NumPy User Guide

Release 1.3

Written by the NumPy community

March 20, 2009

CONTENTS

This guide explains how to make use of different features of Numpy. For a detailed documentation about different functions and classes, see *NumPy Reference* (in *NumPy Reference*).

Warning: This "User Guide" is still very much work in progress; the material is not organized, and many aspects of Numpy are not covered.
More documentation for Numpy can be found on the scipy.org website.

CHAPTER

ONE

HOW TO FIND DOCUMENTATION

See Also:

Numpy-specific help functions (in *NumPy Reference*)

Note: XXX: this part is not yet written. How to find things in NumPy.

CHAPTER

TWO

NUMPY BASICS

Note: XXX: there is overlap between this text extracted from `numpy.doc` and "Guide to Numpy" chapter 2. Needs combining?

2.1 Data types

See Also:

Data type objects (in *NumPy Reference*)

Note: XXX: Combine `numpy.doc.indexing` with material from "Guide to Numpy" (section 2.1 Data-Type descriptors)? Or incorporate the material directly here?

2.1.1 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	Platform integer (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`.
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)

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 *link_here*.

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)
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` and that `float` is `np.float`. 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. See xxx for details. 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, int)
True

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

2.1.2 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.

2.2 Array creation

See Also:

Array creation routines (in *NumPy Reference*)

2.2.1 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 record arrays. Both of those are covered in their own sections.

2.2.2 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]])
```

2.2.3 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=np.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 examples 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.

2.2.4 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: PyTables
FITS: PyFITS
Others? xxx
```

Examples of formats that cannot be read directly but for which it is not hard to convert are 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 (see xxx) 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)

2.3 Indexing

See Also:

Indexing routines (in *NumPy Reference*)

Note: XXX: Combine `numpy.doc.indexing` with material section 2.2 Basic indexing? Or incorporate the material directly here? 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.

2.3.1 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.

2.3.2 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 remaining dimension of lenth 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 (a new view of the same data in other words, see xxx for details). 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 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 (see xxx for exceptions) 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.

2.3.3 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 (see xxx for more explanation of this issue).

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 (see xxx for details). 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 (see xxx for examples)

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. See xxx.

2.3.4 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]])
```

2.3.5 Indexing Multi-dimensional arrays

Things become more complex when multidimensional arrays are indexed, particularly with multidimensional index arrays. These tend to be more unusal 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. Broadcasting won't be discussed here but is discussed in detail in xxx. 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 (size of row, number index elements).

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 resulant 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.

2.3.6 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 array being indexed, or broadcastable to the same shape. 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])
```

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. As with index arrays, what is returned is a copy of the data, not a view as one gets with slices.

With broadcasting, multidimesional arrays may be the result. For example:

```
>>> b[:,5] # use a 1-D boolean that broadcasts with y
array([False, False, False,  True,  True], dtype=bool)
>>> 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.

2.3.7 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]])
```

2.3.8 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]
```

2.3.9 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[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.array([1, 1, 3, 1]) += 1
```

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.

2.3.10 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]
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.where() 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
>>> z[(1,1,1,1)]
40 # returns a single value
```

2.4 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 element-by-element, which requires two arrays to have exactly the same shape:

```
>>> 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 second example is more effective than the first, since here broadcasting moves less memory around during the multiplication (`b` is a scalar, not an array).

2.4.1 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 larger of the two is used. In other words, the smaller of two axes is 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 mismatch
```

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.

2.5 Structured arrays (aka “Record arrays”)

2.5.1 Structured Arrays (aka Record Arrays)

Introduction

Numpy provides powerful capabilities to create arrays of structs or records. These arrays permit one to manipulate the data by the structs or by fields of the struct. A simple example will show what is meant.:

```
>>> x = np.zeros((2,),dtype=('i4,f4,a10'))
>>> x[:] = [(1,2.,'Hello'),(2,3.,"World")]
>>> x
array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
     dtype=[('f0', '>i4'), ('f1', '>f4'), ('f2', '|S10')])
```

Here we have created a one-dimensional array of length 2. Each element of this array is a record that contains three items, a 32-bit integer, a 32-bit float, and a string of length 10 or less. If we index this array at the second position we get the second record:

```
>>> x[1]
(2,3.,"World")
```

The interesting aspect is that we can reference the different fields of the array simply by indexing the array with the string representing the name of the field. In this case the fields have received the default names of ‘f0’, ‘f1’ and ‘f2’.

