NumPy User Guide

Release 1.14.0

Written by the NumPy community

January 08, 2018

CONTENTS

This guide is intended as an introductory overview of NumPy and explains how to install and make use of the most important features of NumPy. For detailed reference documentation of the functions and classes contained in the package, see the reference.

CHAPTER

ONE

SETTING UP

1.1 What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, `a` and `b`, we could iterate over each element:

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

This produces the correct answer, but if `a` and `b` each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

```
for (i = 0; i < rows; i++): {
  c[i] = a[i]*b[i];
}
```

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

```
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

NumPy gives us the best of both worlds: element-by-element operations are the "default mode" when an *ndarray* is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

```
c = a * b
```

does what the earlier examples do, at near-C speeds, but with the code simplicity we expect from something based on Python. Indeed, the NumPy idiom is even simpler! This last example illustrates two of NumPy's features which are the basis of much of its power: vectorization and broadcasting.

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
- vectorization results in more "Pythonic" code. Without vectorization, our code would be littered with inefficient and difficult to read `for` loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, `a` and `b` could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is "expandable" to the shape of the larger in such a way that the resulting broadcast is unambiguous. For detailed "rules" of broadcasting see *`numpy.doc.broadcasting`*.

NumPy fully supports an object-oriented approach, starting, once again, with *ndarray*. For example, *ndarray* is a class, possessing numerous methods and attributes. Many of its methods mirror functions in the outer-most NumPy namespace, giving the programmer complete freedom to code in whichever paradigm she prefers and/or which seems most appropriate to the task at hand.

1.2 Installing NumPy

In most use cases the best way to install NumPy on your system is by using a pre-built package for your operating system. Please see http://scipy.org/install.html for links to available options.

For instructions on building for source package, see *Building from source*. This information is useful mainly for advanced users.

CHAPTER

TWO

QUICKSTART TUTORIAL

2.1 Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the Python tutorial.

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Please see http://scipy.org/install.html for instructions.

2.2 The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called *axes*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy's array class is called `ndarray`. It is also known by the alias `array`. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an `ndarray` object are:

ndarray.ndim
: the number of axes (dimensions) of the array.

ndarray.shape
: the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the number of axes, `ndim`.

ndarray.size
: the total number of elements of the array. This is equal to the product of the elements of `shape`.

ndarray.dtype
: an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize
: the size in bytes of each element of the array. For example, an array of elements of type `float64` has

`itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

ndarray.data
the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

2.2.1 An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

2.2.2 Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the `array` function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

`array` transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function `zeros` creates an array full of zeros, the function `ones` creates an array full of ones, and the function `empty` creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is `float64`.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized, output may vary
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

To create sequences of numbers, NumPy provides a function analogous to `range` that returns arrays instead of lists.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of␣
↪points
>>> f = np.sin(x)
```

See also:

`array`, `zeros`, `zeros_like`, `ones`, `ones_like`, `empty`, `empty_like`, `arange`, `linspace`, `numpy.random.rand`, `numpy.random.randn`, `fromfunction`, `fromfile`

2.2.3 Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

See *below* to get more details on `reshape`.

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> print(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

```
>>> np.set_printoptions(threshold=np.nan)
```

2.2.4 Basic Operations

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False])
```

Unlike in many matrix languages, the product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function or method:

```
>>> A = np.array( [[1,1],
...                [0,1]] )
>>> B = np.array( [[2,0],
...                [3,4]] )
>>> A*B                         # elementwise product
array([[2, 0],
       [0, 4]])
>>> A.dot(B)                    # matrix product
array([[5, 4],
       [3, 4]])
>>> np.dot(A, B)                # another matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
>>> a += b                  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with␣
↪casting rule 'same_kind'
```

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
```

```
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

2.2.5 Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions"(`ufunc`). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
```

```
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

See also:

`all`, `any`, `apply_along_axis`, `argmax`, `argmin`, `argsort`, `average`, `bincount`, `ceil`, `clip`, `conj`, `corrcoef`, `cov`, `cross`, `cumprod`, `cumsum`, `diff`, `dot`, `floor`, `inner`, *inv*, `lexsort`, `max`, `maximum`, `mean`, `median`, `min`, `minimum`, `nonzero`, `outer`, `prod`, `re`, `round`, `sort`, `std`, `sum`, `trace`, `transpose`, `var`, `vdot`, `vectorize`, `where`

2.2.6 Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6,␣
↪exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))
...
nan
1.0
nan
3.0
nan
5.0
6.0
7.0
8.0
9.0
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
```

```
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
>>> b[-1]                                  # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

The expression within brackets in `b[i]` is treated as an `i` followed by as many instances of `:` as needed to represent the remaining axes. NumPy also allows you to write this using dots as `b[i,...]`.

The **dots** (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is an array with 5 axes, then

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`,
- `x[...,3]` to `x[:,:,:,:,3]` and
- `x[4,...,5,:]` to `x[4,:,:,5,:]`.

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
```

```
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

See also:

Indexing, arrays.indexing (reference), `newaxis`, `ndenumerate`, `indices`

2.3 Shape Manipulation

2.3.1 Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

The order of the elements in the array resulting from ravel() is normally "C-style", that is, the rightmost index "changes the fastest", so the element after a[0,0] is a[0,1]. If the array is reshaped to some other shape, again the array is treated as "C-style". NumPy normally creates arrays stored in this order, so ravel() will usually not need to copy its argument, but if the array was made by taking slices of another array or created with unusual options, it may need to be copied. The functions ravel() and reshape() can also be instructed, using an optional argument, to use FORTRAN-style arrays, in which the leftmost index changes the fastest.

The `reshape` function returns its argument with a modified shape, whereas the `ndarray.resize` method modifies the array itself:

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

See also:

`ndarray.shape`, `reshape`, `resize`, `ravel`

2.3.2 Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

The function `column_stack` stacks 1D arrays as columns into a 2D array. It is equivalent to `hstack` only for 2D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))      # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
```

```
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))      # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))            # the result is different
array([ 4., 2., 3., 8.])
>>> a[:,newaxis]                # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

On the other hand, the function `row_stack` is equivalent to `vstack` for any input arrays. In general, for arrays of with more than two dimensions, `hstack` stacks along their second axes, `vstack` stacks along their first axes, and `concatenate` allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Note

In complex cases, `r_` and `c_` are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (":")

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

When used with arrays as arguments, `r_` and `c_` are similar to `vstack` and `hstack` in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

See also:

`hstack`, `vstack`, `column_stack`, `concatenate`, `c_`, `r_`

2.3.3 Splitting one array into several smaller ones

Using `hsplit`, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
       [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

`vsplit` splits along the vertical axis, and `array_split` allows one to specify along which axis to split.

2.4 Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

2.4.1 No Copy at All

Simple assignments make no copy of array objects or of their data.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

Python passes mutable objects as references, so function calls make no copy.

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                           # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

2.4.2 View or Shallow Copy

Different array objects can share the same data. The `view` method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6                      # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Slicing an array returns a view of it:

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,
↪1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and␣
↪s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

2.4.3 Deep Copy

The `copy` method makes a complete copy of the array and its data.

```
>>> d = a.copy()                          # a new array object with new data is␣
↪created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

2.4.4 Functions and Methods Overview

Here is a list of some useful NumPy functions and methods names ordered in categories. See routines for the full list.

Array Creation
: `arange`, `array`, `copy`, `empty`, `empty_like`, `eye`, `fromfile`, `fromfunction`, `identity`, `linspace`, `logspace`, `mgrid`, `ogrid`, `ones`, `ones_like`, *r*, `zeros`, `zeros_like`

Conversions
: `ndarray.astype`, `atleast_1d`, `atleast_2d`, `atleast_3d`, `mat`

Manipulations
: `array_split`, `column_stack`, `concatenate`, `diagonal`, `dsplit`, `dstack`, `hsplit`, `hstack`, `ndarray.item`, `newaxis`, `ravel`, `repeat`, `reshape`, `resize`, `squeeze`, `swapaxes`, `take`, `transpose`, `vsplit`, `vstack`

Questions
: `all`, `any`, `nonzero`, `where`

Ordering
: `argmax`, `argmin`, `argsort`, `max`, `min`, `ptp`, `searchsorted`, `sort`

Operations
: `choose`, `compress`, `cumprod`, `cumsum`, `inner`, `ndarray.fill`, `imag`, `prod`, `put`, `putmask`, `real`, `sum`

Basic Statistics
: `cov`, `mean`, `std`, `var`

Basic Linear Algebra
: `cross`, `dot`, `outer`, `linalg.svd`, `vdot`

2.5 Less Basic

2.5.1 Broadcasting rules

Broadcasting allows universal functions to deal in a meaningful way with inputs that do not have exactly the same shape.

The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

After application of the broadcasting rules, the sizes of all arrays must match. More details can be found in *Broadcasting*.

2.6 Fancy indexing and index tricks

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

2.6.1 Indexing with Arrays of Indices

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional array of indices
>>> a[j]                                       # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

When the indexed array `a` is multidimensional, a single array of indices refers to the first dimension of `a`. The following example shows this behavior by converting an image of labels into a color image using a palette.

```
>>> palette = np.array( [ [0,0,0],                # black
...                       [255,0,0],              # red
...                       [0,255,0],              # green
...                       [0,0,255],              # blue
...                       [255,255,255] ] )       # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],           # each value corresponds to a color␣
↪in the palette
...                    [ 0, 3, 4, 0 ]  ] )
>>> palette[image]                            # the (2,4,3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1],                        # indices for the first dim of a
...                 [1,2] ] )
>>> j = np.array( [ [2,1],                        # indices for the second dim
...                 [3,3] ] )
>>>
>>> a[i,j]                                     # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                                     # i.e., a[ : , j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
       [[10,  9],
        [11, 11]]])
```

Naturally, we can put `i` and `j` in a sequence (say a list) and then do the indexing with the list.

```
>>> l = [i,j]
>>> a[l]                                       # equivalent to a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

However, we can not do this by putting `i` and `j` into an array, because this array will be interpreted as indexing the first dimension of a.

```
>>> s = np.array( [i,j] )
>>> a[s]                                       # not what we want
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
>>>
>>> a[tuple(s)]                                # same as a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

Another common use of indexing with arrays is the search of the maximum value of time-dependent series:

```
>>> time = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
>>> time
array([  20.  ,   51.25,   82.5 ,  113.75,  145.  ])
>>> data
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
```

```
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
>>>
>>> ind = data.argmax(axis=0)                  # index of the maxima for each series
>>> ind
array([2, 0, 3, 1])
>>>
>>> time_max = time[ind]                       # times corresponding to the maxima
>>>
>>> data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
>>>
>>> time_max
array([  82.5 ,   20.  ,  113.75,   51.25])
>>> data_max
array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
>>>
>>> np.all(data_max == data.max(axis=0))
True
```

You can also use indexing with arrays as a target to assign to:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]=[1,2,3]
>>> a
array([2, 1, 3, 3, 4])
```

This is reasonable enough, but watch out if you want to use Python's += construct, as it may not do what you expect:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires "a+=1" to be equivalent to "a = a + 1".

2.6.2 Indexing with Boolean Arrays

When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.

The most natural way one can think of for boolean indexing is to use boolean arrays that have *the same shape* as the original array:

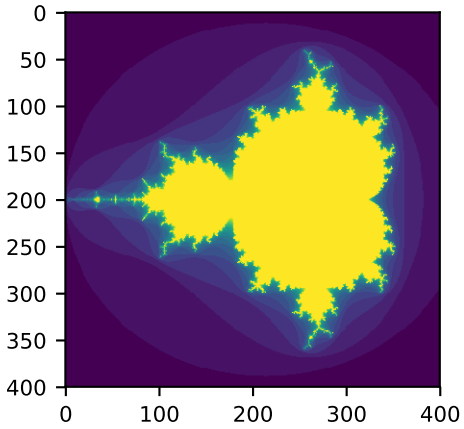
```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                          # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b]                                       # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

This property can be very useful in assignments:

```
>>> a[b] = 0                                   # All elements of 'a' higher than 4␣
↪become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
...     c = x+y*1j
...     z = c
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + c
...         diverge = z*np.conj(z) > 2**2            # who is diverging
...         div_now = diverge & (divtime==maxit)  # who is diverging now
...         divtime[div_now] = i                  # note when
...         z[diverge] = 2                        # avoid diverging too much
...
...     return divtime
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```



The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want:

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])             # first dim selection
>>> b2 = np.array([True,False,True,False])       # second dim selection
>>>
>>> a[b1,:]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                  # a weird thing to do
array([ 4, 10])
```

Note that the length of the 1D boolean array must coincide with the length of the dimension (or axis) you want to slice. In the previous example, `b1` has length 3 (the number of *rows* in `a`), and `b2` (of length 4) is suitable to index the 2nd axis (columns) of `a`.

2.6.3 The ix_() function

The `ix_` function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the a+b*c for all the triplets taken from each of the vectors a, b and c:

```
>>> a = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
>>> ax,bx,cx = np.ix_(a,b,c)
>>> ax
```

```
array([[[2]],
       [[3]],
       [[4]],
       [[5]]])
>>> bx
array([[[8],
        [5],
        [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
>>> result = ax+bx*cx
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],
       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],
       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],
       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

You could also implement the reduce as follows:

```
>>> def ufunc_reduce(ufct, *vectors):
...    vs = np.ix_(*vectors)
...    r = ufct.identity
...    for v in vs:
...        r = ufct(r,v)
...    return r
```

and then use it as:

```
>>> ufunc_reduce(np.add,a,b,c)
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],
       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],
       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],
       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

The advantage of this version of reduce compared to the normal ufunc.reduce is that it makes use of the Broadcasting Rules in order to avoid creating an argument array the size of the output times the number of vectors.

2.6.4 Indexing with strings

See *Structured arrays*.

2.7 Linear Algebra

Work in progress. Basic linear algebra to be included here.

2.7.1 Simple Array Operations

See linalg.py in numpy folder for more.

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]

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

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> np.dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> np.trace(u)  # trace
2.0

>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(a, y)
array([[-3.],
       [ 4.]])

>>> np.linalg.eig(j)
(array([ 0.+1.j,  0.-1.j]), array([[ 0.70710678+0.j        ,  0.70710678-0.j        ],
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]]))
```

```
Parameters:
    square matrix
Returns
    The eigenvalues, each repeated according to its multiplicity.
    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]`` .
```

2.8 Tricks and Tips

Here we give a list of short and useful tips.

2.8.1 "Automatic" Reshaping

To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

```
>>> a = np.arange(30)
>>> a.shape = 2,-1,3  # -1 means "whatever is needed"
>>> a.shape
(2, 5, 3)
>>> 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],
        [24, 25, 26],
        [27, 28, 29]]])
```

2.8.2 Vector Stacking

How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if `x` and `y` are two vectors of the same length you only need do `m=[x;y]`. In NumPy this works via the functions `column_stack`, `dstack`, `hstack` and `vstack`, depending on the dimension in which the stacking is to be done. For example:

```
x = np.arange(0,10,2)                     # x=([0,2,4,6,8])
y = np.arange(5)                          # y=([0,1,2,3,4])
m = np.vstack([x,y])                      # m=([[0,2,4,6,8],
                                          #     [0,1,2,3,4]])
xy = np.hstack([x,y])                     # xy =([0,2,4,6,8,0,1,2,3,4])
```

The logic behind those functions in more than two dimensions can be strange.

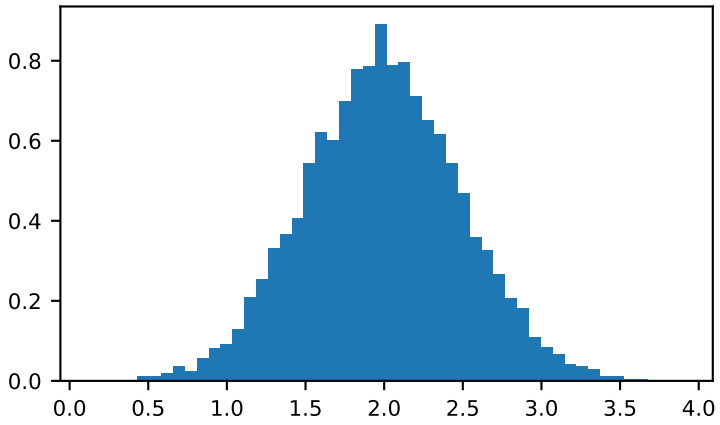
See also:

NumPy for Matlab users

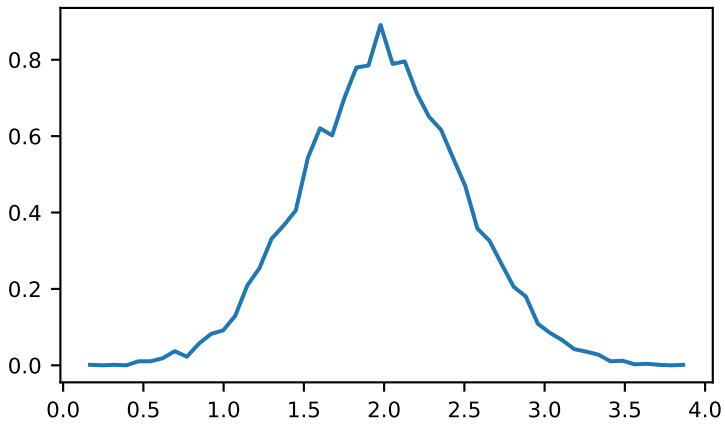
2.8.3 Histograms

The NumPy `histogram` function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist` plots the histogram automatically, while `numpy.histogram` only generates the data.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
>>> mu, sigma = 2, 0.5
>>> v = np.random.normal(mu,sigma,10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, normed=1)       # matplotlib version (plot)
>>> plt.show()
```




```
>>> # Compute the histogram with numpy and then plot it
>>> (n, bins) = np.histogram(v, bins=50, normed=True)  # NumPy version (no plot)
>>> plt.plot(.5*(bins[1:]+bins[:-1]), n)
>>> plt.show()
```



2.9 Further reading

- The Python tutorial
- reference
- SciPy Tutorial
- SciPy Lecture Notes
- A matlab, R, IDL, NumPy/SciPy dictionary

CHAPTER

THREE

NUMPY BASICS

3.1 Data types

See also:

Data type objects

3.1.1 Array types and conversions between types

NumPy supports a much greater variety of numerical types than Python does. This section shows which are available, and how to modify an array's data-type.

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Additionally to `intc` the platform dependent C integer types `short`, `long`, `longlong` and their unsigned versions are defined.

NumPy numerical types are instances of `dtype` (data-type) objects, each having unique characteristics. Once you have imported NumPy using

```
>>> import numpy as np
```

the dtypes are available as `np.bool_`, `np.float32`, etc.

Advanced types, not listed in the table above, are explored in section *Structured arrays*.

There are 5 basic numerical types representing booleans (bool), integers (int), unsigned integers (uint) floating point (float) and complex. Those with numbers in their name indicate the bitsize of the type (i.e. how many bits are needed to represent a single value in memory). Some types, such as `int` and `intp`, have differing bitsizes, dependent on the platforms (e.g. 32-bit vs. 64-bit machines). This should be taken into account when interfacing with low-level code (such as C or Fortran) where the raw memory is addressed.

Data-types can be used as functions to convert python numbers to array scalars (see the array scalar section for an explanation), python sequences of numbers to arrays of that type, or as arguments to the dtype keyword that many numpy functions or methods accept. Some examples:

```
>>> import numpy as np
>>> x = np.float32(1.0)
>>> x
1.0
>>> y = np.int_([1,2,4])
>>> y
array([1, 2, 4])
>>> z = np.arange(3, dtype=np.uint8)
>>> z
array([0, 1, 2], dtype=uint8)
```

Array types can also be referred to by character codes, mostly to retain backward compatibility with older packages such as Numeric. Some documentation may still refer to these, for example:

```
>>> np.array([1, 2, 3], dtype='f')
array([ 1.,  2.,  3.], dtype=float32)
```

We recommend using dtype objects instead.

To convert the type of an array, use the .astype() method (preferred) or the type itself as a function. For example:

```
>>> z.astype(float)
array([  0.,  1.,  2.])
>>> np.int8(z)
array([0, 1, 2], dtype=int8)
```

Note that, above, we use the *Python* float object as a dtype. NumPy knows that `int` refers to `np.int_`, `bool` means `np.bool_`, that `float` is `np.float_` and `complex` is `np.complex_`. The other data-types do not have Python equivalents.

To determine the type of an array, look at the dtype attribute:

```
>>> z.dtype
dtype('uint8')
```

dtype objects also contain information about the type, such as its bit-width and its byte-order. The data type can also be used indirectly to query properties of the type, such as whether it is an integer:

```
>>> d = np.dtype(int)
>>> d
dtype('int32')

>>> np.issubdtype(d, np.integer)
True
```

```
>>> np.issubdtype(d, np.floating)
False
```

3.1.2 Array Scalars

NumPy generally returns elements of arrays as array scalars (a scalar with an associated dtype). Array scalars differ from Python scalars, but for the most part they can be used interchangeably (the primary exception is for versions of Python older than v2.x, where integer array scalars cannot act as indices for lists and tuples). There are some exceptions, such as when code requires very specific attributes of a scalar or when it checks specifically whether a value is a Python scalar. Generally, problems are easily fixed by explicitly converting array scalars to Python scalars, using the corresponding Python type function (e.g., `int`, `float`, `complex`, `str`, `unicode`).

The primary advantage of using array scalars is that they preserve the array type (Python may not have a matching scalar type available, e.g. `int16`). Therefore, the use of array scalars ensures identical behaviour between arrays and scalars, irrespective of whether the value is inside an array or not. NumPy scalars also have many of the same methods arrays do.

3.1.3 Extended Precision

Python's floating-point numbers are usually 64-bit floating-point numbers, nearly equivalent to `np.float64`. In some unusual situations it may be useful to use floating-point numbers with more precision. Whether this is possible in numpy depends on the hardware and on the development environment: specifically, x86 machines provide hardware floating-point with 80-bit precision, and while most C compilers provide this as their `long double` type, MSVC (standard for Windows builds) makes `long double` identical to `double` (64 bits). NumPy makes the compiler's `long double` available as `np.longdouble` (and `np.clongdouble` for the complex numbers). You can find out what your numpy provides with `np.finfo(np.longdouble)`.

NumPy does not provide a dtype with more precision than C `long doubles`; in particular, the 128-bit IEEE quad precision data type (FORTRAN's `REAL*16`) is not available.

For efficient memory alignment, `np.longdouble` is usually stored padded with zero bits, either to 96 or 128 bits. Which is more efficient depends on hardware and development environment; typically on 32-bit systems they are padded to 96 bits, while on 64-bit systems they are typically padded to 128 bits. `np.longdouble` is padded to the system default; `np.float96` and `np.float128` are provided for users who want specific padding. In spite of the names, `np.float96` and `np.float128` provide only as much precision as `np.longdouble`, that is, 80 bits on most x86 machines and 64 bits in standard Windows builds.

Be warned that even if `np.longdouble` offers more precision than python `float`, it is easy to lose that extra precision, since python often forces values to pass through `float`. For example, the `%` formatting operator requires its arguments to be converted to standard python types, and it is therefore impossible to preserve extended precision even if many decimal places are requested. It can be useful to test your code with the value `1 + np.finfo(np.longdouble).eps`.

3.2 Array creation

See also:

Array creation routines

3.2.1 Introduction

There are 5 general mechanisms for creating arrays:

1. Conversion from other Python structures (e.g., lists, tuples)
2. Intrinsic numpy array array creation objects (e.g., arange, ones, zeros, etc.)
3. Reading arrays from disk, either from standard or custom formats
4. Creating arrays from raw bytes through the use of strings or buffers
5. Use of special library functions (e.g., random)

This section will not cover means of replicating, joining, or otherwise expanding or mutating existing arrays. Nor will it cover creating object arrays or structured arrays. Both of those are covered in their own sections.

3.2.2 Converting Python array_like Objects to NumPy Arrays

In general, numerical data arranged in an array-like structure in Python can be converted to arrays through the use of the array() function. The most obvious examples are lists and tuples. See the documentation for array() for details for its use. Some objects may support the array-protocol and allow conversion to arrays this way. A simple way to find out if the object can be converted to a numpy array using array() is simply to try it interactively and see if it works! (The Python Way).

Examples:

```
>>> x = np.array([2,3,1,0])
>>> x = np.array([2, 3, 1, 0])
>>> x = np.array([[1,2.0],[0,0],(1+1j,3.)]) # note mix of tuple and lists,
    and types
>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j], [ 1.+1.j, 3.+0.j]])
```

3.2.3 Intrinsic NumPy Array Creation

NumPy has built-in functions for creating arrays from scratch:

zeros(shape) will create an array filled with 0 values with the specified shape. The default dtype is float64.

```
>>> np.zeros((2, 3)) array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

ones(shape) will create an array filled with 1 values. It is identical to zeros in all other respects.

arange() will create arrays with regularly incrementing values. Check the docstring for complete information on the various ways it can be used. A few examples will be given here:

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=float)
array([ 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

Note that there are some subtleties regarding the last usage that the user should be aware of that are described in the arange docstring.

linspace() will create arrays with a specified number of elements, and spaced equally between the specified beginning and end values. For example:

```
>>> np.linspace(1., 4., 6)
array([ 1. ,  1.6,  2.2,  2.8,  3.4,  4. ])
```

The advantage of this creation function is that one can guarantee the number of elements and the starting and end point, which arange() generally will not do for arbitrary start, stop, and step values.

indices() will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension. An example illustrates much better than a verbal description:

```
>>> np.indices((3,3))
array([[[0, 0, 0], [1, 1, 1], [2, 2, 2]], [[0, 1, 2], [0, 1, 2], [0, 1, 2]]])
```

This is particularly useful for evaluating functions of multiple dimensions on a regular grid.

3.2.4 Reading Arrays From Disk

This is presumably the most common case of large array creation. The details, of course, depend greatly on the format of data on disk and so this section can only give general pointers on how to handle various formats.

Standard Binary Formats

Various fields have standard formats for array data. The following lists the ones with known python libraries to read them and return numpy arrays (there may be others for which it is possible to read and convert to numpy arrays so check the last section as well)

```
HDF5: h5py
FITS: Astropy
```

Examples of formats that cannot be read directly but for which it is not hard to convert are those formats supported by libraries like PIL (able to read and write many image formats such as jpg, png, etc).

Common ASCII Formats

Comma Separated Value files (CSV) are widely used (and an export and import option for programs like Excel). There are a number of ways of reading these files in Python. There are CSV functions in Python and functions in pylab (part of matplotlib).

More generic ascii files can be read using the io package in scipy.

Custom Binary Formats

There are a variety of approaches one can use. If the file has a relatively simple format then one can write a simple I/O library and use the numpy fromfile() function and .tofile() method to read and write numpy arrays directly (mind your byteorder though!) If a good C or C++ library exists that read the data, one can wrap that library with a variety of techniques though that certainly is much more work and requires significantly more advanced knowledge to interface with C or C++.

Use of Special Libraries

There are libraries that can be used to generate arrays for special purposes and it isn't possible to enumerate all of them. The most common uses are use of the many array generation functions in random that can generate arrays of random values, and some utility functions to generate special matrices (e.g. diagonal).

3.3 I/O with NumPy

3.3.1 Importing data with `genfromtxt`

NumPy provides several functions to create arrays from tabular data. We focus here on the `genfromtxt` function.

In a nutshell, `genfromtxt` runs two main loops. The first loop converts each line of the file in a sequence of strings. The second loop converts each string to the appropriate data type. This mechanism is slower than a single loop, but gives more flexibility. In particular, `genfromtxt` is able to take missing data into account, when other faster and simpler functions like `loadtxt` cannot.

Note: When giving examples, we will use the following conventions:

```
>>> import numpy as np
>>> from io import BytesIO
```

Defining the input

The only mandatory argument of `genfromtxt` is the source of the data. It can be a string, a list of strings, or a generator. If a single string is provided, it is assumed to be the name of a local or remote file, or an open file-like object with a `read` method, for example, a file or `StringIO.StringIO` object. If a list of strings or a generator returning strings is provided, each string is treated as one line in a file. When the URL of a remote file is passed, the file is automatically downloaded to the current directory and opened.

Recognized file types are text files and archives. Currently, the function recognizes `gzip` and `bz2` (*bzip2*) archives. The type of the archive is determined from the extension of the file: if the filename ends with `'.gz'`, a `gzip` archive is expected; if it ends with `'bz2'`, a `bzip2` archive is assumed.

Splitting the lines into columns

The `delimiter` argument

Once the file is defined and open for reading, `genfromtxt` splits each non-empty line into a sequence of strings. Empty or commented lines are just skipped. The `delimiter` keyword is used to define how the splitting should take place.

Quite often, a single character marks the separation between columns. For example, comma-separated files (CSV) use a comma (`,`) or a semicolon (`;`) as delimiter:

```
>>> data = "1, 2, 3\n4, 5, 6"
>>> np.genfromtxt(BytesIO(data), delimiter=",")
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Another common separator is `"\t"`, the tabulation character. However, we are not limited to a single character, any string will do. By default, `genfromtxt` assumes `delimiter=None`, meaning that the line is split along white spaces (including tabs) and that consecutive white spaces are considered as a single white space.

