Guide to NumPy

Travis E. Oliphant, PhD

Dec 7, 2006

Contents

List of Tables

Part I

NumPy from Python

Chapter 1

Origins of NumPy

NumPy builds on (and is a successor to) the successful Numeric array object. Its goal is to create the corner-stone for a useful environment for scientific computing. In order to better understand the people surrounding NumPy and (its library-package) SciPy, I will explain a little about how SciPy and (current) NumPy originated. In 1998, as a graduate student studying biomedical imaging at the Mayo Clinic in Rochester, MN, I came across Python and its numerical extension (Numeric) while I was looking for ways to analyze large data sets for Magnetic Resonance Imaging and Ultrasound using a high-level language. I quickly fell in love with Python programming which is a remarkable statement to make about a programming language. If I had not seen others with the same view, I might have seriously doubted my sanity. I became rather involved in the Numeric Python community, adding the C-API chapter to the Numeric documentation (for which Paul Dubois graciously made me a co-author).

As I progressed with my thesis work, programming in Python was so enjoyable that I felt inhibited when I worked with other programming frameworks. As a result, when a task I needed to perform was not available in the core language, or in the Numeric extension, I looked around and found C or Fortran code that performed the needed task, wrapped it into Python (either by hand or using SWIG), and used the new functionality in my programs.

Along the way, I learned a great deal about the underlying structure of Numeric and grew to admire it's simple but elegant structures that grew out of the mechanism by which Python allows itself to be extended.

NOTE

Numeric was originally written in 1995 largely by Jim Hugunin while he was a graduate student at MIT. He received help from many people including Jim Fulton, David Ascher, Paul Dubois, and Konrad Hinsen. These individuals and many others added comments, criticisms, and code which helped the Numeric extension reach stability. Jim Hugunin did not stay long as an active member of the community — moving on to write Jython and, later, Iron Python.

By operating in this need-it-make-it fashion I ended up with a substantial library of extension modules that helped Python + Numeric become easier to use in a scientific setting. These early modules included raw input-output functions, a special function library, an integration library, an ordinary differential equation solver, some least-squares optimizers, and sparse matrix solvers. While I was doing this laborious work, Pearu Peterson noticed that a lot of the routines I was wrapping were written in Fortran and there was no simplified wrapping mechanism for Fortran subroutines (like SWIG for C). He began the task of writing f2py which made it possible to easily wrap Fortran programs into Python. I helped him a little bit, mostly with testing and contributing early function-call-back code, but he put forth the brunt of the work. His result was simply amazing to me. I've always been impressed with f2py, especially because I knew how much effort writing and maintaining extension modules could be. Anybody serious about scientific computing with Python will appreciate that f2py is distributed along with NumPy.

When I finished my Ph.D. in 2001, Eric Jones (who had recently completed his Ph.D. at Duke) contacted me because he had a collection of Python modules he had developed as part of his thesis work as well. He wanted to combine his modules with mine into one super package. Together with Pearu Peterson we joined our efforts, and SciPy was born in 2001. Since then, many people have contributed module code to SciPy including Ed Schofield, Robert Cimrman, David M. Cooke, Charles (Chuck) Harris, Prabhu Ramachandran, Gary Strangman, Jean-Sebastien Roy, and Fernando Perez. Others such as Travis Vaught, David Morrill, Jeff Whitaker, and Louis Luangkesorn have contributed testing and build support.

At the start of 2005, SciPy was at release 0.3 and relatively stable for an early version number. Part of the reason it was difficult to stabilize SciPy was that the array object upon which SciPy builds was undergoing a bit of an upheaval. At about the same time as SciPy was being built, some Numeric users were hitting up against the limited capabilities of Numeric. In particular, the ability to deal with memory mapped files (and associated alignment and swapping issues), record arrays, and altered error checking modes were important but limited or non-existent in Numeric. As a result, numarray was created by Perry Greenfield, Todd Miller, and Rick White at the Space Science Telescope Institute as a replacement for Numeric. Numarray used a very different implementation scheme as a mix of Python classes and C code (which led to slow downs in certain common uses). While improving some capabilities, it was slow to pick up on the more advanced features of Numeric's universal functions (ufuncs) — never re-creating the C-API that SciPy depended on. This made it difficult for SciPy to "convert" to numarray.

Many newcomers to scientific computing with Python were told that numarray was the future and started developing for it. Very useful tools were developed that could not be used with Numeric (because of numarray's change in C-API), and therefore could not be used easily in SciPy. This state of affairs was very discouraging for me personally as it left the community fragmented. Some developed for numarray, others developed as part of SciPy. A few people even rejected adopting Python for scientific computing entirely because of the split. In addition, I estimate that quite a few Python users simply stayed away from both SciPy and numarray, leaving the community smaller than it could have been given the number of people that use Python for science and engineering purposes.

It should be recognized that the split was not intentional, but simply an outgrowth of the different and exacting demands of scientific computing users. My describing these events should not be construed as assigning blame to anyone. I very much admire and appreciate everyone I've met who is involved with scientific computing and Python. Using a stretched biological metaphor, it is only through the process of dividing and merging that better results are born. I think this concept applies to NumPy.

In early 2005, I decided to begin an effort to help bring the diverging community together under a common framework if it were possible. I first looked at numarray to see what could be done to add the missing features to make SciPy work with it as a core array object. After a couple of days of studying numarray, I was not enthusiastic about this approach. My familiarity with the Numeric code base no doubt biased my opinion, but it seemed to me that the features of Numarray could be added back to Numeric with a few fundamental changes to the core object. This would make the transition of SciPy to a more enhanced array object much easier in my mind.

Therefore, I began to construct this hybrid array object complete with an enhanced set of universal (broadcasting) functions that could deal with it. Along the way, quite a few new features and significant enhancements were added to the array object and its surrounding infrastructure. This book describes the result of that year-and-a-half-long effort which culminated with the release of NumPy 0.9.2 in early 2006 and NumPy 1.0 in late 2006. I first named the new package, SciPy Core, and used the scipy namespace. However, after a few months of testing under that name, it became clear that a separate namespace was needed for the new package. As a result, a rapid search for a new name resulted in actually coming back to the NumPy name which was the unofficial name of Numerical Python but never the actual namespace. Because the new package builds on the code-base of and is a successor to Numeric, I think the NumPy name is fitting and hopefully not too confusing to new users.

This book only briefly outlines some of the infrastructure that surrounds the basic objects in NumPy to provide the additional functionality contained in the older Numeric package (*i.e.* LinearAlgebra, RandomArray, FFT). This infrastructure in NumPy includes basic linear algebra routines, Fourier transform capabilities, and random number generators. In addition, the f2py module is described in its own documentation, and so is only briefly mentioned in the second part of the book. There are also extensions to the standard Python distutils and testing frameworks included with NumPy that are useful in constructing your own packages built on top of NumPy. The central purpose of this book, however, is to describe and document the basic NumPy system that is available under the numpy namespace.

NOTE

The numpy namespace includes all names under the numpy.core and numpy.lib namespaces as well. Thus, `import numpy` will also import the names from numpy.core and numpy.lib. This is the recommended way to use numpy.

The following table gives a brief outline of the sub-packages contained in numpy package.

Sub-Package	Purpose	Comments
core	basic objects	all names exported to numpy
lib	additional utilities	all names exported to numpy
linalg	basic linear algebra	old LinearAlgebra from Numeric
fft	discrete Fourier transforms	old FFT from Numeric
random	random number generators	old RandomArray from Numeric
distutils	enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
f2py	automatic wrapping of Fortran code	a useful utility needed by SciPy

Chapter 2

Object Essentials

NumPy provides two fundamental objects: an N-dimensional array object (`ndarray`) and a universal function object (`ufunc`). In addition, there are other objects that build on top of these which you may find useful in your work, and these will be discussed later. The current chapter will provide background information on just the `ndarray` and the `ufunc` that will be important for understanding the attributes and methods to be discussed later.

An N-dimensional array is a homogeneous collection of "items" indexed using N integers. There are two essential pieces of information that define an N-dimensional array: 1) the shape of the array, and 2) the kind of item the array is composed of. The shape of the array is a tuple of N integers (one for each dimension) that provides information on how far the index can vary along that dimension. The other important information describing an array is the kind of item the array is composed of. Because every `ndarray` is a homogeneous collection of exactly the same data-type, every item takes up the same size block of memory, and each block of memory in the array is interpreted in exactly the same way[1].

TIP

All arrays in NumPy are indexed starting at 0 and ending at M-1 following the Python convention.

For example, consider the following piece of code:

[1] By using OBJECT arrays, one can effectively have heterogeneous arrays, but the system still sees each element of the array as exactly the same thing (a reference to a Python object).

```
>>> a = array([[1,2,3],[4,5,6]])
>>> a.shape
(2, 3)
>>> a.dtype
dtype('int32')
```

NOTE

for all code in this book it is assumed that you have first entered `from numpy import *`. In addition, any previously defined arrays are still defined for subsequent examples.

This code defines an array of size 2×3 composed of 4-byte (little-endian) integer elements (on my 32-bit platform). We can index into this two-dimensional array using two integers: the first integer running from 0 to 1 inclusive and the second from 0 to 2 inclusive. For example, index $(1, 1)$ selects the element with value 5:

```
>>> a[1,1]
5
```

All code shown in the shaded-boxes in this book has been (automatically) executed on a particular version of NumPy. The output of the code shown below shows which version of NumPy was used to create all of the output in your copy of this book.

```
>>> import numpy; print numpy.__version__
1.0.2.dev3478
```

2.1 Data-Type Descriptors

In NumPy, an ndarray is an N-dimensional array of items where each item takes up a fixed number of bytes. Typically, this fixed number of bytes represents a number (*e.g.* integer or floating-point). However, this fixed number of bytes could also represent an arbitrary record made up of any collection of other data types. NumPy achieves this flexibility through the use of a data-type (dtype) object. Every array has an associated dtype object which describes the layout of the array data. Every dtype object, in turn, has an associated Python type-object that determines

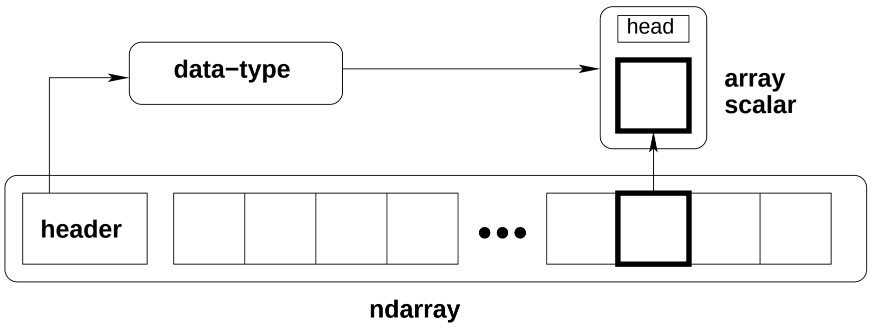

Figure 2.1: Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned when a single element of the array is accessed.

exactly what type of Python object is returned when an element of the array is accessed. The dtype objects are flexible enough to contain references to arrays of other dtype objects and, therefore, can be used to define nested records. This advanced functionality will be described in better detail later as it is mainly useful for the recarray (record array) subclass that will also be defined later. However, all ndarrays can enjoy the flexibility provided by the dtype objects. Figure 2.1 provides a conceptual diagram showing the relationship between the ndarray, its associated data-type object, and an array-scalar that is returned when a single-element of the array is accessed. Note that the data-type points to the type-object of the array scalar. An array scalar is returned using the type-object and a particular element of the ndarray.

Every dtype object is based on one of 21 built-in dtype objects. These built-in objects allow numeric operations on a wide-variety of integer, floating-point, and complex data types. Associated with each data-type is a Python type object whose instances are array scalars. This type-object can be obtained using the `type` attribute of the dtype object. Python typically defines only one data-type of a particular data class (one integer type, one floating-point type, etc.). This can be convenient for some applications that don't need to be concerned with all the ways data can be represented in a computer. For scientific applications, however, this is not always true. As a result, in NumPy, their are 21 different fundamental Python data-type-descriptor objects built-in. These descriptors are mostly based on the types available in the C language that CPython is written in. However, there are a few types that are extremely flexible, such as `str_`, `unicode_`, and `void`.

The fundamental data-types are shown in Table 2.1. Along with their (mostly) C-derived names, the integer, float, and complex data-types are also available using a bit-width convention so that an array of the right size can always be ensured (*e.g.* int8, float64, complex128). The C-like names are also accessible using a character code which is also shown in the table (use of the character codes, however, is discouraged). Names for the data types that would clash with standard Python object names are followed by a trailing underscore, '_'. These data types are so named because they use the same underlying precision as the corresponding Python data types. Most scientific users should be able to use the array-enhanced scalar objects in place of the Python objects. The array-enhanced scalars inherit from the Python objects they can replace and should act like them under all circumstances (except for how errors are handled in math computations).

TIP

The array types **bool_**, **int_**, **complex_**, **float_**, **object_**, **unicode_**, and **str_** are enhanced-scalars. They are very similar to the standard Python types (without the trailing underscore) and inherit from them (except for bool_ and object_). They can be used in place of the standard Python types whenever desired. Whenever a data type is required, as an argument, the standard Python types are recognized as well.

Three of the data types are flexible in that they can have items that are of an arbitrary size: the `str_` type, the `unicode_` type, and the `void` type. While, you can specify an arbitrary size for these types, every item in an array is still of that specified size. The void type, for example, allows for arbitrary records to be defined as elements of the array, and can be used to define exotic types built on top of the basic `ndarray`.

NOTE

The two types `intp` and `uintp` are not separate types. They are names bound to a specific integer type just large enough to hold a memory address (a pointer) on the platform.

Table 2.1: Built-in array-scalar types corresponding to data-types for an ndarray. The bold-face types correspond to standard Python types. The object_ type is special because arrays with dtype='O' do not return an array scalar on item access but instead return the actual object referenced in the array.

Type	Bit-Width	Character
bool_	boolXX	`'?'`
byte	intXX	`'b'`
short		`'h'`
intc		`'i'`
int_		`'l'`
longlong		`'q'`
intp		`'p'`
ubyte	uintXX	`'B'`
ushort		`'H'`
uintc		`'I'`
uint		`'L'`
ulonglong		`'Q'`
uintp		`'P'`
single	floatXX	`'f'`
float_		`'d'`
longfloat		`'g'`
csingle	complexXX	`'F'`
complex_		`'D'`
clongfloat		`'G'`
object_		`'O'`
str_		`'S#'`
unicode_		`'U#'`
void		`'V#'`

WARNING

Numeric Compatibility: If you used old typecode characters in your Numeric code (which was never recommended), you will need to change some of them to the new characters. In particular, the needed changes are 'c->'S1', 'b'->'B', '1'->'b', 's'->'h', 'w'->'H', and 'u'->'I'. These changes make the typecharacter convention more consistent with other Python modules such as the struct module.

The fundamental data-types are arranged into a hierarchy of Python type-objects shown in Figure 2.2. Each of the leaves on this hierarchy correspond to actual data-types that arrays can have (in other words, there is a built in dtype object associated with each of these new types). They also correspond to new Python objects that can be created. These new objects are "scalar" types corresponding to each fundamental data-type. Their purpose is to smooth out the rough edges that result when mixing scalar and array operations. These scalar objects will be discussed in more detail in Chapter 6. The other types in the hierarchy define particular categories of types. These categories can be useful for testing whether or not the object returned by `self.dtype.type` is of a particular class (using `issubclass`).

2.2 Basic indexing (slicing)

Indexing is a powerful tool in Python and NumPy takes full advantage of this power. In fact, some of capabilities of Python's indexing were first established by the needs of Numeric users.[2] Indexing is also sometimes called slicing in Python, and slicing for an `ndarray` works very similarly as it does for other Python sequences. There are three big differences: 1) slicing can be done over multiple dimensions, 2) exactly one ellipsis object can be used to indicate several dimensions at once, 3) slicing cannot be used to expand the size of an array (unlike lists).

A few examples should make slicing more clear. Suppose A is a 10×20 array, then $A[3]$ is the same as $A[3, :]$ and represents the 4th length-20 "row" of the array. On the other hand, $A[:, 3]$ represents the 4th length-10 "column" of the array. Every

[2]For example, the ability to index with a comma separated list of objects and have it correspond to indexing with a tuple is a feature added to Python at the request of the NumPy community. The Ellipsis object was also added to Python explicitly for the NumPy community. Extended slicing (wherein a step can be provided) was also a feature added to Python because of Numeric.

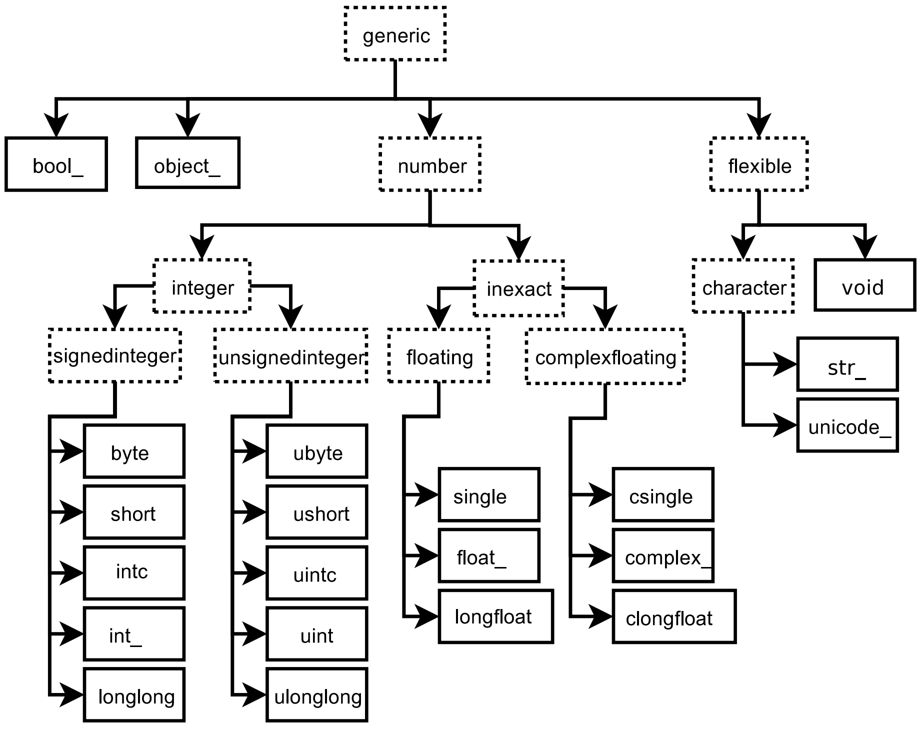

Figure 2.2: Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.

third element of the 4th column can be selected as $A[:: 3, 3]$. Ellipses can be used to replace zero or more ":" terms. In other words, an Ellipsis object expands to zero or more full slice objects (":") so that the total number of dimensions in the slicing tuple matches the number of dimensions in the array. Thus, if A is $10 \times 20 \times 30 \times 40$, then $A[3 :, ..., 4]$ is equivalent to $A[3 :, :, :, 4]$ while $A[..., 3]$ is equivalent to $A[:, :, :, 3]$.

The following code illustrates some of these concepts:

```
>>> a = arange(60).reshape(3,4,5); print a
[[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]]

[[20 21 22 23 24]
[25 26 27 28 29]
[30 31 32 33 34]
[35 36 37 38 39]]

[[40 41 42 43 44]
[45 46 47 48 49]
[50 51 52 53 54]
[55 56 57 58 59]]]
```

```
>>> print a[...,3]
[[ 3  8 13 18]
[23 28 33 38]
[43 48 53 58]]
>>> print a[1,...,3]
[23 28 33 38]
>>> print a[:,:,2]
[[ 2  7 12 17]
[22 27 32 37]
[42 47 52 57]]
>>> print a[0,::2,::2]
[[ 0  2  4]
[10 12 14]]
```

2.3 Memory Layout of `ndarray`

On a fundamental level, an N-dimensional array object is just a one-dimensional sequence of memory with fancy indexing code that maps an N-dimensional index into a one-dimensional index. The one-dimensional index is necessary on some level because that is how memory is addressed in a computer. The fancy indexing, however, can be very helpful for translating our ideas into computer code. This is because many concepts we wish to model on a computer have a natural representation as an N-dimensional array. While this is especially true in science and engineering, it is also applicable to many other arenas which can be appreciated by considering the popularity of the spreadsheet as well as "image processing" applications.

WARNING

Some high-level languages give pre-eminence to a particular use of 2-dimensional arrays as Matrices. In NumPy, however, the core object is the more general N-dimensional array. NumPy defines a matrix object as a sub-class of the N-dimensional array.

In order to more fully understand the array object along with its attributes and methods it is important to learn more about how an N-dimensional array is represented in the computer's memory. A complete understanding of this layout is only essential for optimizing algorithms operating on general purpose arrays. But, even for the casual user, a general understanding of memory layout will help to explain the use of certain array attributes that may otherwise be mysterious.

2.3.1 Contiguous Memory Layout

There is a fundamental ambiguity in how the mapping to a one-dimensional index can take place which is illustrated for a 2-dimensional array in Figure 2.3. In that figure, each block represents a chunk of memory that is needed for representing the underlying array element. For example, each block could represent the 8 bytes needed to represent a double-precision floating point number.

In the figure, two arrays are shown, a $4x3$ array and a $3x4$ array. Each of these arrays takes 12 blocks of memory shown as a single, contiguous segment. How this memory is used to form the abstract 2-dimensional array can vary, however, and the `ndarray` object supports both styles. Which style is in use can be interrogated by the use of the flags attribute which returns a dictionary of the state of array flags.

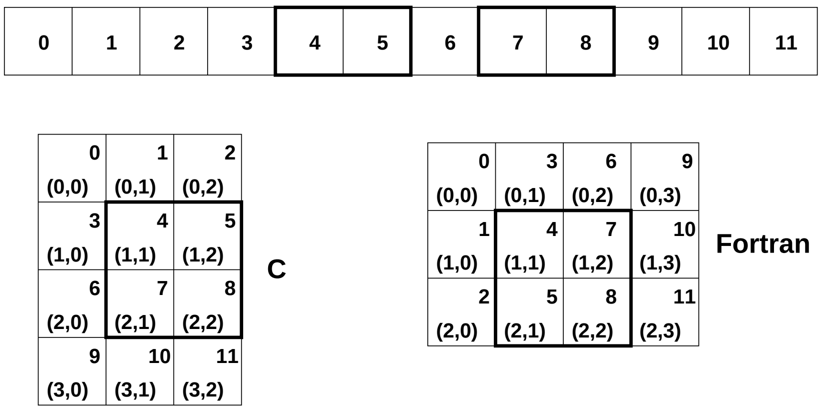

Figure 2.3: Options for memory layout of a 2-dimensional array.

In the C-style of N-dimensional indexing shown on the left of Figure 2.3 the last N-dimensional index "varies the fastest." In other words, to move through computer memory sequentially, the last index is incremented first, followed by the second-to-last index and so forth. Some of the algorithms in NumPy that deal with N-dimensional arrays work best with this kind of data.