```
>>> y = x['f1']
>>> y
array([ 2.,  3.], dtype=float32)
>>> y[:] = 2*y
```

```
>>> y
array([ 4.,  6.], dtype=float32)
>>> x
array([(1, 4.0, 'Hello'), (2, 6.0, 'World')],
      dtype=[('f0', '>i4'), ('f1', '>f4'), ('f2', '|S10')])
```

In these examples, y is a simple float array consisting of the 2nd field in the record. But it is not a copy of the data in the structured array, instead it is a view. It shares exactly the same data. Thus when we updated this array by doubling its values, the structured array shows the corresponding values as doubled as well. Likewise, if one changes the record, the field view changes:

```
>>> x[1] = (-1,-1.,"Master")
>>> x
array([(1, 4.0, 'Hello'), (-1, -1.0, 'Master')],
      dtype=[('f0', '>i4'), ('f1', '>f4'), ('f2', '|S10')])
>>> y
array([ 4., -1.], dtype=float32)
```

Defining Structured Arrays

The definition of a structured array is all done through the dtype object. There are a **lot** of different ways one can define the fields of a record. Some of variants are there to provide backward compatibility with Numeric or numarray, or another module, and should not be used except for such purposes. These will be so noted. One defines records by specifying the structure by 4 general ways, using an argument (as supplied to a dtype function keyword or a dtype object constructor itself) in the form of a: 1) string, 2) tuple, 3) list, or 4) dictionary. Each of these will be briefly described.

1) String argument (as used in the above examples). In this case, the constructor is expecting a comma separated list of type specifiers, optionally with extra shape information. The type specifiers can take 4 different forms:

```
a) b1, i1, i2, i4, i8, u1, u2, u4, u8, f4, f8, c8, c16, a<n>
   (representing bytes, ints, unsigned ints, floats, complex and
    fixed length strings of specified byte lengths)
b) int8,...,uint8,...,float32, float64, complex64, complex128
   (this time with bit sizes)
c) older Numeric/numarray type specifications (e.g. Float32).
   Don't use these in new code!
d) Single character type specifiers (e.g H for unsigned short ints).
   Avoid using these unless you must. Details can be found in the
   Numpy book
```

These different styles can be mixed within the same string (but why would you want to do that?). Furthermore, each type specifier can be prefixed with a repetition number, or a shape. In these cases an array element is created, i.e., an array within a record. That array is still referred to as a single field. An example:

```
>>> x = np.zeros(3, dtype='3int8, float32, (2,3)float64')
>>> x
array([([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]),
       ([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]),
       ([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])],
      dtype=[('f0', '|i1', 3), ('f1', '>f4'), ('f2', '>f8', (2, 3))])
```

By using strings to define the record structure, it precludes being able to name the fields in the original definition. The names can be changed as shown later, however.

2) Tuple argument: The only relevant tuple case that applies to record structures is when a structure is mapped to an existing data type. This is done by pairing in a tuple, the existing data type with a matching dtype definition (using any of the variants being described here). As an example (using a definition using a list, so see 3) for further details):

```
>>> x = zeros(3, dtype=('i4',[('r','u1'), ('g','u1'), ('b','u1'), ('a','u1')]))
>>> x
array([0, 0, 0])
>>> x['r']
array([0, 0, 0], dtype=uint8)
```

In this case, an array is produced that looks and acts like a simple int32 array, but also has definitions for fields that use only one byte of the int32 (a bit like Fortran equivalencing).

3) List argument: In this case the record structure is defined with a list of tuples. Each tuple has 2 or 3 elements specifying: 1) The name of the field (‘’ is permitted), 2) the type of the field, and 3) the shape (optional). For example:

```
>>> x = np.zeros(3, dtype=[('x','f4'),('y',np.float32),('value','f4',(2,2))])
>>> x
array([(0.0, 0.0, [[0.0, 0.0], [0.0, 0.0]]),
       (0.0, 0.0, [[0.0, 0.0], [0.0, 0.0]]),
       (0.0, 0.0, [[0.0, 0.0], [0.0, 0.0]])],
      dtype=[('x', '>f4'), ('y', '>f4'), ('value', '>f4', (2, 2))])
```

4) Dictionary argument: two different forms are permitted. The first consists of a dictionary with two required keys (‘names’ and ‘formats’), each having an equal sized list of values. The format list contains any type/shape specifier allowed in other contexts. The names must be strings. There are two optional keys: ‘offsets’ and ‘titles’. Each must be a correspondingly matching list to the required two where offsets contain integer offsets for each field, and titles are objects containing metadata for each field (these do not have to be strings), where the value of None is permitted. As an example:

```
>>> x = np.zeros(3, dtype={'names':['col1', 'col2'], 'formats':['i4','f4']})
>>> x
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('col1', '>i4'), ('col2', '>f4')])
```

The other dictionary form permitted is a dictionary of name keys with tuple values specifying type, offset, and an optional title.