Alternatively, we may be dealing with a fixed-width file, where columns are defined as a given number of characters. In that case, we need to set `delimiter` to a single integer (if all the columns have the same size) or to a sequence of integers (if columns can have different sizes):

```
>>> data = "  1  2  3\n  4  5 67\n890123  4"
>>> np.genfromtxt(BytesIO(data), delimiter=3)
array([[   1.,    2.,    3.],
       [   4.,    5.,   67.],
       [ 890.,  123.,    4.]])
>>> data = "123456789\n   4  7 9\n   4567 9"
>>> np.genfromtxt(BytesIO(data), delimiter=(4, 3, 2))
array([[ 1234.,   567.,    89.],
       [    4.,     7.,     9.],
       [    4.,   567.,     9.]])
```

The `autostrip` argument

By default, when a line is decomposed into a series of strings, the individual entries are not stripped of leading nor trailing white spaces. This behavior can be overwritten by setting the optional argument `autostrip` to a value of `True`:

```
>>> data = "1, abc , 2\n 3, xxx, 4"
>>> # Without autostrip
>>> np.genfromtxt(BytesIO(data), delimiter=",", dtype="|U5")
array([['1', ' abc ', ' 2'],
       ['3', ' xxx', ' 4']],
      dtype='|U5')
>>> # With autostrip
>>> np.genfromtxt(BytesIO(data), delimiter=",", dtype="|U5", autostrip=True)
array([['1', 'abc', '2'],
       ['3', 'xxx', '4']],
      dtype='|U5')
```

The `comments` argument

The optional argument `comments` is used to define a character string that marks the beginning of a comment. By default, `genfromtxt` assumes `comments='#'`. The comment marker may occur anywhere on the line. Any character present after the comment marker(s) is simply ignored:

```
>>> data = """#
... # Skip me !
... # Skip me too !
... 1, 2
... 3, 4
... 5, 6 #This is the third line of the data
... 7, 8
... # And here comes the last line
... 9, 0
... """
>>> np.genfromtxt(BytesIO(data), comments="#", delimiter=",")
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9.  0.]]
```

Note: There is one notable exception to this behavior: if the optional argument `names=True`, the first commented line will be examined for names.

Skipping lines and choosing columns

The `skip_header` and `skip_footer` arguments

The presence of a header in the file can hinder data processing. In that case, we need to use the `skip_header` optional argument. The values of this argument must be an integer which corresponds to the number of lines to skip at the beginning of the file, before any other action is performed. Similarly, we can skip the last `n` lines of the file by using the `skip_footer` attribute and giving it a value of `n`:

```
>>> data = "\n".join(str(i) for i in range(10))
>>> np.genfromtxt(BytesIO(data),)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> np.genfromtxt(BytesIO(data),
...               skip_header=3, skip_footer=5)
array([ 3.,  4.])
```

By default, `skip_header=0` and `skip_footer=0`, meaning that no lines are skipped.

The `usecols` argument

In some cases, we are not interested in all the columns of the data but only a few of them. We can select which columns to import with the `usecols` argument. This argument accepts a single integer or a sequence of integers corresponding to the indices of the columns to import. Remember that by convention, the first column has an index of 0. Negative integers behave the same as regular Python negative indexes.

For example, if we want to import only the first and the last columns, we can use `usecols=(0, -1)`:

```
>>> data = "1 2 3\n4 5 6"
>>> np.genfromtxt(BytesIO(data), usecols=(0, -1))
array([[ 1.,  3.],
       [ 4.,  6.]])
```

If the columns have names, we can also select which columns to import by giving their name to the `usecols` argument, either as a sequence of strings or a comma-separated string:

```
>>> data = "1 2 3\n4 5 6"
>>> np.genfromtxt(BytesIO(data),
...               names="a, b, c", usecols=("a", "c"))
array([(1.0, 3.0), (4.0, 6.0)],
      dtype=[('a', '<f8'), ('c', '<f8')])
>>> np.genfromtxt(BytesIO(data),
...               names="a, b, c", usecols=("a, c"))
    array([(1.0, 3.0), (4.0, 6.0)],
          dtype=[('a', '<f8'), ('c', '<f8')])
```

Choosing the data type

The main way to control how the sequences of strings we have read from the file are converted to other types is to set the `dtype` argument. Acceptable values for this argument are:

- a single type, such as `dtype=float`. The output will be 2D with the given dtype, unless a name has been associated with each column with the use of the `names` argument (see below). Note that `dtype=float` is the default for `genfromtxt`.
- a sequence of types, such as `dtype=(int, float, float)`.
- a comma-separated string, such as `dtype="i4,f8,|U3"`.
- a dictionary with two keys `'names'` and `'formats'`.

- a sequence of tuples `(name, type)`, such as `dtype=[('A', int), ('B', float)]`.
- an existing `numpy.dtype` object.
- the special value `None`. In that case, the type of the columns will be determined from the data itself (see below).

In all the cases but the first one, the output will be a 1D array with a structured dtype. This dtype has as many fields as items in the sequence. The field names are defined with the `names` keyword.

When `dtype=None`, the type of each column is determined iteratively from its data. We start by checking whether a string can be converted to a boolean (that is, if the string matches `true` or `false` in lower cases); then whether it can be converted to an integer, then to a float, then to a complex and eventually to a string. This behavior may be changed by modifying the default mapper of the `StringConverter` class.

The option `dtype=None` is provided for convenience. However, it is significantly slower than setting the dtype explicitly.

Setting the names

The `names` argument

A natural approach when dealing with tabular data is to allocate a name to each column. A first possibility is to use an explicit structured dtype, as mentioned previously:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=[(_, int) for _ in "abc"])
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

Another simpler possibility is to use the `names` keyword with a sequence of strings or a comma-separated string:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, names="A, B, C")
array([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)],
      dtype=[('A', '<f8'), ('B', '<f8'), ('C', '<f8')])
```

In the example above, we used the fact that by default, `dtype=float`. By giving a sequence of names, we are forcing the output to a structured dtype.

We may sometimes need to define the column names from the data itself. In that case, we must use the `names` keyword with a value of `True`. The names will then be read from the first line (after the `skip_header` ones), even if the line is commented out:

```
>>> data = BytesIO("So it goes\n#a b c\n1 2 3\n 4 5 6")
>>> np.genfromtxt(data, skip_header=1, names=True)
array([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)],
      dtype=[('a', '<f8'), ('b', '<f8'), ('c', '<f8')])
```

The default value of `names` is `None`. If we give any other value to the keyword, the new names will overwrite the field names we may have defined with the dtype:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> ndtype=[('a',int), ('b', float), ('c', int)]
>>> names = ["A", "B", "C"]
>>> np.genfromtxt(data, names=names, dtype=ndtype)
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('A', '<i8'), ('B', '<f8'), ('C', '<i8')])
```

The `defaultfmt` argument

If `names=None` but a structured dtype is expected, names are defined with the standard NumPy default of `"f%i"`, yielding names like `f0`, `f1` and so forth:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int))
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('f0', '<i8'), ('f1', '<f8'), ('f2', '<i8')])
```

In the same way, if we don't give enough names to match the length of the dtype, the missing names will be defined with this default template:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), names="a")
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('a', '<i8'), ('f0', '<f8'), ('f1', '<i8')])
```

We can overwrite this default with the `defaultfmt` argument, that takes any format string:

```
>>> data = BytesIO("1 2 3\n 4 5 6")
>>> np.genfromtxt(data, dtype=(int, float, int), defaultfmt="var_%02i")
array([(1, 2.0, 3), (4, 5.0, 6)],
      dtype=[('var_00', '<i8'), ('var_01', '<f8'), ('var_02', '<i8')])
```

Note: We need to keep in mind that `defaultfmt` is used only if some names are expected but not defined.

Validating names

NumPy arrays with a structured dtype can also be viewed as `recarray`, where a field can be accessed as if it were an attribute. For that reason, we may need to make sure that the field name doesn't contain any space or invalid character, or that it does not correspond to the name of a standard attribute (like `size` or `shape`), which would confuse the interpreter. `genfromtxt` accepts three optional arguments that provide a finer control on the names:

`deletechars`
: Gives a string combining all the characters that must be deleted from the name. By default, invalid characters are `~!@#$%^&*()-=+~\|]}[{';:  /?.>,<`.

`excludelist`
: Gives a list of the names to exclude, such as `return`, `file`, `print`... If one of the input name is part of this list, an underscore character (`'_'`) will be appended to it.

`case_sensitive`
: Whether the names should be case-sensitive (`case_sensitive=True`), converted to upper case (`case_sensitive=False` or `case_sensitive='upper'`) or to lower case (`case_sensitive='lower'`).

Tweaking the conversion

The `converters` argument

Usually, defining a dtype is sufficient to define how the sequence of strings must be converted. However, some additional control may sometimes be required. For example, we may want to make sure that a date in a format `YYYY/MM/DD` is converted to a `datetime` object, or that a string like `xx%` is properly converted to a float between 0 and 1. In such cases, we should define conversion functions with the `converters` arguments.

The value of this argument is typically a dictionary with column indices or column names as keys and a conversion functions as values. These conversion functions can either be actual functions or lambda functions. In any case, they should accept only a string as input and output only a single element of the wanted type.

In the following example, the second column is converted from as string representing a percentage to a float between 0 and 1:

```
>>> convertfunc = lambda x: float(x.strip("%"))/100.
>>> data = "1, 2.3%, 45.\n6, 78.9%, 0"
>>> names = ("i", "p", "n")
>>> # General case .....
>>> np.genfromtxt(BytesIO(data), delimiter=",", names=names)
array([(1.0, nan, 45.0), (6.0, nan, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

We need to keep in mind that by default, `dtype=float`. A float is therefore expected for the second column. However, the strings `' 2.3%'` and `' 78.9%'` cannot be converted to float and we end up having `np.nan` instead. Let's now use a converter:

```
>>> # Converted case ...
>>> np.genfromtxt(BytesIO(data), delimiter=",", names=names,
...               converters={1: convertfunc})
array([(1.0, 0.023, 45.0), (6.0, 0.78900000000000003, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

The same results can be obtained by using the name of the second column (`"p"`) as key instead of its index (1):

```
>>> # Using a name for the converter ...
>>> np.genfromtxt(BytesIO(data), delimiter=",", names=names,
...               converters={"p": convertfunc})
array([(1.0, 0.023, 45.0), (6.0, 0.78900000000000003, 0.0)],
      dtype=[('i', '<f8'), ('p', '<f8'), ('n', '<f8')])
```

Converters can also be used to provide a default for missing entries. In the following example, the converter `convert` transforms a stripped string into the corresponding float or into -999 if the string is empty. We need to explicitly strip the string from white spaces as it is not done by default:

```
>>> data = "1, , 3\n 4, 5, 6"
>>> convert = lambda x: float(x.strip() or -999)
>>> np.genfromtxt(BytesIO(data), delimiter=",",
...               converters={1: convert})
array([[   1., -999.,    3.],
       [   4.,    5.,    6.]])
```

Using missing and filling values

Some entries may be missing in the dataset we are trying to import. In a previous example, we used a converter to transform an empty string into a float. However, user-defined converters may rapidly become cumbersome to manage.

The `genfromtxt` function provides two other complementary mechanisms: the `missing_values` argument is used to recognize missing data and a second argument, `filling_values`, is used to process these missing data.

`missing_values`

By default, any empty string is marked as missing. We can also consider more complex strings, such as `"N/A"` or `"???"` to represent missing or invalid data. The `missing_values` argument accepts three kind of values:

a string or a comma-separated string
: This string will be used as the marker for missing data for all the columns

a sequence of strings
In that case, each item is associated to a column, in order.

a dictionary
Values of the dictionary are strings or sequence of strings. The corresponding keys can be column indices (integers) or column names (strings). In addition, the special key `None` can be used to define a default applicable to all columns.

`filling_values`

We know how to recognize missing data, but we still need to provide a value for these missing entries. By default, this value is determined from the expected dtype according to this table:

Expected type	Default
`bool`	`False`
`int`	`-1`
`float`	`np.nan`
`complex`	`np.nan+0j`
`string`	`'???'`

We can get a finer control on the conversion of missing values with the `filling_values` optional argument. Like `missing_values`, this argument accepts different kind of values:

a single value
This will be the default for all columns

a sequence of values
Each entry will be the default for the corresponding column

a dictionary
Each key can be a column index or a column name, and the corresponding value should be a single object. We can use the special key `None` to define a default for all columns.

In the following example, we suppose that the missing values are flagged with `"N/A"` in the first column and by `"???"` in the third column. We wish to transform these missing values to 0 if they occur in the first and second column, and to -999 if they occur in the last column:

```
>>> data = "N/A, 2, 3\n4, ,???"
>>> kwargs = dict(delimiter=",",
...               dtype=int,
...               names="a,b,c",
...               missing_values={0:"N/A", 'b':" ", 2:"???"},
...               filling_values={0:0, 'b':0, 2:-999})
>>> np.genfromtxt(BytesIO(data), **kwargs)
array([(0, 2, 3), (4, 0, -999)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

`usemask`

We may also want to keep track of the occurrence of missing data by constructing a boolean mask, with `True` entries where data was missing and `False` otherwise. To do that, we just have to set the optional argument `usemask` to `True` (the default is `False`). The output array will then be a `MaskedArray`.

Shortcut functions

In addition to `genfromtxt`, the `numpy.lib.io` module provides several convenience functions derived from `genfromtxt`. These functions work the same way as the original, but they have different default values.

ndfromtxt
Always set `usemask=False`. The output is always a standard `numpy.ndarray`.

mafromtxt
Always set `usemask=True`. The output is always a `MaskedArray`

recfromtxt
Returns a standard `numpy.recarray` (if `usemask=False`) or a `MaskedRecords` array (if `usemaske=True`). The default dtype is `dtype=None`, meaning that the types of each column will be automatically determined.

recfromcsv
Like `recfromtxt`, but with a default `delimiter=","`.

3.4 Indexing

See also:

Indexing routines

Array indexing refers to any use of the square brackets ([]) to index array values. There are many options to indexing, which give numpy indexing great power, but with power comes some complexity and the potential for confusion. This section is just an overview of the various options and issues related to indexing. Aside from single element indexing, the details on most of these options are to be found in related sections.

3.4.1 Assignment vs referencing

Most of the following examples show the use of indexing when referencing data in an array. The examples work just as well when assigning to an array. See the section at the end for specific examples and explanations on how assignments work.

3.4.2 Single element indexing

Single element indexing for a 1-D array is what one expects. It work exactly like that for other standard Python sequences. It is 0-based, and accepts negative indices for indexing from the end of the array.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

Unlike lists and tuples, numpy arrays support multidimensional indexing for multidimensional arrays. That means that it is not necessary to separate each dimension's index into its own set of square brackets.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

Note that if one indexes a multidimensional array with fewer indices than dimensions, one gets a subdimensional array. For example:

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

That is, each index specified selects the array corresponding to the rest of the dimensions selected. In the above example, choosing 0 means that the remaining dimension of length 5 is being left unspecified, and that what is returned is an array of that dimensionality and size. It must be noted that the returned array is not a copy of the original, but points to the same values in memory as does the original array. In this case, the 1-D array at the first position (0) is returned. So using a single index on the returned array, results in a single element being returned. That is:

```
>>> x[0][2]
2
```

So note that `x[0,2] = x[0][2]` though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

Note to those used to IDL or Fortran memory order as it relates to indexing. NumPy uses C-order indexing. That means that the last index usually represents the most rapidly changing memory location, unlike Fortran or IDL, where the first index represents the most rapidly changing location in memory. This difference represents a great potential for confusion.

3.4.3 Other indexing options

It is possible to slice and stride arrays to extract arrays of the same number of dimensions, but of different sizes than the original. The slicing and striding works exactly the same way it does for lists and tuples except that they can be applied to multiple dimensions as well. A few examples illustrates best:

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
```

Note that slices of arrays do not copy the internal array data but also produce new views of the original data.

It is possible to index arrays with other arrays for the purposes of selecting lists of values out of arrays into new arrays. There are two different ways of accomplishing this. One uses one or more arrays of index values. The other involves giving a boolean array of the proper shape to indicate the values to be selected. Index arrays are a very powerful tool that allow one to avoid looping over individual elements in arrays and thus greatly improve performance.

It is possible to use special features to effectively increase the number of dimensions in an array through indexing so the resulting array aquires the shape needed for use in an expression or with a specific function.

3.4.4 Index arrays

NumPy arrays may be indexed with other arrays (or any other sequence- like object that can be converted to an array, such as lists, with the exception of tuples; see the end of this document for why this is). The use of index arrays ranges from simple, straightforward cases to complex, hard-to-understand cases. For all cases of index arrays, what is returned is a copy of the original data, not a view as one gets for slices.

Index arrays must be of integer type. Each value in the array indicates which value in the array to use in place of the index. To illustrate:

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

The index array consisting of the values 3, 3, 1 and 8 correspondingly create an array of length 4 (same as the index array) where each index is replaced by the value the index array has in the array being indexed.

Negative values are permitted and work as they do with single indices or slices:

```
>>> x[np.array([3,3,-3,8])]
array([7, 7, 4, 2])
```

It is an error to have index values out of bounds:

```
>>> x[np.array([3, 3, 20, 8])]
<type 'exceptions.IndexError'>: index 20 out of bounds 0<=index<9
```

Generally speaking, what is returned when index arrays are used is an array with the same shape as the index array, but with the type and values of the array being indexed. As an example, we can use a multidimensional index array instead:

```
>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

3.4.5 Indexing Multi-dimensional arrays

Things become more complex when multidimensional arrays are indexed, particularly with multidimensional index arrays. These tend to be more unusual uses, but they are permitted, and they are useful for some problems. We'll start with the simplest multidimensional case (using the array y from the previous examples):

```
>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])
```

In this case, if the index arrays have a matching shape, and there is an index array for each dimension of the array being indexed, the resultant array has the same shape as the index arrays, and the values correspond to the index set for each position in the index arrays. In this example, the first index value is 0 for both index arrays, and thus the first value of the resultant array is y[0,0]. The next value is y[2,1], and the last is y[4,2].

If the index arrays do not have the same shape, there is an attempt to broadcast them to the same shape. If they cannot be broadcast to the same shape, an exception is raised:

```
>>> y[np.array([0,2,4]), np.array([0,1])]
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be
broadcast to a single shape
```

The broadcasting mechanism permits index arrays to be combined with scalars for other indices. The effect is that the scalar value is used for all the corresponding values of the index arrays:

```
>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])
```

Jumping to the next level of complexity, it is possible to only partially index an array with index arrays. It takes a bit of thought to understand what happens in such cases. For example if we just use one index array with y:

```
>>> y[np.array([0,2,4])]
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])
```

What results is the construction of a new array where each value of the index array selects one row from the array being indexed and the resultant array has the resulting shape (number of index elements, size of row).

An example of where this may be useful is for a color lookup table where we want to map the values of an image into RGB triples for display. The lookup table could have a shape (nlookup, 3). Indexing such an array with an image with shape (ny, nx) with dtype=np.uint8 (or any integer type so long as values are with the bounds of the lookup table) will result in an array of shape (ny, nx, 3) where a triple of RGB values is associated with each pixel location.

In general, the shape of the resultant array will be the concatenation of the shape of the index array (or the shape that all the index arrays were broadcast to) with the shape of any unused dimensions (those not indexed) in the array being indexed.

3.4.6 Boolean or "mask" index arrays

Boolean arrays used as indices are treated in a different manner entirely than index arrays. Boolean arrays must be of the same shape as the initial dimensions of the array being indexed. In the most straightforward case, the boolean array has the same shape:

```
>>> b = y>20
>>> y[b]
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
```

Unlike in the case of integer index arrays, in the boolean case, the result is a 1-D array containing all the elements in the indexed array corresponding to all the true elements in the boolean array. The elements in the indexed array are always iterated and returned in row-major (C-style) order. The result is also identical to `y[np.nonzero(b)]`. As with index arrays, what is returned is a copy of the data, not a view as one gets with slices.

The result will be multidimensional if y has more dimensions than b. For example:

```
>>> b[:,5] # use a 1-D boolean whose first dim agrees with the first dim of y
array([False, False, False,  True,  True])
>>> y[b[:,5]]
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

Here the 4th and 5th rows are selected from the indexed array and combined to make a 2-D array.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to y[b, ...], which means y is indexed by b followed by as many : as are needed to fill out the rank of y. Thus the shape of the result is one dimension containing the number of True elements of the boolean array, followed by the remaining dimensions of the array being indexed.

For example, using a 2-D boolean array of shape (2,3) with four True elements to select rows from a 3-D array of shape (2,3,5) results in a 2-D result of shape (4,5):

```
>>> x = np.arange(30).reshape(2,3,5)
>>> 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, 27, 28, 29]]])
>>> b = np.array([[True, True, False], [False, True, True]])
>>> x[b]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

For further details, consult the numpy reference documentation on array indexing.

3.4.7 Combining index arrays with slices

Index arrays may be combined with slices. For example:

```
>>> y[np.array([0,2,4]),1:3]
array([[ 1,  2],
       [15, 16],
       [29, 30]])
```

In effect, the slice is converted to an index array np.array([[1,2]]) (shape (1,2)) that is broadcast with the index array to produce a resultant array of shape (3,2).

Likewise, slicing can be combined with broadcasted boolean indices:

```
>>> y[b[:,5],1:3]
array([[22, 23],
       [29, 30]])
```

3.4.8 Structural indexing tools

To facilitate easy matching of array shapes with expressions and in assignments, the np.newaxis object can be used within array indices to add new dimensions with a size of 1. For example:

```
>>> y.shape
(5, 7)
>>> y[:,np.newaxis,:].shape
(5, 1, 7)
```

Note that there are no new elements in the array, just that the dimensionality is increased. This can be handy to combine two arrays in a way that otherwise would require explicitly reshaping operations. For example:

```
>>> x = np.arange(5)
>>> x[:,np.newaxis] + x[np.newaxis,:]
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
```

The ellipsis syntax maybe used to indicate selecting in full any remaining unspecified dimensions. For example:

```
>>> z = np.arange(81).reshape(3,3,3,3)
>>> z[1,...,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

This is equivalent to:

```
>>> z[1,:,:,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

3.4.9 Assigning values to indexed arrays

As mentioned, one can select a subset of an array to assign to using a single index, slices, and index and mask arrays. The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces). For example, it is permitted to assign a constant to a slice:

```
>>> x = np.arange(10)
>>> x[2:7] = 1
```

or an array of the right size:

```
>>> x[2:7] = np.arange(5)
```

Note that assignments may result in changes if assigning higher types to lower types (like floats to ints) or even exceptions (assigning complex to floats or ints):

```
>>> x[1] = 1.2
>>> x[1]
1
>>> x[1] = 1.2j
<type 'exceptions.TypeError'>: can't convert complex to long; use
long(abs(z))
```

Unlike some of the references (such as array and mask indices) assignments are always made to the original data in the array (indeed, nothing else would make sense!). Note though, that some actions may not work as one may naively expect. This particular example is often surprising to people:

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
>>> x[np.array([1, 1, 3, 1])] += 1
>>> x
array([ 0, 11, 20, 31, 40])
```

Where people expect that the 1st location will be incremented by 3. In fact, it will only be incremented by 1. The reason is because a new array is extracted from the original (as a temporary) containing the values at 1, 1, 3, 1, then the value 1 is added to the temporary, and then the temporary is assigned back to the original array. Thus the value of the array at x[1]+1 is assigned to x[1] three times, rather than being incremented 3 times.

3.4.10 Dealing with variable numbers of indices within programs

The index syntax is very powerful but limiting when dealing with a variable number of indices. For example, if you want to write a function that can handle arguments with various numbers of dimensions without having to write special case code for each number of possible dimensions, how can that be done? If one supplies to the index a tuple, the tuple will be interpreted as a list of indices. For example (using the previous definition for the array z):

```
>>> indices = (1,1,1,1)
>>> z[indices]
40
```

So one can use code to construct tuples of any number of indices and then use these within an index.

Slices can be specified within programs by using the slice() function in Python. For example:

```
>>> indices = (1,1,1,slice(0,2)) # same as [1,1,1,0:2]
>>> z[indices]
array([39, 40])
```

Likewise, ellipsis can be specified by code by using the Ellipsis object:

```
>>> indices = (1, Ellipsis, 1) # same as [1,...,1]
>>> z[indices]
array([[28, 31, 34],
       [37, 40, 43],
       [46, 49, 52]])
```

For this reason it is possible to use the output from the np.nonzero() function directly as an index since it always returns a tuple of index arrays.

Because the special treatment of tuples, they are not automatically converted to an array as a list would be. As an example:

```
>>> z[[1,1,1,1]] # produces a large array
array([[[[27, 28, 29],
         [30, 31, 32], ...
>>> z[(1,1,1,1)] # returns a single value
40
```

3.5 Broadcasting

See also:

`numpy.broadcast`

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
```

```
>>> a * b
array([ 2.,  4.,  6.])
```

NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies, so that broadcasting operations are as memory and computationally efficient as possible.

The code in the second example is more efficient than that in the first because broadcasting moves less memory around during the multiplication (`b` is a scalar rather than an array).

3.5.1 General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a `ValueError: frames are not aligned` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

Arrays do not need to have the same *number* of dimensions. For example, if you have a `256x256x3` array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or "copied" to match the other.

In the following example, both the `A` and `B` arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4
```

```
A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

Here are examples of shapes that do not broadcast:

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

An example of broadcasting in practice:

```
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))

>>> x.shape
(4,)

>>> y.shape
(5,)

>>> x + y
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be broadcast to a␣
↪single shape

>>> xx.shape
(4, 1)

>>> y.shape
(5,)

>>> (xx + y).shape
(4, 5)

>>> xx + y
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])

>>> x.shape
```

```
(4,)

>>> z.shape
(3, 4)

>>> (x + z).shape
(3, 4)

>>> x + z
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays. The following example shows an outer addition operation of two 1-d arrays:

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```

Here the `newaxis` index operator inserts a new axis into `a`, making it a two-dimensional `4x1` array. Combining the `4x1` array with `b`, which has shape `(3,)`, yields a `4x3` array.

See this article for illustrations of broadcasting concepts.

3.6 Byte-swapping

3.6.1 Introduction to byte ordering and ndarrays

The `ndarray` is an object that provide a python array interface to data in memory.

It often happens that the memory that you want to view with an array is not of the same byte ordering as the computer on which you are running Python.

For example, I might be working on a computer with a little-endian CPU - such as an Intel Pentium, but I have loaded some data from a file written by a computer that is big-endian. Let's say I have loaded 4 bytes from a file written by a Sun (big-endian) computer. I know that these 4 bytes represent two 16-bit integers. On a big-endian machine, a two-byte integer is stored with the Most Significant Byte (MSB) first, and then the Least Significant Byte (LSB). Thus the bytes are, in memory order:

1. MSB integer 1
2. LSB integer 1
3. MSB integer 2
4. LSB integer 2

Let's say the two integers were in fact 1 and 770. Because 770 = 256 * 3 + 2, the 4 bytes in memory would contain respectively: 0, 1, 3, 2. The bytes I have loaded from the file would have these contents:

```
>>> big_end_str = chr(0) + chr(1) + chr(3) + chr(2)
>>> big_end_str
'\x00\x01\x03\x02'
```

We might want to use an `ndarray` to access these integers. In that case, we can create an array around this memory, and tell numpy that there are two integers, and that they are 16 bit and big-endian:

```
>>> import numpy as np
>>> big_end_arr = np.ndarray(shape=(2,),dtype='>i2', buffer=big_end_str)
>>> big_end_arr[0]
1
>>> big_end_arr[1]
770
```

Note the array `dtype` above of `>i2`. The `>` means 'big-endian' (`<` is little-endian) and `i2` means 'signed 2-byte integer'. For example, if our data represented a single unsigned 4-byte little-endian integer, the dtype string would be `<u4`.

In fact, why don't we try that?

```
>>> little_end_u4 = np.ndarray(shape=(1,),dtype='<u4', buffer=big_end_str)
>>> little_end_u4[0] == 1 * 256**1 + 3 * 256**2 + 2 * 256**3
True
```

Returning to our `big_end_arr` - in this case our underlying data is big-endian (data endianness) and we've set the dtype to match (the dtype is also big-endian). However, sometimes you need to flip these around.