In the Fortran-style of N-dimensional indexing shown on the right of Figure 2.3, the first N-dimensional index "varies the fastest." Thus, to move through computer memory sequentially, the first index is incremented first until it reaches the limit in that dimension, then the second index is incremented and the first index is reset to zero. While NumPy can be compiled without the use of a Fortran compiler, several modules of SciPy (available separately) rely on underlying algorithms written in Fortran. Algorithms that work on N-dimensional arrays that are written in Fortran typically expect Fortran-style arrays.

The two-styles of memory layout for arrays are connected through the transpose operation. Thus, if A is a (contiguous) C-style array, then the same block of memory can be used to represent A^T as a (contiguous) Fortran-style array. This kind of understanding can be useful when trying to optimize the wrapping of Fortran subroutines, or if a more detailed understanding of how to write algorithms for generally-indexed arrays is desired. But, fortunately, the casual user who does not care if an array is copied occasionally to get it into the right orientation needed for a particular algorithm can forget about how the array is stored in memory and just visualize it as an N-dimensional array (that is, after all, the whole point of creating the `ndarray` object in the first place).

2.3.2 Non-contiguous memory layout

Both of the examples presented above are *single-segment* arrays where the entire array is visited by sequentially marching through memory one element at a time. When an algorithm in C or Fortran expects an N-dimensional array, this single segment (of a certain fundamental type) is usually what is expected along with the shape N-tuple. With a single-segment of memory representing the array, the one-dimensional index into computer memory can always be computed from the N-dimensional index. This concept is explored further in the following paragraphs.

Let n_i be the value of the i^{th} index into an array whose shape is represented by the N integers d_i $(i = 0 \ldots N-1)$. Then, the one-dimensional index into a C-style contiguous array is

$$n^C = \sum_{i=0}^{N-1} n_i \prod_{j=i+1}^{N-1} d_j$$

while the one-dimensional index into a Fortran-style contiguous array is

$$n^F = \sum_{i=0}^{N-1} n_i \prod_{j=0}^{i-1} d_j.$$

In these formulas we are assuming that

$$\prod_{j=k}^{m} d_j = d_k d_{k+1} \cdots d_{m-1} d_m$$

so that if $m < k$, the product is 1. While perfectly general, these formulas may be a bit confusing at first glimpse. Let's see how they expand out for determining the one-dimensional index corresponding to the element $(1, 3, 2)$ of a $4 \times 5 \times 6$ array. If the array is stored as Fortran contiguous, then

$$\begin{aligned} n^F &= n_0 \cdot (1) + n_1 \cdot (4) + n_2 \cdot (4 \cdot 5) \\ &= 1 + 3 \cdot 4 + 2 \cdot 20 = 53. \end{aligned}$$

On the other hand, if the array is stored as C contiguous, then

$$\begin{aligned} n^C &= n_0 \cdot (5 \cdot 6) + n_1 \cdot (6) + n_2 \cdot (1) \\ &= 1 \cdot 30 + 3 \cdot 6 + 2 \cdot 1 = 50. \end{aligned}$$

The general pattern should be more clear from these examples.

The formulas for the one-dimensional index of the N-dimensional arrays reveal what results in an important generalization for memory layout. Notice that each formula can be written as

$$n^X = \sum_{i=0}^{N-1} n_i s_i^X$$

where s_i^X gives the *stride* for dimension i.[3] Thus, for C and Fortran contiguous arrays respectively we have

$$\begin{aligned} s_i^C &= \prod_{j=i+1}^{N-1} d_j = d_{i+1} d_{i+2} \cdots d_{N-1}, \\ s_i^F &= \prod_{j=0}^{i-1} d_j = d_0 d_1 \cdots d_{i-1}. \end{aligned}$$

The stride is how many elements in the underlying one-dimensional layout of the array one must jump in order to get to the next array element of a specific dimension in the N-dimensional layout. Thus, in a C-style $4 \times 5 \times 6$ array one must jump over 30 elements to increment the first index by one, so 30 is the stride for the first dimension ($s_0^C = 30$). If, for each array, we define a strides tuple with N integers, then we have pre-computed and stored an important piece of how to map the N-dimensional index to the one-dimensional one used by the computer.

In addition to providing a pre-computed table for index mapping, by allowing the strides tuple to consist of arbitrary integers we have provided a more general layout for the N-dimensional array. As long as we always use the stride information to move around in the N-dimensional array, we can use any convenient layout we wish for the underlying representation as long as it is regular enough to be defined by constant jumps in each dimension. The `ndarray` object of NumPy uses this stride information and therefore the underlying memory of an `ndarray` can be laid out dis-contiguously.

NOTE

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

[3] Our definition of stride here is an element-based stride, while the strides attribute returns a byte-based stride. The byte-based stride is the element itemsize multiplied by the element-based stride.

An important situation where irregularly strided arrays occur is array indexing. Consider again Figure 2.3. In that figure a high-lighted sub-array is shown. Define C to be the 4×3 C contiguous array and F to be the 3×4 Fortran contiguous array. The highlighted areas can be written respectively as C[1:3,1:3] and F[1:3,1:3]. As evidenced by the corresponding highlighted region in the one-dimensional view of the memory, these sub-arrays are neither C contiguous nor Fortran contiguous. However, they can still be represented by an `ndarray` object using the same striding tuple as the original array used. Therefore, a regular indexing expression on an `ndarray` can always produce an `ndarray` object *without* copying any data. This is sometimes referred to as the "view" feature of array indexing, and one can see that it is enabled by the use of striding information in the underlying `ndarray` object. The greatest benefit of this feature is that it allows indexing to be done very rapidly and without exploding memory usage (because no copies of the data are made).

2.4 Universal Functions for arrays

NumPy provides a wealth of mathematical functions that operate on then ndarray object. From algebraic functions such as addition and multiplication to trigonometric functions such as sin, and cos. Each universal function (`ufunc`) is an instance of a general class so that function behavior is the same. All ufuncs perform element-by-element operations over an array or a set of arrays (for multi-input functions). The ufuncs themselves and their methods are documented in Part 9.

One important aspect of ufunc behavior that should be introduced early, however, is the idea of *broadcasting*. Broadcasting is used in several places throughout NumPy and is therefore worth early exposure. To understand the idea of broadcasting, you first have to be conscious of the fact that all ufuncs are always element-by-element operations. In other words, suppose we have a ufunc with two inputs and one output (*e.g.* addition) and the inputs are both arrays of shape $4 \times 6 \times 5$. Then, the output is going to be $4 \times 6 \times 5$, and will be the result of applying the underlying function (*e.g.* $+$) to each pair of inputs to produce the output at the corresponding N-dimensional location.

Broadcasting allows ufuncs to deal in a meaningful way with inputs that do not have exactly the same shape. In particular, the first rule of broadcasting is that if all input arrays do not have the same number of dimensions, then a "1" will be repeatedly pre-pended to the shapes of the smaller arrays until all the arrays have the same number of dimensions. The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcasted" array. After application of the broadcasting rules, the sizes of all arrays must match.

While a little tedious to explain, the broadcasting rules are easy to pick up by looking at a couple of examples. Suppose there is a `ufunc` with two inputs, A and B. Now supposed that A has shape $4 \times 6 \times 5$ while B has shape $4 \times 6 \times 1$. The ufunc will proceed to compute the $4 \times 6 \times 5$ output as if B had been $4 \times 6 \times 5$ by assuming that $B[..., k] = B[..., 0]$ for $k = 1, 2, 3, 4$.

Another example illustrates the idea of adding 1's to the beginning of the array shape-tuple. Suppose A is the same as above, but B is a length 5 array. Because of the first rule, B will be interpreted as a $1 \times 1 \times 5$ array, and then because of the second rule B will be interpreted as a $4 \times 6 \times 5$ array by repeating the elements of B in the obvious way.

The most common alteration needed is to route-around the automatic pre-pending of 1's to the shape of the array. If it is desired, to add 1's to the end of the array shape, then dimensions can always be added using the `newaxis` name in NumPy: $B[..., \text{newaxis}, \text{newaxis}]$ returns an array with 2 additional 1's appended to the shape of B.

One important aspect of broadcasting is the calculation of functions on regularly spaced grids. For example, suppose it is desired to show a portion of the multiplication table by computing the function $a * b$ on a grid with a running from 6 to 9 and b running from 12 to 16. The following code illustrates how this could be done using ufuncs and broadcasting.

```
>>> a = arange(6, 10); print a
[6 7 8 9]
>>> b = arange(12, 17); print b
[12 13 14 15 16]
>>> table = a[:,newaxis] * b
>>> print table
[[ 72  78  84  90  96]
[ 84  91  98 105 112]
[ 96 104 112 120 128]
[108 117 126 135 144]]
```

2.5 Summary of new features

More information about using arrays in Python can be found in the old Numeric documentation at http://numeric.scipy.org `http://numeric.scipy.org`. Quite a bit of that documentation is still accurate, especially in the discussion of array basics. There are significant differences, however, and this book seeks to explain them in detail. The following list tries to summarize the significant new features (over Numeric) available in the `ndarray` and `ufunc` objects of NumPy:

1. more data types (all standard C-data types plus complex floats, Boolean, string, unicode, and void *);
2. flexible data types where each array can have a different itemsize (but all elements of the same array still have the same itemsize);
3. there is a true Python scalar type (contained in a hierarchy of types) for every data-type an array can have;
4. data-type objects define the data-type with support for data-type objects with fields and subarrays which allow record arrays with nested records;
5. many more array methods in addition to functional counterparts;
6. attributes more clearly distinguished from methods (attributes are intrinsic parts of an array so that setting them changes the array itself);
7. array scalars covering all data types which inherit from Python scalars when appropriate;
8. arrays can be misaligned, swapped, and in Fortran order in memory (facilitates memory-mapped arrays);
9. arrays can be more easily read from text files and created from buffers and iterators;
10. arrays can be quickly written to files in text and/or binary mode;
11. arrays support the removal of the 64-bit memory limitation as long as you have Python 2.5 or later;
12. fancy indexing can be done on arrays using integer sequences and Boolean masks;

13. coercion rules are altered for mixed scalar / array operations so that scalars (anything that produces a 0-dimensional array internally) will not determine the output type in such cases.

14. when coercion is needed, temporary buffer-memory allocation is limited to a user-adjustable size;

15. errors are handled through the IEEE floating point status flags and there is flexibility on a per-thread level for handling these errors;

16. one can register an error callback function in Python to handle errors are set to 'call' for their error handling;

17. ufunc reduce, accumulate, and reduceat can take place using a different type then the array type if desired (without copying the entire array);

18. ufunc output arrays passed in can be a different type than expected from the calculation;

19. ufuncs take keyword arguments which can specify 1) the error handling explicitly and 2) the specific 1-d loop to use by-passing the type-coercion detection.

20. arbitrary classes can be passed through ufuncs (__array_wrap__ and __array_priority__ expand previous __array__ method);

21. ufuncs can be easily created from Python functions;

22. ufuncs have attributes to detail their behavior, including a dynamic doc string that automatically generates the calling signature;

23. several new ufuncs (frexp, modf, ldexp, isnan, isfinite, isinf, signbit);

24. new types can be registered with the system so that specialized ufunc loops can be written over new type objects;

25. new types can also register casting functions and rules for fitting into the "can-cast" hierarchy;

26. C-API enhanced so that more of the functionality is available from compiled code;

27. C-API enhanced so array structure access can take place through macros;

28. new iterator objects created for easy handling in C of non-contiguous arrays;

29. new multi-iterator object created for easy handling in C of broadcasting;
30. types have more functions associated with them (no magic function lists in the C-code). Any function needed is part of the type structure.

All of these enhancements will be documented more thoroughly in the remaining portions of this book.

2.6 Summary of differences with Numeric

An effort was made to retain backwards compatibility with Numeric all the way to the C-level. This was mostly accomplished, with a few changes that needed to be made for consistency of the new system. If you are just starting out with NumPy, then this section may be skipped.

There are two steps (one required and one optional) to converting code that works with Numeric to work fully with NumPy The first step uses a compatibility layer and requires only small changes which can be handled by the numpy.oldnumeric.alter_code1 module. Code written to the compatibility layer will work and be supported. The purpose of the compatibility layer is to make it easy to convert to NumPy and many codes may only take this first step and work fine with NumPy. The second step is optional as it removes dependency on the compatibility layer and therefore requires a few more extensive changes. Many of these changes can be performed by the numpy.oldnumeric.alter_code2 module, but you may still need to do some final tweaking by hand. Because many users will probably be content to only use the first step, the alter_code2 module for second-stage migration may not be as complete as it otherwise could be.

2.6.1 First-step changes

In order to use the compatibility layer there are still a few changes that need to be made to your code. Many of these changes can be made by running the alter_code1 module with your code as input.

1. Importing (the alter_code1 module handles all these changes)
   (a) import Numeric –> import numpy.oldnumeric as Numeric
   (b) import Numeric as XX –> import numpy.oldnumeric as XX
   (c) from Numeric import <name1>,...<nameN> –> from numpy.oldnumeric import <name1>,...,<nameN>

   (d) from Numeric import * –> from numpy.oldnumeric import *
   (e) Similar name changes need to be made for Matrix, MLab, UserArray, LinearAlgebra, RandomArray RNG, RNG.Statistics, and FFT. The new names are numpy.oldnumeric.<pkg> where <pkg> is matrix, mlab, user_array, linear_algebra, random_array, rng, rng_stats, and fft.
   (f) multiarray and umath (if you used them directly) are now numpy.core.multiarray and numpy.core.umath, but it is more future proof to replace usages of these internal modules with numpy.oldnumeric.

2. Method name changes and methods converted to attributes. The alter_code1 module handles all these changes.

   (a) *arr*.typecode() –> *arr*.dtype.char
   (b) *arr*.iscontiguous() –> *arr*.flags.contiguous
   (c) *arr*.byteswapped() –> *arr*.byteswap()
   (d) *arr*.toscalar() –> *arr*.item()
   (e) *arr*.itemsize() –> *arr*.itemsize
   (f) *arr*.spacesaver() eliminated
   (g) *arr*.savespace() eliminated

3. Some of the typecode characters have changed to be more consistent with other Python modules (array and struct). You should only notice this change if you used the actual typecode characters (instead of the named constants). The alter_code1 module will change uses of 'b' to 'B' for internal Numeric functions that it knows about because NumPy will interpret 'b' to mean a signed byte type (instead of the old unsigned). It will also change the character codes when they are used explicitly in the .astype method. In the compatibility layer (and only in the compatibility layer), typecode-requiring function calls (*e.g.* zeros, array) understand the old typecode characters.
   The changes are (Numeric –> NumPy):

   (a) 'b' –> 'B'
   (b) '1' –> 'b'
   (c) 's' –> 'h'
   (d) 'w' –> 'H'

    (e) 'u' –> 'I'

4. *arr*.flat now returns an indexable 1-D iterator. This behaves correctly when passed to a function, but if you expected methods or attributes on *arr*.flat — besides .copy() — then you will need to replace *arr*.flat with *arr*.ravel() (copies only when necessary) or *arr*.flatten() (always copies). The alter_code1 module will change *arr*.flat to *arr*.ravel() unless you used the construct *arr*.flat = obj or *arr*.flat[ind].

5. If you used type-equality testing on the objects returned from arrays, then you need to change this to isinstance testing. Thus type(a[0]) is float or type(a[0]) == float should be changed to isinstance(a[0], float). This is because array scalar objects are now returned from arrays. These inherit from the Python scalars where they can, but define their own methods and attributes. This conversion is done by alter_code1 for the types (float, int, complex, and ArrayType)

6. If your code should produce 0-d arrays. These no-longer have a length as they should be interpreted similarly to real scalars which don't have a length.

7. Arrays cannot be tested for truth value unless they are empty (returns False) or have only one element. This means that if Z: where Z is an array will fail (unless Z is empty or has only one element). Also the 'and' and 'or' operations (which test for object truth value) will also fail on arrays of more than one element. Use the .any() and .all() methods to test for truth value of an array.

8. Masked arrays return a special nomask object instead of None when there is no mask on the array for the functions getmask and attribute access *arr*.mask

9. Masked array functions have a default axis of None (meaning ravel), make sure to specify an axis if your masked arrays are larger than 1-d.

10. If you used the construct `arr.shape=<tuple>`, this will not work for array scalars (which can be returned from array operations). You cannot set the shape of an array-scalar (you can read it though). As a result, for more general code you should use `arr=arr.reshape(<tuple>)` which works for both array-scalars and arrays.

The alter_code1 script should handle the changes outlined in steps 1-5 above. The final incompatibilities in 6-9 are less common and must be modified by hand if necessary.

2.6.2 Second-step changes

During the second phase of migration (should it be necessary) the compatibility layer is dropped. This phase requires additional changes to your code. There is another conversion module (alter_code2) which can help but it is not complete. The changes required to drop dependency on the compatibility layer are

1. Importing
   (a) numpy.oldnumeric –> numpy
   (b) from numpy.oldnumeric import * –> from numpy import * (this may clobber more names and therefore require further fixes to your code but then you didn't do this regularly anyway did you). The recommended procedure if this replacement causes problems is to fix the use of from numpy.oldnumeric import * to extract only the required names and then continue.
   (c) numpy.oldnumeric.mlab –> None, the functions come from other places.
   (d) numpy.oldnumeric.linear_algebra –> numpy.lilnalg with name changes to the functions (made lower case and shorter).
   (e) numpy.oldnumeric.random_array –> numpy.random with some name changes to the functions.
   (f) numpy.oldnumeic.fft –> numpy.fft with some name changes to the functions.
   (g) numpy.oldnumeric.rng –> None
   (h) numpy.oldnumeric.rng_stats –> None
   (i) numpy.oldnumeric.user_array –> numpy.lib.user_array
   (j) numpy.oldnumeric.matrix –> numpy
2. The typecode names are all lower-case and refer to type-objects corresponding to array scalars. The character codes are understood by array-creation functions but are not given names. All named type constants should be replaced with their lower-case equivalents. Also, the old character codes '1', 's', 'w', and 'u' are not understood as data-types. It is probably easiest to manually replace these with Int8, Int16, UInt16, and UInt32 and let the alter_code2 script convert the names to lower-case typeobjects.
3. Keyword and argument changes

   (a) All `typecode=` keywords must be changed to `dtype=`.
   (b) The `savespace` keyword argument has been removed from all functions where it was present (array, sarray, asarray, ones, and zeros). The sarray function is equivalent to asarray.

4. The default data-type in NumPy is float unlike in Numeric (and numpy.oldnumeric) where it was int. There are several functions affected by this so that if your code was relying on the default data-type, then it must be changed to explicitly add dtype=int.

5. The nonzero function in NumPy returns a tuple of index arrays just like the corresponding method. There is a flatnonzero function that first ravels the array and then returns a single index array. This function should be interchangeable with the old use of nonzero.

6. The default axis is None (instead of 0) to match the methods for the functions take, repeat, sum, average, product, sometrue, alltrue, cumsum, and cumproduct (from Numeric) and also for the functions average, max, min, ptp, prod, std, and mean (from MLab).

7. The default axis is None (instead of -1) to match the methods for the functions argmin, argmax, compress

2.6.3 Updating code that uses Numeric using alter_codeN

Despite the long list of changes that might be needed given above, it is likely that your code does not use any of the incompatible corners and it should not be too difficult to convert from Numeric to NumPy. For example all of SciPy was converted in about 2-3 days. The needed changes are largely search-and replace type changes, and the alter_codeN modules can help. The modules have two functions which help the process:

convertfile (filename, orig=1)

Convert the file with the given filename to use NumPy. If orig is True, then a backup is first made and given the name filename.orig. Then, the file is converted and the updated code written over the top of the old file.

convertall (direc=os.path.curdir, orig=1)

Converts all the ".py" files in the given directory to use NumPy. Backups of all the files are first made if orig is True as explained for the convertfile function.

convertsrc (direc=os.path.curdir, ext=None, orig=1)

Replace `''Numeric/arrayobject.h''` with `''numpy/oldnumeric.h''` in all files ending in the list of extensions given by ext (if ext is None, then all files are updated). If orig is True, then first make a backup file with ".orig" as the extension.

converttree (direc=os.path.curdir)

Walks the tree pointed to by direc and converts all ".py" modules in each subdirectory to use NumPy. No backups of the files are made. Also, converts all .h and .c files to replace `''Numeric/arrayobject.h''` with `''numpy/oldnumeric.h''` so that NumPy is used.

2.6.4 Changes to think about

Even if you don't make changes to your old code. If you are used to coding in Numeric, then you may need to adjust your coding style a bit. This list provides some helpful things to remember.

1. Switch from using typecode characters to bitwidth type names or c-type names
2. Convert use of uppercase type-names Int32, Float, etc., to lower case int32, float, etc.
3. Convert use of functions to method calls where appropriate but explicitly specify any axis arguments for arrays greater than 1-d.
4. The names for standard computations like Fourier transforms, linear algebra, and random-number generation have changed to conform to the standard of lower-case names possibly separated by an underscore.
5. Look for ways to take advantage of advanced slicing, but remember it always returns a copy and may be slower at times.
6. Remove any kludges you inserted to eliminate problems with Numeric that are now gone.
7. Look for ways to take advantage of new features like expanded data-types (record-arrays).

8. See if you can inherit from the ndarray directly, rather than using user_array.container (UserArray). However, if you were using UserArray in a multiple-inheritance hierarchy this is going to be more difficult and you can continue to use the standard container class in user_array (but notice the name change).

9. Watch your usage of scalars extracted from arrays. Treating Numeric arrays like lists and then doing math on the elements 1 by 1 was always about 2x slower than using real lists in Python. This can now be 3x-6x slower than using lists in NumPy because of the increased complexity of both the indexing of ndarrays and the math of array scalars. If you must select individual scalars from NumPy, you can get some speed increases by using the item method to get a standard Python scalar from an N-d array and by using the itemset method to place a scalar into a particular location in the N-d array. This complicates the appearance of the code, however. Also, when using these methods inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Throughout this book, warnings are inserted when compatibility issues with old Numeric are raised. While you may not need to make any changes to get code to run with the ndarray object, you will likely want to make changes to take advantage of the new features of NumPy. If you get into a jam during the conversion process, you should be aware that Numeric and NumPy can both be used together and they will not interfere with each other. In addition, if you have Numeric 24.0 or newer, they can even share the same memory. This makes it easy to use NumPy as well as third-party tools that have not made the switch from Numeric yet.

2.7 Summary of differences with Numarray

Conversion from Numarray can also be relatively painless, depending on how dependent your code is on the specific structure of the Numarray ufuncs, cfuncs, and various array-like objects. The internals of Numarray can be quite different and so depending on how intimately you used those internals adapting to NumPy can be more or less difficult. C-code that used the Numarray C-API can be easily adapted because NumPy includes a Numarray-compatible C-API module. All you need to do is replace usage of "numarray/libnumarray.h" with "numpy/libnumarray.h" and be sure the directory returned from the Python command numpy.get_numarray_include() is included in the list of directories used for compilation.

On the Python-side the largest number of differences are in the methods and attributes of the array and the way array data-types are represented. In addition, arrays containing Python Objects, strings, and records are an integral part of the array object and not handled using a separate class (although enhanced separate classes do exist for the case of character arrays and record arrays).