```
>>> x = np.zeros(3, dtype={'col1':('i1',0,'title 1'), 'col2':('f4',1,'title 2')})
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[(('title 1', 'col1'), '|i1'), (('title 2', 'col2'), '>f4')])
```

Accessing and modifying field names

The field names are an attribute of the dtype object defining the record structure. For the last example:

```
>>> x.dtype.names
('col1', 'col2')
>>> x.dtype.names = ('x', 'y')
>>> x
array([(0, 0.0), (0, 0.0), (0, 0.0)],
     dtype=[(('title 1', 'x'), '|i1'), (('title 2', 'y'), '>f4')])
>>> x.dtype.names = ('x', 'y', 'z') # wrong number of names
<type 'exceptions.ValueError'>: must replace all names at once with a sequence of length 2
```

Accessing field titles

The field titles provide a standard place to put associated info for fields. They do not have to be strings.

```
>>> x.dtype.fields['x'][2]
'title 1'
```

2.6 Subclassing ndarray

2.6.1 Credits

This page is based with thanks on the wiki page on subclassing by Pierre Gerard-Marchant - http://www.scipy.org/Subclasses.

2.6.2 Introduction

Subclassing ndarray is relatively simple, but you will need to understand some behavior of ndarrays to understand some minor complications to subclassing. There are examples at the bottom of the page, but you will probably want to read the background to understand why subclassing works as it does.

ndarrays and object creation

The creation of ndarrays is complicated by the need to return views of ndarrays, that are also ndarrays. For example:

```
>>> import numpy as np
>>> arr = np.zeros((3,))
>>> type(arr)
<type 'numpy.ndarray'>
>>> v = arr[1:]
>>> type(v)
<type 'numpy.ndarray'>
>>> v is arr
False
```

So, when we take a view (here a slice) from the ndarray, we return a new ndarray, that points to the data in the original. When we subclass ndarray, taking a view (such as a slice) needs to return an object of our own class. There is machinery to do this, but it is this machinery that makes subclassing slightly non-standard.

To allow subclassing, and views of subclasses, ndarray uses the ndarray `__new__` method for the main work of object initialization, rather then the more usual `__init__` method.

`__new__` and `__init__`

`__new__` is a standard python method, and, if present, is called before `__init__` when we create a class instance. Consider the following:

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

```
        print 'Args in __init__:', args

C('hello')
```

The code gives the following output:

```
cls is: <class '__main__.C'>
Args in __new__: ('hello',)
self is : <__main__.C object at 0xb7dc720c>
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 C(object):
    def __new__(cls, *args):
        print 'cls is:', cls
        print 'Args in __new__:', args
        return object.__new__(cls, *args)
    def __init__(self, *args):
        print 'self is :', self
        print 'Args in __init__:', args

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):
        print 'D self is :', self
        print 'D args in __init__:', args

D('hello')
```

which gives:

```
D cls is: <class '__main__.D'>
D args in __new__: ('hello',)
cls is: <class '__main__.C'>
Args in __new__: ('hello',)
```

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).

So, when creating a new view object of our subclass, we need to be able to set any extra attributes from the original object of our class. This is the role of the `__array_finalize__` method of ndarray. `__array_finalize__` is called from within the ndarray machinery, each time we create an ndarray of our own class, and passes in the new view object, created as above, as well as the old object from which the view has been taken. In it we can take any attributes from the old object and put then into the new view object, or do any other related processing. Now we are ready for a simple example.

2.6.3 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
        # input arguments.  This will call the standard ndarray
        # constructor, but return an object of our type
        obj = np.ndarray.__new__(subtype, shape, dtype, buffer, offset, strides,
                         order)
        # 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):
        # reset the attribute from passed original object
        self.info = getattr(obj, 'info', None)
        # We do not need to return anything

obj = InfoArray(shape=(3,), info='information')
print type(obj)
print obj.info
v = obj[1:]
print type(v)
print v.info
```

which gives:

```
<class '__main__.InfoArray'>
information
<class '__main__.InfoArray'>
information
```

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 to be able to take an already formed ndarray from the usual numpy calls to `np.array` and return an object.

2.6.4 Slightly more realistic example - attribute added to existing array

Here is a class (with thanks to Pierre GM for the original example), that takes array 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):
        # reset the attribute from passed original object
        self.info = getattr(obj, 'info', None)
        # We do not need to return anything

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

which gives:

```
<class '__main__.RealisticInfoArray'>
information
<class '__main__.RealisticInfoArray'>
information
```

2.6.5 `__array_wrap__` for ufuncs

Let's say you have an instance `obj` of your new subclass, `RealisticInfoArray`, and you pass it into a ufunc with another array:

```
arr = np.arange(5)
ret = np.multiply.outer(arr, obj)
```

When a numpy ufunc is called on a subclass of ndarray, the __array_wrap__ method is called to transform the result into a new instance of the subclass. By default, __array_wrap__ will call __array_finalize__, and the attributes will be inherited.

By defining a specific __array_wrap__ method for our subclass, we can tweak the output. The __array_wrap__ method requires one argument, the object on which the ufunc is applied, and an optional parameter *context*. This parameter is returned by some ufuncs as a 3-element tuple: (name of the ufunc, argument of the ufunc, domain of the ufunc). See the masked array subclass for an implementation.