Warning: Scalars currently do not include byte order information, so extracting a scalar from an array will return an integer in native byte order. Hence:

```
>>> big_end_arr[0].dtype.byteorder == little_end_u4[0].dtype.byteorder
True
```

3.6.2 Changing byte ordering

As you can imagine from the introduction, there are two ways you can affect the relationship between the byte ordering of the array and the underlying memory it is looking at:

- Change the byte-ordering information in the array dtype so that it interprets the underlying data as being in a different byte order. This is the role of `arr.newbyteorder()`
- Change the byte-ordering of the underlying data, leaving the dtype interpretation as it was. This is what `arr.byteswap()` does.

The common situations in which you need to change byte ordering are:

1. Your data and dtype endianess don't match, and you want to change the dtype so that it matches the data.
2. Your data and dtype endianess don't match, and you want to swap the data so that they match the dtype
3. Your data and dtype endianess match, but you want the data swapped and the dtype to reflect this

Data and dtype endianness don't match, change dtype to match data

We make something where they don't match:

```
>>> wrong_end_dtype_arr = np.ndarray(shape=(2,),dtype='<i2', buffer=big_end_str)
>>> wrong_end_dtype_arr[0]
256
```

The obvious fix for this situation is to change the dtype so it gives the correct endianness:

```
>>> fixed_end_dtype_arr = wrong_end_dtype_arr.newbyteorder()
>>> fixed_end_dtype_arr[0]
1
```

Note the array has not changed in memory:

```
>>> fixed_end_dtype_arr.tobytes() == big_end_str
True
```

Data and type endianness don't match, change data to match dtype

You might want to do this if you need the data in memory to be a certain ordering. For example you might be writing the memory out to a file that needs a certain byte ordering.

```
>>> fixed_end_mem_arr = wrong_end_dtype_arr.byteswap()
>>> fixed_end_mem_arr[0]
1
```

Now the array *has* changed in memory:

```
>>> fixed_end_mem_arr.tobytes() == big_end_str
False
```

Data and dtype endianness match, swap data and dtype

You may have a correctly specified array dtype, but you need the array to have the opposite byte order in memory, and you want the dtype to match so the array values make sense. In this case you just do both of the previous operations:

```
>>> swapped_end_arr = big_end_arr.byteswap().newbyteorder()
>>> swapped_end_arr[0]
1
>>> swapped_end_arr.tobytes() == big_end_str
False
```

An easier way of casting the data to a specific dtype and byte ordering can be achieved with the ndarray astype method:

```
>>> swapped_end_arr = big_end_arr.astype('<i2')
>>> swapped_end_arr[0]
1
>>> swapped_end_arr.tobytes() == big_end_str
False
```

3.7 Structured arrays

3.7.1 Introduction

Structured arrays are ndarrays whose datatype is a composition of simpler datatypes organized as a sequence of named fields. For example,

```
>>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
...              dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])
>>> x
array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
      dtype=[('name', 'S10'), ('age', '<i4'), ('weight', '<f4')])
```

Here `x` is a one-dimensional array of length two whose datatype is a structure with three fields: 1. A string of length 10 or less named 'name', 2. a 32-bit integer named 'age', and 3. a 32-bit float named 'weight'.

If you index `x` at position 1 you get a structure:

```
>>> x[1]
('Fido', 3, 27.0)
```

You can access and modify individual fields of a structured array by indexing with the field name:

```
>>> x['age']
array([9, 3], dtype=int32)
>>> x['age'] = 5
>>> x
array([('Rex', 5, 81.0), ('Fido', 5, 27.0)],
      dtype=[('name', 'S10'), ('age', '<i4'), ('weight', '<f4')])
```

Structured arrays are designed for low-level manipulation of structured data, for example, for interpreting binary blobs. Structured datatypes are designed to mimic 'structs' in the C language, making them also useful for interfacing with C code. For these purposes, numpy supports specialized features such as subarrays and nested datatypes, and allows manual control over the memory layout of the structure.

For simple manipulation of tabular data other pydata projects, such as pandas, xarray, or DataArray, provide higher-level interfaces that may be more suitable. These projects may also give better performance for tabular data analysis because the C-struct-like memory layout of structured arrays can lead to poor cache behavior.

3.7.2 Structured Datatypes

To use structured arrays one first needs to define a structured datatype.

A structured datatype can be thought of as a sequence of bytes of a certain length (the structure's itemsize) which is interpreted as a collection of fields. Each field has a name, a datatype, and a byte offset within the structure. The datatype of a field may be any numpy datatype including other structured datatypes, and it may also be a sub-array which behaves like an ndarray of a specified shape. The offsets of the fields are arbitrary, and fields may even overlap. These offsets are usually determined automatically by numpy, but can also be specified.

Structured Datatype Creation

Structured datatypes may be created using the function `numpy.dtype`. There are 4 alternative forms of specification which vary in flexibility and conciseness. These are further documented in the Data Type Objects reference page, and in summary they are:

1. A list of tuples, one tuple per field

   Each tuple has the form `(fieldname, datatype, shape)` where shape is optional. `fieldname` is a string (or tuple if titles are used, see *Field Titles* below), `datatype` may be any object convertible to a datatype, and `shape` is a tuple of integers specifying subarray shape.

```
>>> np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2,2))])
dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
```

   If `fieldname` is the empty string `''`, the field will be given a default name of the form `f#`, where `#` is the integer index of the field, counting from 0 from the left:

```
>>> np.dtype([('x', 'f4'),('', 'i4'),('z', 'i8')])
dtype([('x', '<f4'), ('f1', '<i4'), ('z', '<i8')])
```

   The byte offsets of the fields within the structure and the total structure itemsize are determined automatically.

2. A string of comma-separated dtype specifications

   In this shorthand notation any of the string dtype specifications may be used in a string and separated by commas. The itemsize and byte offsets of the fields are determined automatically, and the field names are given the default names `f0`, `f1`, etc.

```
>>> np.dtype('i8,f4,S3')
dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
>>> np.dtype('3int8, float32, (2,3)float64')
dtype([('f0', 'i1', 3), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

3. A dictionary of field parameter arrays

   This is the most flexible form of specification since it allows control over the byte-offsets of the fields and the itemsize of the structure.

   The dictionary has two required keys, 'names' and 'formats', and four optional keys, 'offsets', 'itemsize', 'aligned' and 'titles'. The values for 'names' and 'formats' should respectively be a list of field names and a list of dtype specifications, of the same length. The optional 'offsets' value should be a list of integer byte-offsets, one for each field within the structure. If 'offsets' is not given the offsets are determined automatically. The optional 'itemsize' value should be an integer describing the total size in bytes of the dtype, which must be large enough to contain all the fields.

```
>>> np.dtype({'names': ['col1', 'col2'], 'formats': ['i4','f4']})
dtype([('col1', '<i4'), ('col2', '<f4')])
>>> np.dtype({'names': ['col1', 'col2'],
...           'formats': ['i4','f4'],
...           'offsets': [0, 4],
...           'itemsize': 12})
dtype({'names':['col1','col2'], 'formats':['<i4','<f4'], 'offsets':[0,4],
↪'itemsize':12})
```

   Offsets may be chosen such that the fields overlap, though this will mean that assigning to one field may clobber any overlapping field's data. As an exception, fields of `numpy.object` type .. (see object arrays) cannot overlap with other fields, because of the risk of clobbering the internal object pointer and then dereferencing it.

   The optional 'aligned' value can be set to `True` to make the automatic offset computation use aligned offsets (see *Automatic Byte Offsets and Alignment*), as if the 'align' keyword argument of `numpy.dtype` had been set to True.

   The optional 'titles' value should be a list of titles of the same length as 'names', see *Field Titles* below.

4. A dictionary of field names

The use of this form of specification is discouraged, but documented here because older numpy code may use it. The keys of the dictionary are the field names and the values are tuples specifying type and offset:

```
>>> np.dtype={'col1': ('i1',0), 'col2': ('f4',1)})
dtype([(('col1'), 'i1'), (('col2'), '>f4')])
```

This form is discouraged because Python dictionaries do not preserve order in Python versions before Python 3.6, and the order of the fields in a structured dtype has meaning. *Field Titles* may be specified by using a 3-tuple, see below.

Manipulating and Displaying Structured Datatypes

The list of field names of a structured datatype can be found in the `names` attribute of the dtype object:

```
>>> d = np.dtype([('x', 'i8'), ('y', 'f4')])
>>> d.names
('x', 'y')
```

The field names may be modified by assigning to the `names` attribute using a sequence of strings of the same length.

The dtype object also has a dictionary-like attribute, `fields`, whose keys are the field names (and *Field Titles*, see below) and whose values are tuples containing the dtype and byte offset of each field.

```
>>> d.fields
mappingproxy({'x': (dtype('int64'), 0), 'y': (dtype('float32'), 8)})
```

Both the `names` and `fields` attributes will equal `None` for unstructured arrays.

The string representation of a structured datatype is shown in the "list of tuples" form if possible, otherwise numpy falls back to using the more general dictionary form.

Automatic Byte Offsets and Alignment

Numpy uses one of two methods to automatically determine the field byte offsets and the overall itemsize of a structured datatype, depending on whether `align=True` was specified as a keyword argument to `numpy.dtype`.

By default (`align=False`), numpy will pack the fields together such that each field starts at the byte offset the previous field ended, and the fields are contiguous in memory.

```
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] for name in d.names])
...     print("itemsize:", d.itemsize)
>>> print_offsets(np.dtype('u1,u1,i4,u1,i8,u2'))
offsets: [0, 1, 2, 6, 7, 15]
itemsize: 17
```

If `align=True` is set, numpy will pad the structure in the same way many C compilers would pad a C-struct. Aligned structures can give a performance improvement in some cases, at the cost of increased datatype size. Padding bytes are inserted between fields such that each field's byte offset will be a multiple of that field's alignment, which is usually equal to the field's size in bytes for simple datatypes, see `PyArray_Descr.alignment`. The structure will also have trailing padding added so that its itemsize is a multiple of the largest field's alignment.

```
>>> print_offsets(np.dtype('u1,u1,i4,u1,i8,u2', align=True))
offsets: [0, 1, 4, 8, 16, 24]
itemsize: 32
```

Note that although almost all modern C compilers pad in this way by default, padding in C structs is C-implementation-dependent so this memory layout is not guaranteed to exactly match that of a corresponding struct in a C program. Some work may be needed, either on the numpy side or the C side, to obtain exact correspondence.

If offsets were specified using the optional `offsets` key in the dictionary-based dtype specification, setting `align=True` will check that each field's offset is a multiple of its size and that the itemsize is a multiple of the largest field size, and raise an exception if not.

If the offsets of the fields and itemsize of a structured array satisfy the alignment conditions, the array will have the `ALIGNED` flag set.

Field Titles

In addition to field names, fields may also have an associated title, an alternate name, which is sometimes used as an additional description or alias for the field. The title may be used to index an array, just like a field name.

To add titles when using the list-of-tuples form of dtype specification, the field name may be be specified as a tuple of two strings instead of a single string, which will be the field's title and field name respectively. For example:

```
>>> np.dtype([(('my title', 'name'), 'f4')])
```

When using the first form of dictionary-based specification, the titles may be supplied as an extra `'titles'` key as described above. When using the second (discouraged) dictionary-based specification, the title can be supplied by providing a 3-element tuple `(datatype, offset, title)` instead of the usual 2-element tuple:

```
>>> np.dtype({'name': ('i4', 0, 'my title')})
```

The `dtype.fields` dictionary will contain titles as keys, if any titles are used. This means effectively that a field with a title will be represented twice in the fields dictionary. The tuple values for these fields will also have a third element, the field title. Because of this, and because the `names` attribute preserves the field order while the `fields` attribute may not, it is recommended to iterate through the fields of a dtype using the `names` attribute of the dtype, which will not list titles, as in:

```
>>> for name in d.names:
...     print(d.fields[name][:2])
```

Union types

Structured datatypes are implemented in numpy to have base type `numpy.void` by default, but it is possible to interpret other numpy types as structured types using the `(base_dtype, dtype)` form of dtype specification described in Data Type Objects. Here, `base_dtype` is the desired underlying dtype, and fields and flags will be copied from `dtype`. This dtype is similar to a 'union' in C.

3.7.3 Indexing and Assignment to Structured arrays

Assigning data to a Structured Array

There are a number of ways to assign values to a structured array: Using python tuples, using scalar values, or using other structured arrays.

Assignment from Python Native Types (Tuples)

The simplest way to assign values to a structured array is using python tuples. Each assigned value should be a tuple of length equal to the number of fields in the array, and not a list or array as these will trigger numpy's broadcasting rules. The tuple's elements are assigned to the successive fields of the array, from left to right:

```
>>> x = np.array([(1,2,3),(4,5,6)], dtype='i8,f4,f8')
>>> x[1] = (7,8,9)
>>> x
array([(1, 2., 3.), (7, 8., 9.)],
     dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '<f8')])
```

Assignment from Scalars

A scalar assigned to a structured element will be assigned to all fields. This happens when a scalar is assigned to a structured array, or when an unstructured array is assigned to a structured array:

```
>>> x = np.zeros(2, dtype='i8,f4,?,S1')
>>> x[:] = 3
>>> x
array([(3, 3.0, True, b'3'), (3, 3.0, True, b'3')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
>>> x[:] = np.arange(2)
>>> x
array([(0, 0.0, False, b'0'), (1, 1.0, True, b'1')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
```

Structured arrays can also be assigned to unstructured arrays, but only if the structured datatype has just a single field:

```
>>> twofield = np.zeros(2, dtype=[('A', 'i4'), ('B', 'i4')])
>>> onefield = np.zeros(2, dtype=[('A', 'i4')])
>>> nostruct = np.zeros(2, dtype='i4')
>>> nostruct[:] = twofield
ValueError: Can't cast from structure to non-structure, except if the structure only␣
↪has a single field.
>>> nostruct[:] = onefield
>>> nostruct
array([0, 0], dtype=int32)
```

Assignment from other Structured Arrays

Assignment between two structured arrays occurs as if the source elements had been converted to tuples and then assigned to the destination elements. That is, the first field of the source array is assigned to the first field of the destination array, and the second field likewise, and so on, regardless of field names. Structured arrays with a different number of fields cannot be assigned to each other. Bytes of the destination structure which are not included in any of the fields are unaffected.

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'f4'), ('c', 'S3')])
>>> b = np.ones(3, dtype=[('x', 'f4'), ('y', 'S3'), ('z', 'O')])
>>> b[:] = a
>>> b
array([(0.0, b'0.0', b''), (0.0, b'0.0', b''), (0.0, b'0.0', b'')],
      dtype=[('x', '<f4'), ('y', 'S3'), ('z', 'O')])
```

Assignment involving subarrays

When assigning to fields which are subarrays, the assigned value will first be broadcast to the shape of the subarray.

Indexing Structured Arrays

Accessing Individual Fields

Individual fields of a structured array may be accessed and modified by indexing the array with the field name.

```
>>> x = np.array([(1,2),(3,4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> x['foo']
array([1, 3])
>>> x['foo'] = 10
>>> x
array([(10, 2.), (10, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

The resulting array is a view into the original array. It shares the same memory locations and writing to the view will modify the original array.

```
>>> y = x['bar']
>>> y[:] = 10
>>> x
array([(10, 5.), (10, 5.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

This view has the same dtype and itemsize as the indexed field, so it is typically a non-structured array, except in the case of nested structures.

```
>>> y.dtype, y.shape, y.strides
(dtype('float32'), (2,), (12,))
```

Accessing Multiple Fields

One can index a structured array with a multi-field index, where the index is a list of field names:

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'i4'), ('c', 'f8')])
>>> a[['a', 'c']]
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype={'names':['a','c'], 'formats':['<i8','<f8'], 'offsets':[0,11], 'itemsize
↪':19})
>>> a[['a', 'c']] = (2, 3)
>>> a
array([(2, 0, 3.0), (2, 0, 3.0), (2, 0, 3.0)],
      dtype=[('a', '<i8'), ('b', '<i4'), ('c', '<f8')])
```

The resulting array is a view into the original array, such that assignment to the view modifies the original array. The view's fields will be in the order they were indexed. Note that unlike for single-field indexing, the view's dtype has the same itemsize as the original array, and has fields at the same offsets as in the original array, and unindexed fields are merely missing.

Since the view is a structured array itself, it obeys the assignment rules described above. For example, this means that one can swap the values of two fields using appropriate multi-field indexes:

```
>>> a[['a', 'c']] = a[['c', 'a']]
```

Indexing with an Integer to get a Structured Scalar

Indexing a single element of a structured array (with an integer index) returns a structured scalar:

```
>>> x = np.array([(1, 2., 3.)], dtype='i,f,f')
>>> scalar = x[0]
>>> scalar
(1, 2., 3.)
>>> type(scalar)
numpy.void
```

Unlike other numpy scalars, structured scalars are mutable and act like views into the original array, such that modifying the scalar will modify the original array. Structured scalars also support access and assignment by field name:

```
>>> x = np.array([(1,2),(3,4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> s = x[0]
>>> s['bar'] = 100
>>> x
array([(1, 100.), (3, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Similarly to tuples, structured scalars can also be indexed with an integer:

```
>>> scalar = np.array([(1, 2., 3.)], dtype='i,f,f')[0]
>>> scalar[0]
1
>>> scalar[1] = 4
```

Thus, tuples might be thought of as the native Python equivalent to numpy's structured types, much like native python integers are the equivalent to numpy's integer types. Structured scalars may be converted to a tuple by calling `ndarray.item`:

```
>>> scalar.item(), type(scalar.item())
((1, 2.0, 3.0), tuple)
```

Viewing Structured Arrays Containing Objects

In order to prevent clobbering object pointers in fields of `numpy.object` type, numpy currently does not allow views of structured arrays containing objects.

Structure Comparison

If the dtypes of two void structured arrays are equal, testing the equality of the arrays will result in a boolean array with the dimensions of the original arrays, with elements set to `True` where all fields of the corresponding structures are equal. Structured dtypes are equal if the field names, dtypes and titles are the same, ignoring endianness, and the fields are in the same order:

```
>>> a = np.zeros(2, dtype=[('a', 'i4'), ('b', 'i4')])
>>> b = np.ones(2, dtype=[('a', 'i4'), ('b', 'i4')])
>>> a == b
array([False, False])
```

Currently, if the dtypes of two void structured arrays are not equivalent the comparison fails, returning the scalar value `False`. This behavior is deprecated as of numpy 1.10 and will raise an error or perform elementwise comparison in the future.

The < and > operators always return `False` when comparing void structured arrays, and arithmetic and bitwise operations are not supported.

3.7.4 Record Arrays

As an optional convenience numpy provides an ndarray subclass, `numpy.recarray`, and associated helper functions in the `numpy.rec` submodule, that allows access to fields of structured arrays by attribute instead of only by index. Record arrays also use a special datatype, `numpy.record`, that allows field access by attribute on the structured scalars obtained from the array.

The simplest way to create a record array is with `numpy.rec.array`:

```
>>> recordarr = np.rec.array([(1,2.,'Hello'),(2,3.,"World")],
...                    dtype=[('foo', 'i4'),('bar', 'f4'), ('baz', 'S10')])
>>> recordarr.bar
array([ 2.,  3.], dtype=float32)
>>> recordarr[1:2]
rec.array([(2, 3.0, 'World')],
      dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])
>>> recordarr[1:2].foo
array([2], dtype=int32)
>>> recordarr.foo[1:2]
array([2], dtype=int32)
>>> recordarr[1].baz
'World'
```

`numpy.rec.array` can convert a wide variety of arguments into record arrays, including structured arrays:

```
>>> arr = array([(1,2.,'Hello'),(2,3.,"World")],
...             dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])
>>> recordarr = np.rec.array(arr)
```

The `numpy.rec` module provides a number of other convenience functions for creating record arrays, see record array creation routines.

A record array representation of a structured array can be obtained using the appropriate view:

```
>>> arr = np.array([(1,2.,'Hello'),(2,3.,"World")],
...                dtype=[('foo', 'i4'),('bar', 'f4'), ('baz', 'a10')])
>>> recordarr = arr.view(dtype=dtype((np.record, arr.dtype)),
...                      type=np.recarray)
```

For convenience, viewing an ndarray as type `np.recarray` will automatically convert to `np.record` datatype, so the dtype can be left out of the view:

```
>>> recordarr = arr.view(np.recarray)
>>> recordarr.dtype
dtype((numpy.record, [('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')]))
```

To get back to a plain ndarray both the dtype and type must be reset. The following view does so, taking into account the unusual case that the recordarr was not a structured type:

```
>>> arr2 = recordarr.view(recordarr.dtype.fields or recordarr.dtype, np.ndarray)
```

Record array fields accessed by index or by attribute are returned as a record array if the field has a structured type but as a plain ndarray otherwise.

```
>>> recordarr = np.rec.array([('Hello', (1,2)),("World", (3,4))],
...                 dtype=[('foo', 'S6'),('bar', [('A', int), ('B', int)])])
>>> type(recordarr.foo)
<type 'numpy.ndarray'>
```

```
>>> type(recordarr.bar)
<class 'numpy.core.records.recarray'>
```

Note that if a field has the same name as an ndarray attribute, the ndarray attribute takes precedence. Such fields will be inaccessible by attribute but will still be accessible by index.

3.8 Subclassing ndarray

3.8.1 Introduction

Subclassing ndarray is relatively simple, but it has some complications compared to other Python objects. On this page we explain the machinery that allows you to subclass ndarray, and the implications for implementing a subclass.

ndarrays and object creation

Subclassing ndarray is complicated by the fact that new instances of ndarray classes can come about in three different ways. These are:

1. Explicit constructor call - as in `MySubClass(params)`. This is the usual route to Python instance creation.
2. View casting - casting an existing ndarray as a given subclass
3. New from template - creating a new instance from a template instance. Examples include returning slices from a subclassed array, creating return types from ufuncs, and copying arrays. See *Creating new from template* for more details

The last two are characteristics of ndarrays - in order to support things like array slicing. The complications of subclassing ndarray are due to the mechanisms numpy has to support these latter two routes of instance creation.

3.8.2 View casting

View casting is the standard ndarray mechanism by which you take an ndarray of any subclass, and return a view of the array as another (specified) subclass:

```
>>> import numpy as np
>>> # create a completely useless ndarray subclass
>>> class C(np.ndarray): pass
>>> # create a standard ndarray
>>> arr = np.zeros((3,))
>>> # take a view of it, as our useless subclass
>>> c_arr = arr.view(C)
>>> type(c_arr)
<class 'C'>
```

3.8.3 Creating new from template

New instances of an ndarray subclass can also come about by a very similar mechanism to *View casting*, when numpy finds it needs to create a new instance from a template instance. The most obvious place this has to happen is when you are taking slices of subclassed arrays. For example:

```
>>> v = c_arr[1:]
>>> type(v) # the view is of type 'C'
<class 'C'>
>>> v is c_arr # but it's a new instance
False
```

The slice is a *view* onto the original `c_arr` data. So, when we take a view from the ndarray, we return a new ndarray, of the same class, that points to the data in the original.

There are other points in the use of ndarrays where we need such views, such as copying arrays (`c_arr.copy()`), creating ufunc output arrays (see also *__array_wrap__ for ufuncs and other functions*), and reducing methods (like `c_arr.mean()`.

3.8.4 Relationship of view casting and new-from-template

These paths both use the same machinery. We make the distinction here, because they result in different input to your methods. Specifically, *View casting* means you have created a new instance of your array type from any potential subclass of ndarray. *Creating new from template* means you have created a new instance of your class from a pre-existing instance, allowing you - for example - to copy across attributes that are particular to your subclass.

3.8.5 Implications for subclassing

If we subclass ndarray, we need to deal not only with explicit construction of our array type, but also *View casting* or *Creating new from template*. NumPy has the machinery to do this, and this machinery that makes subclassing slightly non-standard.

There are two aspects to the machinery that ndarray uses to support views and new-from-template in subclasses.

The first is the use of the `ndarray.__new__` method for the main work of object initialization, rather then the more usual `__init__` method. The second is the use of the `__array_finalize__` method to allow subclasses to clean up after the creation of views and new instances from templates.

A brief Python primer on `__new__` and `__init__`

`__new__` is a standard Python method, and, if present, is called before `__init__` when we create a class instance. See the python `__new__` documentation for more detail.

For example, consider the following Python code:

```
class C(object):
    def __new__(cls, *args):
        print('Cls in __new__:', cls)
        print('Args in __new__:', args)
        return object.__new__(cls, *args)

    def __init__(self, *args):
        print('type(self) in __init__:', type(self))
        print('Args in __init__:', args)
```

meaning that we get:

```
>>> c = C('hello')
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
```

```
type(self) in __init__: <class 'C'>
Args in __init__: ('hello',)
```

When we call `C('hello')`, the `__new__` method gets its own class as first argument, and the passed argument, which is the string `'hello'`. After python calls `__new__`, it usually (see below) calls our `__init__` method, with the output of `__new__` as the first argument (now a class instance), and the passed arguments following.

As you can see, the object can be initialized in the `__new__` method or the `__init__` method, or both, and in fact ndarray does not have an `__init__` method, because all the initialization is done in the `__new__` method.

Why use `__new__` rather than just the usual `__init__`? Because in some cases, as for ndarray, we want to be able to return an object of some other class. Consider the following:

```
class D(C):
    def __new__(cls, *args):
        print('D cls is:', cls)
        print('D args in __new__:', args)
        return C.__new__(C, *args)

    def __init__(self, *args):
        # we never get here
        print('In D __init__')
```

meaning that:

```
>>> obj = D('hello')
D cls is: <class 'D'>
D args in __new__: ('hello',)
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
>>> type(obj)
<class 'C'>
```

The definition of `C` is the same as before, but for `D`, the `__new__` method returns an instance of class `C` rather than `D`. Note that the `__init__` method of `D` does not get called. In general, when the `__new__` method returns an object of class other than the class in which it is defined, the `__init__` method of that class is not called.

This is how subclasses of the ndarray class are able to return views that preserve the class type. When taking a view, the standard ndarray machinery creates the new ndarray object with something like:

```
obj = ndarray.__new__(subtype, shape, ...
```

where `subdtype` is the subclass. Thus the returned view is of the same class as the subclass, rather than being of class `ndarray`.

That solves the problem of returning views of the same type, but now we have a new problem. The machinery of ndarray can set the class this way, in its standard methods for taking views, but the ndarray `__new__` method knows nothing of what we have done in our own `__new__` method in order to set attributes, and so on. (Aside - why not call `obj = subdtype.__new__(...` then? Because we may not have a `__new__` method with the same call signature).

The role of `__array_finalize__`

`__array_finalize__` is the mechanism that numpy provides to allow subclasses to handle the various ways that new instances get created.

Remember that subclass instances can come about in these three ways:

1. explicit constructor call (`obj = MySubClass(params)`). This will call the usual sequence of `MySubClass.__new__` then (if it exists) `MySubClass.__init__`.
2. *View casting*
3. *Creating new from template*

Our `MySubClass.__new__` method only gets called in the case of the explicit constructor call, so we can't rely on `MySubClass.__new__` or `MySubClass.__init__` to deal with the view casting and new-from-template. It turns out that `MySubClass.__array_finalize__` *does* get called for all three methods of object creation, so this is where our object creation housekeeping usually goes.

- For the explicit constructor call, our subclass will need to create a new ndarray instance of its own class. In practice this means that we, the authors of the code, will need to make a call to `ndarray.__new__(MySubClass,...)`, a class-hierarchy prepared call to `super(MySubClass, cls).__new__(cls, ...)`, or do view casting of an existing array (see below)
- For view casting and new-from-template, the equivalent of `ndarray.__new__(MySubClass,...` is called, at the C level.

The arguments that `__array_finalize__` receives differ for the three methods of instance creation above.

The following code allows us to look at the call sequences and arguments:

```
import numpy as np

class C(np.ndarray):
    def __new__(cls, *args, **kwargs):
        print('In __new__ with class %s' % cls)
        return super(C, cls).__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        # in practice you probably will not need or want an __init__
        # method for your subclass
        print('In __init__ with class %s' % self.__class__)

    def __array_finalize__(self, obj):
        print('In array_finalize:')
        print('   self type is %s' % type(self))
        print('   obj type is %s' % type(obj))
```

Now:

```
>>> # Explicit constructor
>>> c = C((10,))
In __new__ with class <class 'C'>
In array_finalize:
   self type is <class 'C'>
   obj type is <type 'NoneType'>
In __init__ with class <class 'C'>
>>> # View casting
>>> a = np.arange(10)
>>> cast_a = a.view(C)
In array_finalize:
   self type is <class 'C'>
   obj type is <type 'numpy.ndarray'>
>>> # Slicing (example of new-from-template)
>>> cv = c[:1]
In array_finalize:
```

```
   self type is <class 'C'>
   obj type is <class 'C'>
```

The signature of `__array_finalize__` is:

```
def __array_finalize__(self, obj):
```

One sees that the `super` call, which goes to `ndarray.__new__`, passes `__array_finalize__` the new object, of our own class (`self`) as well as the object from which the view has been taken (`obj`). As you can see from the output above, the `self` is always a newly created instance of our subclass, and the type of `obj` differs for the three instance creation methods:

- When called from the explicit constructor, `obj` is `None`
- When called from view casting, `obj` can be an instance of any subclass of ndarray, including our own.
- When called in new-from-template, `obj` is another instance of our own subclass, that we might use to update the new `self` instance.