As is the case with Numeric, there is a two-step process available for migrating code written for Numarray to work with NumPy. This process involves running functions in the modules alter_code1 and alter_code2 located in the numarray sub-package of NumPy. These modules have interfaces identical to the ones that convert Numeric code, but they work to convert code written for numarray. The first module will convert your code to use the numarray compatibility module (numpy.numarray), while the second will try and help convert code to move away from dependency on the compatibility module. Because many users will probably be content to only use the first step, the alter_code2 module for second-stage migration may not be as complete as it otherwise could be.

Also, the alter_code1 module is not guaranteed to convert every piece of working numarray code to use NumPy. If your code relied on the internal module structure of numarray or on how the class hierarchy was laid out, then it will need to be changed manually to run with NumPy. Of course you can still use your code with Numarray installed side-by-side and the two array objects should be able to exchange data without copying.

2.7.1 First-step changes

The alter_code1 script makes the following import and attribute/method changes

2.7.1.1 Import changes

- import numarray –> import numpy.numarray as numarray
- import numarray.package –> import numpy.numarray.package as numarray_package with all usages of numarray.package in the code replaced by numarray_package
- import numarray as <name> –> import numpy.numarray s <name>
- import numarray.package as <name> –> import numpy.numarray.package as <name>

- from numarray import <names> –> from numpy.numarray import <names>
- from numarray.package import <names> –> from numpy.numarray.package import <names>

2.7.1.2 Attribute and method changes

- .imaginary –> .imag
- .flat –> probably .ravel() (Many usages will still work correctly because you can index and assign to self.flat)
- .byteswapped() –> .byteswap(False)
- .byteswap() –> .byteswap(True) (Returns a reference to self instead of None).
- self.info() –> numarray.info(self)
- .isaligned() –> .flags.aligned
- .isbyteswapped() –> not .dtype.isnative (the byte-order is a property of the data-type object not the array itself in NumPy).
- .iscontiguous() –> .flags.c_contiguous
- .is_c_array() –> .dtype.isnative and .flags.carray
- .is_fortran_contiguous() –> .flags.f_contiguous
- .is_f_array() –> .dtype.isnative and .flags.farray
- .itemsize() –> .itemsize
- .nelements() –> .size
- self.new(type) –> numarray.newobj(self, type)
- .repeat(r) –> .repeat(r, axis=0)
- .size() –> .size
- .type() –> numarray.typefrom(self)
- .typecode() –> .dtype.char
- .stddev() –> .std()

- .togglebyteorder() –> numarray.togglebyteorder(self)
- .getshape() –> .shape
- .setshape(obj) –> .shape = obj
- .getflat() –> .ravel()
- .getreal() –> .real
- .setreal(obj) –> .real = obj
- .getimag() –> .imag
- .setimag(obj) –> .imag = obj
- .getimaginary() –> .imag
- .setimaginary(obj) –> .imag = obj

2.7.2 Second-step changes

One of the notable differences is that several functions (array, arange, fromfile, and fromstring) do not take the shape= keyword argument. Instead you simply reshape the result using the reshape method. Another notable difference is that instead of allowing typecode=, type=, and dtype= variants for specifying the data-types, you must use the dtype= keyword. Other differences include

- matrixmultiply(a,b) –> dot(a,b)
- innerproduct(a,b) –> inner(a,b)
- outerproduct(a,b) –> outer(a,b)
- kroneckerproduct(a,b) –> kron(a,b)
- tensormultiply(a,b) –> None

2.7.3 Additional Extension modules

There are three extension packages that come included with numarray which are now downloaded separately. Stubs for these packages exist in numpy.numarray but they try and find the actual code by looking at what is currently installed. These packages are available in SciPy but can be installed separately as well:

- nd_image –> scipy.ndimage
- convolve –> scipy.stsci.convolve
- image –> scipy.stsci.image

If you don't want to install all of scipy, you can grab just these packages from SVN using

```
svn co http://svn.scipy.org/svn/scipy/trunk/Lib/ndimage ndimage
svn co http://svn.scipy.org/svn/scipy/trunk/Lib/stsci stsci
```

and then run

```
cd ndimage; sudo python setup.py install
cd stsci; sudo python setup.py install
```

On a Windows system, you can use the Tortoise SVN client which is integrated into the Windows Explorer. It can be downloaded from http://tortoisesvn.tigris.org. Instructions on how to use it are also provided on that site. After downloading the packages from SVN, installation will still require a C-compiler (the mingw32 compiler works fine even with MSVC-compiled Python as long as you specify –compiler=mingw32). Alternatively you can download binary releases of scipy from http://www.scipy.org to get the needed functionality or use the Enthon edition of Python.

Chapter 3

The Array Object

3.1 `ndarray` Attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Table 3.1 shows all the attributes with a brief description. Detailed information on each attribute is given below.

WARNING

Numeric Compatibility: you should check your old use of the .flat attribute. This attribute now returns an iterator object which acts like a 1-d array in terms of indexing. while it does not share all the attributes or methods of an array, it will be interpreted as an array in functions that take objects and convert them to arrays. Furthermore, Any changes in an array converted from a 1-d iterator will be reflected back in the original array when the converted array is deleted.

3.1.1 Memory Layout attributes

flags

Table 3.1: Attributes of the `ndarray`.

Attribute	Settable	Description
flags	No	special array-connected dictionary-like object with attributes showing the state of flags in this array; only the flags WRITEABLE, ALIGNED, and UPDATEIFCOPY can be modified by setting attributes of this object
shape	Yes	tuple showing the array shape; setting this attribute re-shapes the array
strides	Yes	tuple showing how many *bytes* must be jumped in the data segment to get from one entry to the next
ndim	No	number of dimensions in array
data	Yes	buffer object loosely wrapping the array data (only works for single-segment arrays)
size	No	total number of elements
itemsize	No	size (in bytes) of each element
nbytes	No	total number of bytes used
base	No	object this array is using for its data buffer, or None if it owns its own memory
dtype	Yes	data-type object for this array
real	Yes	real part of the array; setting copies data to real part of current array
imag	Yes	imaginary part, or read-only zero array if type is not complex; setting works only if type is complex
flat	Yes	one-dimensional, indexable iterator object that acts somewhat like a 1-d array
ctypes	No	object to simplify the interaction of this array with the ctypes module
__array_interface__	No	dictionary with keys (data, typestr, descr, shape, strides) for compliance with Python side of array protocol
__array_struct__	No	array interface on C-level
__array_priority__	No	always 0.0 for base type `ndarray`

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use which govern whether or not:

C_CONTIGUOUS (C) the data is in a single, C-style contiguous segment;

F_CONTIGUOUS (F) the data is in a single, Fortran-style contiguous segment;

OWNDATA (O) the array owns the memory it uses or if it borrows it from another object (if this is False, the base attribute retrieves a reference to the object this array obtained its data from);

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by `.base`). When this array is deallocated, the base array will be updated with the contents of this array.

Only the **UPDATEIFCOPY**, **WRITEABLE**, and **ALIGNED** flags can be changed by the user. This can be done using the special array-connected, dictionary-like object that the flags attribute returns. By setting elements in this dictionary, the underlying array obect's flags are altered. Flags can also be changed using the method `setflags`(...). All flags in the dictionary can be accessed using their first (upper case) letter as well as the full name.

Certain logical combinations of flags can also be read using named keys to the special flags dictionary. These combinations are

FNC Returns F_CONTIGUOUS and not C_CONTIGUOUS

FORC Returns F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) Returns ALIGNED and WRITEABLE

CARRAY (CA) Returns BEHAVED and C_CONTIGUOUS

FARRAY_(FA) Returns BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS

NOTE

The array flags cannot be set arbitrarily. UPDATEIFCOPY can only be set False. the ALIGNED flag can only be set True if the data is truly aligned. The flag WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface (or is a string). The exception for string is made so that unpickling can be done without copying memory.

Flags can also be set and read using attribute access with the lower-case key equivalent (without first letter support). Thus, for example, self.flags.c_contiguous returns whether or not the array is C-style contiguous, and self.flags.writeable=True changes the array to be writeable (if possible).

shape

The shape of the array is a tuple giving the number of elements in each dimension. The shape can be reset for single-segment arrays by setting this attribute to another tuple. The total number of elements cannot change. However, a -1 may be used in a dimension entry to indicate that the array length in that dimension should be computed so that the total number of elements does not change. `a.shape=x` is equivalent to `a=a.reshape(x)` except the latter can be used even if the array is not single-segment and even if *a* is an array scalar.

NOTE

Setting the shape attribute to () for a 1-element array will turn self into a 0-dimensional array. This is one of the few ways to get a 0-dimensional array in Python. Most other operations will return an array scalar. Other ways to get a 0-dimensional array in Python include calling array with a scalar argument and calling the squeeze method of an array whose shape is all 1's.

strides

The strides of an array is a tuple showing for each dimension how many *bytes* must be skipped to get to the next element in that dimension. Setting this attribute to another tuple will change the way the memory is viewed. This attribute can only be set to a tuple that will not cause the array to access unavailable memory. If an attempt is made to do so, ValueError is raised.

ndim

The number of dimensions of an array is sometimes called the rank of the array. Getting this attribute reveals the length of the shape tuple and the strides tuple.

data

A buffer object referencing the actual data for this array if this array is single-segment. If the array is not single-segment, then an AttributeError is raised. The buffer object is writeable depending on the status of self.flags.writeable.

size

The total number of elements in the array.

itemsize

The number of bytes each element of the array requires.

nbytes

The total number of bytes used by the array. This is equal to `self.itemsize*self.size`.

base

If the array does not own its own memory, then this attribute returns the object whose memory this array is referencing. The returned object may not be the original allocator of the memory, but may be borrowing it from still another object. If this array does own its own memory, then None is returned unless the UPDATEIFCOPY flag is True in which case self.base is the array that will be updated when self is deleted. UPDATEIFCOPY gets set for an array that is created as a behaved copy of a general array. The intent is for the misaligned array to get any changes that occur to the copy.

3.1.2 Data Type attributes

There are several ways to specify the kind of data that the array is composed of. The fullest description that preserves field information is always obtained using an actual dtype object. See Chapter 7 for more discussion on data-type objects and acceptable arguments to construct data-type objects. Three commonly-used attributes of the data-type object returned are also documented here.

dtype

A data-type object that fully describes (including any defined fields) each fixed-length item in the array. Whether or not the data is in machine byte-order is also determined by the data-type. The data-type attribute can be set to anything that can be interpreted as a data-type (see Chapter 7 for more information). Setting this attribute allows you to change the interpretation of the data in the array. The new data-type must be compatible with the array's current data-type. The new data-type is compatible if it has the same itemsize as the current data-type descriptor, or (if the array is a single-segment array) if the the array with the new data-type fits in the memory already consumed by the array.

dtype.type

A Python type object gives the typeobject whose instances represent elements of the array. This type object can be used to instantiate a scalar of that type.

dtype.char

A typecode character unique to each of the 21 built-in types.

dtype.str

This string consists of a required first character giving the "endianness" of the data ("<" for little endian, ">" for big endian, and "|" for irrelevant), the second character is a code for the kind of data ('b' for Boolean, 'i' for signed integer, 'u' for unsigned integer, 'f' for floating-point, 'c' for complex floating point, 'O' for object, 'S' for ASCII string, 'U' for unicode, and 'V' for void), the final characters give the number of bytes each element uses.

3.1.3 Other attributes

T

Equivalent to self.transpose(). For self.ndim < 2, it returns a view of self.

real

The real part of an array. For arrays that are not complex this attribute returns the array itself. Setting this attribute allows setting just the real part of an array. If the array is already real then setting this attribute is equivalent to self[...] = values.

imag

A view of the imaginary part of an array. For arrays that are not complex, this returns a read-only array of zeros. Setting this array allows in-place alteration of the complex part of an imaginary array. If the array is not complex, then trying to set this attribute raises an Error.

flat

Return an iterator object (numpy.flatiter) that acts like a 1-d version of the array. 1-d indexing works on this array and it can be passed in to most routines as an array wherein a 1-d array will be constructed from it. The new 1-d array will reference this array's data if this array is C-style contiguous, otherwise, new memory will be allocated for the 1-d array, the UPDATEIFCOPY flag will be set for the new array, and this array will have its WRITEABLE flag set FALSE until the the last reference to the new array disappears. When the last reference to the new 1-d array disappears, the data will be copied over to this non-contiguous array. This is done so that a.flat effectively references the current array regardless of whether or not it is contiguous or non-contiguous. As an example, consider the following code:

```
>>> a = zeros((4,5))
>>> b = ones(6)
>>> add(b,b,a[1:3,0:3].flat)
array([[ 2.,  2.,  2.],
[ 2.,  2.,  2.]])
>>> print a
[[ 0.  0.  0.  0.  0.]
[ 2.  2.  2.  0.  0.]
[ 2.  2.  2.  0.  0.]
[ 0.  0.  0.  0.  0.]]
```

The numpy.flatiter object has two methods: **__array__()** and **copy()** and one attribute: **base**. The base attribute returns a reference to the underlying array.

__array_priority__

The array priority attribute is a floating point number useful in mixed operations involving two subtypes to decide which subtype is returned. The base ndarray object has priority 0.0 and 1.0 is the default subtype priority.

3.1.4 Array Interface attributes

The array interface (sometimes called array protocol) was created in 2005 as a means for array-like Python objects to re-use each other's data buffers intelligently whenever possible. The ndarray object supports both the Python-side and the C-side of the array interface. The system is able to consume objects that expose the array interface, and array objects can expose their inner workings to other objects that support the array interface.

__array_interface__

The python-side of the array interface. It is a dictionary with the following attributes:

data A 2-tuple (dataptr, read-only flag). The dataptr is a string giving the address (in hexadecimal format) of the array data. The read-only flag is True if the array memory is read-only.

strides The strides tuple. Same as **strides** attribute except None is returned if the array is C-style contiguous.

shape The shape tuple. Same as **shape** attribute.

typestr A string giving the format of the data. Same as **dtype.str** attribute.

descr A list of tuples providing the detailed description of this data type. This information is obtained from the arrdescr attribute of the dtypedescr object associated with each array. For arrays with fields, this will return a valid array-protocol descriptor list. For arrays without defined fields, this returns [(”,typestr)].

__array_struct__

A PyCObject that wraps a pointer to a PyArrayInterface structure. This is only useful on the C-level for rapid implementation of the array interface, using a single attribute lookup.

ctypes

This attribute creates an object that makes it easier to use arrays when calling out to shared libraries with the ctypes module. The returned object has data, shape, and strides attributes which return ctypes objects that can be used as arguments to a shared library. These attributes are:

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

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

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

_as_parameter_ (c_void_p) Returns the data-pointer to the array as a ctypes object. Among other possible uses, this enables this ctypes object to be used directly in a ctypes-loaded call to an arbitrary function. Be sure to respect the flags on the array and the size and strides of the array so as not to use this memory in-appropriately (see the **ndpointer** function for how to return a class that can be used with the argtypes attribute of ctypes functions).

The ctypes object also has several methods which can alter how the shape, strides, and data of the underlying object is returned.

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

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

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

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

3.2 `ndarray` Methods

In NumPy, the `ndarray` object has many methods which operate on or with the array in some fashion, typically returning an array result. In Numeric, many of these methods were only library calls. These methods are explained in this chapter. Whenever the array whose method is being called needs to be referenced it will be referred to as *this array*, or *self*. Keyword arguments will be shown. Methods that only take one argument do not have keyword arguments. Default values for one argument methods will be shown in braces {default}.

WARNING

If you are converting code from Numeric, then you will need to make the following (search and replace) conversions: `.typecode() --> .dtype.char`; `.iscontiguous() --> .flags.contiguous`; `.byteswapped() --> .byteswap()`; `.toscalar() --> .item()`; and `.itemsize() --> .itemsize`. The numpy.oldnumeric.alter_code1 module can automate this for you.

3.2.1 Array conversion

tolist ()

The contents of self as a nested list.

```
>>> a = array([[1,2,3],[4,5,6]]); print a.tolist()
[[1, 2, 3], [4, 5, 6]]
```

item (*args)

If no arguments are passed in, then this method only works for arrays with one element (a.size == 1). In this case, it returns a standard Python scalar object (if possible) copied from the first element of self. When the data type of self is longdouble or clongdouble, this returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item() unless fields are defined in which case a tuple is returned.

```
>>> asc = a[0,0].item()
>>> type(asc)
<type 'int'>
>>> asc
1
>>> type(a[0,0])
<type 'numpy.int32'>
```

If arguments are provided, then they indicate indices into the array (either a flat index or an nd-index). A standard Python scalar corresponding to the item at the given location is then returned. This is very similar to self[args] except instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

itemset (*args)

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

tostring (order='C')

A Python string showing a copy of the raw contents of data memory. The string can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, then 'Fortran' order.

tofile (file=, sep='', format='')

Write the contents of self to the open file object. If file is a string, then open a file of that name first. If sep is the empty string, then write the file in binary mode. If sep is any other string, write the array in simple text mode separating each element with the value of the sep string. When the file is written in text mode, the format string can be used to alter the appearance of each entry. If format is the empty string, then it is equivalent to `‘‘%s’’`. Each element of the array will be converted to a Python scalar, `o`, and written to the file as `‘‘format’’ % o`. Note that writing an array to a file does not store any information about the shape, type, or endianness of an array. When written in binary mode, tofile is functionally equivalent to `fid.write(self.tostring())`.

```
>>> a.tofile('myfile.txt',sep=':',format='%03d')
Contents of myfile.txt
001:002:003:004:005:006
```

dump (file)

Pickle the contents of self to the file object represented by file. Equivalent to cPickle.dump(self, file, 2)

dumps ()

Return pickled representation of self as a string. Equivalent to cPickle.dumps(self, 2)

astype ({None})

Force conversion of this array to an array with the data type provided as the argument. If the argument is None, or equal to the data type of self, then return a copy of the array.

byteswap ({False})

Byteswap the elements of the array and return the byteswapped array. If the argument is True, then byteswap in-place and return a reference to self. Otherwise, return a copy of the array with the elements byteswapped. The data-type descriptor is not changed so the array will have changed numbers.

copy ()

Return a copy of the array (which is always single-segment, and ALIGNED). However, the data-type is preserved (including whether or not the data is byteswapped).

view ({None})

Return a new array using the same memory area as self. If the optional argument is given, it can be either a typeobject that is a sub-type of the ndarray or an object that can be converted to a data-type descriptor. If the argument is a typeobject then a new array of that Python type is returned that uses the information from self. If the argument is a data-type descriptor, then a new array of the same Python type as self is returned using the given data-type.

```
>>> print a.view(single)
[[  1.40129846e-45   2.80259693e-45   4.20389539e-45]
[  5.60519386e-45   7.00649232e-45   8.40779079e-45]]
>>> a.view(ubyte)
array([[1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0],
[4, 0, 0, 0, 5, 0, 0, 0, 6, 0, 0, 0]], dtype=uint8)
```

getfield (dtype=, offset=0)

Return a *field* of the given array as an array of the given data type. A field is a view of the array's data at a certain byte offset interpreted as a given data type. The returned array is a reference into self, therefore changes made to the returned array will be reflected in self. This method is particularly useful for record arrays that use a void data type, but it can also be used to extract the low (high)-order bytes of other array types as well. For example, using getfield, you could extract fixed-length substrings from an array of strings.

```
>>> a = array(['Hello','World','NumPy'])
>>> a.getfield('S2',1)
array(['el', 'or', 'um'],
dtype='|S2')
```

setflags (write=None, align=None, uic=None)

Table 3.2: Array conversion methods

Method	Arguments	Description
astype	(dtype {None})	Cast to another data type
byteswap	(inplace {False})	Byteswap array elements
copy	()	Copy array
dump	(file)	Pickle to stream or file
dumps	()	Get pickled string
fill	(scalar)	Fill an array with scalar value
getfield	(dtype=, offset=0)	Return a field of the array
setflags	(write=None, align=None, uic=None)	Set array flags
tofile	(file=, sep=", format=")	Raw write to file
tolist	()	Array as a nested list
item	(*args)	Python scalar extraction
itemset	(*args)	Insert scalar (last argument) into array
tostring	(order='C')	String of raw memory
view	(obj)	View as another data type or class

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively. The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The UPDATEIFCOPY flag can never be set to True. The flag WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface (or is a string). The exception for string is made so that unpickling can be done without copying memory.

fill (scalar)

Fill an array with the scalar value (appropriately converted to the type of self). If the scalar value is an array or a sequence, then only the first element is used. This method is usually faster than a[...]=scalar or self.flat=scalar, and always interprets its argument as a scalar.

3.2.2 Array shape manipulation

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

reshape (newshape, order='C')

Return an array that uses the same data as this array but has a new shape given by the newshape tuple (or a scalar to reshape as 1-d). The new shape must define an array with the same total number of elements. If one of the elements of the new shape tuple is -1, then that dimension will be determined such that the overall number of items in the array stays constant. If possible, the new array will reference the data of the old one. If the data must be moved in order to accomplish the reshape, then then the new array will contain a copy of the data in self. The order argument specifies how the array data should be viewed during the reshape (either in 'C' or 'FORTRAN' order). This order argument specifies both how the intrinsic raveling to a 1-d array should occur as well as how that 1-d array should be used to fill-up the new output array.

resize (newshape, refcheck=1, order='C')

Resize an array in-place. This changes self (in-place) to be an array with the new shape, reallocating space for the data area if necessary. If the data memory must be changed because the number of new elements is different than self.size, then an error will occur if this array does not own its data or if another object is referencing this one. Only a single-segment array can be resized. The method returns None. To bypass the reference count check, then set refcheck=0. The purpose of the reference count check is to make sure you don't use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array to another Python object, then you may safely set refcheck=0.

transpose (<None>)

Return an array view with the shape transposed according to the argument. An argument of None is equivalent to range(self.ndim)[::-1]. The argument can either be a tuple or multiple integer arguments. This method returns a new array with permuted shape and strides attributes using the same data as self.

```
>>> a = arange(40).reshape((2,4,5))
>>> b = a.transpose(2,0,1)
>>> print a.shape, b.shape
(2, 4, 5) (5, 2, 4)
>>> print a.strides, b.strides
(80, 20, 4) (4, 80, 20)
>>> print a
[[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]]

[[20 21 22 23 24]
[25 26 27 28 29]
[30 31 32 33 34]
[35 36 37 38 39]]]
>>> print b
[[[ 0  5 10 15]
[20 25 30 35]]

[[ 1  6 11 16]
[21 26 31 36]]

[[ 2  7 12 17]
[22 27 32 37]]

[[ 3  8 13 18]
[23 28 33 38]]

[[ 4  9 14 19]
[24 29 34 39]]]
```

swapaxes (axis1, axis2)

Return an array view with axis1 and axis2 swapped. This is a special case of the transpose method with argument equal to arg=range(self.ndim); arg[axis1], arg[axis2] = arg[axis2], arg[axis1]. See the **rollaxis** function for a routine that transposes the array with the axes rolled instead of swapped.

flatten (order='C')

Return a new 1-d array with data copied from self. Equivalent to but slightly faster then a.flat.copy().

ravel (order='C')

Return a 1-d version of self. If self is single-segment, then the new array references self, otherwise, a copy is made.

squeeze ()

Return an array with all unit-length dimensions squeezed out.

