</tr>
<tr>
<td>PyArray.MultiplyIntList</td>
<td>(C function), 1365</td>
</tr>
<tr>
<td>PyArray.MultiplyList</td>
<td>(C function), 1364</td>
</tr>
<tr>
<td>PyArray.NBYTES</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray.NDIM</td>
<td>(C function), 1340</td>
</tr>
<tr>
<td>PyArray.NeighborhoodIterNew</td>
<td>(C function), 1368</td>
</tr>
<tr>
<td>PyArray.New</td>
<td>(C function), 1343</td>
</tr>
<tr>
<td>PyArray.NewCopy</td>
<td>(C function), 1358</td>
</tr>
<tr>
<td>PyArray.NewFromDescr</td>
<td>(C function), 1342</td>
</tr>
<tr>
<td>PyArray.NewLikeArray</td>
<td>(C function), 1342</td>
</tr>
<tr>
<td>PyArray.Newshape</td>
<td>(C function), 1359</td>
</tr>
<tr>
<td>PyArray.Nonzero</td>
<td>(C function), 1361</td>
</tr>
<tr>
<td>PyArray.ObjectType</td>
<td>(C function), 1353</td>
</tr>
<tr>
<td>PyArray.One</td>
<td>(C function), 1354</td>
</tr>
<tr>
<td>PyArray.OrderConverter</td>
<td>(C function), 1372</td>
</tr>
<tr>
<td>PyArray.OutputConverter</td>
<td>(C function), 1371</td>
</tr>
<tr>
<td>PyArray_Partition</td>
<td>(C function), 1361</td>
</tr>
<tr>
<td>PyArray.Prod</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray PromoteTypes</td>
<td>(C function), 1353</td>
</tr>
<tr>
<td>PyArray.Ptp</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray.PutMask</td>
<td>(C function), 1360</td>
</tr>
<tr>
<td>PyArray.PutTo</td>
<td>(C function), 1360</td>
</tr>
<tr>
<td>PyArray.PyIntAsInt</td>
<td>(C function), 1372</td>
</tr>
<tr>
<td>PyArray.PyIntAsIntp</td>
<td>(C function), 1372</td>
</tr>
<tr>
<td>PyArray.ResultType</td>
<td>(C function), 1377</td>
</tr>
<tr>
<td>PyArray.Return</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray.Round</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray.SAMESHAPE</td>
<td>(C function), 1377</td>
</tr>
<tr>
<td>PyArray.Scalar</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray.ScalarAsCtype</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray.ScalarKind</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray.SearchsideConverter</td>
<td>(C function), 1372</td>
</tr>
<tr>
<td>PyArray.SearchedSorted</td>
<td>(C function), 1361</td>
</tr>
<tr>
<td>PyArray.SetBaseObject</td>
<td>(C function), 1344</td>
</tr>
<tr>
<td>PyArray.SetField</td>
<td>(C function), 1358</td>
</tr>
<tr>
<td>PyArray_SETITEM</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray.SetNumericOps</td>
<td>(C function), 1374</td>
</tr>
<tr>
<td>PyArray.Sort</td>
<td>(C function), 1374</td>
</tr>
<tr>
<td>PyArray_SetStringFunction</td>
<td>(C function), 1374</td>
</tr>
<tr>
<td>PyArray_SHAPE</td>
<td>(C function), 1340</td>
</tr>
<tr>
<td>PyArray.SimpleNew</td>
<td>(C function), 1343</td>
</tr>
<tr>
<td>PyArray.SimpleNewFromData</td>
<td>(C function), 1343</td>
</tr>
<tr>
<td>PyArray.SimpleNewFromDescr</td>
<td>(C function), 1343</td>
</tr>
<tr>
<td>PyArray.SIZE</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray.Size</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray.Sort</td>
<td>(C function), 1360</td>
</tr>
<tr>
<td>PyArray.SortkindConverter</td>
<td>(C function), 1372</td>
</tr>
<tr>
<td>PyArray.Squeeze</td>
<td>(C function), 1359</td>
</tr>
<tr>
<td>PyArray.Std</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray_STRIDE</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray_Strides</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray.Sum</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray.SwapAxes</td>
<td>(C function), 1359</td>
</tr>
<tr>
<td>PyArray.TakeFrom</td>
<td>(C function), 1360</td>
</tr>
<tr>
<td>PyArray.getFile</td>
<td>(C function), 1358</td>
</tr>
<tr>
<td>PyArray.ToList</td>
<td>(C function), 1358</td>
</tr>
<tr>
<td>PyArray.ToScalar</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray.ToString</td>
<td>(C function), 1358</td>
</tr>
<tr>
<td>PyArray.Trace</td>
<td>(C function), 1362</td>
</tr>
<tr>
<td>PyArray.Transpose</td>
<td>(C function), 1359</td>
</tr>
<tr>
<td>PyArray_TYPE</td>
<td>(C function), 1341</td>
</tr>
<tr>
<td>PyArray_Type</td>
<td>(C variable), 1320</td>
</tr>
<tr>
<td>PyArray_TypeObjectFromType</td>
<td>(C function), 1369</td>
</tr>
<tr>