Because `__array_finalize__` is the only method that always sees new instances being created, it is the sensible place to fill in instance defaults for new object attributes, among other tasks.

This may be clearer with an example.

3.8.6 Simple example - adding an extra attribute to ndarray

```
import numpy as np

class InfoArray(np.ndarray):

    def __new__(subtype, shape, dtype=float, buffer=None, offset=0,
                strides=None, order=None, info=None):
        # Create the ndarray instance of our type, given the usual
        # ndarray input arguments.  This will call the standard
        # ndarray constructor, but return an object of our type.
        # It also triggers a call to InfoArray.__array_finalize__
        obj = super(InfoArray, subtype).__new__(subtype, shape, dtype,
                                                buffer, offset, strides,
                                                order)
        # set the new 'info' attribute to the value passed
        obj.info = info
        # Finally, we must return the newly created object:
        return obj

    def __array_finalize__(self, obj):
        # ``self`` is a new object resulting from
        # ndarray.__new__(InfoArray, ...), therefore it only has
        # attributes that the ndarray.__new__ constructor gave it -
        # i.e. those of a standard ndarray.
        #
        # We could have got to the ndarray.__new__ call in 3 ways:
        # From an explicit constructor - e.g. InfoArray():
        #    obj is None
        #    (we're in the middle of the InfoArray.__new__
        #    constructor, and self.info will be set when we return to
        #    InfoArray.__new__)
        if obj is None: return
        # From view casting - e.g arr.view(InfoArray):
```

```
        #    obj is arr
        #    (type(obj) can be InfoArray)
        # From new-from-template - e.g infoarr[:3]
        #    type(obj) is InfoArray
        #
        # Note that it is here, rather than in the __new__ method,
        # that we set the default value for 'info', because this
        # method sees all creation of default objects - with the
        # InfoArray.__new__ constructor, but also with
        # arr.view(InfoArray).
        self.info = getattr(obj, 'info', None)
        # We do not need to return anything
```

Using the object looks like this:

```
>>> obj = InfoArray(shape=(3,)) # explicit constructor
>>> type(obj)
<class 'InfoArray'>
>>> obj.info is None
True
>>> obj = InfoArray(shape=(3,), info='information')
>>> obj.info
'information'
>>> v = obj[1:] # new-from-template - here - slicing
>>> type(v)
<class 'InfoArray'>
>>> v.info
'information'
>>> arr = np.arange(10)
>>> cast_arr = arr.view(InfoArray) # view casting
>>> type(cast_arr)
<class 'InfoArray'>
>>> cast_arr.info is None
True
```

This class isn't very useful, because it has the same constructor as the bare ndarray object, including passing in buffers and shapes and so on. We would probably prefer the constructor to be able to take an already formed ndarray from the usual numpy calls to `np.array` and return an object.

3.8.7 Slightly more realistic example - attribute added to existing array

Here is a class that takes a standard ndarray that already exists, casts as our type, and adds an extra attribute.

```
import numpy as np

class RealisticInfoArray(np.ndarray):

    def __new__(cls, input_array, info=None):
        # Input array is an already formed ndarray instance
        # We first cast to be our class type
        obj = np.asarray(input_array).view(cls)
        # add the new attribute to the created instance
        obj.info = info
        # Finally, we must return the newly created object:
        return obj
```

```
    def __array_finalize__(self, obj):
        # see InfoArray.__array_finalize__ for comments
        if obj is None: return
        self.info = getattr(obj, 'info', None)
```

So:

```
>>> arr = np.arange(5)
>>> obj = RealisticInfoArray(arr, info='information')
>>> type(obj)
<class 'RealisticInfoArray'>
>>> obj.info
'information'
>>> v = obj[1:]
>>> type(v)
<class 'RealisticInfoArray'>
>>> v.info
'information'
```

3.8.8 `__array_ufunc__` for ufuncs

New in version 1.13.

A subclass can override what happens when executing numpy ufuncs on it by overriding the default `ndarray.__array_ufunc__` method. This method is executed *instead* of the ufunc and should return either the result of the operation, or `NotImplemented` if the operation requested is not implemented.

The signature of `__array_ufunc__` is:

```
def __array_ufunc__(ufunc, method, *inputs, **kwargs):

- *ufunc* is the ufunc object that was called.
- *method* is a string indicating how the Ufunc was called, either
  ``"__call__"`` to indicate it was called directly, or one of its
  :ref:`methods<ufuncs.methods>`: ``"reduce"``, ``"accumulate"``,
  ``"reduceat"``, ``"outer"``, or ``"at"``.
- *inputs* is a tuple of the input arguments to the ``ufunc``
- *kwargs* contains any optional or keyword arguments passed to the
  function. This includes any ``out`` arguments, which are always
  contained in a tuple.
```

A typical implementation would convert any inputs or ouputs that are instances of one's own class, pass everything on to a superclass using `super()`, and finally return the results after possible back-conversion. An example, taken from the test case `test_ufunc_override_with_super` in `core/tests/test_umath.py`, is the following.

```
input numpy as np

class A(np.ndarray):
    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
        args = []
        in_no = []
        for i, input_ in enumerate(inputs):
            if isinstance(input_, A):
                in_no.append(i)
                args.append(input_.view(np.ndarray))
            else:
                args.append(input_)
```

```
        outputs = kwargs.pop('out', None)
        out_no = []
        if outputs:
            out_args = []
            for j, output in enumerate(outputs):
                if isinstance(output, A):
                    out_no.append(j)
                    out_args.append(output.view(np.ndarray))
                else:
                    out_args.append(output)
            kwargs['out'] = tuple(out_args)
        else:
            outputs = (None,) * ufunc.nout

        info = {}
        if in_no:
            info['inputs'] = in_no
        if out_no:
            info['outputs'] = out_no

        results = super(A, self).__array_ufunc__(ufunc, method,
                                                 *args, **kwargs)
        if results is NotImplemented:
            return NotImplemented

        if method == 'at':
            if isinstance(inputs[0], A):
                inputs[0].info = info
            return

        if ufunc.nout == 1:
            results = (results,)

        results = tuple((np.asarray(result).view(A)
                         if output is None else output)
                        for result, output in zip(results, outputs))
        if results and isinstance(results[0], A):
            results[0].info = info

        return results[0] if len(results) == 1 else results
```

So, this class does not actually do anything interesting: it just converts any instances of its own to regular ndarray (otherwise, we'd get infinite recursion!), and adds an `info` dictionary that tells which inputs and outputs it converted. Hence, e.g.,

```
>>> a = np.arange(5.).view(A)
>>> b = np.sin(a)
>>> b.info
{'inputs': [0]}
>>> b = np.sin(np.arange(5.), out=(a,))
>>> b.info
{'outputs': [0]}
>>> a = np.arange(5.).view(A)
>>> b = np.ones(1).view(A)
>>> c = a + b
>>> c.info
{'inputs': [0, 1]}
```

```
>>> a += b
>>> a.info
{'inputs': [0, 1], 'outputs': [0]}
```

Note that another approach would be to to use `getattr(ufunc, methods)(*inputs, **kwargs)` instead of the `super` call. For this example, the result would be identical, but there is a difference if another operand also defines `__array_ufunc__`. E.g., lets assume that we evalulate `np.add(a, b)`, where `b` is an instance of another class `B` that has an override. If you use `super` as in the example, `ndarray.__array_ufunc__` will notice that `b` has an override, which means it cannot evaluate the result itself. Thus, it will return *NotImplemented* and so will our class `A`. Then, control will be passed over to `b`, which either knows how to deal with us and produces a result, or does not and returns *NotImplemented*, raising a `TypeError`.

If instead, we replace our `super` call with `getattr(ufunc, method)`, we effectively do `np.add(a.view(np.ndarray), b)`. Again, `B.__array_ufunc__` will be called, but now it sees an `ndarray` as the other argument. Likely, it will know how to handle this, and return a new instance of the `B` class to us. Our example class is not set up to handle this, but it might well be the best approach if, e.g., one were to re-implement `MaskedArray` using `__array_ufunc__`.

As a final note: if the `super` route is suited to a given class, an advantage of using it is that it helps in constructing class hierarchies. E.g., suppose that our other class `B` also used the `super` in its `__array_ufunc__` implementation, and we created a class `C` that depended on both, i.e., `class C(A, B)` (with, for simplicity, not another `__array_ufunc__` override). Then any ufunc on an instance of `C` would pass on to `A.__array_ufunc__`, the `super` call in `A` would go to `B.__array_ufunc__`, and the `super` call in `B` would go to `ndarray.__array_ufunc__`, thus allowing `A` and `B` to collaborate.

3.8.9 `__array_wrap__` for ufuncs and other functions

Prior to numpy 1.13, the behaviour of ufuncs could only be tuned using `__array_wrap__` and `__array_prepare__`. These two allowed one to change the output type of a ufunc, but, in constrast to `__array_ufunc__`, did not allow one to make any changes to the inputs. It is hoped to eventually deprecate these, but `__array_wrap__` is also used by other numpy functions and methods, such as `squeeze`, so at the present time is still needed for full functionality.

Conceptually, `__array_wrap__` "wraps up the action" in the sense of allowing a subclass to set the type of the return value and update attributes and metadata. Let's show how this works with an example. First we return to the simpler example subclass, but with a different name and some print statements:

```
import numpy as np

class MySubClass(np.ndarray):

    def __new__(cls, input_array, info=None):
        obj = np.asarray(input_array).view(cls)
        obj.info = info
        return obj

    def __array_finalize__(self, obj):
        print('In __array_finalize__:')
        print('   self is %s' % repr(self))
        print('   obj is %s' % repr(obj))
        if obj is None: return
        self.info = getattr(obj, 'info', None)

    def __array_wrap__(self, out_arr, context=None):
        print('In __array_wrap__:')
        print('   self is %s' % repr(self))
```

```
        print('   arr is %s' % repr(out_arr))
        # then just call the parent
        return super(MySubClass, self).__array_wrap__(self, out_arr, context)
```

We run a ufunc on an instance of our new array:

```
>>> obj = MySubClass(np.arange(5), info='spam')
In __array_finalize__:
   self is MySubClass([0, 1, 2, 3, 4])
   obj is array([0, 1, 2, 3, 4])
>>> arr2 = np.arange(5)+1
>>> ret = np.add(arr2, obj)
In __array_wrap__:
   self is MySubClass([0, 1, 2, 3, 4])
   arr is array([1, 3, 5, 7, 9])
In __array_finalize__:
   self is MySubClass([1, 3, 5, 7, 9])
   obj is MySubClass([0, 1, 2, 3, 4])
>>> ret
MySubClass([1, 3, 5, 7, 9])
>>> ret.info
'spam'
```

Note that the ufunc (`np.add`) has called the `__array_wrap__` method with arguments `self` as `obj`, and `out_arr` as the (ndarray) result of the addition. In turn, the default `__array_wrap__` (`ndarray.__array_wrap__`) has cast the result to class `MySubClass`, and called `__array_finalize__` - hence the copying of the `info` attribute. This has all happened at the C level.

But, we could do anything we wanted:

```
class SillySubClass(np.ndarray):

    def __array_wrap__(self, arr, context=None):
        return 'I lost your data'
```

```
>>> arr1 = np.arange(5)
>>> obj = arr1.view(SillySubClass)
>>> arr2 = np.arange(5)
>>> ret = np.multiply(obj, arr2)
>>> ret
'I lost your data'
```

So, by defining a specific `__array_wrap__` method for our subclass, we can tweak the output from ufuncs. The `__array_wrap__` method requires `self`, then an argument - which is the result of the ufunc - and an optional parameter *context*. This parameter is returned by ufuncs as a 3-element tuple: (name of the ufunc, arguments of the ufunc, domain of the ufunc), but is not set by other numpy functions. Though, as seen above, it is possible to do otherwise, `__array_wrap__` should return an instance of its containing class. See the masked array subclass for an implementation.

In addition to `__array_wrap__`, which is called on the way out of the ufunc, there is also an `__array_prepare__` method which is called on the way into the ufunc, after the output arrays are created but before any computation has been performed. The default implementation does nothing but pass through the array. `__array_prepare__` should not attempt to access the array data or resize the array, it is intended for setting the output array type, updating attributes and metadata, and performing any checks based on the input that may be desired before computation begins. Like `__array_wrap__`, `__array_prepare__` must return an ndarray or subclass thereof or raise an error.

3.8.10 Extra gotchas - custom `__del__` methods and ndarray.base

One of the problems that ndarray solves is keeping track of memory ownership of ndarrays and their views. Consider the case where we have created an ndarray, `arr` and have taken a slice with `v = arr[1:]`. The two objects are looking at the same memory. NumPy keeps track of where the data came from for a particular array or view, with the `base` attribute:

```
>>> # A normal ndarray, that owns its own data
>>> arr = np.zeros((4,))
>>> # In this case, base is None
>>> arr.base is None
True
>>> # We take a view
>>> v1 = arr[1:]
>>> # base now points to the array that it derived from
>>> v1.base is arr
True
>>> # Take a view of a view
>>> v2 = v1[1:]
>>> # base points to the view it derived from
>>> v2.base is v1
True
```

In general, if the array owns its own memory, as for `arr` in this case, then `arr.base` will be None - there are some exceptions to this - see the numpy book for more details.

The `base` attribute is useful in being able to tell whether we have a view or the original array. This in turn can be useful if we need to know whether or not to do some specific cleanup when the subclassed array is deleted. For example, we may only want to do the cleanup if the original array is deleted, but not the views. For an example of how this can work, have a look at the `memmap` class in `numpy.core`.

3.8.11 Subclassing and Downstream Compatibility

When sub-classing `ndarray` or creating duck-types that mimic the `ndarray` interface, it is your responsibility to decide how aligned your APIs will be with those of numpy. For convenience, many numpy functions that have a corresponding `ndarray` method (e.g., `sum`, `mean`, `take`, `reshape`) work by checking if the first argument to a function has a method of the same name. If it exists, the method is called instead of coercing the arguments to a numpy array.

For example, if you want your sub-class or duck-type to be compatible with numpy's `sum` function, the method signature for this object's `sum` method should be the following:

```
def sum(self, axis=None, dtype=None, out=None, keepdims=False):
...
```

This is the exact same method signature for `np.sum`, so now if a user calls `np.sum` on this object, numpy will call the object's own `sum` method and pass in these arguments enumerated above in the signature, and no errors will be raised because the signatures are completely compatible with each other.

If, however, you decide to deviate from this signature and do something like this:

```
def sum(self, axis=None, dtype=None):
...
```

This object is no longer compatible with `np.sum` because if you call `np.sum`, it will pass in unexpected arguments `out` and `keepdims`, causing a TypeError to be raised.

If you wish to maintain compatibility with numpy and its subsequent versions (which might add new keyword arguments) but do not want to surface all of numpy's arguments, your function's signature should accept `**kwargs`. For example:

```
def sum(self, axis=None, dtype=None, **unused_kwargs):
...
```

This object is now compatible with `np.sum` again because any extraneous arguments (i.e. keywords that are not `axis` or `dtype`) will be hidden away in the `**unused_kwargs` parameter.

CHAPTER

FOUR

MISCELLANEOUS

4.1 IEEE 754 Floating Point Special Values

Special values defined in numpy: nan, inf,

NaNs can be used as a poor-man's mask (if you don't care what the original value was)

Note: cannot use equality to test NaNs. E.g.:

```
>>> myarr = np.array([1., 0., np.nan, 3.])
>>> np.nonzero(myarr == np.nan)
(array([], dtype=int64),)
>>> np.nan == np.nan  # is always False! Use special numpy functions instead.
False
>>> myarr[myarr == np.nan] = 0. # doesn't work
>>> myarr
array([  1.,   0.,  NaN,   3.])
>>> myarr[np.isnan(myarr)] = 0. # use this instead find
>>> myarr
array([ 1.,  0.,  0.,  3.])
```

Other related special value functions:

```
isinf():    True if value is inf
isfinite(): True if not nan or inf
nan_to_num(): Map nan to 0, inf to max float, -inf to min float
```

The following corresponds to the usual functions except that nans are excluded from the results:

```
nansum()
nanmax()
nanmin()
nanargmax()
nanargmin()

>>> x = np.arange(10.)
>>> x[3] = np.nan
>>> x.sum()
nan
>>> np.nansum(x)
42.0
```

4.2 How numpy handles numerical exceptions

The default is to `'warn'` for `invalid`, `divide`, and `overflow` and `'ignore'` for `underflow`. But this can be changed, and it can be set individually for different kinds of exceptions. The different behaviors are:

- '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*.

These behaviors can be set for all kinds of errors or specific ones:

- all : apply to all numeric exceptions
- invalid : when NaNs are generated
- divide : divide by zero (for integers as well!)
- overflow : floating point overflows
- underflow : floating point underflows

Note that integer divide-by-zero is handled by the same machinery. These behaviors are set on a per-thread basis.

4.3 Examples

```
>>> oldsettings = np.seterr(all='warn')
>>> np.zeros(5,dtype=np.float32)/0.
invalid value encountered in divide
>>> j = np.seterr(under='ignore')
>>> np.array([1.e-100])**10
>>> j = np.seterr(invalid='raise')
>>> np.sqrt(np.array([-1.]))
FloatingPointError: invalid value encountered in sqrt
>>> def errorhandler(errstr, errflag):
...      print("saw stupid error!")
>>> np.seterrcall(errorhandler)
<function err_handler at 0x...>
>>> j = np.seterr(all='call')
>>> np.zeros(5, dtype=np.int32)/0
FloatingPointError: invalid value encountered in divide
saw stupid error!
>>> j = np.seterr(**oldsettings) # restore previous
...                              # error-handling settings
```

4.4 Interfacing to C

Only a survey of the choices. Little detail on how each works.

1. Bare metal, wrap your own C-code manually.

- Plusses:
  - Efficient
  - No dependencies on other tools
- Minuses:
  - Lots of learning overhead:
    - need to learn basics of Python C API
    - need to learn basics of numpy C API
    - need to learn how to handle reference counting and love it.
  - Reference counting often difficult to get right.
    - getting it wrong leads to memory leaks, and worse, segfaults
  - API will change for Python 3.0!

2. Cython

- Plusses:
  - avoid learning C API's
  - no dealing with reference counting
  - can code in pseudo python and generate C code
  - can also interface to existing C code
  - should shield you from changes to Python C api
  - has become the de-facto standard within the scientific Python community
  - fast indexing support for arrays
- Minuses:
  - Can write code in non-standard form which may become obsolete
  - Not as flexible as manual wrapping

3. ctypes

- Plusses:
  - part of Python standard library
  - good for interfacing to existing sharable libraries, particularly Windows DLLs
  - avoids API/reference counting issues
  - good numpy support: arrays have all these in their ctypes attribute:

```
a.ctypes.data              a.ctypes.get_strides
a.ctypes.data_as           a.ctypes.shape
a.ctypes.get_as_parameter  a.ctypes.shape_as
a.ctypes.get_data          a.ctypes.strides
a.ctypes.get_shape         a.ctypes.strides_as
```

- Minuses:
  - can't use for writing code to be turned into C extensions, only a wrapper tool.

4. SWIG (automatic wrapper generator)

- Plusses:
  - around a long time
  - multiple scripting language support
  - C++ support
  - Good for wrapping large (many functions) existing C libraries
- Minuses:
  - generates lots of code between Python and the C code
  - can cause performance problems that are nearly impossible to optimize out
  - interface files can be hard to write
  - doesn't necessarily avoid reference counting issues or needing to know API's

5. scipy.weave

- Plusses:
  - can turn many numpy expressions into C code
  - dynamic compiling and loading of generated C code
  - can embed pure C code in Python module and have weave extract, generate interfaces and compile, etc.
- Minuses:
  - Future very uncertain: it's the only part of Scipy not ported to Python 3 and is effectively deprecated in favor of Cython.

6. Psyco

- Plusses:
  - Turns pure python into efficient machine code through jit-like optimizations
  - very fast when it optimizes well
- Minuses:
  - Only on intel (windows?)
  - Doesn't do much for numpy?

4.5 Interfacing to Fortran:

The clear choice to wrap Fortran code is f2py.

Pyfort is an older alternative, but not supported any longer. Fwrap is a newer project that looked promising but isn't being developed any longer.

4.6 Interfacing to C++:

1. Cython
2. CXX
3. Boost.python

4. SWIG
5. SIP (used mainly in PyQT)

CHAPTER

FIVE

NUMPY FOR MATLAB USERS

5.1 Introduction

MATLAB® and NumPy/SciPy have a lot in common. But there are many differences. NumPy and SciPy were created to do numerical and scientific computing in the most natural way with Python, not to be MATLAB® clones. This page is intended to be a place to collect wisdom about the differences, mostly for the purpose of helping proficient MATLAB® users become proficient NumPy and SciPy users.

5.2 Some Key Differences

In MATLAB®, the basic data type is a multidimensional array of double precision floating point numbers. Most expressions take such arrays and return such arrays. Operations on the 2-D instances of these arrays are designed to act more or less like matrix operations in linear algebra.	In NumPy the basic type is a multidimensional `array`. Operations on these arrays in all dimensionalities including 2D are element-wise operations. However, there is a special `matrix` type for doing linear algebra, which is just a subclass of the `array` class. Operations on matrix-class arrays are linear algebra operations.
MATLAB® uses 1 (one) based indexing. The initial element of a sequence is found using a(1). *See note INDEXING*	Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].
MATLAB®'s scripting language was created for doing linear algebra. The syntax for basic matrix operations is nice and clean, but the API for adding GUIs and making full-fledged applications is more or less an afterthought.	NumPy is based on Python, which was designed from the outset to be an excellent general-purpose programming language. While Matlab's syntax for some array manipulations is more compact than NumPy's, NumPy (by virtue of being an add-on to Python) can do many things that Matlab just cannot, for instance subclassing the main array type to do both array and matrix math cleanly.
In MATLAB®, arrays have pass-by-value semantics, with a lazy copy-on-write scheme to prevent actually creating copies until they are actually needed. Slice operations copy parts of the array.	In NumPy arrays have pass-by-reference semantics. Slice operations are views into an array.

5.3 'array' or 'matrix'? Which should I use?

NumPy provides, in addition to `np.ndarray`, an additional matrix type that you may see used in some existing code. Which one to use?

5.3.1 Short answer

Use arrays.

- They are the standard vector/matrix/tensor type of numpy. Many numpy functions return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.
- You can have standard vectors or row/column vectors if you like.

Until Python 3.5 the only disadvantage of using the array type was that you had to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.). Since Python 3.5 you can use the matrix multiplication `@` operator.

5.3.2 Long answer

NumPy contains both an `array` class and a `matrix` class. The `array` class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while `matrix` is intended to facilitate linear algebra computations specifically. In practice there are only a handful of key differences between the two.

- Operator `*`, `dot()`, and `multiply()`:
  - For `array`, **‘‘*’’ means element-wise multiplication**, and the `dot()` function is used for matrix multiplication.
  - For `matrix`, **‘‘*’’ means matrix multiplication**, and the `multiply()` function is used for element-wise multiplication.
- Handling of vectors (one-dimensional arrays)
  - For `array`, the **vector shapes 1xN, Nx1, and N are all different things**. Operations like `A[:,1]` return a one-dimensional array of shape N, not a two-dimensional array of shape Nx1. Transpose on a one-dimensional `array` does nothing.
  - For `matrix`, **one-dimensional arrays are always upconverted to 1xN or Nx1 matrices** (row or column vectors). `A[:,1]` returns a two-dimensional matrix of shape Nx1.
- Handling of higher-dimensional arrays (ndim > 2)
  - `array` objects **can have number of dimensions > 2**;
  - `matrix` objects **always have exactly two dimensions**.
- Convenience attributes
  - `array` **has a .T attribute**, which returns the transpose of the data.
  - `matrix` **also has .H, .I, and .A attributes**, which return the conjugate transpose, inverse, and `asarray()` of the matrix, respectively.
- Convenience constructor
  - The `array` constructor **takes (nested) Python sequences as initializers**. As in, `array([[1,2,3],[4,5,6]])`.
  - The `matrix` constructor additionally **takes a convenient string initializer**. As in `matrix("[1 2 3; 4 5 6]")`.

There are pros and cons to using both:

- `array`
  - `:)` You can treat one-dimensional arrays as *either* row or column vectors. `dot(A,v)` treats `v` as a column vector, while `dot(v,A)` treats `v` as a row vector. This can save you having to type a lot of transposes.

  - <: (Having to use the `dot()` function for matrix-multiply is messy – `dot(dot(A,B),C)` vs. `A*B*C`. This isn't an issue with Python >= 3.5 because the `@` operator allows it to be written as `A @ B @ C`.
  - :) Element-wise multiplication is easy: `A*B`.
  - :) `array` is the "default" NumPy type, so it gets the most testing, and is the type most likely to be returned by 3rd party code that uses NumPy.
  - :) Is quite at home handling data of any number of dimensions.
  - :) Closer in semantics to tensor algebra, if you are familiar with that.
  - :) *All* operations (`*`, `/`, `+`, `-` etc.) are element-wise.

- `matrix`
  - :\\ Behavior is more like that of MATLAB® matrices.
  - <: (Maximum of two-dimensional. To hold three-dimensional data you need `array` or perhaps a Python list of `matrix`.
  - <: (Minimum of two-dimensional. You cannot have vectors. They must be cast as single-column or single-row matrices.
  - <: (Since `array` is the default in NumPy, some functions may return an `array` even if you give them a `matrix` as an argument. This shouldn't happen with NumPy functions (if it does it's a bug), but 3rd party code based on NumPy may not honor type preservation like NumPy does.
  - :) `A*B` is matrix multiplication, so more convenient for linear algebra (For Python >= 3.5 plain arrays have the same convenience with the `@` operator).
  - <: (Element-wise multiplication requires calling a function, `multiply(A,B)`.
  - <: (The use of operator overloading is a bit illogical: `*` does not work element-wise but `/` does.

The `array` is thus much more advisable to use.

5.4 Facilities for Matrix Users

NumPy has some features that facilitate the use of the `matrix` type, which hopefully make things easier for Matlab converts.

- A `matlib` module has been added that contains matrix versions of common array constructors like `ones()`, `zeros()`, `empty()`, `eye()`, `rand()`, `repmat()`, etc. Normally these functions return `array`s, but the `matlib` versions return `matrix` objects.
- `mat` has been changed to be a synonym for `asmatrix`, rather than `matrix`, thus making it a concise way to convert an `array` to a `matrix` without copying the data.
- Some top-level functions have been removed. For example `numpy.rand()` now needs to be accessed as `numpy.random.rand()`. Or use the `rand()` from the `matlib` module. But the "numpythonic" way is to use `numpy.random.random()`, which takes a tuple for the shape, like other numpy functions.

5.5 Table of Rough MATLAB-NumPy Equivalents

The table below gives rough equivalents for some common MATLAB® expressions. **These are not exact equivalents**, but rather should be taken as hints to get you going in the right direction. For more detail read the built-in documentation on the NumPy functions.

Some care is necessary when writing functions that take arrays or matrices as arguments — if you are expecting an `array` and are given a `matrix`, or vice versa, then '*' (multiplication) will give you unexpected results. You can convert back and forth between arrays and matrices using

- `asarray`: always returns an object of type `array`
- `asmatrix` or `mat`: always return an object of type `matrix`
- `asanyarray`: always returns an `array` object or a subclass derived from it, depending on the input. For instance if you pass in a `matrix` it returns a `matrix`.

These functions all accept both arrays and matrices (among other things like Python lists), and thus are useful when writing functions that should accept any array-like object.

In the table below, it is assumed that you have executed the following commands in Python:

```
from numpy import *
import scipy.linalg
```

Also assume below that if the Notes talk about "matrix" that the arguments are two-dimensional entities.