3.2.3 Array item selection and manipulation

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

take (indices=, axis=None, out=None, mode='raise')

The functionality of this method is available using the advanced indexing ability of the `ndarray` object. However, for doing selection along a single axis it is usually faster to use take. If axis is not None, this method is equivalent to `self[indxobj]` preceeded by `indxobj=[slice(None)]*self.ndim; indxobj[`**`axis`**`] = `**`indices`**. It returns the elements or sub-arrays from self indicated by the index numbers in indices. If axis is None, then this method is equivalent to `self.flat[indices]`. The out and mode arguments allow for specification of the output array and how out-of-bounds indices will be handled ('raise': raise an error, 'wrap': wrap around, 'clip': clip to range)

put (indices=, values=, mode='raise')

Performs the equivalent of

```
for n in indices:
    self.flat[n] = values[n]
```

Values is repeated if it is too short. The mode argument specifies what to do if n is too large.

repeat (repeats=, axis=None)

Copy elements (or sub-arrays selected along axis) of self **repeats** times. The repeats argument must be a sequence of length self.shape[axis] or a scalar. The repeats argument dictates how many times the element (or sub-array) will be repeated in the result array.

choose (choices, out=None, mode='raise')

The array must be an integer (or bool) array with entries from 0 to n. Choices is a tuple of n choice arrays: $b0$, $b1$, . . . , bn. (Alternatively, choices can be replaced with n arguments where each argument is a choice array). The return array will be formed from the elements of the choice arrays according to the value of the elements of self. In other words, the output array will merge the choice arrays together by using the value of self at a particular position to select which choice array should be used for the output at a particular position. The out keyword allows specification of an output array and the clip keyword allows different behavior when self contains entries outside the number of choices. The acceptable arguments to mode are 'raise' (RAISE), 'wrap' (WRAP), and 'clip' (CLIP) ('raise' produces an error, 'wrap' converts the number into range by periodic wrapping so that numbers $<$0 have n repeatedly added and numbers $>=$ n have n repeatedly subtracted, and 'clip' will clip all entries to be within the range [0,n).

```
>>> a = array([0,3,2,1])
>>> a.choose([0,1,2,3],[10,11,12,13],
... [20,21,22,23],[30,31,32,33])
array([ 0, 31, 22, 13])
```

sort (axis=-1, kind='quick', order=None)

Sort the array in-place and return None. The sort takes place over the given axis using an underlying sort algorithm specified by kind. The sorting algorithms available are 'quick', 'heap', and 'merge'. For flexible types only the quicksort algorithm is available. For arrays with fields defined, the order keyword allows specification of the order in which to use the field names in the sort. If order is a string then it is the field name to use to define the sort. If order is a list (or tuple) of strings, then it specifies a lexicographic ordering so that the first listed field name is compared first if that results in equality, the second listed field name is used for the comparison and so on. If order is None, then arrays with fields use the first field for comparison.

```
>>> a=array([[0.2,1.3,2.5],[1.5,0.1,1.4]]);
>>> b=a.copy(); b.sort(0); print b
[[ 0.2  0.1  1.4]
[ 1.5  1.3  2.5]]
>>> b=a.copy(); b.sort(1); print b
[[ 0.2  1.3  2.5]
[ 0.1  1.4  1.5]]
```

argsort (axis=-1, kind='quick', order=None)

Return an index array of the same size as self showing which indices along the given axis should be selected to sort self along that axis. Uses an underlying sort algorithm specified by kind. The sorting algorithms available are 'quick', 'heap', and 'merge'. For arrays with fields defined, the order keyword allows specification of the order in which to use the field names in the sort. If order is a string then it is the field name to use to define the sort. If order is a list (or tuple) of strings, then it specifies a lexicographic ordering so that the first listed field name is compared first if that results in equality, the second listed field name is used for the comparison and so on. If order is None, then arrays with fields use the first field for comparison.

```
>>> b=a.copy(); print b.argsort(0)
[[0 1 1]
[1 0 0]]
>>> b=a.copy(); print b.argsort(1)
[[0 1 2]
[1 2 0]]
```

TIP

Complex valued arrays sort lexicographically by comparing first the real parts and then the imaginary parts if the real parts are the same.

searchsorted (values, side='left')

Return an index array (dtype=intp) of the same shape as values showing the index where the value would fit in self. The index is such that self[index-1]

$<$ value $\leq$ self[index] when side is 'left'. In this formula self[self.size]=∞ and self[-1]=$-\infty$. Therefore, if value is larger than all elements of self, then index is self.size. If value is smaller than all elements of self, then index is 0. Self must be a sorted 1-d array. If elements of self are repeated, the index of the first occurrence is used. If side is 'right', then the search rule is switched so that the $<$ sign is on the "right" instead of the left in the search rule. In other words, the index returned is such that self[index-1] $\leq$value $<$ self[index].

```
>>> b=a.ravel(); b.sort()
>>> b.searchsorted([0.0, 1.35, 2.0, 3.0])
array([0, 3, 5, 6])
```

nonzero ()

Return the n-dimensional indices for elements of the n-dimensional array self that are nonzero into an n-tuple of equal-length index arrays. In particular, notice that a 0-dimensional array always returns an empty tuple.

```
>>> x = arange(15); y=x.reshape(3,5)
>>> (x>8).nonzero()
(array([ 9, 10, 11, 12, 13, 14]),)
>>> (y>8).nonzero()
(array([1, 2, 2, 2, 2, 2]), array([4, 0, 1, 2, 3, 4]))
```

compress (condition=, axis=None, out=None)

This method expects condition to be a one-dimensional mask array of the same length as self.shape[axis]. If the array is less than self.shape[axis], then False is assumed for the missing elements. The method returns the elements (or sub-arrays along the given axis) of self where condition is true. The shape of the return array is self.shape with the axis dimension replaced by the number of True elements of condition. The same effect can often be accomplished using array indexing.

```
>>> x=array([0,1,2,3])
>>> x.compress(x > 2)
array([3])
>>> x[x>2]
array([3])
```

diagonal (offset=0, axis1=0, axis2=1)

If self is 2-d, return the **offset** (from the main diagonal) diagonal of self. If self is larger than 2-d, then return an array constructed from all the diagonals created from all the 2-d sub-arrays formed using all of axis1 and axis2. The offset parameter is with respect to axis2. The shape of the returned array is found by removing the axis1 and axis2 entries from self.shape and then appending the length of the offset diagonal of each 2-d sub-array.

```
>>> a=arange(25).reshape(5,5); print a
[[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]
[20 21 22 23 24]]
>>> print a.diagonal()
[ 0  6 12 18 24]
>>> print a.diagonal(1)
[ 1  7 13 19]
>>> print a.diagonal(-1)
[ 5 11 17 23]
```

3.2.4 Array calculation

Many of these methods take an argument named axis. In such cases, if axis is None (the default), the array is treated as a 1-d array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. If axis is an integer, then the operation is done over the given axis (for each 1-d subarray that can be created along the given axis). The parameter dtype specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of self. To avoid overflow, it can be useful to perform the reduction using a larger data type. For several methods, an optional out argument can be provided and the result will be placed into the output array given. The out argument must be an ndarray and have the same number of elements. It can be of a different type in which case casting will be performed.

max (axis=None, out=None)

Table 3.3: Array item selection and shape manipulation methods. If axis is an argument, then the calculation is performed along that axis. An axis value of None means the array is flattened before calculation proceeds.

Method	Arguments	Description
argsort	argsort (axis=None, kind='quick')	Indices showing how to sort array.
choose	choose (c0, c1 , ..., cn, out=None, clip='raise')	Choose from different arrays based on value of self.
compress	(condition=, axis=None, out=None)	Elements of self where condition is true.
diagonal	(offset=0, axis1=0, axis2=1)	Return a diagonal from self.
flatten	(order='C')	A 1-d copy of self.
nonzero	()	True where self is not zero.
put	(indices=, values=, mode='raise')	Place values at 1-d index locations of self.
ravel	(order='C')	1-d version of self (no data copy if self is C-style contiguous).
repeat	(repeats=, axis=None)	Repeat elements of self.
reshape	(d1,d2,...,dn, order='C')	Return reshaped version of self.
resize	(d1,d2,...,dn, refcheck=1, order='Any')	Resize self in-place.
searchsorted	(values)	Show where values would be placed in self (assumed sorted).
sort	(axis=None, kind='quick')	Copy of self sorted along axis.
squeeze	()	Squeeze out all length-1 dimensions.
swapaxes	(axis1, axis2)	Swap two dimensions of self.
take	(indices=, axis=None, out=None, mode='raise')	Select elements of self along axis according to indices.
transpose	(permute <None>)	Rearrange shape of self according to permute.

Return the largest value in self. This is a better way to compute the maximum over an array, than using max(self). The latter uses the generic sequence interface to self. This will be slower, and will try to get an answer by comparing whole sub-arrays of self. This will be incorrect for arrays larger than 1-d.

argmax (axis=None, out=None)

Return the (first, 1-d) index of the largest value in self.

min (axis=None, out=None)

Return the smallest value in self. This is a better way to compute the minimum over an array, than using min(self). The latter uses the generic sequence interface to self. This will be slower, and will try to get an answer by comparing whole sub-arrays of self. This will be incorrect for arrays larger than 1-d.

argmin (axis=None, out=None)

Return the (first, 1-d) index of the smallest value in self.

ptp (axis=None, out=None)

Return the difference of the largest to the smallest value in self. Equivalent to self.max(axis) - self.min(axis)

clip (min=,max=, out=None)

Return a new array where any element in self less than min is set to min and any element less than max is set to max. Equivalent to self[self<min]=min; self[self>max]=max.

conj (out=None)

conjugate (out=None)

Return the conjugate of elements of the array.

round (decimals=0, out=None)

Round the elements of the array to the nearest decimal. For decimals < 0, the rounding is done to the nearest tens, hundreds, etc. Rounding of exactly the half-interval is to the nearest even integer. This is the only difference with standard Python rounding.

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

Perform a summation along each diagonal specified by offset, axis1, and axis2.
Equivalent to diagonal(offset,axis1,axis2).sum(axis=-1, dtype=dtype)

sum (axis=None, dtype=None, out=None)

Return the sum

$$\sum_{i=0}^{N-1} \text{self}[\underbrace{:,\ldots,:}_{\text{axis}},i]$$

where axis ':' objects are placed before the i.

cumsum (axis=None, dtype=None, out=None)

Return the cumulative sum. If ret is the return array of the same shape as self, then

$$\text{ret}[\underbrace{:,\ldots,:}_{\text{axis}},j] = \sum_{i=0}^{j} \text{self}[\underbrace{:,\ldots,:}_{\text{axis}},i].$$

mean (axis=None, dtype=None, out=None)

Return the average value caculated as

$$\frac{1}{N}\sum_{i=0}^{N-1} \text{self}[\underbrace{:,\ldots,:}_{\text{axis}},i]$$

where N is self.shape[axis] and axis ':' objects are placed before the i. The sum is done in the data-type of self unless self is an integer or Boolean data-type and then it is done over the float data-type.

var (axis=None, dtype=None, out=None)

Return the variance of the data calculated as

$$\frac{1}{N}\sum_{i=0}^{N-1} \left(\text{self}[\underbrace{:,\ldots,:}_{\text{axis}},i] - \mu\right)^2$$

where N is self.shape[axis] and μ is the mean (restored to the same number of dimensions as self with μ copied along the axis dimension). This is equivalent to (self**2).mean - self.mean()**2 and ((self-self.mean())**2).mean(). The value of $N-1$ was not chosen for normalization because while it gives an "unbiased" estimate, it is not always prudent to return unbiased estimates as they may have larger mean-square error. The sum is done using a float data-type if self has integer or Boolean data-type, otherwise it is done using the same data-type as self.

std (axis=None, dtype=None, out=None)

Return the standard deviation calculated as

$$\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(\text{self}[\underbrace{:,\ldots,:,}_{\text{axis}} i] - \mu\right)^2}$$

where N is self.shape[axis] and μ is the mean (restored to the same number of dimensions as self with μ copied along the axis dimension). The sum is done using the same data-type as self unless self is an integer or Boolean data-type and then it is done using a float data-type.

prod (axis=None, dtype=None, out=None)

Return the product calculated as

$$\prod_{i=0}^{N-1}\text{self}[\underbrace{:,\ldots,:,}_{\text{axis}} i].$$

cumprod (axis=None, dtype=None, out=None)

Return the cumulative product so that the return array, ret, is the same shape as self and

$$\text{ret}[\underbrace{:,\ldots,:,}_{\text{axis}} j] = \prod_{i=0}^{j}\text{self}[\underbrace{:,\ldots,:,}_{\text{axis}} i].$$

all (axis=None, out=None)

Return True if all entries along axis evaluate True, otherwise return False.

any (axis=None, out=None)

Return True if any entries along axis evaluate True, otherwise return False.

Table 3.4: Array object calculation methods. If axis is an argument, then the calculation is performed along that axis. An axis value of None means the array is flattened before calculation proceeds. All of these methods can take an optional out= argument which can specify the output array to write the results into.

Method	Arguments	Description
all	(axis=None)	true if all entries are true.
any	(axis=None)	true if any entries are true.
argmax	(axis=None)	index of largest value.
argmin	(axis=None)	index of smallest value.
clip	(min=, max=)	self[self>max]=max; self[self<min]=min
conj	()	complex conjugate
cumprod	(axis=None, dtype=None)	cumulative product
cumsum	(axis=None, dtype=None)	cumulative sum
max	(axis=None)	maximum of self
mean	(axis=None, dtype=None)	mean of self
min	(axis=None)	minimum of self
prod	(axis=None, dtype=None)	multiply elements of self together
ptp	(axis=None)	self.max(axis)-self.min(axis)
var	(axis=None, dtype=None)	variance of self
std	(axis=None, dtype=None)	standard deviation of self
sum	(axis=None, dtype=None)	add elements of self together
trace	(offset, axis1=0, axis2=0, dtype=None)	sum along a diagonal

3.3 Array Special Methods

Methods in this chapter are not generally meant to be called directly by the user. They are called by Python and are used to customize behavior of the ndarray object as it interacts with the Python language and standard library.

3.3.1 Methods for standard library functions

__copy__ ()

To allow copy.copy(a) to perform a shallow copy of an array. Exactly the same as self.copy() (contents of object arrays are not copied).

__deepcopy__ (memodict)

To allow copy.deepcopy(a) to perform a deep copy. This is the same as a shallow copy unless self is an object array. Then, after the shallow copy is made, a copy.deepcopy(item) is called for every item in the object array.

__reduce__ ()

__setstate__ (shape, typestr, isfortran, data)

Pickling support for arrays is provided by these two methods. When an array needs to be pickled, the __reduce__() method is called to provide a 3-tuple of already-pickleable objects. To construct a new object from the pickle, the first two elements of the 3-tuple are used to construct a new (0-length) array of the correct type and the last element of the 3-tuple, which is itself a 4-tuple of (shape, typestr, isfortran, data) is passed to the __setstate__ method of the newly created array to restore its contents.

The reduce method returns a 3-tuple consisting of (callable, args, state) where callable is a simple constructor function that handles subclasses of the ndarray. Also, args is a 3-tuple of arguments to pass to this constructor function (type(self), (0,), self.dtypechar), and state is a 4-tuple of information giving the object's state (self.shape, self.dtypedescr, isfortran, string_or_list). In this tuple, isfortran is a Bool stating whether the following flattened data is in Fortran order or not, and string_or_list is a string formed by self.tostring() if the data type is not object. If the data type of self is an object array, then string_or_list is a flat list equivalent to self.ravel().tolist().

On load from a pickle, the pickling code uses the first two elements from the tuple returned by reduce to construct an empty 0-dimensional subclass of the correct type. The last element is then passed to the __setstate__ method of the newly created array to restore its contents.

NOTE

When data is a string, the __setstate__ method will directly use the string memory as the array memory (new.base will point to the string). The typestr contains enough information to decode how the memory should be interpreted.

3.3.2 Basic customization

__new__ (subtype, shape=, dtype=long_, buffer=None, offset=0, strides=None, order=None)

This method creates a new ndarray. It is typically only used in the __new__ method of a subclass. This method is called to construct a new array whenever the object name is called, `a=ndarray(...)`. It supports two basic modes of array creation:

1. a single-segment array of the specified shape and data-type from newly allocated memory;
   (a) uses shape, dtype, strides, and order arguments; others are ignored;
   (b) The order argument allows specification of a Fortran-style contiguous memory segment (order='Fortran');
   (c) If strides is given, then it specifies the new strides of the array (and the order keyword is ignored). The strides will be checked for consistency with the dimension size so that steps outside of the memory won't occur.
2. an array of the given shape and data type using the provided object, buffer, which must export the buffer interface.
   (a) all arguments can be used;
   (b) strides can be given and will be checked for consistency with the shape, data type, and available memory in buffer;

(c) order indicates whether the data buffer should be interpreted as Fortran-style contiguous (order='Fortran') or not;

(d) offset can be used to start the array data at some offset in the buffer.

> **NOTE**
>
> The ndarray uses the default no-op __init__ function because the array is completely initialized after __new__ is called.

__array__ (dtype {None})

This is a special method that should always return an object of type ndarray. Useful for subclasses that need to get to the ndarray object.

__array_wrap__ (arr)

This is a special method that always returns an object of the same Python type as self using the array passed as an argument. This is mainly useful for subclasses as it is an easy way to get the subclass back from an ndarray.

__lt__ (other)

__le__ (other)

__gt__ (other)

__ge__ (other)

__eq__ (other)

__ne__ (other)

Defined to support rich comparisons (<, <=, >, >=, ==, !=) on ndarrays using universal functions.

__str__ ()

__repr__ ()

These functions print the array when called by str(self) and repr(self) respectively. Array printing can be changed using set_string_function(..). Default array printing has been borrowed from numarray whose printing code was written by Perry Greenfield and J. Todd Miller. By default, arrays print such that

1. The last axis is always printed left to right.
2. The next-to-last axis is printed top to bottom.
3. Remaining axes are printed top to bottom with increasing numbers of separators.

Five parameters of the printing can be set using keyword arguments with `set_printoptions(...)`. The parameters can all be retrieved using `get_printoptions()`. These printing options are

precision the number of digits of precision for floating point output (default 8);

threshold total number of array elements which triggers summarization rather than full representation (default 1000);

edgeitems number of array items in summary at beginning an end of each dimension (default 3);

linewidth the number of characters per line for the purpose of inserting line breaks (default 71);

suppress Boolean indicating whether or not to suppress printing of small floating point values using scientific notation (default False).

__nonzero__ ()

Truth-value testing for the array as a whole. It is called whenever the truth value of the ndarray as a whole object is required. This raises an error if the number of elements in the the array is larger than 1 because the truth value of such arrays is ambiguous. Use .any() and .all() instead to be clear about what is meant in such cases. If the number of elements is 0 then False is returned. If there is one element in the array, then the truth-value of this element is returned.

3.3.3 Container customization

__len__ ()

Returns self.shape[0]. It is called in response to len(self). Use self.size to get the total number of elements in the array.

Notice that the default Python iterator for sequences is used when arrays are used in places that expect an iterator. This iterator returns successively self[0], self[1], ..., self[self.__len__()]. Use self.flat to get an iterator that walks through the entire array one element at a time.

__getitem__ (key)

Called when evaluating self[key] construct. Items from the array can be selected using this customization. This construct has both standard and extended indexing abilities which are explained in Section 3.4. A named field can be retrieved if key is a string and fields are defined in the dtypedescr object associated with this array.

__setitem__ (key, value)

Called when evaluating self[key]=value. Items in the array can be set using this construct. This construct is explained in Section 3.4. A named field can be set if key is a string and fields are defined in the dtypedescr object associated with this array.

__getslice__ (i, j)

Equivalent to self.__getitem__(slice(i,j)) but defined mainly so that C code can use the sequence interface. Called to evaluate self[i:j]

__setslice__ (i, j, value)

Equivalent to self.__setitem__(slice(i,j), value) but defined mainly so C code can use the sequence interface. Called to evaluate self[i:j] = value.

__contains__ (item)

Called to determine truth value of the `item in self` construct. Returns the equivalent of (self==item).any()

3.3.4 Arithmetic customization

3.3.4.1 Binary

__add__ (other)

__sub__ (other)

__mul__ (self, other)

__div__ (other)

__truediv__ (other)

__floordiv__ (other)

__mod__ (other)

__divmod__ (other)

__pow__ (other[,modulo])

__lshift__ (other)

__rshift__ (other)

__and__ (other)

__or__ (other)

__xor__ (other)

These methods are defined for ndarrays to implement the operations (`+`, `-`, `*`, `/`, `/`, `//`, `%`, `divmod()`, `**` or `pow()`, `<<`, `>>`, `&`, `^`, `|`). This is done using calls to the corresponding universal function object (add, subtract, multiply, divide, true_divide, floor_divide, remainder, divide and remainder, power, left_shift, right_shift, bitwise_and, bitwise_xor, bitwise_or). These implement element-by-element operations for arrays that are broadcastable to the same shape.

- any third argument to `pow()` is silently ignored as the underlying ufunc (power) only takes two arguments.
- the three division operators are all defined, div is active by default, truediv is active when __future__.division is in effect.

NOTE

Because it is a built-in type (written in C), the __r<op>__ special methods are not directly defined for the ndarray.

3.3.4.2 In-place

__iadd__ (other)

__isub__ (other)

__imul__ (other)

__idiv__ (other)

__itruediv__ (other)

__ifloordiv__ (other)

__imod__ (other)

__ipow__ (other)

__ilshift__ (other)

__irshift__ (other)

__iand__ (other)

__ixor__ (other)

__ior__ (other)

These methods are implemented to handle the inplace operations (`+=`, `-=`, `*=`, `/=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). The inplace operators are implemented using the corresponding ufunc and its ability to take an output argument (which is set as self). Using inplace operations can save space and time and is therefore encouraged whenever appropriate.

WARNING

In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, a <op>= B can be different than a = a <op> B. For example, suppose a=ones((3,3)). Then a+=3j is different than a=a+3j While they both perform the same computation, a+=3j casts the result to fit back in a, while a=a+3j re-binds the name a to the result.

3.3.4.3 Unary operations

__neg__ (self)

__pos__ (self)

__abs__ (self)

__invert__ (self)

These functions are called in response to the unary operations (`-`, `+`, `abs()`, ˜). With the exception of __pos__, these are implemented using ufuncs (negative, absolute, invert). The unary `+` operator, however simply calls self.copy(), and can therefore be used to get a copy of an array.

__complex__ (self)

__int__ (self)

__long__ (self)

__float__ (self)

__oct__ (self)

__hex__ (self)

These functions are also defined for the `ndarray` object to handle the operations `complex()`, `int()`, `long()`, `float()`, `oct()`, and `hex()`. They work only on arrays that have one element in them and return the appropriate scalar.

TIP

The function called to implement many arithmetic special methods for arrays can be modified using the function set_numeric_ops. This function is called with keyword arguments indicating which operation(s) to replace. A dictionary is returned containing showing the old functions. By default, these functions are set to the corresponding ufunc.

3.4 Array indexing

More powerful array indexing was an important extension introduced by numarray, and was therefore an important part of the development of NumPy. In particular, the desire to select arbitrary elements based on their position in the array, and according to a mask was desirable.