2.6.6 Extra gotchas - custom __del__ methods and ndarray.base

One of the problems that ndarray solves is that of memory ownership of ndarrays and their views. Consider the case where we have created an ndarray, `arr` and then taken a view with `v = arr[1:]`. If we then do `del v`, we need to make sure that the `del` does not delete the memory pointed to by the view, because we still need it for the original `arr` object. Numpy therefore keeps track of where the data came from for a particular array or view, with the `base` attribute:

```
import numpy as np

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

The assertions all succeed in this case. 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`.

CHAPTER

THREE

PERFORMANCE

Note: XXX: This section is not yet written. Placeholder for Improving Performance documentation.

CHAPTER

FOUR

MISCELLANEOUS

Note: XXX: This section is not yet written.

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.:

```
>>> np.where(myarr == np.nan)
>>> nan == nan  # is always False! Use special numpy functions instead.

>>> np.nan == np.nan
False
>>> myarr = np.array([1., 0., np.nan, 3.])
>>> 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
```

How numpy handles numerical exceptions

Default is to "warn" But this can be changed, and it can be set individually for different kinds of exceptions. The different behaviors are:

```
'ignore' : ignore completely
'warn'   : print a warning (once only)
'raise'  : raise an exception
'call'   : call a user-supplied function (set using 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-thead basis.

4.2 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)
>>> 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
```

4.3 Interfacing to C:

Only a survey 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!

1. pyrex

- Plusses:
  - avoid learning C API's
  - no dealing with reference counting
  - can code in psuedo python and generate C code
  - can also interface to existing C code
  - should shield you from changes to Python C api
  - become pretty popular within Python community
- Minuses:
  - Can write code in non-standard form which may become obsolete
  - Not as flexible as manual wrapping
  - Maintainers not easily adaptable to new features

Thus:

1. cython - fork of pyrex to allow needed features for SAGE

- being considered as the standard scipy/numpy wrapping tool
- fast indexing support for arrays

1. 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.

1. 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

1. Weave

- Plusses:
  - Phenomenal tool
  - 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 uncertain–lacks a champion

1. 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?

4.4 Interfacing to Fortran:

Fortran: Clear choice is f2py. (Pyfort is an older alternative, but not supported any longer)

4.5 Interfacing to C++:

1. CXX
2. Boost.python
3. SWIG
4. Sage has used cython to wrap C++ (not pretty, but it can be done)
5. SIP (used mainly in PyQT)

4.6 Methods vs. Functions

Placeholder for Methods vs. Functions documentation.

CHAPTER

FIVE

USING NUMPY C-API

5.1 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*

5.1.1 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.

5.1.2 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 to 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 itmes 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.

5.1.3 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 ave 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 Paramters 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, pyrex, 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.

5.1.4 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 convertable 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_ENSUREARRAY` in the requirements flag. A copy is made only if necessary. If you want to guarantee a copy, then pass in `NPY_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_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_IN_ARRAY, NPY_OUT_ARRAY, and NPY_INOUT_ARRAY:

NPY_IN_ARRAY
Equivalent to NPY_CONTIGUOUS | NPY_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_OUT_ARRAY
Equivalent to NPY_CONTIGUOUS | NPY_ALIGNED | NPY_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_INOUT_ARRAY
Equivalent to NPY_CONTIGUOUS | NPY_ALIGNED | NPY_WRITEABLE | NPY_UPDATEIFCOPY. This combination of flags is useful to specify an array that will be used for both input and output. If a copy is needed, then when the temporary is deleted (by your use of Py_DECREF at the end of the interface routine), the temporary array will be copied back into the original array passed in. Use of the UPDATEIFCOPY flag 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_DECREF_ERR (obj) on an array with the NPY_UPDATEIFCOPY flag set. This will delete the array 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_FORCECAST
Cast to the desired type, even if it can't be done without losing information.

NPY_ENSURECOPY
Make sure the resulting array is a copy of the original.

NPY_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_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 `PyArray_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 `PyArray_FLOAT` or `PyArray_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_C_CONTIGUOUS` and :cdata'NPY_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.

5.1.5 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.

```
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,
        &PyArrayType, *out)) return NULL;

    arr1 = PyArray_FROM_OTF(arg1, NPY_DOUBLE, NPY_IN_ARRAY);
    if (arr1 == NULL) return NULL;
    arr2 = PyArray_FROM_OTF(arg2, NPY_DOUBLE, NPY_IN_ARRAY);
    if (arr2 == NULL) goto fail;
    oarr = PyArray_FROM_OTF(out, NPY_DOUBLE, NPY_INOUT_ARRAY);
    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);
    Py_DECREF(oarr);
    Py_INCREF(Py_None);
    return Py_None;

 fail:
    Py_XDECREF(arr1);
    Py_XDECREF(arr2);
    PyArray_XDECREF_ERR(oarr);
    return NULL;
}
```

5.2 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 applicaton 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.

The http://www.scipy.org website also contains a great deal of useful information about many of these tools. For example, there is a nice description of using several of the tools explained in this chapter at http://www.scipy.org/PerformancePython. This link provides several ways to solve the same problem showing how to use and connect with compiled code to get the best performance. In the process you can get a taste for several of the approaches that will be discussed in this chapter.

5.2.1 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 (included in the standard distribution with Python 2.5). The first method is the most common (but with the inclusion of ctypes into Python 2.5 this status may change).

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.

5.2.2 Hand-generated wrappers

Extension modules were discussed in Chapter 1 . 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 session.

5.2.3 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- defintion-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 subroutin 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).

The new interface has docstring:

```
>>> 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 module. This provides a facility that is somewhat similar to the use of weave.ext_tools described below. An example of the final interface executed using Python code is:

```
import numpy.f2py as f2py
fid = open('add.f')
source = fid.read()
fid.close()
f2py.compile(source, 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 commerical 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.

5.2.4 weave

Weave is a scipy package that can be used to automate the process of extending Python with C/C++ code. It can be used to speed up evaluation of an array expression that would otherwise create temporary variables, to directly "inline" C/C++ code into Python, or to create a fully-named extension module. You must either install scipy or get the weave package separately and install it using the standard python setup.py install. You must also have a C/C++-compiler installed and useable by Python distutils in order to use weave. Somewhat dated, but still useful documentation for weave can be found at the link http://www.scipy/Weave. There are also many examples found in the examples directory which is installed under the weave directory in the place where weave is installed on your system.

Speed up code involving arrays (also see scipy.numexpr)

This is the easiest way to use weave and requires minimal changes to your Python code. It involves placing quotes around the expression of interest and calling weave.blitz. Weave will parse the code and generate C++ code using Blitz C++ arrays. It will then compile the code and catalog the shared library so that the next time this exact string is asked for (and the array types are the same), the already- compiled shared library will be loaded and used. Because Blitz makes extensive use of C++ templating, it can take a long time to compile the first time. After that, however, the code should evaluate more quickly than the equivalent NumPy expression. This is especially true if your array sizes are large and the expression would require NumPy to create several temporaries. Only expressions involving basic arithmetic operations and basic array slicing can be converted to Blitz C++ code.

For example, consider the expression:

```
d = 4*a + 5*a*b + 6*b*c
```

where a, b, and c are all arrays of the same type and shape. When the data-type is double-precision and the size is 1000x1000, this expression takes about 0.5 seconds to compute on an 1.1Ghz AMD Athlon machine. When this expression is executed instead using blitz:

```
d = empty(a.shape, 'd'); weave.blitz(expr)
```

execution time is only about 0.20 seconds (about 0.14 seconds spent in weave and the rest in allocating space for d). Thus, we've sped up the code by a factor of 2 using only a simnple command (weave.blitz). Your mileage may vary, but factors of 2-8 speed-ups are possible with this very simple technique.

If you are interested in using weave in this way, then you should also look at scipy.numexpr which is another similar way to speed up expressions by eliminating the need for temporary variables. Using numexpr does not require a C/C++ compiler.

Inline C-code

Probably the most widely-used method of employing weave is to "in-line" C/C++ code into Python in order to speed up a time-critical section of Python code. In this method of using weave, you define a string containing useful C-code and then pass it to the function **weave.inline** (`code_string`, `variables`), where code_string is a string of valid C/C++ code and variables is a list of variables that should be passed in from Python. The C/C++ code should refer to the variables with the same names as they are defined with in Python. If weave.line should return anything the the special value return_val should be set to whatever object should be returned. The following example shows how to use weave on basic Python objects:

```
code = r"""
int i;
py::tuple results(2);
for (i=0; i<a.length(); i++) {
     a[i] = i;
}
results[0] = 3.0;
results[1] = 4.0;
return_val = results;
"""
a = [None]*10
res = weave.inline(code,['a'])
```

The C++ code shown in the code string uses the name 'a' to refer to the Python list that is passed in. Because the Python List is a mutable type, the elements of the list itself are modified by the C++ code. A set of C++ classes are used to access Python objects using simple syntax.

The main advantage of using C-code, however, is to speed up processing on an array of data. Accessing a NumPy array in C++ code using weave, depends on what kind of type converter is chosen in going from NumPy arrays to C++ code. The default converter creates 5 variables for the C-code for every NumPy array passed in to weave.inline. The following table shows these variables which can all be used in the C++ code. The table assumes that `myvar` is the name of the array in Python with data-type {dtype} (i.e. float64, float32, int8, etc.)

Variable	Type	Contents
myvar	{dtype}*	Pointer to the first element of the array
Nmyvar	npy_intp*	A pointer to the dimensions array
Smyvar	npy_intp*	A pointer to the strides array
Dmyvar	int	The number of dimensions
myvar_array	PyArrayObject*	The entire structure for the array

The in-lined code can contain references to any of these variables as well as to the standard macros MYVAR1(i), MYVAR2(i,j), MYVAR3(i,j,k), and MYVAR4(i,j,k,l). These name-based macros (they are the Python name capitalized followed by the number of dimensions needed) will de- reference the memory for the array at the given location with no error checking (be-sure to use the correct macro and ensure the array is aligned and in correct byte-swap order in order to get useful results). The following code shows how you might use these variables and macros to code a loop in C that computes a simple 2-d weighted averaging filter.