5.5.1 General Purpose Equivalents

MATLAB	numpy	Notes
`help func`	`info(func)` or `help(func)` or `func?` (in Ipython)	get help on the function *func*
`which func`	see note HELP	find out where *func* is defined
`type func`	`source(func)` or `func??` (in Ipython)	print source for *func* (if not a native function)
`a && b`	`a and b`	short-circuiting logical AND operator (Python native operator); scalar arguments only
`a || b`	`a or b`	short-circuiting logical OR operator (Python native operator); scalar arguments only
`1*i, 1*j, 1i, 1j`	`1j`	complex numbers
`eps`	`np.spacing(1)`	Distance between 1 and the nearest floating point number.
`ode45`	`scipy.integrate.solve_ivp(f)`	integrate an ODE with Runge-Kutta 4,5
`ode15s`	`scipy.integrate.solve_ivp(f, method='BDF')`	integrate an ODE with BDF method

5.5.2 Linear Algebra Equivalents

MATLAB	NumPy
`ndims(a)`	`ndim(a)` or `a.ndim`
`numel(a)`	`size(a)` or `a.size`
`size(a)`	`shape(a)` or `a.shape`
`size(a,n)`	`a.shape[n-1]`
`[ 1 2 3; 4 5 6 ]`	`array([[1.,2.,3.], [4.,5.,6.]])`

MATLAB	NumPy
`[ a b; c d ]`	`vstack([hstack([a,b]), hstack([c,d])])` or `bmat('a b; c d')`
`a(end)`	`a[-1]`
`a(2,5)`	`a[1,4]`
`a(2,:)`	`a[1]` or `a[1,:]`
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5,:]`
`a(end-4:end,:)`	`a[-5:]`
`a(1:3,5:9)`	`a[0:3][:,4:9]`
`a([2,4,5],[1,3])`	`a[ix_([1,3,4],[0,2])]`
`a(3:2:21,:)`	`a[ 2:21:2,:]`
`a(1:2:end,:)`	`a[ ::2,:]`
`a(end:-1:1,:)` or `flipud(a)`	`a[ ::-1,:]`
`a([1:end 1],:)`	`a[r_[:len(a),0]]`
`a.'`	`a.transpose()` or `a.T`
`a'`	`a.conj().transpose()` or `a.conj().T`
`a * b`	`a.dot(b)`
`a .* b`	`a * b`
`a./b`	`a/b`
`a.^3`	`a**3`
`(a>0.5)`	`(a>0.5)`
`find(a>0.5)`	`nonzero(a>0.5)`
`a(:,find(v>0.5))`	`a[:,nonzero(v>0.5)[0]]`
`a(:,find(v>0.5))`	`a[:,v.T>0.5]`
`a(a<0.5)=0`	`a[a<0.5]=0`
`a .* (a>0.5)`	`a * (a>0.5)`
`a(:) = 3`	`a[:] = 3`
`y=x`	`y = x.copy()`
`y=x(2,:)`	`y = x[1,:].copy()`
`y=x(:)`	`y = x.flatten()`
`1:10`	`arange(1.,11.)` or `r_[1.:11.]` or `r_[1:10:10j]`
`0:9`	`arange(10.)` or `r_[:10.]` or `r_[:9:10j]`
`[1:10]'`	`arange(1.,11.)[:, newaxis]`
`zeros(3,4)`	`zeros((3,4))`
`zeros(3,4,5)`	`zeros((3,4,5))`
`ones(3,4)`	`ones((3,4))`
`eye(3)`	`eye(3)`
`diag(a)`	`diag(a)`
`diag(a,0)`	`diag(a,0)`
`rand(3,4)`	`random.rand(3,4)`
`linspace(1,3,4)`	`linspace(1,3,4)`
`[x,y]=meshgrid(0:8,0:5)`	`mgrid[0:9.,0:6.]` or `meshgrid(r_[0:9.],r_[0:6.]`
	`ogrid[0:9.,0:6.]` or `ix_(r_[0:9.],r_[0:6.]`
`[x,y]=meshgrid([1,2,4],[2,4,5])`	`meshgrid([1,2,4],[2,4,5])`
	`ix_([1,2,4],[2,4,5])`
`repmat(a, m, n)`	`tile(a, (m, n))`
`[a b]`	`concatenate((a,b),1)` or `hstack((a,b))` or `column_stack((a,b))`
`[a; b]`	`concatenate((a,b))` or `vstack((a,b))` or `r_[a,b]`
`max(max(a))`	`a.max()`
`max(a)`	`a.max(0)`
`max(a,[],2)`	`a.max(1)`

MATLAB	NumPy
`max(a,b)`	`maximum(a, b)`
`norm(v)`	`sqrt(dot(v,v))` or `np.linalg.norm(v)`
`a & b`	`logical_and(a,b)`
`a | b`	`logical_or(a,b)`
`bitand(a,b)`	`a & b`
`bitor(a,b)`	`a | b`
`inv(a)`	`linalg.inv(a)`
`pinv(a)`	`linalg.pinv(a)`
`rank(a)`	`linalg.matrix_rank(a)`
`a\b`	`linalg.solve(a,b)` if `a` is square; `linalg.lstsq(a,b)` otherwise
`b/a`	Solve a.T x.T = b.T instead
`[U,S,V]=svd(a)`	`U, S, Vh = linalg.svd(a), V = Vh.T`
`chol(a)`	`linalg.cholesky(a).T`
`[V,D]=eig(a)`	`D,V = linalg.eig(a)`
`[V,D]=eig(a,b)`	`V,D = np.linalg.eig(a,b)`
`[V,D]=eigs(a,k)`	
`[Q,R,P]=qr(a,0)`	`Q,R = scipy.linalg.qr(a)`
`[L,U,P]=lu(a)`	`L,U = scipy.linalg.lu(a)` or `LU,P=scipy.linalg.lu_factor(a)`
`conjgrad`	`scipy.sparse.linalg.cg`
`fft(a)`	`fft(a)`
`ifft(a)`	`ifft(a)`
`sort(a)`	`sort(a)` or `a.sort()`
`[b,I] = sortrows(a,i)`	`I = argsort(a[:,i]), b=a[I,:]`
`regress(y,X)`	`linalg.lstsq(X,y)`
`decimate(x, q)`	`scipy.signal.resample(x, len(x)/q)`
`unique(a)`	`unique(a)`
`squeeze(a)`	`a.squeeze()`

5.6 Notes

Submatrix: Assignment to a submatrix can be done with lists of indexes using the `ix_` command. E.g., for 2d array `a`, one might do: `ind=[1,3]; a[np.ix_(ind,ind)]+=100`.

HELP: There is no direct equivalent of MATLAB's `which` command, but the commands `help` and `source` will usually list the filename where the function is located. Python also has an `inspect` module (do `import inspect`) which provides a `getfile` that often works.

INDEXING: MATLAB® uses one based indexing, so the initial element of a sequence has index 1. Python uses zero based indexing, so the initial element of a sequence has index 0. Confusion and flamewars arise because each has advantages and disadvantages. One based indexing is consistent with common human language usage, where the "first" element of a sequence has index 1. Zero based indexing simplifies indexing. See also a text by prof.dr. Edsger W. Dijkstra.

RANGES: In MATLAB®, `0:5` can be used as both a range literal and a 'slice' index (inside parentheses); however, in Python, constructs like `0:5` can *only* be used as a slice index (inside square brackets). Thus the somewhat quirky `r_` object was created to allow numpy to have a similarly terse range construction mechanism. Note that `r_` is not called like a function or a constructor, but rather *indexed* using square brackets, which allows the use of Python's slice syntax in the arguments.

LOGICOPS: & or | in NumPy is bitwise AND/OR, while in Matlab & and | are logical AND/OR. The difference should be clear to anyone with significant programming experience. The two can appear to work the same, but there

are important differences. If you would have used Matlab's & or | operators, you should use the NumPy ufuncs logical_and/logical_or. The notable differences between Matlab's and NumPy's & and | operators are:

- Non-logical {0,1} inputs: NumPy's output is the bitwise AND of the inputs. Matlab treats any non-zero value as 1 and returns the logical AND. For example (3 & 4) in NumPy is 0, while in Matlab both 3 and 4 are considered logical true and (3 & 4) returns 1.
- Precedence: NumPy's & operator is higher precedence than logical operators like < and >; Matlab's is the reverse.

If you know you have boolean arguments, you can get away with using NumPy's bitwise operators, but be careful with parentheses, like this: z = (x > 1) & (x < 2). The absence of NumPy operator forms of logical_and and logical_or is an unfortunate consequence of Python's design.

RESHAPE and LINEAR INDEXING: Matlab always allows multi-dimensional arrays to be accessed using scalar or linear indices, NumPy does not. Linear indices are common in Matlab programs, e.g. find() on a matrix returns them, whereas NumPy's find behaves differently. When converting Matlab code it might be necessary to first reshape a matrix to a linear sequence, perform some indexing operations and then reshape back. As reshape (usually) produces views onto the same storage, it should be possible to do this fairly efficiently. Note that the scan order used by reshape in NumPy defaults to the 'C' order, whereas Matlab uses the Fortran order. If you are simply converting to a linear sequence and back this doesn't matter. But if you are converting reshapes from Matlab code which relies on the scan order, then this Matlab code: z = reshape(x,3,4); should become z = x.reshape(3,4,order='F').copy() in NumPy.

5.7 Customizing Your Environment

In MATLAB® the main tool available to you for customizing the environment is to modify the search path with the locations of your favorite functions. You can put such customizations into a startup script that MATLAB will run on startup.

NumPy, or rather Python, has similar facilities.

- To modify your Python search path to include the locations of your own modules, define the `PYTHONPATH` environment variable.
- To have a particular script file executed when the interactive Python interpreter is started, define the `PYTHONSTARTUP` environment variable to contain the name of your startup script.

Unlike MATLAB®, where anything on your path can be called immediately, with Python you need to first do an 'import' statement to make functions in a particular file accessible.

For example you might make a startup script that looks like this (Note: this is just an example, not a statement of "best practices"):

```
# Make all numpy available via shorter 'num' prefix
import numpy as num
# Make all matlib functions accessible at the top level via M.func()
import numpy.matlib as M
# Make some matlib functions accessible directly at the top level via, e.g. rand(3,3)
from numpy.matlib import rand,zeros,ones,empty,eye
# Define a Hermitian function
def hermitian(A, **kwargs):
    return num.transpose(A,**kwargs).conj()
# Make some shortcuts for transpose,hermitian:
#    num.transpose(A) --> T(A)
#    hermitian(A) --> H(A)
T = num.transpose
H = hermitian
```

5.8 Links

See http://mathesaurus.sf.net/ for another MATLAB®/NumPy cross-reference.

An extensive list of tools for scientific work with python can be found in the topical software page.

MATLAB® and SimuLink® are registered trademarks of The MathWorks.

CHAPTER

SIX

BUILDING FROM SOURCE

A general overview of building NumPy from source is given here, with detailed instructions for specific platforms given separately.

6.1 Prerequisites

Building NumPy requires the following software installed:

1. Python 2.7.x, 3.4.x or newer

   On Debian and derivatives (Ubuntu): python, python-dev (or python3-dev)

   On Windows: the official python installer at www.python.org is enough

   Make sure that the Python package distutils is installed before continuing. For example, in Debian GNU/Linux, installing python-dev also installs distutils.

   Python must also be compiled with the zlib module enabled. This is practically always the case with pre-packaged Pythons.

2. Compilers

   To build any extension modules for Python, you'll need a C compiler. Various NumPy modules use FORTRAN 77 libraries, so you'll also need a FORTRAN 77 compiler installed.

   Note that NumPy is developed mainly using GNU compilers. Compilers from other vendors such as Intel, Absoft, Sun, NAG, Compaq, Vast, Portland, Lahey, HP, IBM, Microsoft are only supported in the form of community feedback, and may not work out of the box. GCC 4.x (and later) compilers are recommended.

3. Linear Algebra libraries

   NumPy does not require any external linear algebra libraries to be installed. However, if these are available, NumPy's setup script can detect them and use them for building. A number of different LAPACK library setups can be used, including optimized LAPACK libraries such as ATLAS, MKL or the Accelerate/vecLib framework on OS X.

4. Cython

   To build development versions of NumPy, you'll need a recent version of Cython. Released NumPy sources on PyPi include the C files generated from Cython code, so for released versions having Cython installed isn't needed.

6.2 Basic Installation

To install NumPy run:

```
python setup.py install
```

To perform an in-place build that can be run from the source folder run:

```
python setup.py build_ext --inplace
```

The NumPy build system uses `setuptools` (from numpy 1.11.0, before that it was plain `distutils`) and `numpy.distutils`. Using `virtualenv` should work as expected.

Note: for build instructions to do development work on NumPy itself, see development-environment.

6.2.1 Parallel builds

From NumPy 1.10.0 on it's also possible to do a parallel build with:

```
python setup.py build -j 4 install --prefix $HOME/.local
```

This will compile numpy on 4 CPUs and install it into the specified prefix. to perform a parallel in-place build, run:

```
python setup.py build_ext --inplace -j 4
```

The number of build jobs can also be specified via the environment variable `NPY_NUM_BUILD_JOBS`.

6.3 FORTRAN ABI mismatch

The two most popular open source fortran compilers are g77 and gfortran. Unfortunately, they are not ABI compatible, which means that concretely you should avoid mixing libraries built with one with another. In particular, if your blas/lapack/atlas is built with g77, you *must* use g77 when building numpy and scipy; on the contrary, if your atlas is built with gfortran, you *must* build numpy/scipy with gfortran. This applies for most other cases where different FORTRAN compilers might have been used.

6.3.1 Choosing the fortran compiler

To build with gfortran:

```
python setup.py build --fcompiler=gnu95
```

For more information see:

```
python setup.py build --help-fcompiler
```

6.3.2 How to check the ABI of blas/lapack/atlas

One relatively simple and reliable way to check for the compiler used to build a library is to use ldd on the library. If libg2c.so is a dependency, this means that g77 has been used. If libgfortran.so is a dependency, gfortran has been used. If both are dependencies, this means both have been used, which is almost always a very bad idea.

6.4 Disabling ATLAS and other accelerated libraries

Usage of ATLAS and other accelerated libraries in NumPy can be disabled via:

```
BLAS=None LAPACK=None ATLAS=None python setup.py build
```

6.5 Supplying additional compiler flags

Additional compiler flags can be supplied by setting the `OPT`, `FOPT` (for Fortran), and `CC` environment variables.

6.6 Building with ATLAS support

6.6.1 Ubuntu

You can install the necessary package for optimized ATLAS with this command:

```
sudo apt-get install libatlas-base-dev
```

CHAPTER

SEVEN

USING NUMPY C-API

7.1 How to extend NumPy

That which is static and repetitive is boring. That which is dynamic
and random is confusing. In between lies art.
— *John A. Locke*

Science is a differential equation. Religion is a boundary condition.
— *Alan Turing*

7.1.1 Writing an extension module

While the ndarray object is designed to allow rapid computation in Python, it is also designed to be general-purpose and satisfy a wide- variety of computational needs. As a result, if absolute speed is essential, there is no replacement for a well-crafted, compiled loop specific to your application and hardware. This is one of the reasons that numpy includes f2py so that an easy-to-use mechanisms for linking (simple) C/C++ and (arbitrary) Fortran code directly into Python are available. You are encouraged to use and improve this mechanism. The purpose of this section is not to document this tool but to document the more basic steps to writing an extension module that this tool depends on.

When an extension module is written, compiled, and installed to somewhere in the Python path (sys.path), the code can then be imported into Python as if it were a standard python file. It will contain objects and methods that have been defined and compiled in C code. The basic steps for doing this in Python are well-documented and you can find more information in the documentation for Python itself available online at www.python.org .

In addition to the Python C-API, there is a full and rich C-API for NumPy allowing sophisticated manipulations on a C-level. However, for most applications, only a few API calls will typically be used. If all you need to do is extract a pointer to memory along with some shape information to pass to another calculation routine, then you will use very different calls, then if you are trying to create a new array- like type or add a new data type for ndarrays. This chapter documents the API calls and macros that are most commonly used.

7.1.2 Required subroutine

There is exactly one function that must be defined in your C-code in order for Python to use it as an extension module. The function must be called init{name} where {name} is the name of the module from Python. This function must be declared so that it is visible to code outside of the routine. Besides adding the methods and constants you desire, this subroutine must also contain calls like `import_array()` and/or `import_ufunc()` depending on which C-API is needed. Forgetting to place these commands will show itself as an ugly segmentation fault (crash) as soon as any C-API subroutine is actually called. It is actually possible to have multiple init{name} functions in a single file in

which case multiple modules will be defined by that file. However, there are some tricks to get that to work correctly and it is not covered here.

A minimal `init{name}` method looks like:

```
PyMODINIT_FUNC
init{name}(void)
{
    (void)Py_InitModule({name}, mymethods);
    import_array();
}
```

The mymethods must be an array (usually statically declared) of PyMethodDef structures which contain method names, actual C-functions, a variable indicating whether the method uses keyword arguments or not, and docstrings. These are explained in the next section. If you want to add constants to the module, then you store the returned value from Py_InitModule which is a module object. The most general way to add items to the module is to get the module dictionary using PyModule_GetDict(module). With the module dictionary, you can add whatever you like to the module manually. An easier way to add objects to the module is to use one of three additional Python C-API calls that do not require a separate extraction of the module dictionary. These are documented in the Python documentation, but repeated here for convenience:

int **PyModule_AddObject**(PyObject* *module*, char* *name*, PyObject* *value*)

int **PyModule_AddIntConstant**(PyObject* *module*, char* *name*, long *value*)

int **PyModule_AddStringConstant**(PyObject* *module*, char* *name*, char* *value*)

All three of these functions require the *module* object (the return value of Py_InitModule). The *name* is a string that labels the value in the module. Depending on which function is called, the *value* argument is either a general object (*PyModule_AddObject* steals a reference to it), an integer constant, or a string constant.

7.1.3 Defining functions

The second argument passed in to the Py_InitModule function is a structure that makes it easy to to define functions in the module. In the example given above, the mymethods structure would have been defined earlier in the file (usually right before the init{name} subroutine) to:

```
static PyMethodDef mymethods[] = {
    { nokeywordfunc,nokeyword_cfunc,
      METH_VARARGS,
      Doc string},
    { keywordfunc, keyword_cfunc,
      METH_VARARGS|METH_KEYWORDS,
      Doc string},
    {NULL, NULL, 0, NULL} /* Sentinel */
}
```

Each entry in the mymethods array is a `PyMethodDef` structure containing 1) the Python name, 2) the C-function that implements the function, 3) flags indicating whether or not keywords are accepted for this function, and 4) The docstring for the function. Any number of functions may be defined for a single module by adding more entries to this table. The last entry must be all NULL as shown to act as a sentinel. Python looks for this entry to know that all of the functions for the module have been defined.

The last thing that must be done to finish the extension module is to actually write the code that performs the desired functions. There are two kinds of functions: those that don't accept keyword arguments, and those that do.

Functions without keyword arguments

Functions that don't accept keyword arguments should be written as:

```
static PyObject*
nokeyword_cfunc (PyObject *dummy, PyObject *args)
{
    /* convert Python arguments */
    /* do function */
    /* return something */
}
```

The dummy argument is not used in this context and can be safely ignored. The *args* argument contains all of the arguments passed in to the function as a tuple. You can do anything you want at this point, but usually the easiest way to manage the input arguments is to call `PyArg_ParseTuple` (args, format_string, addresses_to_C_variables...) or `PyArg_UnpackTuple` (tuple, "name" , min, max, ...). A good description of how to use the first function is contained in the Python C-API reference manual under section 5.5 (Parsing arguments and building values). You should pay particular attention to the "O&" format which uses converter functions to go between the Python object and the C object. All of the other format functions can be (mostly) thought of as special cases of this general rule. There are several converter functions defined in the NumPy C-API that may be of use. In particular, the `PyArray_DescrConverter` function is very useful to support arbitrary data-type specification. This function transforms any valid data-type Python object into a `PyArray_Descr *` object. Remember to pass in the address of the C-variables that should be filled in.

There are lots of examples of how to use `PyArg_ParseTuple` throughout the NumPy source code. The standard usage is like this:

```
PyObject *input;
PyArray_Descr *dtype;
if (!PyArg_ParseTuple(args, "OO&", &input,
                      PyArray_DescrConverter,
                      &dtype)) return NULL;
```

It is important to keep in mind that you get a *borrowed* reference to the object when using the "O" format string. However, the converter functions usually require some form of memory handling. In this example, if the conversion is successful, *dtype* will hold a new reference to a `PyArray_Descr *` object, while *input* will hold a borrowed reference. Therefore, if this conversion were mixed with another conversion (say to an integer) and the data-type conversion was successful but the integer conversion failed, then you would need to release the reference count to the data-type object before returning. A typical way to do this is to set *dtype* to `NULL` before calling `PyArg_ParseTuple` and then use `Py_XDECREF` on *dtype* before returning.

After the input arguments are processed, the code that actually does the work is written (likely calling other functions as needed). The final step of the C-function is to return something. If an error is encountered then `NULL` should be returned (making sure an error has actually been set). If nothing should be returned then increment `Py_None` and return it. If a single object should be returned then it is returned (ensuring that you own a reference to it first). If multiple objects should be returned then you need to return a tuple. The `Py_BuildValue` (format_string, c_variables...) function makes it easy to build tuples of Python objects from C variables. Pay special attention to the difference between 'N' and 'O' in the format string or you can easily create memory leaks. The 'O' format string increments the reference count of the `PyObject *` C-variable it corresponds to, while the 'N' format string steals a reference to the corresponding `PyObject *` C-variable. You should use 'N' if you have already created a reference for the object and just want to give that reference to the tuple. You should use 'O' if you only have a borrowed reference to an object and need to create one to provide for the tuple.

Functions with keyword arguments

These functions are very similar to functions without keyword arguments. The only difference is that the function signature is:

```
static PyObject*
keyword_cfunc (PyObject *dummy, PyObject *args, PyObject *kwds)
{
...
}
```

The kwds argument holds a Python dictionary whose keys are the names of the keyword arguments and whose values are the corresponding keyword-argument values. This dictionary can be processed however you see fit. The easiest way to handle it, however, is to replace the `PyArg_ParseTuple` (args, format_string, addresses...) function with a call to `PyArg_ParseTupleAndKeywords` (args, kwds, format_string, char *kwlist[], addresses...). The kwlist parameter to this function is a `NULL` -terminated array of strings providing the expected keyword arguments. There should be one string for each entry in the format_string. Using this function will raise a TypeError if invalid keyword arguments are passed in.

For more help on this function please see section 1.8 (Keyword Parameters for Extension Functions) of the Extending and Embedding tutorial in the Python documentation.

Reference counting

The biggest difficulty when writing extension modules is reference counting. It is an important reason for the popularity of f2py, weave, Cython, ctypes, etc.... If you mis-handle reference counts you can get problems from memory-leaks to segmentation faults. The only strategy I know of to handle reference counts correctly is blood, sweat, and tears. First, you force it into your head that every Python variable has a reference count. Then, you understand exactly what each function does to the reference count of your objects, so that you can properly use DECREF and INCREF when you need them. Reference counting can really test the amount of patience and diligence you have towards your programming craft. Despite the grim depiction, most cases of reference counting are quite straightforward with the most common difficulty being not using DECREF on objects before exiting early from a routine due to some error. In second place, is the common error of not owning the reference on an object that is passed to a function or macro that is going to steal the reference (*e.g.* `PyTuple_SET_ITEM`, and most functions that take `PyArray_Descr` objects).

Typically you get a new reference to a variable when it is created or is the return value of some function (there are some prominent exceptions, however — such as getting an item out of a tuple or a dictionary). When you own the reference, you are responsible to make sure that `Py_DECREF` (var) is called when the variable is no longer necessary (and no other function has "stolen" its reference). Also, if you are passing a Python object to a function that will "steal" the reference, then you need to make sure you own it (or use `Py_INCREF` to get your own reference). You will also encounter the notion of borrowing a reference. A function that borrows a reference does not alter the reference count of the object and does not expect to "hold on "to the reference. It's just going to use the object temporarily. When you use `PyArg_ParseTuple` or `PyArg_UnpackTuple` you receive a borrowed reference to the objects in the tuple and should not alter their reference count inside your function. With practice, you can learn to get reference counting right, but it can be frustrating at first.

One common source of reference-count errors is the `Py_BuildValue` function. Pay careful attention to the difference between the 'N' format character and the 'O' format character. If you create a new object in your subroutine (such as an output array), and you are passing it back in a tuple of return values, then you should most- likely use the 'N' format character in `Py_BuildValue`. The 'O' character will increase the reference count by one. This will leave the caller with two reference counts for a brand-new array. When the variable is deleted and the reference count decremented by one, there will still be that extra reference count, and the array will never be deallocated. You will have a reference-counting induced memory leak. Using the 'N' character will avoid this situation as it will return to the caller an object (inside the tuple) with a single reference count.

7.1.4 Dealing with array objects

Most extension modules for NumPy will need to access the memory for an ndarray object (or one of it's sub-classes). The easiest way to do this doesn't require you to know much about the internals of NumPy. The method is to

1. Ensure you are dealing with a well-behaved array (aligned, in machine byte-order and single-segment) of the correct type and number of dimensions.
   (a) By converting it from some Python object using `PyArray_FromAny` or a macro built on it.
   (b) By constructing a new ndarray of your desired shape and type using `PyArray_NewFromDescr` or a simpler macro or function based on it.
2. Get the shape of the array and a pointer to its actual data.
3. Pass the data and shape information on to a subroutine or other section of code that actually performs the computation.
4. If you are writing the algorithm, then I recommend that you use the stride information contained in the array to access the elements of the array (the `PyArray_GETPTR` macros make this painless). Then, you can relax your requirements so as not to force a single-segment array and the data-copying that might result.

Each of these sub-topics is covered in the following sub-sections.

Converting an arbitrary sequence object

The main routine for obtaining an array from any Python object that can be converted to an array is `PyArray_FromAny`. This function is very flexible with many input arguments. Several macros make it easier to use the basic function. *`PyArray_FROM_OTF`* is arguably the most useful of these macros for the most common uses. It allows you to convert an arbitrary Python object to an array of a specific builtin data-type (*e.g.* float), while specifying a particular set of requirements (*e.g.* contiguous, aligned, and writeable). The syntax is

PyObject ***PyArray_FROM_OTF**(PyObject* *obj*, int *typenum*, int *requirements*)

Return an ndarray from any Python object, *obj*, that can be converted to an array. The number of dimensions in the returned array is determined by the object. The desired data-type of the returned array is provided in *typenum* which should be one of the enumerated types. The *requirements* for the returned array can be any combination of standard array flags. Each of these arguments is explained in more detail below. You receive a new reference to the array on success. On failure, `NULL` is returned and an exception is set.

obj

The object can be any Python object convertible to an ndarray. If the object is already (a subclass of) the ndarray that satisfies the requirements then a new reference is returned. Otherwise, a new array is constructed. The contents of *obj* are copied to the new array unless the array interface is used so that data does not have to be copied. Objects that can be converted to an array include: 1) any nested sequence object, 2) any object exposing the array interface, 3) any object with an `__array__` method (which should return an ndarray), and 4) any scalar object (becomes a zero-dimensional array). Sub-classes of the ndarray that otherwise fit the requirements will be passed through. If you want to ensure a base-class ndarray, then use *`NPY_ARRAY_ENSUREARRAY`* in the requirements flag. A copy is made only if necessary. If you want to guarantee a copy, then pass in *`NPY_ARRAY_ENSURECOPY`* to the requirements flag.

typenum

One of the enumerated types or `NPY_NOTYPE` if the data-type should be determined from the object itself. The C-based names can be used:

`NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT, NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG,`

NPY_DOUBLE, NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE, NPY_CLONGDOUBLE, NPY_OBJECT.

Alternatively, the bit-width names can be used as supported on the platform. For example:

NPY_INT8, NPY_INT16, NPY_INT32, NPY_INT64, NPY_UINT8, NPY_UINT16, NPY_UINT32, NPY_UINT64, NPY_FLOAT32, NPY_FLOAT64, NPY_COMPLEX64, NPY_COMPLEX128.

The object will be converted to the desired type only if it can be done without losing precision. Otherwise NULL will be returned and an error raised. Use *NPY_ARRAY_FORCECAST* in the requirements flag to override this behavior.

requirements

The memory model for an ndarray admits arbitrary strides in each dimension to advance to the next element of the array. Often, however, you need to interface with code that expects a C-contiguous or a Fortran-contiguous memory layout. In addition, an ndarray can be misaligned (the address of an element is not at an integral multiple of the size of the element) which can cause your program to crash (or at least work more slowly) if you try and dereference a pointer into the array data. Both of these problems can be solved by converting the Python object into an array that is more "well-behaved" for your specific usage.

The requirements flag allows specification of what kind of array is acceptable. If the object passed in does not satisfy this requirements then a copy is made so that thre returned object will satisfy the requirements. these ndarray can use a very generic pointer to memory. This flag allows specification of the desired properties of the returned array object. All of the flags are explained in the detailed API chapter. The flags most commonly needed are *NPY_ARRAY_IN_ARRAY*, NPY_OUT_ARRAY, and *NPY_ARRAY_INOUT_ARRAY*:

NPY_ARRAY_IN_ARRAY
Equivalent to NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_ALIGNED. This combination of flags is useful for arrays that must be in C-contiguous order and aligned. These kinds of arrays are usually input arrays for some algorithm.