There are two kinds of indexing available using the `X[obj]` syntax: basic slicing, and advanced indexing. For the description of this syntax given below, X is the array to-be-sliced and obj is the *selection* object. Furthermore, define $N \equiv$X.ndim. These two methods of slicing have different behavior and are triggered depending on obj. Adding additional functionality yet remaining compatible with old uses of slicing complicated the rules a little. Hopefully, after studying this section, you will have a firm grasp of what kind of selection will be initiated depending on the selection object.

TIP

in Python X[(exp1, exp2, ..., expN)] is equivalent to X[exp1, exp2, ..., expN] as the latter is just syntactic sugar for the former.

3.4.1 Basic Slicing

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when obj is a slice object (constructed by start:stop:step notation inside of brackets), an integer, or a tuple of slice objects and integers. Ellipsis and newaxis objects can be interspersed with these as well. In order to remain backward compatible with a common usage in Numeric, basic slicing is also initiated if the selection object is any sequence (such as a list) containing slice objects, the ellipsis object, or the newaxis object, but no integer arrays or other embedded sequences.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is '$i : j : k$' where i is the starting index, j is the stopping index, and k is the step ($k \neq 0$). This selects the m elements (in the corresponding dimension) with index values i, $i + k$, $\ldots$, $i + (m - 1)k$ where $m = q + (r \neq 0)$ where q and r are the quotient and remainder obtained by dividing $j - i$ by k: $j - i = qk + r$, so that $i + (m - 1)\,k < j$.

- Assume n is the number of elements in the dimension being sliced. Then, if i is not given it defaults to 0 for $k > 0$ and n for $k < 0$. If j is not given it defaults to n for $k > 0$ and -1 for $k < 0$. If k is not given it defaults to 1. Note that '::' is the same as ':' and means select all indices along this axis.

- If the number of objects in the selection tuple is less than N, then ':' is assumed for any remaining dimensions.
- Ellipsis expand to the number of ':' objects needed to make a selection tuple of the same length as X.ndim. Only one ellipsis is expanded, any others are interpreted as more ':'
- Each newaxis object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the newaxis object in the selection tuple.
- An integer, i, returns the same values as $i : i + 1$ **except** the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the p^{th} element an integer (and all other entries ':') returns the corresponding sub-array with dimension $N - 1$. If $N = 1$, then the returned object is an Array Scalar. These objects are explained in Chapter 6.
- If the selection tuple has all entries ':' except the p^{th} entry which is a slice object $i : j : k$, then the returned array has dimension N formed by concatenating the sub-arrays returned by integer indexing of elements i, $i + k$, $i + (m - 1)k < j$,
- Basic slicing with more than one non-':' entry in the slicing tuple, acts like repeated application of slicing using a single non-':' entry, where the non-':' entries are successively taken (with all other non-':' entries replaced by ':'). Thus, X[ind1,...,ind2,:] acts like X[ind1][...,ind2,:] under basic slicing. Note this is NOT true for advanced slicing.
- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in X[obj] = value must be (broadcastable) to the same shape as X[obj].

Basic slicing always returns another *view* of the array. In other words, the returned array from a basic slicing operation uses the same data as the original array. This can be confusing at first, but it is faster and can save memory. A copy can always be obtained if needed using the unary + operator (which has lower precedence than slicing) or the .copy() method.

TIP

Remember that a slicing tuple can always be constructed as obj and used in the x[obj] notation. Slice objects can be used in the construction in place of the [start:stop:step] notation. For example, `x[1:10:5,::-1]` can also be implemented as `obj=(slice(1,10,5), slice(None,None,-1)); X[obj]`. This can be useful for constructing generic code that works on arrays of arbitrary dimension.

3.4.2 Advanced selection

Advanced selection is triggered when the selection object, obj, is a non-tuple sequence object, an ndarray (of data type integer or bool), or a tuple with at least one sequence object or ndarray (of data type integer or bool). There are two types of advanced indexing: integer and Boolean. Advanced selection always returns a copy of the data (contrast with basic slicing that returns a view).

3.4.2.1 Integer

Integer indexing allows selection of arbitrary items in the array based on their N-dimensional index. This kind of selection occurs when advanced selection is triggered and the selection object is not an array of data type bool. For the discussion below, when the selection object is not a tuple, it will be referred to as if it had been promoted to a 1-tuple, which will be called the selection tuple. The rules of advanced integer-style indexing are:

- if the length of the selection tuple is larger than N(=X.ndim) an error is raised.
- all sequences and scalars in the selection tuple are converted to intp indexing arrays.
- all selection tuple objects must be convertible to intp arrays, or slice objects, or the Ellipsis (...) object.
- Exactly one Ellipsis object will be expanded, any other Ellipsis objects will be treated as full slice (':') objects. The Ellipsis object is replaced with as many full slice (':') objects as needed to make the length of the selection tuple N.

- If the selection tuple is smaller than N, then as many ':' objects as needed are added to the end of the selection tuple so that the modified selection tuple has length N.

- The shape of all the integer indexing arrays must be broadcastable to the same shape. Arrays are broadcastable if any of the following are satisfied

  1. The arrays all have exactly the same shape.
  2. The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
  3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

- The shape of the output (or the needed shape of the object to be used for setting) is the broadcasted shape.

  Example: If a.shape is (5,1), b.shape is (1,6), c.shape is (6,) and d.shape is () so that d is a scalar, then a, b, c, and d are all broadcastable to dimension (5,6). The array "a" acts like a (5,6) array where a[:,0] is broadcast to the other columns, "b" acts like a (5,6) array where b[0,:] is broadcast to the other rows, "c" acts like a (1,6) array and therefore a (5,6) where c[:] is broadcast to every row, and finally "d" acts like a (5,6) array where the single values is repeated.

- After expanding any ellipses and filling out any missing (':') objects in the selection tuple, then let N_t be the number of indexing arrays, and let $N_s = N - N_t$ be the number of slice objects. Note that $N_t > 0$ (or we wouldn't be doing advanced integer indexing).

- If $N_s = 0$ then the M-dimensional result is constructed by varying the index tuple $(i_1, \ldots, i_M)$ over the range of the result shape and for each value of the index tuple setting:

  $$\text{result}[i_1, \ldots, i_M]{=}\text{X}[\text{ind}_1[i_1, \ldots i_M],\ \text{ind}_2[i_1, \ldots, i_M],\ \ \text{etc.,}\ \ \text{ind}_N[i_1, \ldots, i_M].$$

  Example: Suppose the shape of the broadcasted indexing arrays is 3-dimensional and N is 2. Then the result is found by letting i, j, k run over the shape found by broadcasting ind$_1$, and ind$_2$,and for each i, j, k setting result$[i, j, k]$ = X[ind$_1[i, j, k]$, ind$_2[i, j, k]$].

- If $N_s > 0$, then partial indexing is done. This can be somewhat mind-boggling to understand, but if you think in terms of the shapes of the arrays involved, it can be easier to grasp what happens. In simple cases (*i.e.* one indexing array and $N - 1$ slice objects) it does exactly what you would expect (concatenation of repeated application of basic slicing). The rule for partial indexing is that the shape of the result (or the interpreted shape of the object to be used in setting) is the shape of X with the indexed subspace replaced with the broadcasted indexing subspace. If the index subspaces are right next to each other, then the broadcasted indexing space directly replaces all of the indexed subspaces in X. If the indexing subspaces are separated (by slice objects), then the broadcasted indexing space is first, followed by the sliced subspace of X.

Example 1: Suppose X.shape is (10,20,30) and ind is a (2,3,4) indexing intp array, then result=X[...,ind,:] has shape (10,2,3,4,30) because the (20,)-shaped subspace has been replaced with a (2,3,4)-shaped broadcasted indexing subspace. If we let i, j, k loop over the (2,3,4)-shaped subspace then result[...,i,j,k,:] = X[...,ind[i,j,k],:]. This example produces the same result as X.take(ind,axis=-2).

Example 2: Now let X.shape be (10,20,30,40,50) and suppose ind$_1$ and ind$_2$ are broadcastable to the shape (2,3,4). Then X[:,ind$_1$,ind$_2$] has shape (10,2,3,4,40,50) because the (20,30)-shaped subspace from X has been replaced with the (2,3,4) subspace from the indices. However, X[:,ind$_1$,:,ind$_2$,:] has shape (2,3,4,10,30,50) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use .transpose() to move the sups pace anywhere desired. This example cannot be replicated using take.

3.4.2.2 Boolean

This advanced selection occurs when obj is an array object of Boolean type (such as may be returned from comparison operators). It is always equivalent to (but faster than) X[obj.nonzero()] where as described above obj.nonzero() returns a tuple (of length obj.ndim) of integer index arrays showing the True elements of obj.

The special case when obj.ndim == X.ndim is worth mentioning. In this case X[obj] returns a 1-dimensional array filled with the elements of X corresponding to the True values of obj. It The search order will be C-style (last index varies the fastest). If obj has True values at entries that are outside of the bounds of X, then an index error will be raised.

You can also use Boolean arrays as element of the selection tuple. In such instances, they will always be interpreted as nonzero(obj) and the equivalent integer indexing will be done. In general you can think of indexing with Boolean arrays as indexing with nonzero(<Boolean>).

WARNING

the definition of advanced selection means that X[(1,2,3),] is fundamentally different than X[(1,2,3)]. The latter is equivalent to X[1,2,3] which will trigger basic selection while the former will trigger advanced selection. Be sure to understand why this is True. You should also recognize that x[[1,2,3]] will trigger advanced selection, but X[[1,2,slice(None)]] will trigger basic selection.

3.4.3 Flat Iterator indexing

As mentioned previously, X.flat returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that X.flat is a 1-dimensional view. X.flat can be used for integer indexing using 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

Chapter 4

Basic Routines

4.1 Creating arrays

array (object=, dtype=None, copy=True, order=None, subok=False, ndmin=0)

Create a new ndarray of data type, dtype (or determined from object if dtype is None). The shape of the new array will be determined from object. If copy is True, then ensure a copy of the object is made. If copy is False, then the returned object is a copy of the array only if dtype is not equivalent to the data type of object. If order is 'Fortran' then the resulting array will be in Fortran order, otherwise it is in C order. If subok (subclasses are O.K.) is True then pass through subclasses of the array object if possible. If subok is False then only ndarray objects may be returned. The ndmin parameter specifies that the returned array must have at least the given number of dimensions.

asarray (object=, dtype=None, order=None)

Exactly the same as array(...) except the default copy argument is False, and subok is always False. Using this function always returns the base class ndarray.

asanyarray (object, dtype=None, order=None)

Thin wrapper around array(...) with subok=1. You should use this routine if you are only making use of the array attributes, and believe the calculations that will follow would work with any subclass of the array. Use of this routine increases the chance that array subclasses will interact seamlessly with your function — returning the same subclasses.

require (object, dtype=None, requirements=None)

Require a Python object to be an ndarray (or a sub-class) of the given data-type if it can be cast safely, otherwise raise an error. The requirements, if given, are a sequence containing the requested combination of the flags 'C_CONTIGUOUS' ('C'), 'F_CONTIGUOUS' ('F'), 'ALIGNED' ('A'), 'WRITEABLE' ('W'), 'OWNDATA' ('O'), and the special directive 'ENSUREARRAY' ('E'). These strings dictate which flags should be set on the return array (note only one of 'F_CONTIGUOUS' or 'C_CONTIGUOUS' should be used and 'F_CONTIGUOUS' over-rides 'C_CONTIGUOUS'). The special directive 'ENSUREARRAY' makes sure that a base-class ndarray is returned instead of allowing sub-classes to pass through. This function is particularly useful in a Python interface to C-code (say called using ctypes).

arange (start=, stop=None, step=1, dtype=None)

Function similar to Python's built-in range() function except it returns an ndarray object. Return a 1-d array of data type, dtype (or determined from the start, stop, and step objects if None), that starts at start, ends *before* stop and is incremented by step. The returned array has length n where

$$n = \left\lceil \frac{\text{stop} - \text{start}}{\text{step}} \right\rceil$$

with element i equal to start $+ i \cdot$ step. If stop is None, then the first argument is interpreted as stop and start is 0.

NOTE

By definition of the ceiling function (denoted by $\lceil x \rceil$), we know that $x \leq \lceil x \rceil < x + 1$, therefore this definition of the length of arange guarantees that start$+n\cdot$step $\geq$ stop as well as start$+(n-1)\cdot$step $<$ stop.

isfortran (arr)

Equivalent to `arr.flags.fnc` and therefore returns True only if arr is Fortran-contiguous but not also C-contiguous.

empty (shape=, dtype=int, order='C')

Return an uninitialized array of data type, dtype, and given shape. The memory layout defaults to C-style contiguous, but can be made Fortran-style contiguous with a 'Fortran' order keyword.

empty_like (arr)

Syntactic sugar for empty(a.shape, a.dtype, isfortran(arr))

zeros (shape=, dtype=int, order='C')

Return an array of data type dtype and given shape filled with zeros. The memory layout may be altered from the default C-style contiguous with the order keyword.

zeros_like (arr)

Syntactic sugar for zeros(a.shape, a.dtype, isfortran(arr))

ones (shape=, dtype=int, order='C')

Syntactic sugar for a = zeros(shape, dtype, order); a+= 1.

fromstring (string=,dtype=int, count=-1, sep='')

If sep is '', then return a new 1-d array with data-type descriptor given by dtype and with memory initialized (copied) from the raw binary data in string. If count is non-negative, the new array will have count elements (with a `ValueError` raised if count requires more data than the string offers), otherwise the size of the string must be a multiple of the itemsize implied by dtype, and count will be the length of the string divided by the itemsize.

If sep is not '', then interpret the string in ASCII mode with the provided separator and convert the string to an array of numbers. Any additional white-space will be ignored.

fromfile (file=, dtype=int, count=-1, sep='')

Return a 1-d array of data type, dtype, from a file (open file object or string with the name of a file to read). The file will be read in binary mode if sep is the empty string. Otherwise, the file will be read in text mode with sep providing the separator string between the entries. If count is -1, then the size will be determined from the file, otherwise, up to count items will be read from the file. If fewer than count items are read, then a RunTimeWarning is issued indicating the number of items read.

frombuffer (buffer, dtype=intp, count=-1, offset=0)

Very similar to (binary-mode) fromstring in interpretation of the arguments, except buffer can be any object exposing the buffer interface (or any object with a __buffer__ attribute that returns a buffer exposing the buffer protocol). The new array shares memory with the buffer object. The new array will be read-only if the buffer does not expose a writeable buffer.

fromiter (iterator_or_generator, dtype=None)

Construct an array from an iterator or a generator. Only handles 1-dimensional cases. By default the data-type is determined from the objects returned from the iterator.

load (file)

Load a pickled array from an open file. If file is a string, then open a file with that name first. Except for the automatic file opening equivalent to cPickle.load(file)

loads (str)

Load a pickled array from a string. Equivalent to cPickle.loads(str).

indices (dimensions, dtype=intp)

Return an array of dtype representing n(=len(dimensions)) grids of indices each with variation in a single direction. The returned array has shape (n,)+dimensions. Compare with mgrid.

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

[[0, 1, 2],
[0, 1, 2]]])
```

fromfunction (function, dimensions, **kwargs)

Construct an array from a function called on a tuple of index grids. The function should be able to take array arguments and process them like ufuncs (use vectorize if it doesn't). The function should accept as many arguments as there are dimensions which is a sequence of numbers indicating the length of the desired output for each axis. Keyword arguments to function may also be passed in as keywords to fromfunction.

```
>>> print fromfunction(lambda i,j: i+j, (2,3))
[[ 0.  1.  2.]
[ 1.  2.  3.]]
```

identity (n, dtype=intp)

Return a 2-d array of shape (n,n) and data type, dtype with ones along the main diagonal.

where (condition[, x, y])

Returns an array shaped like condition, that has the elements of x and y respectively where condition is respectively true or false. If x and y are not given, then it is equivalent to nonzero(condition).

flatnonzero (arr)

Return indices that are non-zero in a flattened version of arr. Equivalent to a.ravel().nonzero()[0].

putmask (arr=, mask=, values=)

Performs the equivalent of

```
for n, obj in enumerate(mask.flat):
    if obj:
        self.flat[n] = values[n]
```

The values array is repeated if it is too short. In particular, this means that indexing on the values array is modular it's length, which might be surprising you are expecting putmask to work the same as arr[mask]=values.

lexsort (keys=, axis=-1)

Return an array of indices similar to argsort except sorting is done using all of the provided keys. First a sort is computed using key[0], then the indices are further altered by sorting on key[1]. This is repeated until sorting has been performed on all of the keys. This is a useful function for multiple-field sorting.

```
>>> a = [1,2,1,3,1,5]; b = [0,4,5,6,2,3]
>>> ind = lexsort((b,a))
>>> print take(a,ind)
[1 1 1 2 3 5]
>>> print take(b,ind)
[0 2 5 4 6 3]
```

Notice the order the keys had to be used in order to get a lexicographical sorting order. To clarify, suppose three equal-length sequences are fields of an underlying data-type: (f1,f2,f3). If we want to sort first on f1 and then on f2 and then on f3, the indices that would accomplish that sort are obtained as lexsort((f3,f2,f1)).

4.2 Operations on two or more arrays

concatenate (seq=, axis=0)

Construct a new array from elements of the sequence object seq concatenated along the given axis. The elements of the sequence object must have compatible types and be the same shape. If axis is None, then flatten each element of seq before concatenating together to construct a 1-d array.

correlate (x, y, mode='valid')

Compute the 1-d cross correlation of x and y keeping portions determined by mode which may be 'valid' (0), 'same' (1), or 'full' (2). The 'full' cross-correlation between two 1-d arrays is computed as

$$z[n] = \sum_{i=\max(n-M,0)}^{\min(n,K)} x[i]\, y[n+i]\,,$$

for $n = 0 \ldots K + M$ where K=len(x)-1 and M=len(y)-1, and we assume $K \geq M$ (without loss of generality because we can interchange the roles of x and y without effect). For this formula to work, we assume that $x[i] = 0$ when $i \notin [0, K-1]$ and $y[j] = 0$ when $j \neq [0, M-1]$.

If mode is 'same' then only the K middle values are returned starting at $n = \left\lfloor \frac{M-1}{2} \right\rfloor$. If the flag has a value of 'valid' then only the middle $K - M + 1 = (K+1) - (M+1) + 1$ output values are returned starting at $n = M$.

convolve (x, y, mode='valid')

Convolution is very similar to correlation except it is defined with one sequence reversed:

$$z[n] = \sum_i x[i]y[n-i].$$

The mode keyword has the same effect as it does for correlation. Convolution ('full') between two 1-d arrays implements polynomial multiplication where the array entries are viewed as coefficients for polynomials.

Example: Consider that $(x^3 + 4x^2 + 2)(x^4 + 3x + 1) = x^7 + 4x^6 + 5x^4 + 13x^3 + 4x^2 + 6x + 2$. This can be determined by using the code `convolve([1,4,0,2], [1,0,0,3,1])` which returns `[1,4,0,5,13,4,6,2].` Notice the one-to-one alignment between the elements of the arrays and the coefficients on powers of x in the polynomial.

outer (a, b)

compute an outerproduct which is syntactic sugar for a.ravel() [:,newaxis] * b.ravel() [newaxis,:] (after first converting a and b to ndarrays).