```
int i,j;
for(i=1;i<Na[0]-1;i++) {
   for(j=1;j<Na[1]-1;j++) {
        B2(i,j) = A2(i,j) + (A2(i-1,j) +
                  A2(i+1,j)+A2(i,j-1)
                  + A2(i,j+1))*0.5
                  + (A2(i-1,j-1)
                  + A2(i-1,j+1)
                  + A2(i+1,j-1)
                  + A2(i+1,j+1))*0.25
   }
}
```

The above code doesn't have any error checking and so could fail with a Python crash if, `a` had the wrong number of dimensions, or `b` did not have the same shape as `a`. However, it could be placed inside a standard Python function with the necessary error checking to produce a robust but fast subroutine.

One final note about weave.inline: if you have additional code you want to include in the final extension module such as supporting function calls, include statments, etc. you can pass this code in as a string using the keyword support_code: `weave.inline(code, variables, support_code=support)`. If you need the extension module to link against an additional library then you can also pass in distutils-style keyword arguments such as library_dirs, libraries, and/or runtime_library_dirs which point to the appropriate libraries and directories.

Simplify creation of an extension module

The inline function creates one extension module for each function to- be inlined. It also generates a lot of intermediate code that is duplicated for each extension module. If you have several related codes to execute in C, it would be better to make them all separate functions in a single extension module with multiple functions. You can also use the tools weave provides to produce this larger extension module. In fact, the weave.inline function just uses these more general tools to do its work.

The approach is to:

1. construct a extension module object using ext_tools.ext_module(`module_name`);
2. create function objects using ext_tools.ext_function(`func_name`, `code`, `variables`);
3. (optional) add support code to the function using the .customize.add_support_code(`support_code`) method of the function object;
4. add the functions to the extension module object using the .add_function(`func`) method;
5. when all the functions are added, compile the extension with its .compile() method.

Several examples are available in the examples directory where weave is installed on your system. Look particularly at ramp2.py, increment_example.py and fibonacii.py

Conclusion

Weave is a useful tool for quickly routines in C/C++ and linking them into Python. It's caching-mechanism allows for on-the-fly compilation which makes it particularly attractive for in-house code. Because of the requirement that the user have a C++-compiler, it can be difficult (but not impossible) to distribute a package that uses weave to other users who don't have a compiler installed. Of course, weave could be used to construct an extension module which is then distributed in the normal way *(* using a setup.py file). While you can use weave to build larger extension modules with many methods, creating methods with a variable- number of arguments is not possible. Thus, for a more sophisticated module, you will still probably want a Python-layer that calls the weave-produced extension.

5.2.5 Pyrex

Pyrex is a way to write C-extension modules using Python-like syntax. It is an interesting way to generate extension modules that is growing in popularity, particularly among people who have rusty or non- existent C-skills. It does require the user to write the "interface" code and so is more time-consuming than SWIG or f2py if you are trying to interface to a large library of code. However, if you are writing an extension module that will include quite a bit of your own algorithmic code, as well, then Pyrex is a good match. A big weakness perhaps is the inability to easily and quickly access the elements of a multidimensional array. Notice that Pyrex is an extension-module generator only. Unlike weave or 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 Pyrex.Distutils import build_ext
from distutils.extension import Extension
from distutils.core import setup

import numpy
py_ext = Extension('mine', ['mine.pyx'],
         include_dirs=[numpy.get_include()])

setup(name='mine', description='Nothing',
      ext_modules=[pyx_ext],
      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 I assume you are using Pyrex 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 Pyrex 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.

Pyrex does not natively understand NumPy arrays. However, it is not difficult to include information that lets Pyrex deal with them usefully. In fact, the numpy.random.mtrand module was written using Pyrex so an example of Pyrex usage is already included in the NumPy source distribution. That experience led to the creation of a standard c_numpy.pxd file that you can use to simplify interacting with NumPy array objects in a Pyrex-written extension. The file may not be complete (it wasn't at the time of this writing). If you have additions you'd like to contribute, please send them. The file is located in the .../site-packages/numpy/doc/pyrex directory where you have Python installed. There is also an example in that directory of using Pyrex to construct a simple extension module. It shows that Pyrex looks a lot like Python but also contains some new syntax that is necessary in order to get C-like speed.

If you just use Pyrex to compile a standard Python module, then you will get a C-extension module that runs either as fast or, possibly, more slowly than the equivalent Python module. Speed increases are possible only when you use cdef to statically define C variables and use a special construct to create for loops:

```
cdef int i
for i from start <= i < stop
```

Let's look at two examples we've seen before to see how they might be implemented using Pyrex. These examples were compiled into extension modules using Pyrex-0.9.3.1.

Pyrex-add

Here is part of a Pyrex-file I named add.pyx which implements the add functions we previously implemented using f2py:

```
cimport c_numpy
from c_numpy cimport import_array, ndarray, npy_intp, npy_cdouble, \
     npy_cfloat, NPY_DOUBLE, NPY_CDOUBLE, NPY_FLOAT, \
     NPY_CFLOAT

#We need to initialize NumPy
import_array()

def zadd(object ao, object bo):
    cdef ndarray c, a, b
    cdef npy_intp i
    a = c_numpy.PyArray_ContiguousFromAny(ao,
                  NPY_CDOUBLE, 1, 1)
    b = c_numpy.PyArray_ContiguousFromAny(bo,
                  NPY_CDOUBLE, 1, 1)
    c = c_numpy.PyArray_SimpleNew(a.nd, a.dimensions,
                 a.descr.type_num)
    for i from 0 <= i < a.dimensions[0]:
        (<npy_cdouble *>c.data)[i].real = \
             (<npy_cdouble *>a.data)[i].real + \
             (<npy_cdouble *>b.data)[i].real
        (<npy_cdouble *>c.data)[i].imag = \
             (<npy_cdouble *>a.data)[i].imag + \
             (<npy_cdouble *>b.data)[i].imag
    return c
```

This module shows use of the `cimport` statement to load the definitions from the c_numpy.pxd file. As shown, both versions of the import statement are supported. It also shows use of the NumPy C-API to construct NumPy arrays from arbitrary input objects. The array c is created using PyArray_SimpleNew. Then the c-array is filled by addition. Casting to a particiular data-type is accomplished using <cast *>. Pointers are de-referenced with bracket notation and members of structures are accessed using '.' notation even if the object is techinically a pointer to a structure. The use of the special for loop construct ensures that the underlying code will have a similar C-loop so the addition calculation will proceed quickly. Notice that we have not checked for NULL after calling to the C-API — a cardinal sin when writing C-code. For routines that return Python objects, Pyrex inserts the checks for NULL into the C-code for you and returns with failure if need be. There is also a way to get Pyrex to automatically check for exceptions when you call functions that don't return Python objects. See the documentation of Pyrex for details.

Pyrex-filter

The two-dimensional example we created using weave is a bit uglierto implement in Pyrex because two-dimensional indexing using Pyrex is not as simple. But, it is straightforward (and possibly faster because of pre-computed indices). Here is the Pyrex-file I named image.pyx.

```
cimport c_numpy
from c_numpy cimport import_array, ndarray, npy_intp,\
     NPY_DOUBLE, NPY_CDOUBLE, \
     NPY_FLOAT, NPY_CFLOAT, NPY_ALIGNED \
```

```
#We need to initialize NumPy
import_array()
def filter(object ao):
    cdef ndarray a, b
    cdef npy_intp i, j, M, N, oS
    cdef npy_intp r,rm1,rp1,c,cm1,cp1
    cdef double value
    # Require an ALIGNED array
    # (but not necessarily contiguous)
    #  We will use strides to access the elements.
    a = c_numpy.PyArray_FROMANY(ao, NPY_DOUBLE, \
                2, 2, NPY_ALIGNED)
    b = c_numpy.PyArray_SimpleNew(a.nd,a.dimensions, \
                                  a.descr.type_num)
    M = a.dimensions[0]
    N = a.dimensions[1]
    S0 = a.strides[0]
    S1 = a.strides[1]
    for i from 1 <= i < M-1:
        r = i*S0
        rm1 = r-S0
        rp1 = r+S0
        oS = i*N
        for j from 1 <= j < N-1:
            c = j*S1
            cm1 = c-S1
            cp1 = c+S1
            (<double *>b.data)[oS+j] = \
               (<double *>(a.data+r+c))[0] + \
               ((<double *>(a.data+rm1+c))[0] + \
                (<double *>(a.data+rp1+c))[0] + \
                (<double *>(a.data+r+cm1))[0] + \
                (<double *>(a.data+r+cp1))[0])*0.5 + \
               ((<double *>(a.data+rm1+cm1))[0] + \
                (<double *>(a.data+rp1+cm1))[0] + \
                (<double *>(a.data+rp1+cp1))[0] + \
                (<double *>(a.data+rm1+cp1))[0])*0.25
    return b
```

This 2-d averaging filter runs quickly because the loop is in C and the pointer computations are done only as needed. However, it is not particularly easy to understand what is happening. A 2-d image, `in` , can be filtered using this code very quickly using:

```
import image
out = image.filter(in)
```

Conclusion

There are several disadvantages of using Pyrex:

1. The syntax for Pyrex can get a bit bulky, and it can be confusing at first to understand what kind of objects you are getting and how to interface them with C-like constructs.
2. Inappropriate Pyrex syntax or incorrect calls to C-code or type- mismatches can result in failures such as
   (a) Pyrex failing to generate the extension module source code,

(b) Compiler failure while generating the extension module binary due to incorrect C syntax,

(c) Python failure when trying to use the module.

3. 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.

4. Multi-dimensional arrays are "bulky" to index (appropriate macros may be able to fix this).

5. The C-code generated by Prex is hard to read and modify (and typically compiles with annoying but harmless warnings).

Writing a good Pyrex extension module still takes a bit of effort because not only does it require (a little) familiarity with C, but also with Pyrex's brand of Python-mixed-with C. One big advantage of Pyrex-generated extension modules is that they are easy to distribute using distutils. In summary, Pyrex 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-code or Fortran code. But, if you are already able to write simple subroutines in C or Fortran, then I would use one of the other approaches such as f2py (for Fortran), ctypes (for C shared- libraries), or weave (for inline C-code).

5.2.6 ctypes

Ctypes is a python extension module (downloaded separately for Python <2.5 and included with Python 2.5) 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 neglible 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 c-types 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 c-types 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 c-types 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 c-types 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 c-types 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 c-types 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 passsed 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 c-types compatible types corresponding to the data-area, the shape, and the strides of the array. The data attribute reutrns 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 (it's 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
   a has strides that are multiples of sizeof(double)
*/
void
dfilter2d(double *a, double *b, int *astrides, int *dims)
{
    int i, j, M, N, S0, S1;
    int 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 a 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 preceeding 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. It's 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

It's 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 it's 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 is still the easiest way to extend Python for package creation. However, ctypes is a close second and will probably be growing in popularity now that it is part of the Python distribution. This should bring more features to ctypes that should eliminate the difficulty in extending Python and distributing the extension using ctypes.

5.2.7 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 I see them as either older ways to do things more modernly handled by f2py, weave, Pyrex, or ctypes (SWIG, PyFort, PyInline) or because I don't know much about them (SIP, Boost, Instant). 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/doc/swig directory under numpy.i along 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++ structrues 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.

Instant

This is a relatively new package (called pyinstant at sourceforge) that builds on top of SWIG to make it easy to inline C and C++ code in Python very much like weave. However, Instant builds extension modules on the fly with specific module names and specific method names. In this repsect it is more more like f2py in its behavior. The extension modules are built on-the fly (as long as the SWIG is installed). They can then be imported. Here is an example of using Instant with NumPy arrays (adapted from the test2 included in the Instant distribution):

```
code="""
PyObject* add(PyObject* a_, PyObject* b_){
  /*
  various checks
  */
  PyArrayObject* a=(PyArrayObject*) a_;
  PyArrayObject* b=(PyArrayObject*) b_;
  int n = a->dimensions[0];
  int dims[1];
  dims[0] = n;
  PyArrayObject* ret;
  ret = (PyArrayObject*) PyArray_FromDims(1, dims, NPY_DOUBLE);
  int i;
  char *aj=a->data;
  char *bj=b->data;
  double *retj = (double *)ret->data;
  for (i=0; i < n; i++) {
    *retj++ = *((double *)aj) + *((double *)bj);
    aj += a->strides[0];
    bj += b->strides[0];
  }
return (PyObject *)ret;
}
"""
import Instant, numpy
ext = Instant.Instant()
ext.create_extension(code=s, headers=["numpy/arrayobject.h"],
                     include_dirs=[numpy.get_include()],
```

```
                    init_code='import_array();', module="test2b_ext")
import test2b_ext
a = numpy.arange(1000)
b = numpy.arange(1000)
d = test2b_ext.add(a,b)
```

Except perhaps for the dependence on SWIG, Instant is a straightforward utility for writing extension modules.

PyInline

This is a much older module that allows automatic building of extension modules so that C-code can be included with Python code. It's latest release (version 0.03) was in 2001, and it appears that it is not being updated.

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.

5.3 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*

5.3.1 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 desireable 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_RemoveLargest ( 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.

5.3.2 Creating a new universal function

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.

`PyObject * **PyUFunc_FromFuncAndData**(`*PyUFuncGenericFunction* func, void** data, char* types, int ntypes, int nin, int nout, int identity, char* name, char* doc, int check_return*`)`

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

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.

check_return

Not presently used, but this integer value does get set in the structure-member of similar name.

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);
...
```

5.3.3 User-defined data-types

NumPy comes with 21 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 intialize 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 particlar 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 `PyArray_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 `PyArray_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.

5.3.4 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 inheritence 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 proceedure must be followed except only behaviors that are different require new entries in the type- object structure. All other entires 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.

Note: XXX: some of the documentation below needs to be moved to the reference guide.

The __array_finalize__ method

`__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

__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

__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