NPY_ARRAY_OUT_ARRAY
Equivalent to NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE. This combination of flags is useful to specify an array that is in C-contiguous order, is aligned, and can be written to as well. Such an array is usually returned as output (although normally such output arrays are created from scratch).

NPY_ARRAY_INOUT_ARRAY
Equivalent to NPY_ARRAY_C_CONTIGUOUS | NPY_ARRAY_ALIGNED | NPY_ARRAY_WRITEABLE | NPY_ARRAY_WRITEBACKIFCOPY | NPY_ARRAY_UPDATEIFCOPY. This combination of flags is useful to specify an array that will be used for both input and output. PyArray_ResolveWritebackIfCopy must be called before Py_DECREF at the end of the interface routine to write back the temporary data into the original array passed in. Use of the NPY_ARRAY_WRITEBACKIFCOPY or NPY_ARRAY_UPDATEIFCOPY flags requires that the input object is already an array (because other objects cannot be automatically updated in this fashion). If an error occurs use PyArray_DiscardWritebackIfCopy (obj) on an array with these flags set. This will set the underlying base array writable without causing the contents to be copied back into the original array.

Other useful flags that can be OR'd as additional requirements are:

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 and not a sub- class.

Note: Whether or not an array is byte-swapped is determined by the data-type of the array. Native byte-order arrays are always requested by *PyArray_FROM_OTF* and so there is no need for a NPY_ARRAY_NOTSWAPPED flag in the requirements argument. There is also no way to get a byte-swapped array from this routine.

Creating a brand-new ndarray

Quite often new arrays must be created from within extension-module code. Perhaps an output array is needed and you don't want the caller to have to supply it. Perhaps only a temporary array is needed to hold an intermediate calculation. Whatever the need there are simple ways to get an ndarray object of whatever data-type is needed. The most general function for doing this is PyArray_NewFromDescr. All array creation functions go through this heavily re-used code. Because of its flexibility, it can be somewhat confusing to use. As a result, simpler forms exist that are easier to use.

PyObject ***PyArray_SimpleNew**(int *nd*, npy_intp* *dims*, int *typenum*)
This function allocates new memory and places it in an ndarray with *nd* dimensions whose shape is determined by the array of at least *nd* items pointed to by *dims*. The memory for the array is uninitialized (unless typenum is NPY_OBJECT in which case each element in the array is set to NULL). The *typenum* argument allows specification of any of the builtin data-types such as NPY_FLOAT or NPY_LONG. The memory for the array can be set to zero if desired using PyArray_FILLWBYTE (return_object, 0).

PyObject ***PyArray_SimpleNewFromData**(int *nd*, npy_intp* *dims*, int *typenum*, void* *data*)
Sometimes, you want to wrap memory allocated elsewhere into an ndarray object for downstream use. This routine makes it straightforward to do that. The first three arguments are the same as in *PyArray_SimpleNew*, the final argument is a pointer to a block of contiguous memory that the ndarray should use as it's data-buffer which will be interpreted in C-style contiguous fashion. A new reference to an ndarray is returned, but the ndarray will not own its data. When this ndarray is deallocated, the pointer will not be freed.

You should ensure that the provided memory is not freed while the returned array is in existence. The easiest way to handle this is if data comes from another reference-counted Python object. The reference count on this object should be increased after the pointer is passed in, and the base member of the returned ndarray should point to the Python object that owns the data. Then, when the ndarray is deallocated, the base-member will be DECREF'd appropriately. If you want the memory to be freed as soon as the ndarray is deallocated then simply set the OWNDATA flag on the returned ndarray.

Getting at ndarray memory and accessing elements of the ndarray

If obj is an ndarray (PyArrayObject *), then the data-area of the ndarray is pointed to by the void* pointer PyArray_DATA (obj) or the char* pointer PyArray_BYTES (obj). Remember that (in general) this data-area may not be aligned according to the data-type, it may represent byte-swapped data, and/or it may not be writeable. If the data area is aligned and in native byte-order, then how to get at a specific element of the array is determined only by the array of npy_intp variables, PyArray_STRIDES (obj). In particular, this c-array of integers shows how many **bytes** must be added to the current element pointer to get to the next element in each dimension. For arrays less than 4-dimensions there are PyArray_GETPTR{k} (obj, ...) macros where {k} is the integer 1, 2, 3, or 4 that make using the array strides easier. The arguments .... represent {k} non- negative integer indices into the array. For example, suppose E is a 3-dimensional ndarray. A (void*) pointer to the element E[i,j,k] is obtained as PyArray_GETPTR3 (E, i, j, k).

As explained previously, C-style contiguous arrays and Fortran-style contiguous arrays have particular striding patterns. Two array flags (`NPY_ARRAY_C_CONTIGUOUS` and `NPY_ARRAY_F_CONTIGUOUS`) indicate whether or not the striding pattern of a particular array matches the C-style contiguous or Fortran-style contiguous or neither. Whether or not the striding pattern matches a standard C or Fortran one can be tested Using `PyArray_ISCONTIGUOUS` (obj) and `PyArray_ISFORTRAN` (obj) respectively. Most third-party libraries expect contiguous arrays. But, often it is not difficult to support general-purpose striding. I encourage you to use the striding information in your own code whenever possible, and reserve single-segment requirements for wrapping third-party code. Using the striding information provided with the ndarray rather than requiring a contiguous striding reduces copying that otherwise must be made.

7.1.5 Example

The following example shows how you might write a wrapper that accepts two input arguments (that will be converted to an array) and an output argument (that must be an array). The function returns None and updates the output array. Note the updated use of WRITEBACKIFCOPY semantics for NumPy v1.14 and above

```
static PyObject *
example_wrapper(PyObject *dummy, PyObject *args)
{
    PyObject *arg1=NULL, *arg2=NULL, *out=NULL;
    PyObject *arr1=NULL, *arr2=NULL, *oarr=NULL;

    if (!PyArg_ParseTuple(args, "OOO!", &arg1, &arg2,
        &PyArray_Type, &out)) return NULL;

    arr1 = PyArray_FROM_OTF(arg1, NPY_DOUBLE, NPY_ARRAY_IN_ARRAY);
    if (arr1 == NULL) return NULL;
    arr2 = PyArray_FROM_OTF(arg2, NPY_DOUBLE, NPY_ARRAY_IN_ARRAY);
    if (arr2 == NULL) goto fail;
#if NPY_API_VERSION >= 0x0000000c
    oarr = PyArray_FROM_OTF(out, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY2);
#else
    oarr = PyArray_FROM_OTF(out, NPY_DOUBLE, NPY_ARRAY_INOUT_ARRAY);
#endif
    if (oarr == NULL) goto fail;

    /* code that makes use of arguments */
    /* You will probably need at least
       nd = PyArray_NDIM(<..>)    -- number of dimensions
       dims = PyArray_DIMS(<..>)  -- npy_intp array of length nd
                                     showing length in each dim.
       dptr = (double *)PyArray_DATA(<..>) -- pointer to data.

       If an error occurs goto fail.
     */

    Py_DECREF(arr1);
    Py_DECREF(arr2);
#if NPY_API_VERSION >= 0x0000000c
    PyArray_ResolveWritebackIfCopy(oarr);
#endif
    Py_DECREF(oarr);
    Py_INCREF(Py_None);
    return Py_None;

 fail:
```

```
    Py_XDECREF(arr1);
    Py_XDECREF(arr2);
#if NPY_API_VERSION >= 0x0000000c
    PyArray_DiscardWritebackIfCopy(oarr);
#endif
    Py_XDECREF(oarr);
    return NULL;
}
```

7.2 Using Python as glue

There is no conversation more boring than the one where everybody agrees.
— *Michel de Montaigne*

Duct tape is like the force. It has a light side, and a dark side, and it holds the universe together.
— *Carl Zwanzig*

Many people like to say that Python is a fantastic glue language. Hopefully, this Chapter will convince you that this is true. The first adopters of Python for science were typically people who used it to glue together large application codes running on super-computers. Not only was it much nicer to code in Python than in a shell script or Perl, in addition, the ability to easily extend Python made it relatively easy to create new classes and types specifically adapted to the problems being solved. From the interactions of these early contributors, Numeric emerged as an array-like object that could be used to pass data between these applications.

As Numeric has matured and developed into NumPy, people have been able to write more code directly in NumPy. Often this code is fast-enough for production use, but there are still times that there is a need to access compiled code. Either to get that last bit of efficiency out of the algorithm or to make it easier to access widely-available codes written in C/C++ or Fortran.

This chapter will review many of the tools that are available for the purpose of accessing code written in other compiled languages. There are many resources available for learning to call other compiled libraries from Python and the purpose of this Chapter is not to make you an expert. The main goal is to make you aware of some of the possibilities so that you will know what to "Google" in order to learn more.

7.2.1 Calling other compiled libraries from Python

While Python is a great language and a pleasure to code in, its dynamic nature results in overhead that can cause some code (*i.e.* raw computations inside of for loops) to be up 10-100 times slower than equivalent code written in a static compiled language. In addition, it can cause memory usage to be larger than necessary as temporary arrays are created and destroyed during computation. For many types of computing needs, the extra slow-down and memory consumption can often not be spared (at least for time- or memory- critical portions of your code). Therefore one of the most common needs is to call out from Python code to a fast, machine-code routine (e.g. compiled using C/C++ or Fortran). The fact that this is relatively easy to do is a big reason why Python is such an excellent high-level language for scientific and engineering programming.

Their are two basic approaches to calling compiled code: writing an extension module that is then imported to Python using the import command, or calling a shared-library subroutine directly from Python using the ctypes module. Writing an extension module is the most common method.

Warning: Calling C-code from Python can result in Python crashes if you are not careful. None of the approaches in this chapter are immune. You have to know something about the way data is handled by both NumPy and by the third-party library being used.

7.2.2 Hand-generated wrappers

Extension modules were discussed in *Writing an extension module*. The most basic way to interface with compiled code is to write an extension module and construct a module method that calls the compiled code. For improved readability, your method should take advantage of the `PyArg_ParseTuple` call to convert between Python objects and C data-types. For standard C data-types there is probably already a built-in converter. For others you may need to write your own converter and use the `"O&"` format string which allows you to specify a function that will be used to perform the conversion from the Python object to whatever C-structures are needed.

Once the conversions to the appropriate C-structures and C data-types have been performed, the next step in the wrapper is to call the underlying function. This is straightforward if the underlying function is in C or C++. However, in order to call Fortran code you must be familiar with how Fortran subroutines are called from C/C++ using your compiler and platform. This can vary somewhat platforms and compilers (which is another reason f2py makes life much simpler for interfacing Fortran code) but generally involves underscore mangling of the name and the fact that all variables are passed by reference (i.e. all arguments are pointers).

The advantage of the hand-generated wrapper is that you have complete control over how the C-library gets used and called which can lead to a lean and tight interface with minimal over-head. The disadvantage is that you have to write, debug, and maintain C-code, although most of it can be adapted using the time-honored technique of "cutting-pasting-and-modifying" from other extension modules. Because, the procedure of calling out to additional C-code is fairly regimented, code-generation procedures have been developed to make this process easier. One of these code-generation techniques is distributed with NumPy and allows easy integration with Fortran and (simple) C code. This package, f2py, will be covered briefly in the next section.

7.2.3 f2py

F2py allows you to automatically construct an extension module that interfaces to routines in Fortran 77/90/95 code. It has the ability to parse Fortran 77/90/95 code and automatically generate Python signatures for the subroutines it encounters, or you can guide how the subroutine interfaces with Python by constructing an interface-definition-file (or modifying the f2py-produced one).

Creating source for a basic extension module

Probably the easiest way to introduce f2py is to offer a simple example. Here is one of the subroutines contained in a file named `add.f`:

```
C
      SUBROUTINE ZADD(A,B,C,N)
C
      DOUBLE COMPLEX A(*)
      DOUBLE COMPLEX B(*)
      DOUBLE COMPLEX C(*)
      INTEGER N
      DO 20 J = 1, N
         C(J) = A(J)+B(J)
 20   CONTINUE
      END
```

This routine simply adds the elements in two contiguous arrays and places the result in a third. The memory for all three arrays must be provided by the calling routine. A very basic interface to this routine can be automatically generated by f2py:

```
f2py -m add add.f
```

You should be able to run this command assuming your search-path is set-up properly. This command will produce an extension module named addmodule.c in the current directory. This extension module can now be compiled and used from Python just like any other extension module.

Creating a compiled extension module

You can also get f2py to compile add.f and also compile its produced extension module leaving only a shared-library extension file that can be imported from Python:

```
f2py -c -m add add.f
```

This command leaves a file named add.{ext} in the current directory (where {ext} is the appropriate extension for a python extension module on your platform — so, pyd, *etc.*). This module may then be imported from Python. It will contain a method for each subroutine in add (zadd, cadd, dadd, sadd). The docstring of each method contains information about how the module method may be called:

```
>>> import add
>>> print add.zadd.__doc__
zadd - Function signature:
  zadd(a,b,c,n)
Required arguments:
  a : input rank-1 array('D') with bounds (*)
  b : input rank-1 array('D') with bounds (*)
  c : input rank-1 array('D') with bounds (*)
  n : input int
```

Improving the basic interface

The default interface is a very literal translation of the fortran code into Python. The Fortran array arguments must now be NumPy arrays and the integer argument should be an integer. The interface will attempt to convert all arguments to their required types (and shapes) and issue an error if unsuccessful. However, because it knows nothing about the semantics of the arguments (such that C is an output and n should really match the array sizes), it is possible to abuse this function in ways that can cause Python to crash. For example:

```
>>> add.zadd([1,2,3], [1,2], [3,4], 1000)
```

will cause a program crash on most systems. Under the covers, the lists are being converted to proper arrays but then the underlying add loop is told to cycle way beyond the borders of the allocated memory.

In order to improve the interface, directives should be provided. This is accomplished by constructing an interface definition file. It is usually best to start from the interface file that f2py can produce (where it gets its default behavior from). To get f2py to generate the interface file use the -h option:

```
f2py -h add.pyf -m add add.f
```

This command leaves the file add.pyf in the current directory. The section of this file corresponding to zadd is:

```
subroutine zadd(a,b,c,n) ! in :add:add.f
   double complex dimension(*) :: a
   double complex dimension(*) :: b
   double complex dimension(*) :: c
   integer :: n
end subroutine zadd
```

By placing intent directives and checking code, the interface can be cleaned up quite a bit until the Python module method is both easier to use and more robust.

```
subroutine zadd(a,b,c,n) ! in :add:add.f
   double complex dimension(n) :: a
   double complex dimension(n) :: b
   double complex intent(out),dimension(n) :: c
   integer intent(hide),depend(a) :: n=len(a)
end subroutine zadd
```

The intent directive, intent(out) is used to tell f2py that `c` is an output variable and should be created by the interface before being passed to the underlying code. The intent(hide) directive tells f2py to not allow the user to specify the variable, `n`, but instead to get it from the size of `a`. The depend(`a`) directive is necessary to tell f2py that the value of n depends on the input a (so that it won't try to create the variable n until the variable a is created).

After modifying `add.pyf`, the new python module file can be generated by compiling both `add.f95` and `add.pyf`:

```
f2py -c add.pyf add.f95
```

The new interface has docstring:

```
>>> import add
>>> print add.zadd.__doc__
zadd - Function signature:
  c = zadd(a,b)
Required arguments:
  a : input rank-1 array('D') with bounds (n)
  b : input rank-1 array('D') with bounds (n)
Return objects:
  c : rank-1 array('D') with bounds (n)
```

Now, the function can be called in a much more robust way:

```
>>> add.zadd([1,2,3],[4,5,6])
array([ 5.+0.j,  7.+0.j,  9.+0.j])
```

Notice the automatic conversion to the correct format that occurred.

Inserting directives in Fortran source

The nice interface can also be generated automatically by placing the variable directives as special comments in the original fortran code. Thus, if I modify the source code to contain:

```
C
      SUBROUTINE ZADD(A,B,C,N)
C
CF2PY INTENT(OUT) :: C
CF2PY INTENT(HIDE) :: N
CF2PY DOUBLE COMPLEX :: A(N)
CF2PY DOUBLE COMPLEX :: B(N)
```

```
CF2PY DOUBLE COMPLEX :: C(N)
      DOUBLE COMPLEX A(*)
      DOUBLE COMPLEX B(*)
      DOUBLE COMPLEX C(*)
      INTEGER N
      DO 20 J = 1, N
         C(J) = A(J) + B(J)
 20   CONTINUE
      END
```

Then, I can compile the extension module using:

```
f2py -c -m add add.f
```

The resulting signature for the function add.zadd is exactly the same one that was created previously. If the original source code had contained `A(N)` instead of `A(*)` and so forth with `B` and `C`, then I could obtain (nearly) the same interface simply by placing the `INTENT(OUT) :: C` comment line in the source code. The only difference is that `N` would be an optional input that would default to the length of `A`.

A filtering example

For comparison with the other methods to be discussed. Here is another example of a function that filters a two-dimensional array of double precision floating-point numbers using a fixed averaging filter. The advantage of using Fortran to index into multi-dimensional arrays should be clear from this example.

```
      SUBROUTINE DFILTER2D(A,B,M,N)
C
      DOUBLE PRECISION A(M,N)
      DOUBLE PRECISION B(M,N)
      INTEGER N, M
CF2PY INTENT(OUT) :: B
CF2PY INTENT(HIDE) :: N
CF2PY INTENT(HIDE) :: M
      DO 20 I = 2,M-1
         DO 40 J=2,N-1
            B(I,J) = A(I,J) +
     $           (A(I-1,J)+A(I+1,J) +
     $            A(I,J-1)+A(I,J+1) )*0.5D0 +
     $           (A(I-1,J-1) + A(I-1,J+1) +
     $            A(I+1,J-1) + A(I+1,J+1))*0.25D0
 40      CONTINUE
 20   CONTINUE
      END
```

This code can be compiled and linked into an extension module named filter using:

```
f2py -c -m filter filter.f
```

This will produce an extension module named filter.so in the current directory with a method named dfilter2d that returns a filtered version of the input.

Calling f2py from Python

The f2py program is written in Python and can be run from inside your code to compile Fortran code at runtime, as follows:

```
from numpy import f2py
with open("add.f") as sourcefile:
    sourcecode = sourcefile.read()
f2py.compile(sourcecode, modulename='add')
import add
```

The source string can be any valid Fortran code. If you want to save the extension-module source code then a suitable file-name can be provided by the `source_fn` keyword to the compile function.

Automatic extension module generation

If you want to distribute your f2py extension module, then you only need to include the .pyf file and the Fortran code. The distutils extensions in NumPy allow you to define an extension module entirely in terms of this interface file. A valid `setup.py` file allowing distribution of the `add.f` module (as part of the package `f2py_examples` so that it would be loaded as `f2py_examples.add`) is:

```
def configuration(parent_package='', top_path=None)
    from numpy.distutils.misc_util import Configuration
    config = Configuration('f2py_examples',parent_package, top_path)
    config.add_extension('add', sources=['add.pyf','add.f'])
    return config

if __name__ == '__main__':
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())
```

Installation of the new package is easy using:

```
python setup.py install
```

assuming you have the proper permissions to write to the main site- packages directory for the version of Python you are using. For the resulting package to work, you need to create a file named `__init__.py` (in the same directory as `add.pyf`). Notice the extension module is defined entirely in terms of the `add.pyf` and `add.f` files. The conversion of the .pyf file to a .c file is handled by *numpy.disutils*.

Conclusion

The interface definition file (.pyf) is how you can fine-tune the interface between Python and Fortran. There is decent documentation for f2py found in the numpy/f2py/docs directory where-ever NumPy is installed on your system (usually under site-packages). There is also more information on using f2py (including how to use it to wrap C codes) at http://www.scipy.org/Cookbook under the "Using NumPy with Other Languages" heading.

The f2py method of linking compiled code is currently the most sophisticated and integrated approach. It allows clean separation of Python with compiled code while still allowing for separate distribution of the extension module. The only draw-back is that it requires the existence of a Fortran compiler in order for a user to install the code. However, with the existence of the free-compilers g77, gfortran, and g95, as well as high-quality commercial compilers, this restriction is not particularly onerous. In my opinion, Fortran is still the easiest way to write fast and clear code for scientific computing. It handles complex numbers, and multi-dimensional indexing in the most straightforward way. Be aware, however, that some Fortran compilers will not be able to optimize code as well as good hand- written C-code.

7.2.4 Cython

Cython is a compiler for a Python dialect that adds (optional) static typing for speed, and allows mixing C or C++

code into your modules. It produces C or C++ extensions that can be compiled and imported in Python code.

If you are writing an extension module that will include quite a bit of your own algorithmic code as well, then Cython is a good match. Among its features is the ability to easily and quickly work with multidimensional arrays.

Notice that Cython is an extension-module generator only. Unlike f2py, it includes no automatic facility for compiling and linking the extension module (which must be done in the usual fashion). It does provide a modified distutils class called `build_ext` which lets you build an extension module from a `.pyx` source. Thus, you could write in a `setup.py` file:

```
from Cython.Distutils import build_ext
from distutils.extension import Extension
from distutils.core import setup
import numpy

setup(name='mine', description='Nothing',
      ext_modules=[Extension('filter', ['filter.pyx'],
                             include_dirs=[numpy.get_include()])],
      cmdclass = {'build_ext':build_ext})
```

Adding the NumPy include directory is, of course, only necessary if you are using NumPy arrays in the extension module (which is what we assume you are using Cython for). The distutils extensions in NumPy also include support for automatically producing the extension-module and linking it from a `.pyx` file. It works so that if the user does not have Cython installed, then it looks for a file with the same file-name but a `.c` extension which it then uses instead of trying to produce the `.c` file again.

If you just use Cython to compile a standard Python module, then you will get a C extension module that typically runs a bit faster than the equivalent Python module. Further speed increases can be gained by using the `cdef` keyword to statically define C variables.

Let's look at two examples we've seen before to see how they might be implemented using Cython. These examples were compiled into extension modules using Cython 0.21.1.

Complex addition in Cython

Here is part of a Cython module named `add.pyx` which implements the complex addition functions we previously implemented using f2py:

```
cimport cython
cimport numpy as np
import numpy as np

# We need to initialize NumPy.
np.import_array()

#@cython.boundscheck(False)
def zadd(in1, in2):
    cdef double complex[:] a = in1.ravel()
    cdef double complex[:] b = in2.ravel()

    out = np.empty(a.shape[0], np.complex64)
    cdef double complex[:] c = out.ravel()

    for i in range(c.shape[0]):
        c[i].real = a[i].real + b[i].real
        c[i].imag = a[i].imag + b[i].imag

    return out
```

This module shows use of the `cimport` statement to load the definitions from the `numpy.pxd` header that ships with Cython. It looks like NumPy is imported twice; `cimport` only makes the NumPy C-API available, while the regular `import` causes a Python-style import at runtime and makes it possible to call into the familiar NumPy Python API.

The example also demonstrates Cython's "typed memoryviews", which are like NumPy arrays at the C level, in the sense that they are shaped and strided arrays that know their own extent (unlike a C array addressed through a bare pointer). The syntax `double complex[:]` denotes a one-dimensional array (vector) of doubles, with arbitrary strides. A contiguous array of ints would be `int[::1]`, while a matrix of floats would be `float[:, :]`.

Shown commented is the `cython.boundscheck` decorator, which turns bounds-checking for memory view accesses on or off on a per-function basis. We can use this to further speed up our code, at the expense of safety (or a manual check prior to entering the loop).

Other than the view syntax, the function is immediately readable to a Python programmer. Static typing of the variable `i` is implicit. Instead of the view syntax, we could also have used Cython's special NumPy array syntax, but the view syntax is preferred.

Image filter in Cython

The two-dimensional example we created using Fortran is just as easy to write in Cython:

```
cimport numpy as np
import numpy as np

np.import_array()

def filter(img):
    cdef double[:, :] a = np.asarray(img, dtype=np.double)
    out = np.zeros(img.shape, dtype=np.double)
    cdef double[:, ::1] b = out

    cdef np.npy_intp i, j

    for i in range(1, a.shape[0] - 1):
        for j in range(1, a.shape[1] - 1):
            b[i, j] = (a[i, j]
                       + .5 * (  a[i-1, j] + a[i+1, j]
                               + a[i, j-1] + a[i, j+1])
                       + .25 * (  a[i-1, j-1] + a[i-1, j+1]
                                + a[i+1, j-1] + a[i+1, j+1]))

    return out
```

This 2-d averaging filter runs quickly because the loop is in C and the pointer computations are done only as needed. If the code above is compiled as a module `image`, then a 2-d image, `img`, can be filtered using this code very quickly using:

```
import image
out = image.filter(img)
```

Regarding the code, two things are of note: firstly, it is impossible to return a memory view to Python. Instead, a NumPy array `out` is first created, and then a view `b` onto this array is used for the computation. Secondly, the view `b` is typed `double[:, ::1]`. This means 2-d array with contiguous rows, i.e., C matrix order. Specifying the order explicitly can speed up some algorithms since they can skip stride computations.

Conclusion

Cython is the extension mechanism of choice for several scientific Python libraries, including Scipy, Pandas, SAGE, scikit-image and scikit-learn, as well as the XML processing library LXML. The language and compiler are well-maintained.

There are several disadvantages of using Cython:

1. When coding custom algorithms, and sometimes when wrapping existing C libraries, some familiarity with C is required. In particular, when using C memory management (`malloc` and friends), it's easy to introduce memory leaks. However, just compiling a Python module renamed to `.pyx` can already speed it up, and adding a few type declarations can give dramatic speedups in some code.
2. It is easy to lose a clean separation between Python and C which makes re-using your C-code for other non-Python-related projects more difficult.
3. The C-code generated by Cython is hard to read and modify (and typically compiles with annoying but harmless warnings).

One big advantage of Cython-generated extension modules is that they are easy to distribute. In summary, Cython is a very capable tool for either gluing C code or generating an extension module quickly and should not be over-looked. It is especially useful for people that can't or won't write C or Fortran code.

7.2.5 ctypes

Ctypes is a Python extension module, included in the stdlib, that allows you to call an arbitrary function in a shared library directly from Python. This approach allows you to interface with C-code directly from Python. This opens up an enormous number of libraries for use from Python. The drawback, however, is that coding mistakes can lead to ugly program crashes very easily (just as can happen in C) because there is little type or bounds checking done on the parameters. This is especially true when array data is passed in as a pointer to a raw memory location. The responsibility is then on you that the subroutine will not access memory outside the actual array area. But, if you don't mind living a little dangerously ctypes can be an effective tool for quickly taking advantage of a large shared library (or writing extended functionality in your own shared library).

Because the ctypes approach exposes a raw interface to the compiled code it is not always tolerant of user mistakes. Robust use of the ctypes module typically involves an additional layer of Python code in order to check the data types and array bounds of objects passed to the underlying subroutine. This additional layer of checking (not to mention the conversion from ctypes objects to C-data-types that ctypes itself performs), will make the interface slower than a hand-written extension-module interface. However, this overhead should be negligible if the C-routine being called is doing any significant amount of work. If you are a great Python programmer with weak C skills, ctypes is an easy way to write a useful interface to a (shared) library of compiled code.

To use ctypes you must

1. Have a shared library.
2. Load the shared library.
3. Convert the python objects to ctypes-understood arguments.
4. Call the function from the library with the ctypes arguments.

Having a shared library

There are several requirements for a shared library that can be used with ctypes that are platform specific. This guide assumes you have some familiarity with making a shared library on your system (or simply have a shared library available to you). Items to remember are:

- A shared library must be compiled in a special way (*e.g.* using the `-shared` flag with gcc).
- On some platforms (*e.g.* Windows) , a shared library requires a .def file that specifies the functions to be exported. For example a mylib.def file might contain:

```
LIBRARY mylib.dll
EXPORTS
cool_function1
cool_function2
```

  Alternatively, you may be able to use the storage-class specifier `__declspec(dllexport)` in the C-definition of the function to avoid the need for this `.def` file.

There is no standard way in Python distutils to create a standard shared library (an extension module is a "special" shared library Python understands) in a cross-platform manner. Thus, a big disadvantage of ctypes at the time of writing this book is that it is difficult to distribute in a cross-platform manner a Python extension that uses ctypes and includes your own code which should be compiled as a shared library on the users system.

Loading the shared library

A simple, but robust way to load the shared library is to get the absolute path name and load it using the cdll object of ctypes:

```
lib = ctypes.cdll[<full_path_name>]
```

However, on Windows accessing an attribute of the `cdll` method will load the first DLL by that name found in the current directory or on the PATH. Loading the absolute path name requires a little finesse for cross-platform work since the extension of shared libraries varies. There is a `ctypes.util.find_library` utility available that can simplify the process of finding the library to load but it is not foolproof. Complicating matters, different platforms have different default extensions used by shared libraries (e.g. .dll – Windows, .so – Linux, .dylib – Mac OS X). This must also be taken into account if you are using ctypes to wrap code that needs to work on several platforms.