```
>>> print outer([1,2,3],[10,100,1000])
[[  10  100 1000]
[  20  200 2000]
[  30  300 3000]]
```

inner (a, b)

Computes the inner product between two arrays. This is an array that has shape a.shape[:-1] + b.shape[:-1] with elements computed as the sum of the product of the elements from the last dimensions of a and b. In particular, let I and J be the super[1] indices selecting the 1-dimensional arrays $a[I,:]$ and $b[J,:]$, then the resulting array, r, is

$$r[I,J] = \sum_k a[I,k]b[J,k].$$

[1] A super index is 0 or more integer indices used to index into an N-dimensional array. How many indices a super index represents should be implied by context.

dot (a, b)

Computes the dot (matrix) product between two arrays. The product-sum is over the last dimension of a and the second-to-last dimension of b. Specifically, if I and J are super indices for $a[I,:]$ and $b[J,:,j]$ so that j is the index of the last dimension of b. Then, the shape of the resulting array is a.shape[:-1] + b.shape[:-2] + (b.shape[-1],) with elements.

$$r[I,J,j] = \sum_k a[I,k]b[J,k,j],$$

vdot (a, b)

Computes the dot product between two arrays (flattened into one-dimensional vectors) after conjugating the first vector. This is an inner-product following the physicists convention of conjugating the first argument.

$$r = \sum_k \overline{\text{a.flat}[k]}\text{b.flat}[k].$$

tensordot (a, b, axes=(-1,0))

Computes a dot-product between two arrays where the sum is taken over the axes specified by the 2-sequence which can have either scalar or sequence entries. The axes specified are summed over and the remaining axes are used to construct the result. So, for example, if a is $3 \times 4 \times 5$ and b is $4 \times 3 \times 2$ then if axes=([1,0],[0,1]) (or axes=([0,1],[1,0])) the result will be 5×2. Let I represent the indices of the un-summed axes in a, let J represent the indices of the un-summed axes in b and let K represent the the indices of the axes summed over in both a and b. Also, let a_t represent a transposed version of a where the axes to be summed over are pushed to the end, and let b_t represent a transposed version of b where the axes to be summed over are pushed to the front. Then, using $\sum_K$ to represent a multi-index sum, the result can be written as

$$r[I,J] = \sum_K a_t[I,K]b_t[K,J]$$

cross (a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)

Returns the cross product of two (arrays of) vectors. The cross product is performed over the axes of the input arrays indicated by the axisa, and axisb arguments. For both arrays, the axis used must have dimension either 2 or 3. If both axes used have dimension 2, then only the z-component of the equivalent 3-d cross product is returned. Otherwise, the entire vector is returned. The axisc argument gives the axis of the vectors in the returned cross-product result. If axis is not None, then it is assumed that axisa=axisb=axisc=axis (regardless of what else is specified).

allclose (a, b, rtol=10^{-5}, atol=10^{-8})

Returns true if all components of a and b are equal subject to the given relative and absolute tolerances. This returns true if every element of a and b satisfy

$$|a - b| < \text{atol} + \text{rtol}\,|b|\,.$$

4.3 Printing arrays

array2string (a)

The default printing mechanism uses this function to produce a string from an array.

set_printoptions (precision=None, theshold=None, edgeitems=None, linewidth=None, suppress=None)

Set options associated with representing an array.

precision the default number of digits of precision for floating point output (default 8);

threshold total number of array elements which triggers printing only the "ends" of the array rather than a full representation (default 1000);

edgeitems number of array elements in summary at beginning and end of each dimension (default 3);

linewidth the number of characters per line (default 75);

suppress Boolean value indicating whether or not to suppress printing of small floating point values using scientific notation (default False).

get_printoptions ()

Returns the values of precision, threshold, edgeitems, linewidth, and suppress that control printing of arrays.

set_string_function (func, repr=1)

Set the function to use in response to str(array) or repr(array). By default this function is array2string. The function passed in must take an array argument and return a string. If func is None, then the print function is reset to a simple internal function.

4.4 Functions redundant with methods

Several functions are available primarily for purposes of backward compatibility with old Numeric, and are therefore redundant. The functions are all simple wrappers for asarray(a).<function>(*args, **kwds), or are replaceable by attribute access. The following list documents them. It is not recommended that these functions be used in new programs, but there are no plans for removing them as in functional form they work with arbitrary sequences which is sometimes desirable. The functions that mirror methods and attributes are: **take**, **reshape**, **squeeze**, **choose**, **repeat**, **put**, **swapaxes**, **transpose**, **real**, **imag**, **sort**, **argsort**, **amax, argmax**, **amin**, **argmin**, **ptp**, **alen**, **searchsorted**, **diagonal**, **trace**, **ravel**, **nonzero**, **shape**, **compress**, **clip**, **std**, **var**, **mean**, **sum**, **cumsum**, **product**, **cumproduct**, **sometrue** (method is .any), **alltrue** (method is .all), **around** (method is .round), **rank** (attribute is .ndim), **shape**, **size** (.size or .shape[axis]), and **copy**.

4.5 Dealing with data types

dtype (obj, align=0)

Return a data-type object from any object. See Chapter 7 for a more detailed explanation of what can be interpreted as a data-type object and the meaning of the align keyword.

maximum_sctype (arg)

Returns the array-scalar type of highest precision of the same general kind as arg which can be any recognized form for describing a data-type.

issctype (obj)

Returns True if obj is an array data type (or a recognized alias for one)

obj2sctype (obj, default=None)

Returns the array type object corresponding to obj which can be an array type already, a python type object, an actual array, or any recognized alias for an array type object. If no suitable data type object can be determined, return default.

sctype2char (sctype)

Return the typecode character associated with an array-scalar type dtype. The first argument is first converted to a dtype if it needs to be.

TIP

the type attribute of data-type objects are actual Python type objects subclassed in a hierarchy of types. This can often be useful to check data types generically. For example, issubclass(dtype.type, integer) can check to see if the data type is one of the 10 different integer types. The issubclass function, however, raises an error if either argument is not an actual type object. NumPy defines _(arg1, arg2) that will return false instead of raise an error. Alternatively, dtype.kind is a character describing the class of the data-type so dtype.kind in 'iu' would also check to see if the data-type is an integer type.

can_cast (from=d1, to=d2)

Return Boolean value indicating whether or not data type d1 can be cast to data type d2 safely (without losing precision or information).

Chapter 5

Additional Convenience Routines

5.1 Shape functions

atleast_1d (a1,a2,...,an)

Force a sequence of arrays (including array scalars) to each be at least 1-d.

atleast_2d (a1,a2,...,an)

Force a sequence of arrays (including array scalars) to each be at least 2-d. Dimensions of length 1 are pre-pended to reach a two-dimensional array.

atleast_3d (a1,a2,...,an)

Force a sequence of arrays (including array_scalars) to each be at least 3-d. Dimensions of length 1 are pre-pended to reach a two-dimensional array.

roll (arr, shift, axis=None)

Return a new array with the contents of arr shifted (rolled) by the amount given in the integer argument shift along the axis specified. If axis is None, then the shift takes place in the ravelled array (but the returned array has the same shape as arr). Elements that shift outside the array are rolled back into the array from the opposite side.

rollaxis (arr, axis, start)

Return arr transposed so that the provided axis is inserted into the shape before start with the other dimensions rolled. Thus, if arr.shape is (i,j,k,l) then rollaxis(arr, 2, 0) has shape (k,i,j,l) and rollaxis(arr, 1, 3) has shape (i,k,j,l).

vstack (seq)

Stack a sequence of arrays along the first axis (row wise). Arrays in seq must have the same shape along all dimensions but the first. Rebuilds array divided by vsplit. All 1-d arrays will be stacked row-wise.

hstack (seq)

Stack a sequence of arrays along the second axis (column wise). Arrays in seq must have the same shape along all dimensions but the second. Rebuilds array divided by hsplit. Notice that 1-d arrays will be appended into a new 1-d array. Use column_stack to get a 2-d array from 1-d arrays. If some arrays are already 2-d, then the 1-d arrays need to have a dimension added to the end (*e.g.* `y[:,newaxis]`) in order to stack correctly.

column_stack (seq)

Stack a sequence of arrays as columns into a 2-d array. 1-d arrays are converted to 2-d arrays and transposed. All arrays must have shapes so that the resulting array is well defined. Compare with **hstack**.

row_stack (seq)

Stack a sequence of 1-d arrays as rows into a 2-d array (alias for **vstack**).

dstack (seq)

Stack a sequence of arrays along the third axis (depth wise). Arrays in seq must have the same shape along all dimensions but the third. Rebuilds array divided by vsplit.

array_split (ary, i_or_s, axis=0)

Divide ary into a list of sub-arrays along the specified axis. The i_or_s argument stands for indices_or_sections. If i_or_s is an integer, ary is divided into that many equally-sized arrays. If it is impossible to make an even split, each of the leading arrays in the returned list have one additional member. If i_or_s is a list of sorted integer, its entries define the indexes where ary is split. An empty list for i_or_s results in a single sub-array equal to the original array.

split (ary, i_or_s, axis=0)

The same as array_split() except if i_or_s is an integer and it is impossible to make an even split, an error is raised.

hsplit (ary, i_or_s)

Split a single array into multiple columns of sub-arrays (along the first axis if 1-d or along the second second if >1-d). Only works on arrays of 1 or more dimension.

vsplit ()

Split a single array into multiple rows of sub-arrays (along the first axis). Only works on arrays of 2 or more dimensions.

dsplit ()

Split a single array into multiple sub-arrays along the third axis (depth). Only works on arrays of 3 or more dimensions.

apply_along_axis (func1d, axis, arr, *args)

Execute func1d(arr[sel_i], *args) where func1d takes 1-d arrays and arr is an N-d array, where sel_i is a selection object sufficient to select a 1-d sub-array along the given axis. The function is executed for all 1-d arrays along axis in arr.

apply_over_axes (func, a, axes)

For each axis in the axes sequence, call func as `res=func(a, axis)`. If res is the same shape as a then set `a=res` and continue. if `res.ndim = a.ndim -1`, then insert a dimension before axis and continue.

expand_dims (a, axis)

Expand the shape of array a by including newaxis **before** the given axis.

resize (a, new_shape)

Returns a new array with the specified shape which can be any size. The new array is filled with repeated copies of a. This function is similar in spirit to a.resize(new_shape) except that it fills in the new array with repeated copies and returns a new array.

kron (a, b)

Return a composite array with blocks from b scaled by elements of a. The number of dimensions of a and b should be the same. If not, then the input with fewer dimensions is pre-pended with ones (broadcast) to the same shape as the input with more dimensions. The return array has this same number of dimensions with shape given by the product of the shape of a and the shape of b. If either a or b is a scalar then this function is equivalent to multiply(a,b).

For example, if a and b are is 1-d the result is

$$\begin{bmatrix} a[0]*b & a[1]*b & \cdots & a[-1]*b \end{bmatrix}$$

while if a and b are 2-d, the result is

$$\begin{bmatrix} a[0,0]*b & a[0,1]*b & \cdots & a[0,-1]*b \\ a[1,0]*b & a[1,1]*b & \cdots & a[1,-1]*b \\ \vdots & \vdots & \ddots & \vdots \\ a[-1,0]*b & a[-1,1]*b & \cdots & a[-1,-1]*b \end{bmatrix}$$

Example:

```
>>> kron([1,10,100],[5,6,7])
array([  5,   6,   7,  50,  60,  70, 500, 600, 700])
>>> kron([[1,10],[100,1000]],[[2,3],[4,5]])
array([[   2,    3,   20,   30],
[   4,    5,   40,   50],
[ 200,  300, 2000, 3000],
[ 400,  500, 4000, 5000]])
```

tile (a, reps)

Tile an N-dimensional array using the shape information in reps to create a larger N-dimensional array. This is equivalent to kron(ones(reps, a.dtype), a). The number of dimensions of a and the length of shape should be the same or else 1's will be pre-pended to make them the same.

Example:

```
>>> tile([5,6,7],(1,2,3))
array([[[5, 6, 7, 5, 6, 7, 5, 6, 7],
[5, 6, 7, 5, 6, 7, 5, 6, 7]]])
```

5.2 Basic functions

average (a, axis=None, weights=None, returned=0)

Computes the average along the indicated axis. If axis is None, average over the entire array. Inputs can be integer or floating types; result is type float. If weights are given, the result is sum(a*weights)/sum(weights). Therefore, weights must have shape equal to a.shape or be 1-d with length a.shape[axis]. Integer weights are converted to float. If returned is True, then return a tuple showing both the result and the sum of the weights (or count of the values). The shape of these two results will be the same.

cov (x, y=None, rowvar=1, bias=0)

Compute the covariance matrix of data in x. If x is a vector and y is None, then this function is equivalent to asarray(x).var(). Otherwise, x is interpreted as observations of several random variables. If rowvar is True (default), then the variables are in the rows and the observations of the variables are in the columns. Otherwise, the variables are in the columns and the observations are in the rows. If y is given then it is treated as another variable or set of variables to be added to x. By default, a so-called unbiased estimate of the covariance matrix is made. If bias is non-zero, then a biased normalization factor (with better mean-square error performance) is used instead. If $\mathbf{X}$ is a random vector, then the covariance matrix is defined as

$$\mathbf{C} = E\left[\left(\mathbf{X} - E\mathbf{X}\right)\left(\mathbf{X} - E\mathbf{X}\right)^H\right].$$

It can be approximated as

$$\mathbf{C} \approx \frac{1}{P}\sum_{i=0}^{N-1}\left(\mathbf{x}_i - \bar{\mathbf{x}}\right)\left(\mathbf{x}_i - \bar{\mathbf{x}}\right)^H$$

where $\mathbf{x}_i$ is an observation of $\mathbf{X}$ (as a column-vector), N is the number of observations made and $P = N - 1$ for an unbiased estimate or $P = N$ for a biased (but lower mean-squared error) estimate.

corrcoef (x, y=None, rowvar=1, bias=0)

Estimate the correlation coefficient of x. By default, each row of x contains a random variable with observations of the random variable in the columns of x. (If rowvar is False, the each column is a random variable with observations in the rows). The y argument can be used to append additional variables to x. The i^{th} row and j^{th} column of the correlation coefficient matrix is defined as

$$\rho_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}$$

where $\mathbf{C}$ is the covariance matrix. The rowvar and bias arguments are passed on to the cov function to estimate $\mathbf{C}$.

msort (a)

Return a new array, sorted along the first axis. Equivalent to b=a.copy(); b.sort(0)

median (m)

Returns the median of m along its first dimension.

bincount (list=, weights=None)

The list argument is a 1-d integer array. Let r be the returned 1-d array whose length is (list.max()+1). If weights is None, then $r[i]$ is the number of occurrences of i in list. If weight is present, then the i^{th} element is

$$r[i] = \sum_{j:\text{list}[j]=i} \text{weights}[j].$$

Notice that if weights is None, it is equivalent to a weights array of all 1. The length of weights must be the same as the length of list.

digitize (x=,bins=)

Return an array of integers the same length as x with values i such that $\text{bins}[i-1] \le x < \text{bins}[i]$ if bins is monotonically increasing, or $\text{bins}[i] \le x < \text{bins}[i-1]$ if bins is monotonically decreasing. When x is beyond the bounds of bins, return either $i = 0$ or $i = \text{len(bins)}$ as appropriate.

histogram (x=, bins=None, range=None, normed=0)

Construct a histogram for the data in x (treated as one-dimensional array of type float). If bins is not a sequence, then bins should be the number of bins which will be constructed ranging from range[0] to range[1] or x.min() to x.max() if range is None. If normed is True, then the histogram will be normalized and comparable with a probability density function, otherwise it will be a count of the number of items in each bin. The return value is the tuple (n, bins) where n is the histogram.

histogram2d (x, y, bins=10, range=None, normed=False)

Compute the two-dimensional histogram for a dataset (x,y) given the bins. Returns (histogram, xedges, yedges). The bins argument can be either the number of bins or a sequence of the bin edges if the x and y directions should have the same bins. If the bins argument is a sequence of length 2, then separate bin edges will be computed. The first element can be either the number of bins or the bin edges for the x-direction. The second element is interpreted as the number of bins or the bin edges for the y-direction. The returned histogram array, H, is a count of the number of samples in each bin. The array is oriented such that H[i,j] is the number of samples falling into binx[j] and biny[i] (notice the association x<->j and y<->i). Setting normed to True returns a density rather than a bin-count. The range argument allows specifying lower and upper bin edges (in a sequence of length 2 with 2-length sequences in each entry). The default is [[x.min(), x.max()],[y.min(), y.max()]].

histogramdd (sample, bins=10, range=None, normed=False)

Compute the D-dimensional histogram for a (vector) dataset contained in sample give the bins. The dataset is a sequence of D arrays or an $N \times D$ array where N is the number of samples and D is the number of dimensions. Returns (histogram, edges) where histogram is a D-dimensional array of shape given by the number of bins selected in each axis containing the number of counts that a point in the sample data fell into the volume bin specified. The edges sequence has D-entries to specify the edge boundaries for each dimension. The bins argument is a sequence of edge arrays or a sequence of the number of bins. If a scalar is given, it is assumed to be the number of bins for all dimensions. The range is a length-D sequence containing lower and upper bin edges which default to the min and maximum of the respective datasets. If normed is True, then a density rather than a bin-count is returned.

logspace (start, stop, num=50, endpoint=True,base=10.0)

Evenly spaced samples on a logarithmic scale. Returns num evenly spaced (in logspace) samples from base**start to base**stop. If endpoint is True, then the last sample is base**stop.

linspace (start, stop, num=50, endpoint=True, retstep=False):

Evenly spaced samples. Returns num evenly spaced samples from start to stop. If endpoint is True, then the last sample is stop. If retstep is True, then return the computed step size.

meshgrid (x, y)

For 1-d arrays x, y with lengths Nx=len(x) and Ny = len(y), return X, Y where X and Y are (Ny, Nx) shaped arrays with the elements of x and y repeated to fill the array.

```
>>> X,Y = meshgrid([1,2,3], [4,5,6,7]); print X; print Y
[[1 2 3]
[1 2 3]
[1 2 3]
[1 2 3]]
[[4 4 4]
[5 5 5]
[6 6 6]
[7 7 7]]
```

select (condlist, choicelist, default=0)

Returns an array comprised from different elements of choicelist depending on the list of conditions. The condlist argument is a list of Boolean condition arrays. The choicelist argument is a list of choice arrays (of the same size as the arrays in condlist). The result has the same size as the arrays in choicelist. If condlist is $[c_0, \ldots, c_{N-1}]$, then choicelist must be of length N. The elements of choicelist can then be represented as $[v_0, \ldots, v_{N-1}]$. The default choice if none of the conditions are met is given as the default argument. The conditions are tested in order and the first one satisfied is used to select the choice. In other words, the elements of the output array are found from the following tree (evaluated on an element-by-element basis)

```
if c_0: v_0
elif c_1: v_1
...
elif c_{N-1}: v_{N-1}
else: default
```

piecewise (x, condlist, funclist, *args, **kw)

Compute a piecewise-defined function. A piecewise defined function is

$$f(x) = \begin{cases} f_1(x) & x \in S_1, \\ f_2(x) & x \in S_2, \\ \vdots & \vdots \\ f_n(x) & x \in S_n. \end{cases}$$

where S_1 are sets. Thus, the function is defined differently over different sub-domains of the input. Such a function can be computed using `select` but such an implementation means calling each f_i over the entire region of x. The piecewise call guarantees that each function f_i will only be called over those values of x in S_i.

Arguments: x is the array of values over which to call the function; condlist is a sequence of Boolean (indicator) arrays (or a single Boolean array) of the same shape as x that defines the sets (True indicates that element of x is in the set). If needed, to match the length of funclist, an "otherwise" set will be added to condlist. This otherwise set is defined as $S_n = \overline{\bigcup S_i}$. The argument funclist is a list of functions to be called (or items to be inserted) corresponding to the conditions. Each of these functions can take extra arguments and key-word arguments which are passed in as *args, and **kw using standard Python syntax. Each of these functions should return vector output for vector input. If the function is a scalar, then it will simply be inserted where appropriate into the output. It is the equivalent of a constant function.

Example: Suppose we want to compute $f(x) = x^2\Pi\left(\frac{x}{3}\right) + u(x-5)$ where $\Pi(x) = 1$ only when $|x| \leq 1$ and $u(x) = 1$ only when $x \geq 0$. This could be done using the code:

```
>>> f1 = lambda x: x*x
>>> x = r_[-4:6:20j]
>>> y = piecewise(x,abs(x)<=3,[f1,0])+piecewise(x,x>=0,[1,0])
>>> set_printoptions(precision=4); print y
[ 0.      0.      8.687   5.8615  3.59    1.8726  0.7091  0.0997
1.0443  1.5429  2.5956  4.2022  6.3629  9.0776  1.      1.      1.
1.      1.      1.    ]
```

trim_zeros (filt, trim='fb'):

Trim the leading ('f' in trim) and trailing ('b' in trim) zeros from a sequence according to the trim keyword.

trapz (y, x=None, dx=1.0, axis=-1)

If **y** contains samples of a function: $y_i = f(x_i)$ then trapz can be used to approximate the integral of the function using the trapezoidal rule. If the sampling is not evenly spaced use **x** to pass in the sample positions. Otherwise, only the sample-spacing is needed in dx. The trapz function can work with many functions at a time stored in an N-dimensional array. The axis argument controls which axis defines the sampling axis (the other dimensions are different functions). The number of dimensions of the returned result is y.ndim - 1.

diff (x, n=1, axis=-1)

Calculates the n^{th} order, discrete difference along the given axis.

gradient (f, *varargs)

Calculate the gradient of an N-d scalar function, f. Uses central differences on the interior and first differences on boundaries to give the same shape for each component of the gradient. The varargs variable can contain 0, 1, or N scalars corresponding to the sample distances in each direction (default 1.0). If f is N-d, then N arrays are returned each of the same shape as f, giving the derivative of f with respect to each dimension.

angle (z, deg=0)

Return the angle of a complex number z (in degrees if deg is True).

unwrap (p, discont=pi, axis=-1)

Unwraps radian phase p by changing absolute jumps greater than discont to their 2π complement along the given axis.

sort_complex (x)

This is syntactic sugar for asarray(x).sort().astype(<cmplx_type>) where cmplx_type is csingle if x.dtype is integral with fewer bits than intp, clongfloat if x.dtype.type is longfloat, and cdouble for all other types. The sorting is done by comparing the real part of the array, and then the imaginary part if the real parts are the same.

disp (mesg, device=None, linefeed=1)

Display a message to device (defaults to sys.stdout) with or without a closing linefeed.

unique (seq)

Returns unique items in the 1-dimensional seq.

extract (condition, arr)

Equivalent to arr.compress(condition.flat) and arr.flat[bool_(condition.flat)] which extracts the elements of (flattened) arr according to the elements of (flattened) condition that are True.

place (arr, mask, vals)

Inverse of extract. Equivalent to arr[abool(mask)] = vals but it uses a different algorithm.

delete (arr, indices, axis=None)

Return a new array with the sub-arrays indicated by indices along axis removed. If axis is None, then first ravel the array and set axis to -1. The indices argument describes which sub-arrays along the given axis should be removed. It can be an integer, a slice object, or a sequence of integers. A new array is created with the corresponding sub-arrays are removed.

insert (arr, indices, values, axis=None)

Create a new array with values inserted into arr before indices. If axis is None, then first ravel the array and set axis to -1. The indices argument describes which indices along the provided axis the values should be inserted before.

It can be an integer, a slice object, or a sequence of integers. The values argument must be broadcastable to the shape implied by where they will be inserted.

append (arr, values, axis=None)

Return a new array with values appended to the end of the array along axis.

nansum (x, axis=None)

nanmax (x, axis=None)

nanargmax (x, axis=None)

nanargmin (x, axis=None)

nanmin (x, axis=None)

These functions perform their respective operations over the given axis (or the entire array if axis is None), after replacing any nans with appropriate values so as not to affect the calculation.

vectorize (pyfunc, otypes=None, doc=None)

This creates a class whose instances have a call method that invokes a ufunc that has been dynamically built to call the python function pyfunc internally. The output types can be controlled by the otypes argument. If it is None, then the output types will be determined upon first call to the function using the provided inputs. This can be reset, by re-setting the otypes attribute to "". The normal rules of array broadcasting are followed by the returned object.