<td>PyArray_TypestrConvert</td>
<td>(C function), 1373</td>
</tr>
<tr>
<td>PyArray_UpdateFlags</td>
<td>(C function), 1357</td>
</tr>
<tr>
<td>PyArray_ValidType</td>
<td>(C function), 1354</td>
</tr>
</tbody>
</table>
Index
sqrt (in module numpy), 929
square (in module numpy), 930
squeeze (numpy.ma.MaskedArray attribute), 267, 328, 784
squeeze() (in module numpy), 497
squeeze() (in module numpy.ma), 327, 783
squeeze() (numpy.chararray method), 182
squeeze() (numpy.core.defchararray.chararray method), 571
squeeze() (numpy.generic method), 85, 86
squeeze() (numpy.matrix method), 149
squeeze() (numpy.ndarray method), 29, 57
squeeze() (numpy.recarray method), 213
squeeze() (numpy.record method), 225
standard_cauchy() (in module numpy.random), 1170
standard_cauchy() (numpy.random.RandomState method), 1220
standard_exponential() (in module numpy.random), 1170
standard_exponential() (numpy.random.RandomState method), 1221
standard_gamma() (in module numpy.random), 1171
standard_gamma() (numpy.random.RandomState method), 1221
standard_normal() (in module numpy.random), 1172
standard_normal() (numpy.random.RandomState method), 1222
standard_t() (in module numpy.random), 1172
standard_t() (numpy.random.RandomState method), 1223
startswith() (in module numpy.core.defchararray), 549
startswith() (numpy.chararray method), 182
startswith() (numpy.core.defchararray.chararray method), 571
std (in module numpy.ma), 380, 836
std() (in module numpy), 1266
std() (numpy.generic method), 85
std() (numpy.ma.MaskedArray method), 285, 387, 843
std() (numpy.matrix method), 149
std() (numpy.ndarray method), 30, 64
std() (numpy.recarray method), 213
std() (numpy.record method), 225
str (numpy.dtype attribute), 92, 98, 590
stride, 35
strides (numpy.chararray attribute), 167
strides (numpy.core.defchararray.chararray attribute), 556
strides (numpy.generic attribute), 75, 77
strides (numpy.ma.MaskedArray attribute), 257
strides (numpy.matrix attribute), 127
strides (numpy.ndarray attribute), 11, 37
strides (numpy.recarray attribute), 196
strides (numpy.record attribute), 218
strip() (in module numpy.core.defchararray), 539
strip() (numpy.chararray method), 182
strip() (numpy.core.defchararray.chararray method), 571
subarray

dtype, 88, 96
subtype (numpy.dtype attribute), 92, 100, 590
subtract (in module numpy), 920
sum (in module numpy.ma), 381, 837
sum() (in module numpy), 895
sum() (numpy.generic method), 85
sum() (numpy.ma.MaskedArray method), 286, 388, 845
sum() (numpy.matrix method), 150
sum() (numpy.ndarray method), 30, 64
sum() (numpy.recarray method), 214
sum() (numpy.record method), 225
svd() (in module numpy.linalg), 718
swapaxes (in module numpy.ma), 323, 779
swapaxes (numpy.ma.MaskedArray attribute), 267, 323, 779
swapaxes() (in module numpy), 490
swapaxes() (numpy.chararray method), 182
swapaxes() (numpy.core.defchararray.chararray method), 571
swapaxes() (numpy.generic method), 85
swapaxes() (numpy.matrix method), 150
swapaxes() (numpy.ndarray method), 30, 56
swapaxes() (numpy.recarray method), 214
swapaxes() (numpy.record method), 225
swapcaset() (in module numpy.core.defchararray), 540
swapcaset() (numpy.chararray method), 182
swapcaset() (numpy.core.defchararray.chararray method), 571

T
T (numpy.chararray attribute), 162
T (numpy.core.defchararray.chararray attribute), 551
T (numpy.generic attribute), 75, 77
T (numpy.ma.MaskedArray attribute), 268
T (numpy.matrix attribute), 120, 121
T (numpy.ndarray attribute), 6, 40, 491
T (numpy.recarray attribute), 190
T (numpy.record attribute), 217
take() (in module numpy), 662
take() (numpy.chararray method), 182
take() (numpy.core.defchararray.chararray method), 571
take() (numpy.generic method), 85
take() (numpy.ma.MaskedArray method), 276
take() (numpy.matrix method), 150
take() (numpy.ndarray method), 30, 58
take() (numpy.recarray method), 214
take() (numpy.record method), 226	
tan (in module numpy), 877
tanh (in module numpy), 887
tensordot() (in module numpy), 708
tensorinv() (in module numpy.linalg), 735
tensorsolve() (in module numpy.linalg), 731
terminal_has_colors() (in module
numpy.distutils.misc_util), 1306
Index