NumPy provides a convenience function called `ctypeslib.load_library` (name, path). This function takes the name of the shared library (including any prefix like 'lib' but excluding the extension) and a path where the shared library can be located. It returns a ctypes library object or raises an `OSError` if the library cannot be found or raises an `ImportError` if the ctypes module is not available. (Windows users: the ctypes library object loaded using `load_library` is always loaded assuming cdecl calling convention. See the ctypes documentation under `ctypes.windll` and/or `ctypes.oledll` for ways to load libraries under other calling conventions).

The functions in the shared library are available as attributes of the ctypes library object (returned from `ctypeslib.load_library`) or as items using `lib['func_name']` syntax. The latter method for retrieving a function name is particularly useful if the function name contains characters that are not allowable in Python variable names.

Converting arguments

Python ints/longs, strings, and unicode objects are automatically converted as needed to equivalent ctypes arguments The None object is also converted automatically to a NULL pointer. All other Python objects must be converted to ctypes-specific types. There are two ways around this restriction that allow ctypes to integrate with other objects.

1. Don't set the argtypes attribute of the function object and define an `_as_parameter_` method for the object you want to pass in. The `_as_parameter_` method must return a Python int which will be passed directly to the function.
2. Set the argtypes attribute to a list whose entries contain objects with a classmethod named from_param that knows how to convert your object to an object that ctypes can understand (an int/long, string, unicode, or object with the `_as_parameter_` attribute).

NumPy uses both methods with a preference for the second method because it can be safer. The ctypes attribute of the ndarray returns an object that has an `_as_parameter_` attribute which returns an integer representing the address of the ndarray to which it is associated. As a result, one can pass this ctypes attribute object directly to a function expecting a pointer to the data in your ndarray. The caller must be sure that the ndarray object is of the correct type, shape, and has the correct flags set or risk nasty crashes if the data-pointer to inappropriate arrays are passed in.

To implement the second method, NumPy provides the class-factory function *ndpointer* in the `ctypeslib` module. This class-factory function produces an appropriate class that can be placed in an argtypes attribute entry of a ctypes function. The class will contain a from_param method which ctypes will use to convert any ndarray passed in to the function to a ctypes-recognized object. In the process, the conversion will perform checking on any properties of the ndarray that were specified by the user in the call to *ndpointer*. Aspects of the ndarray that can be checked include the data-type, the number-of-dimensions, the shape, and/or the state of the flags on any array passed. The return value of the from_param method is the ctypes attribute of the array which (because it contains the `_as_parameter_` attribute pointing to the array data area) can be used by ctypes directly.

The ctypes attribute of an ndarray is also endowed with additional attributes that may be convenient when passing additional information about the array into a ctypes function. The attributes **data**, **shape**, and **strides** can provide ctypes compatible types corresponding to the data-area, the shape, and the strides of the array. The data attribute returns a `c_void_p` representing a pointer to the data area. The shape and strides attributes each return an array of ctypes integers (or None representing a NULL pointer, if a 0-d array). The base ctype of the array is a ctype integer of the same size as a pointer on the platform. There are also methods `data_as({ctype})`, `shape_as(<base ctype>)`, and `strides_as(<base ctype>)`. These return the data as a ctype object of your choice and the shape/strides arrays using an underlying base type of your choice. For convenience, the `ctypeslib` module also contains `c_intp` as a ctypes integer data-type whose size is the same as the size of `c_void_p` on the platform (its value is None if ctypes is not installed).

Calling the function

The function is accessed as an attribute of or an item from the loaded shared-library. Thus, if `./mylib.so` has a function named `cool_function1` , I could access this function either as:

```
lib = numpy.ctypeslib.load_library('mylib','.')
func1 = lib.cool_function1  # or equivalently
func1 = lib['cool_function1']
```

In ctypes, the return-value of a function is set to be 'int' by default. This behavior can be changed by setting the restype attribute of the function. Use None for the restype if the function has no return value ('void'):

```
func1.restype = None
```

As previously discussed, you can also set the argtypes attribute of the function in order to have ctypes check the types of the input arguments when the function is called. Use the *ndpointer* factory function to generate a ready-made class for data-type, shape, and flags checking on your new function. The *ndpointer* function has the signature

ndpointer(*dtype=None*, *ndim=None*, *shape=None*, *flags=None*)

Keyword arguments with the value `None` are not checked. Specifying a keyword enforces checking of that aspect of the ndarray on conversion to a ctypes-compatible object. The dtype keyword can be any object understood as a data-type object. The ndim keyword should be an integer, and the shape keyword should be an integer or a sequence of integers. The flags keyword specifies the minimal flags that are required on any array passed in. This can be specified as a string of comma separated requirements, an integer indicating the requirement bits OR'd together, or a flags object returned from the flags attribute of an array with the necessary requirements.

Using an ndpointer class in the argtypes method can make it significantly safer to call a C function using ctypes and the data- area of an ndarray. You may still want to wrap the function in an additional Python wrapper to make it

user-friendly (hiding some obvious arguments and making some arguments output arguments). In this process, the `requires` function in NumPy may be useful to return the right kind of array from a given input.

Complete example

In this example, I will show how the addition function and the filter function implemented previously using the other approaches can be implemented using ctypes. First, the C code which implements the algorithms contains the functions `zadd`, `dadd`, `sadd`, `cadd`, and `dfilter2d`. The `zadd` function is:

```
/* Add arrays of contiguous data */
typedef struct {double real; double imag;} cdouble;
typedef struct {float real; float imag;} cfloat;
void zadd(cdouble *a, cdouble *b, cdouble *c, long n)
{
    while (n--) {
        c->real = a->real + b->real;
        c->imag = a->imag + b->imag;
        a++; b++; c++;
    }
}
```

with similar code for `cadd`, `dadd`, and `sadd` that handles complex float, double, and float data-types, respectively:

```
void cadd(cfloat *a, cfloat *b, cfloat *c, long n)
{
        while (n--) {
                c->real = a->real + b->real;
                c->imag = a->imag + b->imag;
                a++; b++; c++;
        }
}
void dadd(double *a, double *b, double *c, long n)
{
        while (n--) {
                *c++ = *a++ + *b++;
        }
}
void sadd(float *a, float *b, float *c, long n)
{
        while (n--) {
                *c++ = *a++ + *b++;
        }
}
```

The `code.c` file also contains the function `dfilter2d`:

```
/*
 * Assumes b is contiguous and has strides that are multiples of
 * sizeof(double)
 */
void
dfilter2d(double *a, double *b, ssize_t *astrides, ssize_t *dims)
{
    ssize_t i, j, M, N, S0, S1;
    ssize_t r, c, rm1, rp1, cp1, cm1;

    M = dims[0]; N = dims[1];
```

```
    S0 = astrides[0]/sizeof(double);
    S1 = astrides[1]/sizeof(double);
    for (i = 1; i < M - 1; i++) {
        r = i*S0;
        rp1 = r + S0;
        rm1 = r - S0;
        for (j = 1; j < N - 1; j++) {
            c = j*S1;
            cp1 = j + S1;
            cm1 = j - S1;
            b[i*N + j] = a[r + c] +
                (a[rp1 + c] + a[rm1 + c] +
                 a[r + cp1] + a[r + cm1])*0.5 +
                (a[rp1 + cp1] + a[rp1 + cm1] +
                 a[rm1 + cp1] + a[rm1 + cp1])*0.25;
        }
    }
}
```

A possible advantage this code has over the Fortran-equivalent code is that it takes arbitrarily strided (i.e. non-contiguous arrays) and may also run faster depending on the optimization capability of your compiler. But, it is an obviously more complicated than the simple code in `filter.f`. This code must be compiled into a shared library. On my Linux system this is accomplished using:

```
gcc -o code.so -shared code.c
```

Which creates a shared_library named code.so in the current directory. On Windows don't forget to either add `__declspec(dllexport)` in front of void on the line preceding each function definition, or write a code.def file that lists the names of the functions to be exported.

A suitable Python interface to this shared library should be constructed. To do this create a file named interface.py with the following lines at the top:

```
__all__ = ['add', 'filter2d']

import numpy as N
import os

_path = os.path.dirname('__file__')
lib = N.ctypeslib.load_library('code', _path)
_typedict = {'zadd' : complex, 'sadd' : N.single,
             'cadd' : N.csingle, 'dadd' : float}
for name in _typedict.keys():
    val = getattr(lib, name)
    val.restype = None
    _type = _typedict[name]
    val.argtypes = [N.ctypeslib.ndpointer(_type,
                      flags='aligned, contiguous'),
                    N.ctypeslib.ndpointer(_type,
                      flags='aligned, contiguous'),
                    N.ctypeslib.ndpointer(_type,
                      flags='aligned, contiguous,'\
                            'writeable'),
                    N.ctypeslib.c_intp]
```

This code loads the shared library named `code.{ext}` located in the same path as this file. It then adds a return type of void to the functions contained in the library. It also adds argument checking to the functions in the library so that ndarrays can be passed as the first three arguments along with an integer (large enough to hold a pointer on the

platform) as the fourth argument.

Setting up the filtering function is similar and allows the filtering function to be called with ndarray arguments as the first two arguments and with pointers to integers (large enough to handle the strides and shape of an ndarray) as the last two arguments.:

```
lib.dfilter2d.restype=None
lib.dfilter2d.argtypes = [N.ctypeslib.ndpointer(float, ndim=2,
                                               flags='aligned'),
                          N.ctypeslib.ndpointer(float, ndim=2,
                                 flags='aligned, contiguous,'\
                                       'writeable'),
                          ctypes.POINTER(N.ctypeslib.c_intp),
                          ctypes.POINTER(N.ctypeslib.c_intp)]
```

Next, define a simple selection function that chooses which addition function to call in the shared library based on the data-type:

```
def select(dtype):
    if dtype.char in ['?bBhHf']:
        return lib.sadd, single
    elif dtype.char in ['F']:
        return lib.cadd, csingle
    elif dtype.char in ['DG']:
        return lib.zadd, complex
    else:
        return lib.dadd, float
    return func, ntype
```

Finally, the two functions to be exported by the interface can be written simply as:

```
def add(a, b):
    requires = ['CONTIGUOUS', 'ALIGNED']
    a = N.asanyarray(a)
    func, dtype = select(a.dtype)
    a = N.require(a, dtype, requires)
    b = N.require(b, dtype, requires)
    c = N.empty_like(a)
    func(a,b,c,a.size)
    return c
```

and:

```
def filter2d(a):
    a = N.require(a, float, ['ALIGNED'])
    b = N.zeros_like(a)
    lib.dfilter2d(a, b, a.ctypes.strides, a.ctypes.shape)
    return b
```

Conclusion

Using ctypes is a powerful way to connect Python with arbitrary C-code. Its advantages for extending Python include

- clean separation of C code from Python code
  - no need to learn a new syntax except Python and C
  - allows re-use of C code

  - functionality in shared libraries written for other purposes can be obtained with a simple Python wrapper and search for the library.

- easy integration with NumPy through the ctypes attribute
- full argument checking with the ndpointer class factory

Its disadvantages include

- It is difficult to distribute an extension module made using ctypes because of a lack of support for building shared libraries in distutils (but I suspect this will change in time).
- You must have shared-libraries of your code (no static libraries).
- Very little support for C++ code and its different library-calling conventions. You will probably need a C wrapper around C++ code to use with ctypes (or just use Boost.Python instead).

Because of the difficulty in distributing an extension module made using ctypes, f2py and Cython are still the easiest ways to extend Python for package creation. However, ctypes is in some cases a useful alternative. This should bring more features to ctypes that should eliminate the difficulty in extending Python and distributing the extension using ctypes.

7.2.6 Additional tools you may find useful

These tools have been found useful by others using Python and so are included here. They are discussed separately because they are either older ways to do things now handled by f2py, Cython, or ctypes (SWIG, PyFort) or because I don't know much about them (SIP, Boost). I have not added links to these methods because my experience is that you can find the most relevant link faster using Google or some other search engine, and any links provided here would be quickly dated. Do not assume that just because it is included in this list, I don't think the package deserves your attention. I'm including information about these packages because many people have found them useful and I'd like to give you as many options as possible for tackling the problem of easily integrating your code.

SWIG

Simplified Wrapper and Interface Generator (SWIG) is an old and fairly stable method for wrapping C/C++-libraries to a large variety of other languages. It does not specifically understand NumPy arrays but can be made useable with NumPy through the use of typemaps. There are some sample typemaps in the numpy/tools/swig directory under numpy.i together with an example module that makes use of them. SWIG excels at wrapping large C/C++ libraries because it can (almost) parse their headers and auto-produce an interface. Technically, you need to generate a `.i` file that defines the interface. Often, however, this `.i` file can be parts of the header itself. The interface usually needs a bit of tweaking to be very useful. This ability to parse C/C++ headers and auto-generate the interface still makes SWIG a useful approach to adding functionalilty from C/C++ into Python, despite the other methods that have emerged that are more targeted to Python. SWIG can actually target extensions for several languages, but the typemaps usually have to be language-specific. Nonetheless, with modifications to the Python-specific typemaps, SWIG can be used to interface a library with other languages such as Perl, Tcl, and Ruby.

My experience with SWIG has been generally positive in that it is relatively easy to use and quite powerful. I used to use it quite often before becoming more proficient at writing C-extensions. However, I struggled writing custom interfaces with SWIG because it must be done using the concept of typemaps which are not Python specific and are written in a C-like syntax. Therefore, I tend to prefer other gluing strategies and would only attempt to use SWIG to wrap a very-large C/C++ library. Nonetheless, there are others who use SWIG quite happily.

SIP

SIP is another tool for wrapping C/C++ libraries that is Python specific and appears to have very good support for C++. Riverbank Computing developed SIP in order to create Python bindings to the QT library. An interface file must

be written to generate the binding, but the interface file looks a lot like a C/C++ header file. While SIP is not a full C++ parser, it understands quite a bit of C++ syntax as well as its own special directives that allow modification of how the Python binding is accomplished. It also allows the user to define mappings between Python types and C/C++ structures and classes.

Boost Python

Boost is a repository of C++ libraries and Boost.Python is one of those libraries which provides a concise interface for binding C++ classes and functions to Python. The amazing part of the Boost.Python approach is that it works entirely in pure C++ without introducing a new syntax. Many users of C++ report that Boost.Python makes it possible to combine the best of both worlds in a seamless fashion. I have not used Boost.Python because I am not a big user of C++ and using Boost to wrap simple C-subroutines is usually over-kill. It's primary purpose is to make C++ classes available in Python. So, if you have a set of C++ classes that need to be integrated cleanly into Python, consider learning about and using Boost.Python.

PyFort

PyFort is a nice tool for wrapping Fortran and Fortran-like C-code into Python with support for Numeric arrays. It was written by Paul Dubois, a distinguished computer scientist and the very first maintainer of Numeric (now retired). It is worth mentioning in the hopes that somebody will update PyFort to work with NumPy arrays as well which now support either Fortran or C-style contiguous arrays.

7.3 Writing your own ufunc

I have the Power!
— *He-Man*

7.3.1 Creating a new universal function

Before reading this, it may help to familiarize yourself with the basics of C extensions for Python by reading/skimming the tutorials in Section 1 of Extending and Embedding the Python Interpreter and in How to extend NumPy

The umath module is a computer-generated C-module that creates many ufuncs. It provides a great many examples of how to create a universal function. Creating your own ufunc that will make use of the ufunc machinery is not difficult either. Suppose you have a function that you want to operate element-by-element over its inputs. By creating a new ufunc you will obtain a function that handles

- broadcasting
- N-dimensional looping
- automatic type-conversions with minimal memory usage
- optional output arrays

It is not difficult to create your own ufunc. All that is required is a 1-d loop for each data-type you want to support. Each 1-d loop must have a specific signature, and only ufuncs for fixed-size data-types can be used. The function call used to create a new ufunc to work on built-in data-types is given below. A different mechanism is used to register ufuncs for user-defined data-types.

In the next several sections we give example code that can be easily modified to create your own ufuncs. The examples are successively more complete or complicated versions of the logit function, a common function in statistical

modeling. Logit is also interesting because, due to the magic of IEEE standards (specifically IEEE 754), all of the logit functions created below automatically have the following behavior.

```
>>> logit(0)
-inf
>>> logit(1)
inf
>>> logit(2)
nan
>>> logit(-2)
nan
```

This is wonderful because the function writer doesn't have to manually propagate infs or nans.

7.3.2 Example Non-ufunc extension

For comparison and general edification of the reader we provide a simple implementation of a C extension of logit that uses no numpy.

To do this we need two files. The first is the C file which contains the actual code, and the second is the setup.py file used to create the module.

```
#include <Python.h>
#include <math.h>

/*
 * spammodule.c
 * This is the C code for a non-numpy Python extension to
 * define the logit function, where logit(p) = log(p/(1-p)).
 * This function will not work on numpy arrays automatically.
 * numpy.vectorize must be called in python to generate
 * a numpy-friendly function.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */


/* This declares the logit function */
static PyObject* spam_logit(PyObject *self, PyObject *args);


/*
 * This tells Python what methods this module has.
 * See the Python-C API for more information.
 */
static PyMethodDef SpamMethods[] = {
    {"logit",
        spam_logit,
        METH_VARARGS, "compute logit"},
    {NULL, NULL, 0, NULL}
};


/*
 * This actually defines the logit function for
```

```
 * input args from Python.
 */

static PyObject* spam_logit(PyObject *self, PyObject *args)
{
    double p;

    /* This parses the Python argument into a double */
    if(!PyArg_ParseTuple(args, "d", &p)) {
        return NULL;
    }

    /* THE ACTUAL LOGIT FUNCTION */
    p = p/(1-p);
    p = log(p);

    /*This builds the answer back into a python object */
    return Py_BuildValue("d", p);
}


/* This initiates the module using the above definitions. */
#if PY_VERSION_HEX >= 0x03000000
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "spam",
    NULL,
    -1,
    SpamMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_spam(void)
{
    PyObject *m;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }
    return m;
}
#else
PyMODINIT_FUNC initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL) {
        return;
    }
}
#endif
```

To use the setup.py file, place setup.py and spammodule.c in the same folder. Then python setup.py build will build the module to import, or setup.py install will install the module to your site-packages directory.

```
'''
    setup.py file for spammodule.c

    Calling
    $python setup.py build_ext --inplace
    will build the extension library in the current file.

    Calling
    $python setup.py build
    will build a file that looks like ./build/lib*, where
    lib* is a file that begins with lib. The library will
    be in this file and end with a C library extension,
    such as .so

    Calling
    $python setup.py install
    will install the module in your site-packages file.

    See the distutils section of
    'Extending and Embedding the Python Interpreter'
    at docs.python.org for more information.
'''


from distutils.core import setup, Extension

module1 = Extension('spam', sources=['spammodule.c'],
                        include_dirs=['/usr/local/lib'])

setup(name = 'spam',
        version='1.0',
        description='This is my spam package',
        ext_modules = [module1])
```

Once the spam module is imported into python, you can call logit via spam.logit. Note that the function used above cannot be applied as-is to numpy arrays. To do so we must call numpy.vectorize on it. For example, if a python interpreter is opened in the file containing the spam library or spam has been installed, one can perform the following commands:

```
>>> import numpy as np
>>> import spam
>>> spam.logit(0)
-inf
>>> spam.logit(1)
inf
>>> spam.logit(0.5)
0.0
>>> x = np.linspace(0,1,10)
>>> spam.logit(x)
TypeError: only length-1 arrays can be converted to Python scalars
>>> f = np.vectorize(spam.logit)
>>> f(x)
array([       -inf, -2.07944154, -1.25276297, -0.69314718, -0.22314355,
    0.22314355,  0.69314718,  1.25276297,  2.07944154,         inf])
```

THE RESULTING LOGIT FUNCTION IS NOT FAST! numpy.vectorize simply loops over spam.logit. The loop is done at the C level, but the numpy array is constantly being parsed and build back up. This is expensive. When the author compared numpy.vectorize(spam.logit) against the logit ufuncs constructed below, the logit ufuncs were almost

exactly 4 times faster. Larger or smaller speedups are, of course, possible depending on the nature of the function.

7.3.3 Example NumPy ufunc for one dtype

For simplicity we give a ufunc for a single dtype, the 'f8' double. As in the previous section, we first give the .c file and then the setup.py file used to create the module containing the ufunc.

The place in the code corresponding to the actual computations for the ufunc are marked with /*BEGIN main ufunc computation*/ and /*END main ufunc computation*/. The code in between those lines is the primary thing that must be changed to create your own ufunc.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npy_3kcompat.h"

/*
 * single_type_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a logit function.
 *
 * In this code we only define the ufunc for
 * a single dtype. The computations that must
 * be replaced to create a ufunc for
 * a different function are marked with BEGIN
 * and END.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */

static PyMethodDef LogitMethods[] = {
        {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void double_logit(char **args, npy_intp *dimensions,
                         npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    double tmp;

    for (i = 0; i < n; i++) {
        /*BEGIN main ufunc computation*/
        tmp = *(double *)in;
        tmp /= 1-tmp;
        *((double *)out) = log(tmp);
        /*END main ufunc computation*/

        in += in_step;
```

```
        out += out_step;
    }
}

/*This a pointer to the above function*/
PyUFuncGenericFunction funcs[1] = {&double_logit};

/* These are the input and return dtypes of logit.*/
static char types[2] = {NPY_DOUBLE, NPY_DOUBLE};

static void *data[1] = {NULL};

#if PY_VERSION_HEX >= 0x03000000
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 1, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}
#else
PyMODINIT_FUNC initnpufunc(void)
{
    PyObject *m, *logit, *d;


    m = Py_InitModule("npufunc", LogitMethods);
    if (m == NULL) {
        return;
    }
```

```
    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 1, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);
}
#endif
```

This is a setup.py file for the above code. As before, the module can be build via calling python setup.py build at the command prompt, or installed to site-packages via python setup.py install.

```
'''
    setup.py file for logit.c
    Note that since this is a numpy extension
    we use numpy.distutils instead of
    distutils from the python standard library.

    Calling
    $python setup.py build_ext --inplace
    will build the extension library in the current file.

    Calling
    $python setup.py build
    will build a file that looks like ./build/lib*, where
    lib* is a file that begins with lib. The library will
    be in this file and end with a C library extension,
    such as .so

    Calling
    $python setup.py install
    will install the module in your site-packages file.

    See the distutils section of
    'Extending and Embedding the Python Interpreter'
    at docs.python.org  and the documentation
    on numpy.distutils for more information.
'''


def configuration(parent_package='', top_path=None):
    import numpy
    from numpy.distutils.misc_util import Configuration

    config = Configuration('npufunc_directory',
                           parent_package,
                           top_path)
    config.add_extension('npufunc', ['single_type_logit.c'])

    return config

if __name__ == "__main__":
    from numpy.distutils.core import setup
```

```
    setup(configuration=configuration)
```

After the above has been installed, it can be imported and used as follows.

```
>>> import numpy as np
>>> import npufunc
>>> npufunc.logit(0.5)
0.0
>>> a = np.linspace(0,1,5)
>>> npufunc.logit(a)
array([       -inf, -1.09861229,  0.        ,  1.09861229,         inf])
```

7.3.4 Example NumPy ufunc with multiple dtypes

We finally give an example of a full ufunc, with inner loops for half-floats, floats, doubles, and long doubles. As in the previous sections we first give the .c file and then the corresponding setup.py file.

The places in the code corresponding to the actual computations for the ufunc are marked with /*BEGIN main ufunc computation*/ and /*END main ufunc computation*/. The code in between those lines is the primary thing that must be changed to create your own ufunc.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"

/*
 * multi_type_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a logit function.
 *
 * Each function of the form type_logit defines the
 * logit function for a different numpy dtype. Each
 * of these functions must be modified when you
 * create your own ufunc. The computations that must
 * be replaced to create a ufunc for
 * a different function are marked with BEGIN
 * and END.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 *
 */


static PyMethodDef LogitMethods[] = {
        {NULL, NULL, 0, NULL}
};

/* The loop definitions must precede the PyMODINIT_FUNC. */

static void long_double_logit(char **args, npy_intp *dimensions,
                              npy_intp* steps, void* data)
{
```

```
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out=args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    long double tmp;

    for (i = 0; i < n; i++) {
        /*BEGIN main ufunc computation*/
        tmp = *(long double *)in;
        tmp /= 1-tmp;
        *((long double *)out) = logl(tmp);
        /*END main ufunc computation*/

        in += in_step;
        out += out_step;
    }
}

static void double_logit(char **args, npy_intp *dimensions,
                         npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    double tmp;

    for (i = 0; i < n; i++) {
        /*BEGIN main ufunc computation*/
        tmp = *(double *)in;
        tmp /= 1-tmp;
        *((double *)out) = log(tmp);
        /*END main ufunc computation*/

        in += in_step;
        out += out_step;
    }
}

static void float_logit(char **args, npy_intp *dimensions,
                        npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in=args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    float tmp;

    for (i = 0; i < n; i++) {
        /*BEGIN main ufunc computation*/
        tmp = *(float *)in;
        tmp /= 1-tmp;
        *((float *)out) = logf(tmp);
        /*END main ufunc computation*/
```

```
        in += in_step;
        out += out_step;
    }
}


static void half_float_logit(char **args, npy_intp *dimensions,
                             npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in = args[0], *out = args[1];
    npy_intp in_step = steps[0], out_step = steps[1];

    float tmp;

    for (i = 0; i < n; i++) {

        /*BEGIN main ufunc computation*/
        tmp = *(npy_half *)in;
        tmp = npy_half_to_float(tmp);
        tmp /= 1-tmp;
        tmp = logf(tmp);
        *((npy_half *)out) = npy_float_to_half(tmp);
        /*END main ufunc computation*/

        in += in_step;
        out += out_step;
    }
}


/*This gives pointers to the above functions*/
PyUFuncGenericFunction funcs[4] = {&half_float_logit,
                                   &float_logit,
                                   &double_logit,
                                   &long_double_logit};

static char types[8] = {NPY_HALF, NPY_HALF,
                NPY_FLOAT, NPY_FLOAT,
                NPY_DOUBLE,NPY_DOUBLE,
                NPY_LONGDOUBLE, NPY_LONGDOUBLE};
static void *data[4] = {NULL, NULL, NULL, NULL};

#if PY_VERSION_HEX >= 0x03000000
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
```

```
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 4, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}
#else
PyMODINIT_FUNC initnpufunc(void)
{
    PyObject *m, *logit, *d;


    m = Py_InitModule("npufunc", LogitMethods);
    if (m == NULL) {
        return;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 4, 1, 1,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);
}
#endif
```

This is a setup.py file for the above code. As before, the module can be build via calling python setup.py build at the command prompt, or installed to site-packages via python setup.py install.

```
'''
    setup.py file for logit.c
    Note that since this is a numpy extension
    we use numpy.distutils instead of
    distutils from the python standard library.

    Calling
    $python setup.py build_ext --inplace
    will build the extension library in the current file.
```

```
    Calling
    $python setup.py build
    will build a file that looks like ./build/lib*, where
    lib* is a file that begins with lib. The library will
    be in this file and end with a C library extension,
    such as .so

    Calling
    $python setup.py install
    will install the module in your site-packages file.

    See the distutils section of
    'Extending and Embedding the Python Interpreter'
    at docs.python.org  and the documentation
    on numpy.distutils for more information.
'''


def configuration(parent_package='', top_path=None):
    import numpy
    from numpy.distutils.misc_util import Configuration
    from numpy.distutils.misc_util import get_info

    #Necessary for the half-float d-type.
    info = get_info('npymath')

    config = Configuration('npufunc_directory',
                           parent_package,
                           top_path)
    config.add_extension('npufunc',
                           ['multi_type_logit.c'],
                           extra_info=info)

    return config

if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(configuration=configuration)
```

After the above has been installed, it can be imported and used as follows.

```
>>> import numpy as np
>>> import npufunc
>>> npufunc.logit(0.5)
0.0
>>> a = np.linspace(0,1,5)
>>> npufunc.logit(a)
array([       -inf, -1.09861229,  0.        ,  1.09861229,         inf])
```

7.3.5 Example NumPy ufunc with multiple arguments/return values

Our final example is a ufunc with multiple arguments. It is a modification of the code for a logit ufunc for data with a single dtype. We compute (A*B, logit(A*B)).