```
>>> def myfunc(a,b):
...     if (a>b): return a
...     else: return b-1
>>> vecfunc = vectorize(myfunc)
>>> vecfunc([[1,2,3],[5,6,9]],[7,4,5])
array([[6, 3, 4],
[6, 6, 9]])
```

asarray_chkfinite (x)

Like asarray(x) except an error is raised if any of the values in x are not finite.

round_ (x, decimals=0)

Return an array with all the elements of x rounded to decimals places. Returns x if array is not floating point and rounds both the real and imaginary parts separately if array is complex. Rounds in the same way as standard python except for half-way values are rounded to the nearest *even* number.

add_docstring (obj, doc)

Adds a docstring to a built-in object, obj, that does not have a docstring defined already. The obj can be a built-in function-or-method, a typeobject, a method descriptor, a getset descriptor, or a member descriptor. This is useful for improving the documentation of objects defined in C-compiled code without re-compiling. If the object already has a docstring, a RuntimeError is raised. If the object is not a supported type the code can add a docstring to, a TypeError is raised.

add_newdoc (place, obj, doc)

Adds a docstring to the *obj* imported from *place* using exec 'from %s import %s' % (place, obj). Thus, both place and obj should be strings. If doc is a string, then a single docstring is added to obj from place. If doc is a 2-tuple, then obj must be an object with attributes that need to be commented. The first element of the doc tuple is the attribute to be commented on and the second element is the actual docstring. If doc is a list, then it must be composed of elements that are 2-tuples indicating that obj has several attributes that need to be documented.

5.3 Polynomial functions

There are two interfaces for dealing with polynomials: a class-based interface, and a collection of functions to deal with a polynomials represented as a simple list of coefficients. This latter representation results from the is a one-to-one correspondence between a length-$(n + 1)$ sequence of coefficients $a_n \equiv a[n]$ and an n^{th} order polynomial:

$$p(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n.$$

Most of the functions below operate on and return a simple sequence of coefficients representing a polynomial. There is, however, a simple polynomial class that provides some utility for doing simple algebra on polynomials.

poly1d (c_or_r, r=0)

This construction returns an instance of a simple polynomial class. It can take either a list of coefficients on polynomial powers, or a sequence of roots (if r=1). The returned polynomial can be added, subtracted, multiplied, divided, and taken to integer powers, resulting in new polynomials.

.r roots of the polynomial

.o order of the polynomial

.c polynomial coefficients as an array (also **__array__()**)

__call__(x) evaluate the polynomial at x (can be an array)

__getitem__(x) p[k] returns the coefficient on the kth power of x (backwards from indexing the coefficient array)

```
>>> p=poly1d([2,5,7])
>>> print p
2
2 x + 5 x + 7
>>> print p*[1,3,1]
4      3      2
2 x + 11 x + 24 x + 26 x + 7
>>> print p([0.5,0.6,3])
[ 10.    10.72  40.  ]
>>> print p.r
[-1.25+1.3919j -1.25-1.3919j]
```

poly (roots_or_matrix)

Return a sequence of coefficients representing a polynomial given the sequence of roots as an argument. Alternatively, if the argument is a 2-d array, then return the characteristic polynomial of the matrix.

roots (poly)

Return the roots of the polynomial represented by coefficients in poly

polyint (poly, m=1, k=None)

Return an exact m^{th}-order integral of the polynomial represented in poly. If k is None, then use 0 for the integrating constants. Otherwise, use the scalars in the sequence k as integrating constants. Also available as .integ (m=1,k=0) method of poly1d objects.

Example:

$$\begin{aligned} p(x) &= x^2+3x+4 \\ \int\int p(x) &= \frac{1}{12}x^4+\frac{1}{2}x^3+2x^2+k_0x+k_1 \end{aligned}$$

```
>>> print polyint([1,3,4],m=2,k=[5,3])
[ 0.0833  0.5     2.      5.      3.    ]
```

polyder (poly, m)

Return an exact m^{th}-order derivative of the polynomial represented in poly. Also available as .deriv(m=1) method of poly1d objects.

Example:

$$\begin{aligned} p(x) &= x^3+2x^2+4x+3 \\ \frac{dp}{dx}(x) &= 3x^2+4x+4 \end{aligned}$$

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

polyadd (p1, p2)

Add the two polynomials represented by coefficients: $p_1(x)+p_2(x)$

polysub (p1, p2)

Return coefficients for the polynomial found by subtracting the two polynomials represented by p_1 and p_2: $p_1(x)-p_2(x)$

polymul (p1, p2)

Return the coefficients for $p_1(x)\,p_2(x)$

polydiv (p1, p2)

Return the quotient, $q(x)$, and remainder, $r(x)$, so that $p_1(x) = q(x)\,p_2(x) + r(x)$, with the order of $r(x)$ less than the order of $p_2(x)$.

polyval (p, y)

Evaluate the polynomial p at y. The argument, y, can be a number or an array or a polynomial object. If x is a polynomial object, then polyval performs polynomial composition: $p(y(x))$, otherwise polyval computes the value of the polynomial at each y. Uses Horner's rule for evaluation, but this can still lead to numerical instabilities for wildly fluctuating coefficients.

polyfit (x,y,N)

Compute a best-fit polynomial in x of order N, to the data, y, in the sense of minimizing averaged-squared error between the measurement and the model. Useful for quick line-fitting.

5.4 Set Operations

The set operations were kindly contributed by Robert Cimrman. These set operations are based on sorting functions and all expect 1-d sequences with unique elements with the exception of unique1d and intersect1d_nu which will flatten N-d nested-sequences to 1-d arrays and can handle non-unique elements.

unique1d (arr, retindx=False)

Return the unique elements of arr as a 1-d array. If retindx is True, then also return the indices, ind, such that arr.flat[ind] is the set of unique values.

intersect1d (a1, a2)

Return the (sorted) intersection of a1 and a2 which is an array containing the elements of a1 that are also in a2.

intersect1d_nu (a1, a2)

Return the (sorted) intersection of a1 and a2 but allow a1 and a2 to be N-d arrays with non-unique elements. Equivalent to intersect1d(unique1d(a1), unique1d(a2)).

union1d (a1, a2)

Return the (sorted) union of a1 and a2 which is an array containing elements that are in either a1 or a2.

setdiff1d (a1, a2)

Return the set-difference of a1 and a2 which is an array containing the elements of a1 that are **not** in a2.

setxor1d (a1, a2)

Return the (sorted) set containing the exclusive-or of the arrays a1 and a2. The exclusive-or contains elements that are in a1 or in a2 as long as the element is not in both a1 and a2.

setmember1d (tocheck, set)

Return a Boolean 1-d array of the length of tocheck which is True whenever that element is contained in set and false when it is not. Equivalent to array([x in set for x in tocheck]).

5.5 Array construction using index tricks

The functions and classes in this category make it simpler to construct arrays.

ix_ (*args)

This indexing cross function is useful for forming indexing arrays necessary to select out the cross-product of N 1-dimensional arrays. Note that the default indexing does not do a cross-product (which might be unexpected for someone coming from other programming environments). The default indexing is more general purpose. Using the ix_ constructor can produce the indexing arrays necessary to select a cross-product.

mgrid [index expression]

This is an instance of a class. It can be used to construct a filled "mesh-grid" using slicing syntax.

ogrid [index expression]

This is similar to mgrid except it returns an open grid, so as to save space and time. The broadcasting rules will ensure that any universal function operating on the grid will act as if the ogrid had been the result of mgrid.

r_ [index expression]

This is a simple way to build up arrays quickly. There are two use cases. 1) If the index expression contains comma separated arrays, then stack them along their first axis. 2) If the index expression contains slice notation or scalars then create a 1-d array with a range indicated by the slice notation. In otherwords the slice syntax start:stop:step is equivalent to arange(start, stop, step) inside of the brackets. However, if step is an imaginary number (i.e. 100j) then its integer portion is interpreted as a number-of-points desired and the start and stop are inclusive. In other words start:stop:step`j` is interpreted as linspace(start, stop, step, endpoint=1) inside of the brackets. After expansion of slice notation, all comma separated sequences are concatenated together.

Optional character strings placed as the first element of the index expression can be used to change the output. The strings 'r' or 'c' result in matrix output. If the result is 1-d and 'r' is specified a $1 \times N$ (row) matrix is produced. If the result is 1-d and 'c' is specified, then a $N \times 1$ (column) matrix is produced. If the result is 2-d then both provide the same matrix result.

```
>>> print r_[-1:1:9j,[0]*10,5,6]
[-1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75  1.    0.    0.
0.    0.    0.    0.    0.    0.    0.    0.    5.    6.  ]
>>> print r_['r',1,2,5,6]
[[1 2 5 6]]
>>> print r_['c',1,2,5,6]
[[1]
[2]
[5]
[6]]
```

A string integer specifies which axis to stack multiple comma separated arrays along.

```
>>> a=arange(6).reshape(2,3)
>>> r_[a,a]
array([[0, 1, 2],
[3, 4, 5],
[0, 1, 2],
[3, 4, 5]])
>>> r_['-1',a,a]
array([[0, 1, 2, 0, 1, 2],
[3, 4, 5, 3, 4, 5]])
```

A string of two comma-separated integers allows indication of the minimum number of dimensions to force each entry into as the second integer (the axis to concatenate along is still the first integer).

```
>>> r_['0,2',[1,2,3],[4,5,6]]
array([[1, 2, 3],
[4, 5, 6]])
>>> r_['1,2',[1,2,3],[4,5,6]]
array([[1, 2, 3, 4, 5, 6]])
```

A string with three comma-separated integers allows specification of the axis to concatenate along, the minimum number of dimensions to force the entries to, and which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the the 1's should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of '0' would place the 1's at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is '-1'.

```
>>> r_['0,2,0', [1,2,3], [4,5,6]]
array([[1],
[2],
[3],
[4],
[5],
[6]])
>>> r_['1,2,0', [1,2,3], [4,5,6]]
array([[1, 4],
[2, 5],
[3, 6]])
```

c_ [index_expression]

This is short-hand for r_['-1,2,0', index_expression] useful because of its common occurence. In particular, arrays will be stacked along their last axis after being upgraded to at least 2-d with 1's post-pended to the shape (column vectors made out of 1-d arrays).

5.6 Other indexing devices

index_exp [index expression]

Return a tuple of Python objects that implements the index expression and can be modified and placed in any other index expression.

```
>>> index_exp[2:5,...,4,::-1]
(slice(2, 5, None), Ellipsis, 4, slice(None, None, -1))
```

s_ [index expression]

Translate index expressions into the equivalent Python objects. This is similar to index_expression except a tuple is not always returned. For example:

```
>>> s_[1:10]
slice(1, 10, None)
>>> s_[1:10,-3:4:0.5]
(slice(1, 10, None), slice(-3, 4, 0.5))
```

This provides a standard way to construct index expressions to pass to functions and methods because Python does not allow slice expressions anywhere except for inside brackets.

ndindex (*seq)

A sequence of N integers are passed in as separate arguments. These integers are used as the upper boundaries of an N-dimensional counter that starts at 0. The object returned is an iterator that implements the counter.

```
>>> for index in ndindex(3,3,2):
...     print index,
(0, 0, 0) (0, 0, 1) (0, 1, 0) (0, 1, 1) (0, 2, 0) (0, 2, 1) (1, 0, 0)
```

unravel_index (indx, dims)

Convert a flat index, indx, into an index tuple for an array of the given shape. Keep in mind that it may be more convenient to use indx with a.flat, then to unravel the index.

5.7 Two-dimensional functions

These functions all deal with or return two dimensional arrays.

eye (N, M=None, k=0, dtype=float)

Return an $N \times M$ array of the given type with ones down the k^{th} diagonal. If M is None, it defaults to N. Alternatively, if M is a valid data type, then it becomes the data-type used.

vander (x, N=None)

The Vandermonde matrix of vector, x. The i^{th} column of the return matrix is the m_i^{th} power of x where $m_i = N - i - 1$. If N is None, it defaults to the length of x.

```
>>> vander([1,2,3,4,5],3)
array([[ 1,  1,  1],
[ 4,  2,  1],
[ 9,  3,  1],
[16,  4,  1],
[25,  5,  1]])
```

diag (v, k=0)

Return the k^{th} diagonal if v is a 2-d array, or returns an array with v as the k^{th} diagonal if v is a 1-d array.

```
>>> diag(arange(12).reshape(4,3),k=1)
array([1, 5])
>>> diag([1,4,5,7],k=-1)
array([[0, 0, 0, 0, 0],
[1, 0, 0, 0, 0],
[0, 4, 0, 0, 0],
[0, 0, 5, 0, 0],
[0, 0, 0, 7, 0]])
```

diagflat (v, k=0)

Return a 2-d array (of the same class as v) by placing a flattened version of v along the k^{th} diagonal. This differs from diag in that it only creates 2-d arrays and will work with any object that can be converted to an array (returning that object if it also defines an __array_wrap__ method).

fliplr (m)

Return the array, m, with rows preserved and columns reversed in the left-right direction. For m.ndim > 2, this works on the first two dimensions (equivalent to m[:,::-1])

flipud (m)

Return the array, m, with columns preserved and rows reversed in the up-down direction. For m.ndim > 1, this works on the first dimension (equivalent to m[::-1])

rot90 (m, k=1)

Rotate the first two dimensions of an array, m, by k*90 degrees in the counter-clockwise direction. Must have m.ndim >=2.

tri (N, M=N, k=0, dtype=aint)

Construct an $N \times M$ array where all the diagonals starting from the lower left corner up to the k$^{\text{th}}$ diagonal are all ones.

triu (m, k=0)

Return a upper-triangular 2-d array from m with all the elements below the kth diagonal set to 0.

tril (m, k=0)

Return a lower-triangular 2-d array from m with all the elements above the kth diagonal set to 0.

mat (data, dtype=None)

Construct a matrix from data. Alias for numpy.asmatrix. The calling syntax is the same as that function. Note that data can be a string in which case the routine uses spaces and semi-colons to construct the matrix:

```
>>> mat('1 3 4; 5 6 9')
matrix([[1, 3, 4],
[5, 6, 9]])
```

bmat (obj, ldict=None, gdict=None)

Build a matrix from sub-blocks. This is similar to mat, except the items in the nested-sequence, or string, should be appropriately shaped 2-d arrays. If obj is a string, then ldict and gdict can be used to alter where the names represented in the string are found (default is current local and global namespace).

```
>>> A=mat('1 2; 3 4'); B=mat('5 6; 7 8')
>>> bmat('A, B; B, A')
matrix([[1, 2, 5, 6],
[3, 4, 7, 8],
[5, 6, 1, 2],
[7, 8, 3, 4]])
```

5.8 More data type functions

issubclass_ (arg1, arg2)

Returns True if arg1 is a sub-class of arg2, otherwise returns False. Similar to the built-in issubclass except it does not raise an error if arg1 or arg2 are not types.

issubdtype (arg1, arg2)

Returns True if the type-object of the data-type represented by arg1 is a sub class of the type-object of the data-type represented by arg2.

iscomplexobj (obj)

Return a single True or False value depending on whether or not obj would be interpreted as an array with complex-valued data type.

isrealobj (obj)

Return a single True or False value depending on whether or not obj would be interpreted as an array with real-valued data type.

isscalar (obj)

True if obj is a scalar (an instance of an array data type, or a standard Python scalar type). There is also a sequence of called ScalarType defined in NumPy, so that this can also be tested as type(obj) in numpy.ScalarType.

nan_to_num (arr)

Returns an array with non-finite numbers changed to finite numbers. The mapping converts `nan` to 0, `inf` to the maximum value for the data type and `-inf` to the minimum value for the data type.

real_if_close (arr, tol=100)

Return a real arr if arr is complex with imaginary parts less than some tolerance. If tol > 1, then it represents a multiplicative factor on the value of epsilon for the data type of arr.

cast [dtype_or_alias] (obj)

Cast obj to an array of the given type. This is equivalent to array(obj, copy=0).astype(dtype_or_alias). When one type is cast to another in this fashion, a very low-level operation takes place. Typically, you get what your C-compiler produces for the cast, but notice that in the case of casting to a bool type, the value becomes either a 0 or a 1.

```
>>> cast[bool]([1,2,0,4,0]).astype(int)
array([1, 1, 0, 1, 0])
```

asfarray (a, dtype=float)

Return an array of inexact data type (floating or complexfloating).

mintypecode (typechars, typeset='GDFgdf', default='d')

Return a minimum data type character from typeset that handles all given typechars. The returned type character must correspond to the data type of the smallest size such that an array of the returned type can handle the data from an array of type t for each t in typechars. If the typechars does not intersect with the typeset, then default is returned. If an element of typechars is not a string, then t=asarray(t).dtypechar is applied.

finfo (dtype)

This class allows exploration of the details of how a floating point number is represented in the computer. It can be instantiated by an inexact data type object (or an alias for one). Complex-valued data types are acceptable and are equivalent to their real-valued counterparts. The attributes of the class are

nmant The number of bits in the floating point mantissa, or fraction.

nexp The number of bits in the floating point exponent

machep Exponent of the smallest (most negative) power of 2 that when added to 1.0 gives something different than 1.0.

eps Floating point precision: 2**machep.

precision Number of decimal digits of precision: int(-log10(eps))

resolution 10**(-precision)

negep Exponent of the smallest power of 2 that, subtracted from 1.0, gives something different than 1.0.

epsneg Floating point precision: 2**negep.

minexp Smallest (most negative) power of 2 producing "normal" numbers (no leading zeros in the mantissa).

tiny The smallest (in magnitude) usable floating point number equal to 2**minexp.

maxexp Smallest (positive) power of 2 that causes overflow.

max The largest usable floating value: (1-epsneg)* (2**maxep)

min The most negative usable floating value: -max

The most useful attributes are probably eps, max, min, and tiny.

5.9 Functions that behave like ufuncs

These functions are Python functions built on top of universal functions (ufuncs) and also take optional output arguments. They broadcast like ufuncs but do not have ufunc attributes.

fix (x, y=None)

Round x to the nearest integer towards zero.

isneginf (x, y=None)

True if $x = -\infty$. Should be the same as `x==NumPy.NINF`.

isposinf (x, y=None)

True if $x = +\infty$. Should be the same as `x==NumPy.PINF`.

log2 (x, y=None)

Compute the logarithm to the base 2 of x. An optional output array may be provided.

5.10 Miscellaneous Functions

Some miscellaneous functions are available in NumPy which are included largely for compatibility with MLab of the old Numeric package. One notable difference, however, is that due to a separate implementation of the modified Bessel function, the kaiser window is available without needing a separate library.

sinc (x)

Compute the sinc function for x which can be a scalar or array. The sinc is defined as $y = \operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$ with the caveat that the limiting value (1.0) of the ratio is taken for $x = 0$.

i0 (x)

Modified Bessel function of the first kind of order 0. Needed to compute the kaiser window. The modified Bessel function is defined as

$$I_0(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\cos\theta} d\theta = \sum_{k=0}^{\infty} \frac{x^{2k}}{4^k (k!)^2}.$$

blackman (M)

Construct an M-point Blackman smoothing window which is sequence of length M with values given for $n = 0 \ldots M-1$ by

$$w[n] = 0.42 - 0.5\cos\left(2\pi\frac{n}{M-1}\right) + 0.08\cos\left(4\pi\frac{n}{M-1}\right).$$

bartlett (M)

Construct an M-point Bartlett (triangular) smoothing window as

$$w[n] = \begin{cases} 2\frac{n}{M-1} & 0 \le n \le \frac{M-1}{2}, \\ 2 - 2\frac{n}{M-1} & \frac{M-1}{2} < n \le M-1. \end{cases}$$

hanning (M)

Construct an M-point Hanning smoothing window defined as

$$w[n] = \frac{1}{2} - \frac{1}{2}\cos\left(2\pi\frac{n}{M-1}\right).$$

hamming (M)

Construct an M-point Hamming smoothing window defined for $n = 0 \ldots M-1$ as

$$w[n] = 0.54 - 0.46\cos\left(2\pi\frac{n}{M-1}\right).$$

All of the windowing functions are smoothing windows that attempt to balance the inherent trade off between side-lobe height (ringing) and main-lobe width (resolution) in the frequency domain. A rectangular window has the smallest main-lobe width but the largest side-lobe height. A windowing (tapering) function tries to can help trade off main-lobe width By sacrificing a little in resolution using a windowing function These windows can be used to smooth

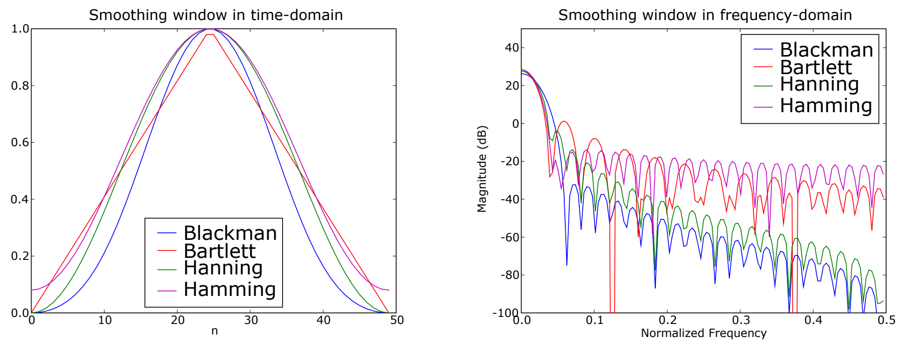

Figure 5.1: Blackman, Bartlett, Hanning, and Hamming windows in the time and frequency domain showing the trade-off between main-lobe width and side-lobe height (Figures made with matplotlib).

data using the convolve function. Figure 5.1 shows the windowing functions described so far and their time- and frequency-domain behavior.

The trade-off between main-lobe and side-lobe has been studied extensively. Solutions that maximize energy in the main-lobe compared to energy in the side-lobes can be found by finding an eigenvector which can be expensive to compute for large window sizes. A good approximation to these prolate-spheroidal windows is the Kaiser window.

kaiser (M, β)

Construct an M-point Kaiser smoothing window. The β parameter controls the width of the window (and the frequency-domain side-lobe height and main-lobe width). The window is defined as

$$w[n] = \frac{1}{I_0(\beta)} I_0\left(\beta\sqrt{1 - \frac{(2n - M - 1)^2}{(M-1)^2}}\right).$$

There is an empirical relationship between β and the side-lobe height which can be used in FIR filter design. To achieve a side-lobe height of $-\alpha$dB, the β parameter is

$$\beta = \begin{cases} 0.1002\,(\alpha - 8.7) & \alpha > 50, \\ 0.5842\,(\alpha - 21)^{0.4} + 0.07886\,(\alpha - 21) & 21 \leq \alpha \leq 50, \\ 0 & \alpha < 21. \end{cases}$$

The length M of the window determines the transition width. To obtain a transition width of $\Delta\omega$rad/s the window-length must be at least:

$$M = \frac{\alpha - 8}{2.285\Delta\omega} + 1.$$

5.11 Utility functions

set_numeric_ops (<op1>=func1, <op2>=func2, ...)

This function can be used to alter the operations used for internal array calculations and array special methods. Replaceable operations (and possible entries for <opN>) are add, subtract, multiply, divide, remainder, power, sqrt, negative, absolute, invert, left_shift, right_shift, bitwise_and, bitwise_or, less, less_equal, equal, not_equal, greater, greater_equal, floor_divide, true_divide, logical_or, logical_and, floor, ceil, maximum, and minimum. The example code below changes, then restores, the old Numeric behavior of remainder (which was changed because it was not consistent with Python).