We only give the C code as the setup.py file is exactly the same as the setup.py file in *Example NumPy ufunc for one dtype*, except that the line

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

is replaced with

```
config.add_extension('npufunc', ['multi_arg_logit.c'])
```

The C file is given below. The ufunc generated takes two arguments A and B. It returns a tuple whose first element is A*B and whose second element is logit(A*B). Note that it automatically supports broadcasting, as well as all other properties of a ufunc.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/halffloat.h"

/*
 * multi_arg_logit.c
 * This is the C code for creating your own
 * NumPy ufunc for a multiple argument, multiple
 * return value ufunc. The places where the
 * ufunc computation is carried out are marked
 * with comments.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 *
 */


static PyMethodDef LogitMethods[] = {
        {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void double_logitprod(char **args, npy_intp *dimensions,
                             npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp n = dimensions[0];
    char *in1 = args[0], *in2 = args[1];
    char *out1 = args[2], *out2 = args[3];
    npy_intp in1_step = steps[0], in2_step = steps[1];
    npy_intp out1_step = steps[2], out2_step = steps[3];

    double tmp;

    for (i = 0; i < n; i++) {
        /*BEGIN main ufunc computation*/
        tmp = *(double *)in1;
        tmp *= *(double *)in2;
        *((double *)out1) = tmp;
        *((double *)out2) = log(tmp/(1-tmp));
        /*END main ufunc computation*/

        in1 += in1_step;
```

```
        in2 += in2_step;
        out1 += out1_step;
        out2 += out2_step;
    }
}


/*This a pointer to the above function*/
PyUFuncGenericFunction funcs[1] = {&double_logitprod};

/* These are the input and return dtypes of logit.*/

static char types[4] = {NPY_DOUBLE, NPY_DOUBLE,
                        NPY_DOUBLE, NPY_DOUBLE};


static void *data[1] = {NULL};

#if PY_VERSION_HEX >= 0x03000000
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "npufunc",
    NULL,
    -1,
    LogitMethods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC PyInit_npufunc(void)
{
    PyObject *m, *logit, *d;
    m = PyModule_Create(&moduledef);
    if (!m) {
        return NULL;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 1, 2, 2,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);

    return m;
}
#else
PyMODINIT_FUNC initnpufunc(void)
{
    PyObject *m, *logit, *d;
```

```
    m = Py_InitModule("npufunc", LogitMethods);
    if (m == NULL) {
        return;
    }

    import_array();
    import_umath();

    logit = PyUFunc_FromFuncAndData(funcs, data, types, 1, 2, 2,
                                    PyUFunc_None, "logit",
                                    "logit_docstring", 0);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "logit", logit);
    Py_DECREF(logit);
}
#endif
```

7.3.6 Example NumPy ufunc with structured array dtype arguments

This example shows how to create a ufunc for a structured array dtype. For the example we show a trivial ufunc for adding two arrays with dtype 'u8,u8,u8'. The process is a bit different from the other examples since a call to PyUFunc_FromFuncAndData doesn't fully register ufuncs for custom dtypes and structured array dtypes. We need to also call PyUFunc_RegisterLoopForDescr to finish setting up the ufunc.

We only give the C code as the setup.py file is exactly the same as the setup.py file in *Example NumPy ufunc for one dtype*, except that the line

```
config.add_extension('npufunc', ['single_type_logit.c'])
```

is replaced with

```
config.add_extension('npufunc', ['add_triplet.c'])
```

The C file is given below.

```
#include "Python.h"
#include "math.h"
#include "numpy/ndarraytypes.h"
#include "numpy/ufuncobject.h"
#include "numpy/npy_3kcompat.h"


/*
 * add_triplet.c
 * This is the C code for creating your own
 * NumPy ufunc for a structured array dtype.
 *
 * Details explaining the Python-C API can be found under
 * 'Extending and Embedding' and 'Python/C API' at
 * docs.python.org .
 */

static PyMethodDef StructUfuncTestMethods[] = {
```

```
    {NULL, NULL, 0, NULL}
};

/* The loop definition must precede the PyMODINIT_FUNC. */

static void add_uint64_triplet(char **args, npy_intp *dimensions,
                               npy_intp* steps, void* data)
{
    npy_intp i;
    npy_intp is1=steps[0];
    npy_intp is2=steps[1];
    npy_intp os=steps[2];
    npy_intp n=dimensions[0];
    uint64_t *x, *y, *z;

    char *i1=args[0];
    char *i2=args[1];
    char *op=args[2];

    for (i = 0; i < n; i++) {

        x = (uint64_t*)i1;
        y = (uint64_t*)i2;
        z = (uint64_t*)op;

        z[0] = x[0] + y[0];
        z[1] = x[1] + y[1];
        z[2] = x[2] + y[2];

        i1 += is1;
        i2 += is2;
        op += os;
    }
}

/* This a pointer to the above function */
PyUFuncGenericFunction funcs[1] = {&add_uint64_triplet};

/* These are the input and return dtypes of add_uint64_triplet. */
static char types[3] = {NPY_UINT64, NPY_UINT64, NPY_UINT64};

static void *data[1] = {NULL};

#if defined(NPY_PY3K)
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "struct_ufunc_test",
    NULL,
    -1,
    StructUfuncTestMethods,
    NULL,
    NULL,
    NULL,
    NULL
};
#endif

#if defined(NPY_PY3K)
```

```
PyMODINIT_FUNC PyInit_struct_ufunc_test(void)
#else
PyMODINIT_FUNC initstruct_ufunc_test(void)
#endif
{
    PyObject *m, *add_triplet, *d;
    PyObject *dtype_dict;
    PyArray_Descr *dtype;
    PyArray_Descr *dtypes[3];

#if defined(NPY_PY3K)
    m = PyModule_Create(&moduledef);
#else
    m = Py_InitModule("struct_ufunc_test", StructUfuncTestMethods);
#endif

    if (m == NULL) {
#if defined(NPY_PY3K)
        return NULL;
#else
        return;
#endif
    }

    import_array();
    import_umath();

    /* Create a new ufunc object */
    add_triplet = PyUFunc_FromFuncAndData(NULL, NULL, NULL, 0, 2, 1,
                                          PyUFunc_None, "add_triplet",
                                          "add_triplet_docstring", 0);

    dtype_dict = Py_BuildValue("[(s, s), (s, s), (s, s)]",
        "f0", "u8", "f1", "u8", "f2", "u8");
    PyArray_DescrConverter(dtype_dict, &dtype);
    Py_DECREF(dtype_dict);

    dtypes[0] = dtype;
    dtypes[1] = dtype;
    dtypes[2] = dtype;

    /* Register ufunc for structured dtype */
    PyUFunc_RegisterLoopForDescr(add_triplet,
                                 dtype,
                                 &add_uint64_triplet,
                                 dtypes,
                                 NULL);

    d = PyModule_GetDict(m);

    PyDict_SetItemString(d, "add_triplet", add_triplet);
    Py_DECREF(add_triplet);
#if defined(NPY_PY3K)
    return m;
#endif
}
```

7.3.7 PyUFunc_FromFuncAndData Specification

What follows is the full specification of PyUFunc_FromFuncAndData, which automatically generates a ufunc from a C function with the correct signature.

PyObject ***PyUFunc_FromFuncAndData**(PyUFuncGenericFunction* *func*, void** *data*, char* *types*, int *ntypes*, int *nin*, int *nout*, int *identity*, char* *name*, char* *doc*, int *unused*)

func

A pointer to an array of 1-d functions to use. This array must be at least ntypes long. Each entry in the array must be a `PyUFuncGenericFunction` function. This function has the following signature. An example of a valid 1d loop function is also given.

void **loop1d**(char** *args*, npy_intp* *dimensions*, npy_intp* *steps*, void* *data*)

args

An array of pointers to the actual data for the input and output arrays. The input arguments are given first followed by the output arguments.

dimensions

A pointer to the size of the dimension over which this function is looping.

steps

A pointer to the number of bytes to jump to get to the next element in this dimension for each of the input and output arguments.

data

Arbitrary data (extra arguments, function names, *etc.*) that can be stored with the ufunc and will be passed in when it is called.

```
static void
double_add(char **args, npy_intp *dimensions, npy_intp *steps,
   void *extra)
{
    npy_intp i;
    npy_intp is1 = steps[0], is2 = steps[1];
    npy_intp os = steps[2], n = dimensions[0];
    char *i1 = args[0], *i2 = args[1], *op = args[2];
    for (i = 0; i < n; i++) {
        *((double *)op) = *((double *)i1) +
                          *((double *)i2);
        i1 += is1;
        i2 += is2;
        op += os;
     }
}
```

data

An array of data. There should be ntypes entries (or NULL) — one for every loop function defined for this ufunc. This data will be passed in to the 1-d loop. One common use of this data variable is to pass in an actual function to call to compute the result when a generic 1-d loop (e.g. `PyUFunc_d_d`) is being used.

types

An array of type-number signatures (type `char`). This array should be of size (nin+nout)*ntypes and contain the data-types for the corresponding 1-d loop. The inputs should be first followed by the

outputs. For example, suppose I have a ufunc that supports 1 integer and 1 double 1-d loop (length-2 func and data arrays) that takes 2 inputs and returns 1 output that is always a complex double, then the types array would be

```
static char types[3] = {NPY_INT, NPY_DOUBLE, NPY_CDOUBLE}
```

The bit-width names can also be used (e.g. `NPY_INT32`, `NPY_COMPLEX128`) if desired.

ntypes

The number of data-types supported. This is equal to the number of 1-d loops provided.

nin

The number of input arguments.

nout

The number of output arguments.

identity

Either `PyUFunc_One`, `PyUFunc_Zero`, `PyUFunc_None`. This specifies what should be returned when an empty array is passed to the reduce method of the ufunc.

name

A `NULL` -terminated string providing the name of this ufunc (should be the Python name it will be called).

doc

A documentation string for this ufunc (will be used in generating the response to `{ufunc_name}.__doc__`). Do not include the function signature or the name as this is generated automatically.

unused

Unused; kept for compatibility. Just set it to zero.

The returned ufunc object is a callable Python object. It should be placed in a (module) dictionary under the same name as was used in the name argument to the ufunc-creation routine. The following example is adapted from the umath module

```
static PyUFuncGenericFunction atan2_functions[] = {
                      PyUFunc_ff_f, PyUFunc_dd_d,
                      PyUFunc_gg_g, PyUFunc_OO_O_method};
static void* atan2_data[] = {
                      (void *)atan2f,(void *) atan2,
                      (void *)atan2l,(void *)"arctan2"};
static char atan2_signatures[] = {
               NPY_FLOAT, NPY_FLOAT, NPY_FLOAT,
               NPY_DOUBLE, NPY_DOUBLE, NPY_DOUBLE,
               NPY_LONGDOUBLE, NPY_LONGDOUBLE, NPY_LONGDOUBLE
               NPY_OBJECT, NPY_OBJECT, NPY_OBJECT};
...
/* in the module initialization code */
PyObject *f, *dict, *module;
...
dict = PyModule_GetDict(module);
...
f = PyUFunc_FromFuncAndData(atan2_functions,
    atan2_data, atan2_signatures, 4, 2, 1,
    PyUFunc_None, "arctan2",
```

```
    "a safe and correct arctan(x1/x2)", 0);
PyDict_SetItemString(dict, "arctan2", f);
Py_DECREF(f);
...
```

7.4 Beyond the Basics

The voyage of discovery is not in seeking new landscapes but in having new eyes.
— *Marcel Proust*

Discovery is seeing what everyone else has seen and thinking what no one else has thought.
— *Albert Szent-Gyorgi*

7.4.1 Iterating over elements in the array

Basic Iteration

One common algorithmic requirement is to be able to walk over all elements in a multidimensional array. The array iterator object makes this easy to do in a generic way that works for arrays of any dimension. Naturally, if you know the number of dimensions you will be using, then you can always write nested for loops to accomplish the iteration. If, however, you want to write code that works with any number of dimensions, then you can make use of the array iterator. An array iterator object is returned when accessing the .flat attribute of an array.

Basic usage is to call `PyArray_IterNew ( array )` where array is an ndarray object (or one of its sub-classes). The returned object is an array-iterator object (the same object returned by the .flat attribute of the ndarray). This object is usually cast to PyArrayIterObject* so that its members can be accessed. The only members that are needed are `iter->size` which contains the total size of the array, `iter->index`, which contains the current 1-d index into the array, and `iter->dataptr` which is a pointer to the data for the current element of the array. Sometimes it is also useful to access `iter->ao` which is a pointer to the underlying ndarray object.

After processing data at the current element of the array, the next element of the array can be obtained using the macro `PyArray_ITER_NEXT ( iter )`. The iteration always proceeds in a C-style contiguous fashion (last index varying the fastest). The `PyArray_ITER_GOTO ( iter, destination )` can be used to jump to a particular point in the array, where `destination` is an array of npy_intp data-type with space to handle at least the number of dimensions in the underlying array. Occasionally it is useful to use `PyArray_ITER_GOTO1D ( iter, index )` which will jump to the 1-d index given by the value of `index`. The most common usage, however, is given in the following example.

```
PyObject *obj; /* assumed to be some ndarray object */
PyArrayIterObject *iter;
...
iter = (PyArrayIterObject *)PyArray_IterNew(obj);
if (iter == NULL) goto fail;   /* Assume fail has clean-up code */
while (iter->index < iter->size) {
    /* do something with the data at it->dataptr */
    PyArray_ITER_NEXT(it);
}
...
```

You can also use `PyArrayIter_Check ( obj )` to ensure you have an iterator object and `PyArray_ITER_RESET ( iter )` to reset an iterator object back to the beginning of the array.

It should be emphasized at this point that you may not need the array iterator if your array is already contiguous (using an array iterator will work but will be slower than the fastest code you could write). The major purpose of array iterators is to encapsulate iteration over N-dimensional arrays with arbitrary strides. They are used in many, many places in the NumPy source code itself. If you already know your array is contiguous (Fortran or C), then simply adding the element- size to a running pointer variable will step you through the array very efficiently. In other words, code like this will probably be faster for you in the contiguous case (assuming doubles).

```
npy_intp size;
double *dptr;  /* could make this any variable type */
size = PyArray_SIZE(obj);
dptr = PyArray_DATA(obj);
while(size--) {
   /* do something with the data at dptr */
   dptr++;
}
```

Iterating over all but one axis

A common algorithm is to loop over all elements of an array and perform some function with each element by issuing a function call. As function calls can be time consuming, one way to speed up this kind of algorithm is to write the function so it takes a vector of data and then write the iteration so the function call is performed for an entire dimension of data at a time. This increases the amount of work done per function call, thereby reducing the function-call overhead to a small(er) fraction of the total time. Even if the interior of the loop is performed without a function call it can be advantageous to perform the inner loop over the dimension with the highest number of elements to take advantage of speed enhancements available on micro- processors that use pipelining to enhance fundmental operations.

The `PyArray_IterAllButAxis ( array, &dim )` constructs an iterator object that is modified so that it will not iterate over the dimension indicated by dim. The only restriction on this iterator object, is that the `PyArray_Iter_GOTO1D ( it, ind )` macro cannot be used (thus flat indexing won't work either if you pass this object back to Python — so you shouldn't do this). Note that the returned object from this routine is still usually cast to PyArrayIterObject *. All that's been done is to modify the strides and dimensions of the returned iterator to simulate iterating over array[...,0,...] where 0 is placed on the dim^{th} dimension. If dim is negative, then the dimension with the largest axis is found and used.

Iterating over multiple arrays

Very often, it is desirable to iterate over several arrays at the same time. The universal functions are an example of this kind of behavior. If all you want to do is iterate over arrays with the same shape, then simply creating several iterator objects is the standard procedure. For example, the following code iterates over two arrays assumed to be the same shape and size (actually obj1 just has to have at least as many total elements as does obj2):

```
/* It is already assumed that obj1 and obj2
   are ndarrays of the same shape and size.
*/
iter1 = (PyArrayIterObject *)PyArray_IterNew(obj1);
if (iter1 == NULL) goto fail;
iter2 = (PyArrayIterObject *)PyArray_IterNew(obj2);
if (iter2 == NULL) goto fail;  /* assume iter1 is DECREF'd at fail */
while (iter2->index < iter2->size)  {
    /* process with iter1->dataptr and iter2->dataptr */
    PyArray_ITER_NEXT(iter1);
```

```
    PyArray_ITER_NEXT(iter2);
}
```

Broadcasting over multiple arrays

When multiple arrays are involved in an operation, you may want to use the same broadcasting rules that the math operations (*i.e.* the ufuncs) use. This can be done easily using the `PyArrayMultiIterObject`. This is the object returned from the Python command numpy.broadcast and it is almost as easy to use from C. The function `PyArray_MultiIterNew ( n, ... )` is used (with `n` input objects in place of `...`). The input objects can be arrays or anything that can be converted into an array. A pointer to a PyArrayMultiIterObject is returned. Broadcasting has already been accomplished which adjusts the iterators so that all that needs to be done to advance to the next element in each array is for PyArray_ITER_NEXT to be called for each of the inputs. This incrementing is automatically performed by `PyArray_MultiIter_NEXT ( obj )` macro (which can handle a multiterator `obj` as either a `PyArrayMultiObject *` or a `PyObject *`). The data from input number `i` is available using `PyArray_MultiIter_DATA ( obj, i )` and the total (broadcasted) size as `PyArray_MultiIter_SIZE ( obj)`. An example of using this feature follows.

```
mobj = PyArray_MultiIterNew(2, obj1, obj2);
size = PyArray_MultiIter_SIZE(obj);
while(size--) {
    ptr1 = PyArray_MultiIter_DATA(mobj, 0);
    ptr2 = PyArray_MultiIter_DATA(mobj, 1);
    /* code using contents of ptr1 and ptr2 */
    PyArray_MultiIter_NEXT(mobj);
}
```

The function `PyArray_RemoveSmallest ( multi )` can be used to take a multi-iterator object and adjust all the iterators so that iteration does not take place over the largest dimension (it makes that dimension of size 1). The code being looped over that makes use of the pointers will very-likely also need the strides data for each of the iterators. This information is stored in multi->iters[i]->strides.

There are several examples of using the multi-iterator in the NumPy source code as it makes N-dimensional broadcasting-code very simple to write. Browse the source for more examples.

7.4.2 User-defined data-types

NumPy comes with 24 builtin data-types. While this covers a large majority of possible use cases, it is conceivable that a user may have a need for an additional data-type. There is some support for adding an additional data-type into the NumPy system. This additional data- type will behave much like a regular data-type except ufuncs must have 1-d loops registered to handle it separately. Also checking for whether or not other data-types can be cast "safely" to and from this new type or not will always return "can cast" unless you also register which types your new data-type can be cast to and from. Adding data-types is one of the less well-tested areas for NumPy 1.0, so there may be bugs remaining in the approach. Only add a new data-type if you can't do what you want to do using the OBJECT or VOID data-types that are already available. As an example of what I consider a useful application of the ability to add data-types is the possibility of adding a data-type of arbitrary precision floats to NumPy.

Adding the new data-type

To begin to make use of the new data-type, you need to first define a new Python type to hold the scalars of your new data-type. It should be acceptable to inherit from one of the array scalars if your new type has a binary compatible layout. This will allow your new data type to have the methods and attributes of array scalars. New data- types

must have a fixed memory size (if you want to define a data-type that needs a flexible representation, like a variable-precision number, then use a pointer to the object as the data-type). The memory layout of the object structure for the new Python type must be PyObject_HEAD followed by the fixed-size memory needed for the data- type. For example, a suitable structure for the new Python type is:

```
typedef struct {
   PyObject_HEAD;
   some_data_type obval;
   /* the name can be whatever you want */
} PySomeDataTypeObject;
```

After you have defined a new Python type object, you must then define a new `PyArray_Descr` structure whose typeobject member will contain a pointer to the data-type you've just defined. In addition, the required functions in the ".f" member must be defined: nonzero, copyswap, copyswapn, setitem, getitem, and cast. The more functions in the ".f" member you define, however, the more useful the new data-type will be. It is very important to initialize unused functions to NULL. This can be achieved using `PyArray_InitArrFuncs` (f).

Once a new `PyArray_Descr` structure is created and filled with the needed information and useful functions you call `PyArray_RegisterDataType` (new_descr). The return value from this call is an integer providing you with a unique type_number that specifies your data-type. This type number should be stored and made available by your module so that other modules can use it to recognize your data-type (the other mechanism for finding a user-defined data-type number is to search based on the name of the type-object associated with the data-type using `PyArray_TypeNumFromName`).

Registering a casting function

You may want to allow builtin (and other user-defined) data-types to be cast automatically to your data-type. In order to make this possible, you must register a casting function with the data-type you want to be able to cast from. This requires writing low-level casting functions for each conversion you want to support and then registering these functions with the data-type descriptor. A low-level casting function has the signature.

void **castfunc**(void* *from*, void* *to*, npy_intp *n*, void* *fromarr*, void* *toarr*)

Cast `n` elements `from` one type `to` another. The data to cast from is in a contiguous, correctly-swapped and aligned chunk of memory pointed to by from. The buffer to cast to is also contiguous, correctly-swapped and aligned. The fromarr and toarr arguments should only be used for flexible-element-sized arrays (string, unicode, void).

An example castfunc is:

```
static void
double_to_float(double *from, float* to, npy_intp n,
       void* ig1, void* ig2);
while (n--) {
      (*to++) = (double) *(from++);
}
```

This could then be registered to convert doubles to floats using the code:

```
doub = PyArray_DescrFromType(NPY_DOUBLE);
PyArray_RegisterCastFunc(doub, NPY_FLOAT,
     (PyArray_VectorUnaryFunc *)double_to_float);
Py_DECREF(doub);
```

Registering coercion rules

By default, all user-defined data-types are not presumed to be safely castable to any builtin data-types. In addition builtin data-types are not presumed to be safely castable to user-defined data-types. This situation limits the ability of user-defined data-types to participate in the coercion system used by ufuncs and other times when automatic coercion takes place in NumPy. This can be changed by registering data-types as safely castable from a particular data-type object. The function `PyArray_RegisterCanCast` (from_descr, totype_number, scalarkind) should be used to specify that the data-type object from_descr can be cast to the data-type with type number totype_number. If you are not trying to alter scalar coercion rules, then use `NPY_NOSCALAR` for the scalarkind argument.

If you want to allow your new data-type to also be able to share in the scalar coercion rules, then you need to specify the scalarkind function in the data-type object's ".f" member to return the kind of scalar the new data-type should be seen as (the value of the scalar is available to that function). Then, you can register data-types that can be cast to separately for each scalar kind that may be returned from your user-defined data-type. If you don't register scalar coercion handling, then all of your user-defined data-types will be seen as `NPY_NOSCALAR`.

Registering a ufunc loop

You may also want to register low-level ufunc loops for your data-type so that an ndarray of your data-type can have math applied to it seamlessly. Registering a new loop with exactly the same arg_types signature, silently replaces any previously registered loops for that data-type.

Before you can register a 1-d loop for a ufunc, the ufunc must be previously created. Then you call *`PyUFunc_RegisterLoopForType`*(...) with the information needed for the loop. The return value of this function is `0` if the process was successful and `-1` with an error condition set if it was not successful.

int **`PyUFunc_RegisterLoopForType`**(PyUFuncObject* *ufunc*, int *usertype*, PyUFuncGenericFunction *function*, int* *arg_types*, void* *data*)

ufunc

The ufunc to attach this loop to.

usertype

The user-defined type this loop should be indexed under. This number must be a user-defined type or an error occurs.

function

The ufunc inner 1-d loop. This function must have the signature as explained in Section *3* .

arg_types

(optional) If given, this should contain an array of integers of at least size ufunc.nargs containing the data-types expected by the loop function. The data will be copied into a NumPy-managed structure so the memory for this argument should be deleted after calling this function. If this is NULL, then it will be assumed that all data-types are of type usertype.

data

(optional) Specify any optional data needed by the function which will be passed when the function is called.

7.4.3 Subtyping the ndarray in C

One of the lesser-used features that has been lurking in Python since 2.2 is the ability to sub-class types in C. This facility is one of the important reasons for basing NumPy off of the Numeric code-base which was already in C. A sub-type in C allows much more flexibility with regards to memory management. Sub-typing in C is not difficult even

if you have only a rudimentary understanding of how to create new types for Python. While it is easiest to sub-type from a single parent type, sub-typing from multiple parent types is also possible. Multiple inheritance in C is generally less useful than it is in Python because a restriction on Python sub-types is that they have a binary compatible memory layout. Perhaps for this reason, it is somewhat easier to sub-type from a single parent type.

All C-structures corresponding to Python objects must begin with `PyObject_HEAD` (or `PyObject_VAR_HEAD`). In the same way, any sub-type must have a C-structure that begins with exactly the same memory layout as the parent type (or all of the parent types in the case of multiple-inheritance). The reason for this is that Python may attempt to access a member of the sub-type structure as if it had the parent structure (*i.e.* it will cast a given pointer to a pointer to the parent structure and then dereference one of it's members). If the memory layouts are not compatible, then this attempt will cause unpredictable behavior (eventually leading to a memory violation and program crash).

One of the elements in `PyObject_HEAD` is a pointer to a type-object structure. A new Python type is created by creating a new type-object structure and populating it with functions and pointers to describe the desired behavior of the type. Typically, a new C-structure is also created to contain the instance-specific information needed for each object of the type as well. For example, `&PyArray_Type` is a pointer to the type-object table for the ndarray while a `PyArrayObject *` variable is a pointer to a particular instance of an ndarray (one of the members of the ndarray structure is, in turn, a pointer to the type- object table `&PyArray_Type`). Finally `PyType_Ready` (<pointer_to_type_object>) must be called for every new Python type.

Creating sub-types

To create a sub-type, a similar procedure must be followed except only behaviors that are different require new entries in the type- object structure. All other entries can be NULL and will be filled in by `PyType_Ready` with appropriate functions from the parent type(s). In particular, to create a sub-type in C follow these steps:

1. If needed create a new C-structure to handle each instance of your type. A typical C-structure would be:

   ```
   typedef _new_struct {
       PyArrayObject base;
       /* new things here */
   } NewArrayObject;
   ```

   Notice that the full PyArrayObject is used as the first entry in order to ensure that the binary layout of instances of the new type is identical to the PyArrayObject.

2. Fill in a new Python type-object structure with pointers to new functions that will over-ride the default behavior while leaving any function that should remain the same unfilled (or NULL). The tp_name element should be different.

3. Fill in the tp_base member of the new type-object structure with a pointer to the (main) parent type object. For multiple-inheritance, also fill in the tp_bases member with a tuple containing all of the parent objects in the order they should be used to define inheritance. Remember, all parent-types must have the same C-structure for multiple inheritance to work properly.

4. Call `PyType_Ready` (<pointer_to_new_type>). If this function returns a negative number, a failure occurred and the type is not initialized. Otherwise, the type is ready to be used. It is generally important to place a reference to the new type into the module dictionary so it can be accessed from Python.

More information on creating sub-types in C can be learned by reading PEP 253 (available at http://www.python.org/dev/peps/pep-0253).

Specific features of ndarray sub-typing

Some special methods and attributes are used by arrays in order to facilitate the interoperation of sub-types with the base ndarray type.

The __array_finalize__ method

ndarray.**__array_finalize__**

Several array-creation functions of the ndarray allow specification of a particular sub-type to be created. This allows sub-types to be handled seamlessly in many routines. When a sub-type is created in such a fashion, however, neither the __new__ method nor the __init__ method gets called. Instead, the sub-type is allocated and the appropriate instance-structure members are filled in. Finally, the `__array_finalize__` attribute is looked-up in the object dictionary. If it is present and not None, then it can be either a CObject containing a pointer to a `PyArray_FinalizeFunc` or it can be a method taking a single argument (which could be None).

If the `__array_finalize__` attribute is a CObject, then the pointer must be a pointer to a function with the signature:

```
(int) (PyArrayObject *, PyObject *)
```

The first argument is the newly created sub-type. The second argument (if not NULL) is the "parent" array (if the array was created using slicing or some other operation where a clearly-distinguishable parent is present). This routine can do anything it wants to. It should return a -1 on error and 0 otherwise.

If the `__array_finalize__` attribute is not None nor a CObject, then it must be a Python method that takes the parent array as an argument (which could be None if there is no parent), and returns nothing. Errors in this method will be caught and handled.

The __array_priority__ attribute

ndarray.**__array_priority__**

This attribute allows simple but flexible determination of which sub- type should be considered "primary" when an operation involving two or more sub-types arises. In operations where different sub-types are being used, the sub-type with the largest `__array_priority__` attribute will determine the sub-type of the output(s). If two sub- types have the same `__array_priority__` then the sub-type of the first argument determines the output. The default `__array_priority__` attribute returns a value of 0.0 for the base ndarray type and 1.0 for a sub-type. This attribute can also be defined by objects that are not sub-types of the ndarray and can be used to determine which `__array_wrap__` method should be called for the return output.

The __array_wrap__ method

ndarray.**__array_wrap__**

Any class or type can define this method which should take an ndarray argument and return an instance of the type. It can be seen as the opposite of the `__array__` method. This method is used by the ufuncs (and other NumPy functions) to allow other objects to pass through. For Python >2.4, it can also be used to write a decorator that converts a function that works only with ndarrays to one that works with any type with `__array__` and `__array_wrap__` methods.

INDEX

Symbols

A

B

C

D

E

F

L

N

P

R

S

U