```
>>> a = array([-3.,-2,-1,0,1,2,3])
>>> print a % -2.1
[-0.9 -2.  -1.   0.  -1.1 -0.1 -1.2]
>>> oldops = set_numeric_ops(remainder=fmod)
>>> print a % -2.1
[-0.9 -2.  -1.   0.   1.   2.   0.9]
>>> newops = set_numeric_ops(**oldops)
>>> print a % -2.1
[-0.9 -2.  -1.   0.  -1.1 -0.1 -1.2]
>>> print 3 % -2.1 # comparison
-1.2
```

get_include ()

Return the directory that contains the numpy include files. The numpy.distutils automatically includes this directory in building extensions.

get_numarray_include (type=None)

Return the directory that contains the numarray compatible C-API include files. If type is not None, then return a list containing both the numarray compatible

C-API include files and the numpy include files. The latter form is only needed when building an extension without the use of numpy.distutils.

deprecate (func, oldname, newname)

Return a deprecated function named ’oldname’ that has been replaced by ’newname’. This new deprecated function issues a warning before calling the old function. The name and docs of the function are also updated to be oldname instead of the name that func has. Example usage. If you want to deprecate the function named ’old’ in favor of a new function named ’new’ which has the same calling conention then this could be done with the assignment

```
old = deprecate(new, ’old’, ’new’)
```

Chapter 6

Scalar objects

One important new feature of NumPy is the addition of a new scalar object for each of the 21 different data types that an array can have. Do not confuse these scalar objects with the data-type objects. There is one data-type object. It contains a `.type` attribute which points to the Python type that each element of the array will be returned as[1]. The built-in data-types point have `.type` attributes that point to these scalar objects. Five (or six) of these new scalar objects are essentially equivalent to fundamental Python types and therefore inherit from them as well as from the generic array scalar type. The bool_ data type is very similar to the Python BooleanType but does not inherit from it because Python's BooleanType does not allow itself to be inherited from, and on the C-level the size of the actual bool_ data is not the same as a Python Boolean scalar. Table 6.1 shows which array scalars inherit from basic Python types.

The array scalars have the same attributes and methods as arrays and live in a hierarchy of scalar types so they can be easily classified based on their type objects. However, because array scalars are immutable, and attributes change intrinsic properties of the object, the **array scalar attributes are not settable**.

Array scalars can be detected using the hierarchy of data types. For example, `isinstance(val, generic)` will return True if val is an array scalar object. Alternatively, what kind of array scalar is present can be determined using other members of the data type hierarchy. Thus, for example `isinstance(val, complexfloating)` will return True if val is a complex valued type, while `isinstance(val, flexible)` will return true if val is one of the flexible item-

[1]with the exception of object data-types which return the underlying object and not a "scalar" type.

Table 6.1: Array scalar types that inherit from basic Python types. The intc array data type might also inherit from the IntType if it has the same number of bits as the int_ array data type on your platform.

array data type	Python type
int_	IntType
float_	FloatType
complex_	ComplexType
str_	StringType
unicode_	UnicodeType

size array types (string, unicode, void).

WARNING

The bool_ type is not a subclass of the int_ type (the bool_ type is not even a number type). This is different than Python's default implementation of bool as a sub-class of int.

6.1 Attributes of array scalars

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

flags

Returns True for CONTIGUOUS, OWNDATA, FORTRAN, and ALIGNED. Always returns False for WRITEABLE, and UPDATEIFCOPY.

shape

Returns ().

strides

Returns ().

ndim

Returns 0.

data

A read-only buffer object of size self.itemsize,

size

Return 1.

itemsize

The number of bytes this scalar requires.

base

Returns None.

dtype

Returns data type descriptor corresponding to this array scalar.

real

The real part of the scalar.

imag

The imaginary part of the scalar (or 0 if this is real).

flat

Return a 1-d iterator object (of size 1).

T

Return a reference to self.

__array_interface__

The Python-side to the array interface.

__array_struct__

The C-side to the array interface

__array_priority__

-100.0 (very low-priority).

__array_wrap__ (obj)

Returns an array scalar from an array

6.2 Methods of array scalars

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

__new__ (obj)

The default behavior is to return a new array or array scalar by calling array(obj) with the corresponding data type. There are two situations when this default behavior is delayed until another approach is tried. First, when the array scalar type inherits from a Python type, then the Python types new method is called first and the default method is called only if that approach fails. The second situation is for the `void` data type where a single integer-like argument will cause a void scalar of that size to be created and initialized to 0.

Notice that because array(obj) is called for new, if obj is a nested sequence, then the return object could actually be an `ndarray`. Thus, arrays of the correct type can also be created by calling the array data type name directly:

```
>>> uint32([[5,6,7,8],[1,2,3,4]])
array([[5, 6, 7, 8],
[1, 2, 3, 4]], dtype=uint32)
```

__array__ (<None>)

Returns a 0-dimensional array of the given data type, or of type(self) if argument is None.

__array_wrap__ (array)

Returns a scalar array object from the first-element of the array.

__squeeze__ ()

Returns self.

byteswap (<False>)

Trying to set the first (inplace) argument to True raises a ValueError. Otherwise, this returns a new array scalar with the data byteswapped.

__reduce__ ()

This is called to pickle an array scalar. It returns a tuple of (numpy.core.multiarray.scalar, self.dtypestr, obj or self.tostring()) which can be used to reconstruct the scalar on unpickling. Notice that no state is written, because the entire scalar can be constructed from just the string. Also, if this is an object array scalar, then the Python object being referenced is written.

__setstate__ ()

Does nothing but return None.

setflags ()

Does nothing, as flags cannot be set for scalars.

6.3 Defining New Types

There are two ways to effectively define a new type of array. One way is to simply subclass the ndarray and overwrite the methods of interest. This will work to a degree, but internally certain behaviors are fixed by the data type of the array. To fully customize the data type of an array you need to define a new data-type for the array, and register it with NumPy. This new type can only be defined in C. How to define a new data type in C will be discussed in the next part of the book.

Chapter 7

Data-type (`dtype`) Objects

It is important not to confuse the "array-scalars" with the "data-type objects." It is true that an array-scalar can be interpreted as a data-type object and so can be used to refer to the data-type of an array. However, the data-type object is a separate Python object. Every ndarray has an associated data-type object that completely defines the data in the array (including any named fields). For every built-in data-type object there is an associated type object whose instances are the array-scalars. Because of the association between each data-type object and a type-object of the corresponding array scalar, the array-scalar type-objects can also be thought of as data-types. However, for the type objects of flexible array-scalars (string, unicode_, and void), the type-objects alone are not enough to specify the full data-type because the length is not given. The data-type constructor, **numpy.dtype**, converts any object that can be considered as a data-type into a data-type object which is the actual object an ndarray looks to in order to interpret each element of its data region. Whenever a data-type is required in a NumPy function or method, supplying a dtype object is always fastest. If the object supplied is not a dtype object, then it will be converted to one using dtype(obj). Therefore, understanding data-type objects is the key to understanding how data types are really represented and understood in NumPy.

7.1 Attributes

type The type object used to instantiate a scalar of this data-type.

kind A character code (one of 'biufcSUV') identifying the general kind of data.

char A unique character code for each of the 21 different built-in types.

num A unique number for each of the 21 different built-in types roughly ordered from least-to-most precision.

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

name A bit-width name for this data-type (un-sized flexible data-type objects are missing the width).

byteorder A character indicating the byte-order of this data-type object ('=' : native, '<' : little-endian, '>' : big-endian, '|' : not applicable). All built-in data-type objects have byteorder either '=' or '|'.

itemsize The element size of this data-type object. For 18 of the 21 types this number is fixed by the data-type. For the flexible data-types, this number can be anything.

alignment The required alignment (in bytes) of this data-type according to the compiler. More information is available in the C-API section.

fields A dictionary showing any named fields that have been defined for this data-type (or None if there are no named fields). Fields can be assigned to any built-in data-type (*e.g.* using the tuple input to the dtype constructor). However, fields are most useful for (subtypes of) void data-types which can be any size. Fields are a convenient way to keep track of fixed-size sub-parts of the total fixed-size array-element, or record. A field is defined in terms of another dtype object and an offset (in bytes) into the current record.

The fields dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field: (dtype, offset[, title]). If present, the optional title can actually be any object (if it is string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also, that the first two elements of the tuple can be passed directly as arguments to the getfield and setfield attributes of an ndarray. If field names are not specified in a constructor, they default to 'f0', 'f2', ..., 'f<n-1>'.

names An ordered list of field names. This can be used to walk through all of the named fields in offset order. Notice that the defined fields do not have to "cover" the record, but the itemsize of the container data-type object must always be at least as large as the itemsizes of the data-type objects in the defined fields. This attribute is None if there are no fields.

subdtype Numarray introduced the concept of a fixed-length record having fields that were themselves arrays of another data-type. This is supported at a fundamental level in NumPy using this attribute which maintains the simplicity of defining a field by another data-type object. It either returns None or a tuple (base dtype, shape) where shape is a tuple showing the size of the C-contiguous array and the base dtype object indicates the data-type in each element of the subarray. If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by the shape are tacked on to the end of the retrieved array.

descr An array-interface-compliant full description of the data-type. The format is that required by the 'descr' key in the __array_interface__.

isbuiltin A 1 if self is one of the built-in dtype objects; a 2 if self is a user-defined dtype object; a 0, otherwise.

isnative True if this data-type object has a byteorder that is native to the platform; otherwise False.

hasobject True if self contains reference-counted objects in any of it's fields or sub data-types. Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required and this attribute is useful for distinguishing data-types that may contain arbitrary Python objects and data-types that won't.

flags Bit-flags for the data-type describing how the data-type will be interpreted. Bit-masks are in numpy.core.multiarray as the constants ITEM_HASOBJECT, LIST_PICKLE, ITEM_IS_POINTER, NEEDS_INIT, NEEDS_PYAPI, USE_GETITEM, USE_SETITEM. A full explanation of these flags is in the second part of this book. These flags are largely useful for user-defined data-types.

7.2 Construction

dtype (obj, align=0, copy=0)

Return a new data-type object from obj. The keyword argument, align, can only be nonzero if obj is a dictionary, or a comma-separated string. If it is non-zero in those cases it is used to add padding as needed to the fields to match what the compiler that compiled NumPy would do to a similar C-struct. The copy argument guarantees a new copy of the data-type object, otherwise, the result may just be a reference to a built-in data-type object.

Objects that can be converted to a data-type object are described in the following list. Because every object in this list can be converted to a data-type object it can also be used whenever a `dtype` is requested by a function or method in NumPy.

dtype Returns itself.

None Returns the default data-type descriptor object: float.

type-object Many Python type objects can be converted to data-type objects.

1. Array-scalar types: The type-objects of the 21 built-in array scalars all convert to an associated data-type object. This is true for subclasses as well. Not all data-type information can be supplied with a type-object. Flexible data-types with default itemsizes of 0, for example, require an itemsize to be useful.

   Examples: int32, float64, uint16, complex128

2. Generic types: The generic hierarchical type objects convert to corresponding dtype objects according to the associations: (numeric, inexact, floating) –> float; complexfloating –> cfloat; (integer, signedinteger) –> int_; unsignedinteger –> uint; character –> string; (generic, flexible) –> void.

3. Builtin types: Several python types are equivalent to a corresponding array scalar when used to generate a dtype object: int –> int_; bool –> bool_; float –> float_; complex –> cfloat; str –> string; unicode –> unicode_; buffer –> void; (all others) –> object_.

   Examples: object, str, float, int

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

string Several kinds of strings can be converted. Recognized strings can be pre-pended with '>', or '<', to specify the byteorder.

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

   Examples: 'b', 'H', 'f', 'd', 'F', 'D', Float64, Int32, UInt16

2. Array-protocol type strings: The first character specifies the kind of data and the remaining characters specify how many bytes of data. The supported kinds are 'b' –> Boolean, 'i' –> (signed) integer, 'u' –> unsigned integer, 'f' –> floating-point, 'c' –> complex-floating point, 'S', 'a' –> string, 'U' –> unicode, 'V' –> anything (void).

   Examples: 'i4', 'f8', 'c16', 'b1', 'S10', 'a25'

3. Comma-separated field formats: numarray introduced a short-hand notation for specifying the format of a record as a comma-separated string of basic formats. A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it is greater than 1-d. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. This data-type defines fields named 'f0', 'f2', ..., 'f<N-1>' where N (>1) is the number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field contains a subdtype attribute providing the shape.

   Examples: "i4, (2,3)f8, f4"; "a3, 3u8, (3,4)a10"

4. Any string in NumPy.sctypeDict.keys():

   Examples: 'uint32', 'Int16', 'Uint64', 'Float64', 'Complex64'

tuple Three kinds of tuples each of length 2 can be converted into a data-type object:

1. (flexible dtype, itemsize): The first argument must be an object that is converted to a flexible data-type object (one whose element size is 0), the second argument is an integer providing the desired itemsize.

   Examples: (void, 10); (str, 35), ('U', 10)

2. (fixed dtype, shape): The first argument is any object that can be converted into a fixed-size data-type object. The second argument is the desired shape of this type. If the shape parameter is 1, then the data-type object is equivalent to fixed dtype.

   Examples: (int32, (2,5)); ('S10', 1)=='S10'; ('i4, (2,3)f8, f4', (2,3))

3. (base dtype, new dtype): Both arguments must be convertible to data-type objects in this case. The base dtype is the data-type object that the new data-type builds on. This is how you could assign named fields to any built-in data-type object.

   Examples: (int32, {'real':(int16,0), 'imag':(int16,2)}); (int32, (int8, 4));
   ('i4', [('r','u1'),('g','u1'),('b','u1'),('a','u1')])

list (array description interface): This style is more fully described at this site http://numpy.scipy.org/array_interface.html. It consists of a list of fields where each field is described by a tuple of length 2 or 3. The first element of the tuple is the field name (if this is '' then a standard field name, 'f#', is assigned). The field name may also be a 2-tuple of strings where the first string is either a "title" (which may be any string or unicode string) or meta-data for the field which can be any object, and the second string is the "name" which must be a valid Python identifier. The second element of the tuple can be anything that can be interpreted as a data-type. The optional third element of the tuple contains the shape if this field represents an array of the data-type in the second element. This style does not accept align=1 as it is assumed that all of the memory is accounted for by the array interface description. See the web-page for more examples. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.

Examples: [('big','>i4'), ('little','<i4')]; [('R','u1'), ('G','u1'), ('B','u1'), ('A','u1')]

dictionary There are two dictionary styles. The first is a standard dictionary format while the second accepted format allows the fields attribute of dtype objects to be interpreted as a data-type.

1. names and formats: This style has two required and two optional keys. The 'names' and 'formats' keys are required. Their respective values are equal-length lists with the field names and the field formats. The field names must be strings and the field formats can be any object accepted by dtypedescr constructor. The optional keys in the dictionary are 'offsets' and 'titles' and their values must each be lists of the same length as the 'names' and 'formats' lists. The 'offsets' value is a list of integer offsets for each field, while the 'titles' value is a list of titles for each field (None can be used if no title is

desired for that field). The titles can be any string or unicode object and will add another entry to the fields dictionary keyed by the title and referencing the same field tuple which will contain the title as an additional tuple member.

Examples: {'names': ['r','g','b','a'], 'formats': [uint8, uint8, uint8, uint8]}; {'names':['r','b'], 'formats': ['u1', 'u1'], 'offsets': [0, 2], 'titles': ['Red pixel', 'Blue pixel']}

2. data-type object fields: This style is patterned after the format of the fields dictionary in a data-type object. It contains string or unicode keys that refer to (data-type, offset) or (data-type, offset, title) tuples.

   Examples: {'col1': ('S10', 0), 'col2': (float32, 10), 'col3': (int, 14)}

7.3 Methods

newbyteorder (<'swap'>)

Construct a new copy of self with its byteorder changed according to the optional argument. All changes are also propagated to the data-type objects of all fields and sub-arrays. If a byteorder of '|' (meaning ignore) is encountered it is left unchanged. The default behavior is to swap the byteorder. Other possible arguments are 'big' ('>'), 'little' ('<'), and 'native' ('=') which recursively forces the byteorder of self (and it's field data-type objects and any sub-arrays) to the corresponding byteorder.

__reduce__ ()

__setstate__ (state)

Data-type objects can be pickled because of these two methods. The __reduce__() method returns a 3-tuple consisting of (callable object, args, state), where the callable object is numpy.core.multiarray.dtype and args is (typestring, 0, 1) unless the data-type inherits from void (or is user-defined) in which case args is (typeobj, 0, 1). The state is an 8-tuple with (version, endian, self.subdtype, self.names, self.fields, self.itemsize, self.alignment, self.flags). The self.itemsize and self.alignment entries are both -1 if the data-type object is built-in and not flexible (because they are fixed on creation). The setstate method takes the saved state and updates the date-type object.

Chapter 8

Standard Classes

The ndarray in NumPy is a “new-style” Python built-in-type. Therefore, it can be inherited from (in Python or in C) if desired. Therefore, it can form a foundation for many useful classes. Often whether to sub-class the array object or to simply use the core array component as an internal part of a new class is a difficult decision, and can be simply a matter of choice. NumPy has several tools for simplifying how your new object interacts with other array objects, and so the choice may not be significant in the end. One way to simplify the question is by asking yourself if the object you are interested can be replaced as a single array or does it really require two or more arrays at it’s core. For example, in the standard NumPy distribution, the matrix and records classes inherit from the ndarray, while masked arrays use two ndarrays as objects of its internal structure.

Note that asarray(a) always returns the base-class ndarray. If you are confident that your use of the array object can handle any subclass of an ndarray, then asanyarray(a) can be used to allow subclasses to propagate more cleanly through your subroutine. In principal a subclass could redefine any aspect of the array and therefore, under strict guidelines, asanyarray(a) would rarely be useful. However, most subclasses of the arrayobject will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One of important example, however, of why your subroutine may not be able to handle an arbitrary subclass of an array is that matrices redefine the ’*’ operator to be matrix-multiplication, rather than element-by-element multiplication.

8.1 Special attributes and methods recognized by NumPy

__array_finalize__ (obj)

This method is called whenever the system internally allocates a new array from obj, where obj is a subclass (subtype) of the (big)ndarray. It can be used to change attributes of self after construction (so as to ensure a 2-d matrix for example), or to update meta-information from the "parent." Subclasses inherit a default implementation of this method that does nothing.

__array_wrap__ (array)

This method should return an instance of the class from the ndarray object passed in. For example, this is called after every ufunc for the object with the highest __array_priority__. The ufunc-computed array object is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method.

__array__ (dtype <None>)

This method is called to obtain an ndarray object when needed. You should always guarantee this returns an actual ndarray object. Subclasses inherit a default implementation of this method.

__array_priority__

The value of this attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. Subclasses inherit a default value of 1.0 for this attribute.

8.2 Matrix Objects

Matrix objects inherit from the ndarray and therefore, they have the same attributes and methods of ndarrays. There are six important differences of matrix objects, however that may lead to unexpected results when you use matrices but expect them to act like arrays:

1. Matrix objects can be created using a string notation to allow Matlab-style syntax where spaces separate columns and semicolons (';') separate rows.

2. Matrix objects are always two-dimensional. This has far-reaching implications, in that m.ravel() is still two-dimensional (with a 1 in the first dimension) and item selection returns two-dimensional objects so that sequence behavior is fundamentally different than arrays.

3. Matrix objects over-ride multiplication to be matrix-multiplication. **Make sure you understand this for functions that you may want to receive matrices. Especially in light of the fact that asanyarray(m) returns a matrix when m is a matrix.**

4. Matrix objects over-ride power to be matrix raised to a power. The same warning about using power inside a function that uses asanyarray(...) to get an array object holds for this fact.

5. The default __array_priority__ of matrix objects is 10.0, and therefore mixed operations with ndarrays always produce matrices.

6. Matrices have special attributes which make calculations easier. These are

   (a) .T — return the transpose of self

   (b) .H — return the conjugate transpose of self

   (c) .I — return the inverse of self

   (d) .A — return a view of the data of self as a 2d array (no copy is done).

WARNING

Matrix objects over-ride multiplication, '*', and power, '**', to be matrix-multiplication and matrix power, respectively. If your subroutine can accept sub-classes and you do not convert to base-class arrays, then you must use the ufuncs multiply and power to be sure that you are performing the correct operation for all inputs.

The matrix class is a Python subclass of the ndarray and can be used as a reference for how to construct your own subclass of the ndarray. Matrices can be created from other matrices, strings, and anything else that can be converted to an `ndarray`. The name "mat" is an alias for "matrix" in NumPy.

Example 1: Matrix creation from a string

```
>>> a=mat('1 2 3; 4 5 3')
>>> print (a*a.T).I
[[ 0.2924 -0.1345]
[-0.1345  0.0819]]
```

Example 2: Matrix creation from nested sequence

```
>>> mat([[1,5,10],[1.0,3,4j]])
matrix([[  1.+0.j,   5.+0.j,  10.+0.j],
[  1.+0.j,   3.+0.j,   0.+4.j]])
```

Example 3: Matrix creation from an array

```
>>> mat(random.rand(3,3)).T
matrix([[ 0.7699,  0.7922,  0.3294],
[ 0.2792,  0.0101,  0.9219],
[ 0.3398,  0.7571,  0.8197]])
```

matrix (data, dtype=None, copy=True)

The sequence to convert to a matrix is passed in as data. If dtype is None, then the data-type is determined from the data. If copy is True, then a copy of the data is made, otherwise, the same data buffer is used. If no buffer can be found for data, then a copy is also made. Note: The matrix object is actually a class and so using this syntax calls matrix.__new__(matrix, data, dtype, copy) which is what happens whenever you "call" any class object as a function.

mat

Just another name for matrix.

asmatrix (data, dtype=None)

Returns the data without copying. Equivalent to matrix(data, dtype, copy=False).

bmat (obj, ldict=None, gdict=None)

Build a matrix object from a string, nested sequence or an array. This command lets you build up matrices from other other objects. The ldict and gdict parameters are local and module (global) dictionaries that are only used when obj is a string. If they are not provided, then the local and module dictionaries present when bmat is called are used.

```
>>> A = mat('2 2; 2 2'); B=mat('1 1; 1 1');
>>> print bmat('A B; B A')
[[2 2 1 1]
[2 2 1 1]
[1 1 2 2]
[1 1 2 2]]
```

8.3 Memory-mapped-file arrays

Memory-mapped files are useful for reading and/or modifying small segments of a large file with regular layout, without reading the entire file into memory. A simple subclass of the ndarray uses a memory-mapped file for the data buffer of the array. For small files, the over-head of reading the entire file into memory is typically not significant, however for large files using memory mapping can save considerable resources.

NOTE

Memory-mapped arrays use the the Python memory-map object which (prior to Python 2.5) does not allow files to be larger than a certain size depending on the platform. This size is always $<$ 2GB even on 64-bit systems.

The class is called memmap and is available in the NumPy namespace. The __new__ method of the class has been re-written to have the following syntax:

__new__ (cls, filename, dtype=uint8, mode='r+', offset=0, shape=None, order=0)

filename The file name to be used as the array data buffer

dtype A data-type object used to interpret the file contents (including byte-order).

mode The mode to open the file in. Valid modes are 'readonly' or 'r', 'copyonwrite' or 'c', 'readwrite' or 'r+', and 'write' or 'w+'. This mode determines the WRITEABLE flag of the returned array.

offset An offset into the file to start the array data.

shape The desired shape of the array. If this is None, then the returned array will be 1-d with the number of elements determined by the file size and data type.

order Either 'C' or 'Fortran' to indicate the order that an N-D array should be interpreted. This only has an effect if the shape is greater than 2-D.

Memory-mapped-file arrays have one additional method (besides those they inherit from the ndarray): self.**sync**() which must be called manually by the user to ensure that any changes to the array actually get written to disk.

Example:

```
>>> a = memmap('newfile.dat', dtype=float, mode='w+', shape=1000)
>>> a[10] = 10.0
>>> a[30] = 30.0
>>> del a
>>> b = fromfile('newfile.dat', dtype=float)
>>> print b[10], b[30]
10.0 30.0
>>> a = memmap('newfile.dat', dtype=float)
>>> print a[10], a[30]
10.0 30.0
```

8.4 Character arrays (numpy.char)

These are enhanced arrays of either string type or unicode_ type. These arrays inherit from the ndarray, but specially-define the operations +, *, and % on a (broadcasting) element-by-element basis. These operations are not available on the standard ndarray of character type. In addition, the chararray has all of the standard string (and unicode) methods, executing them on an element-by-element basis. Perhaps the easiest way to create a chararray is to use self.view(chararray) where self is an ndarray of string or unicode data-type. However, a chararray can also be created using the numpy.chararray.__new__ method.

__new__ (shape, itemsize, unicode=False, buffer=None, offset=0, strides=None, order=None)

Create a new character array of string or unicode type and itemsize characters. Create the array using buffer (with offset and strides) if it is not None. If buffer is None, then construct a new array with strides in Fortran order if len(shape) >=2 and order is 'Fortran' (otherwise the strides will be in 'C' order).

char.array (obj, itemsize=None, copy=True, unicode=False, order=None)

Create a chararray from the nested sequence obj. If obj is an ndarray of data-type unicode_ or string, then its data is wrapped by the chararray object and converted to the desired type (string or unicode).

Another difference with the standard ndarray of string data-type is that the chararray inherits the feature introduced by Numarray that white-space at the end of any element in the array will be ignored on item retrieval and comparison operations.

8.5 Record Arrays (numpy.rec)

NumPy provides a powerful data-type object that allows any ndarray to hold (arbitrarily nested) record-like items with named-field access to the sub-types. This is possible without any special record-array sub-class. Consider the example where each item in the array is a simple record of name, age, and weight. You could specify a data-type for an array of such records using the following data-type object:

```
>>> desc = dtype({'names': ['name', 'age', 'weight'], 'formats': ['S30
>>> a = array([('Bill',31,260.0),('Fred', 15, 145.0)],dtype=desc)
>>> print a[0]
('Bill', 31, 260.0)
>>> print a['name']
['Bill' 'Fred']
>>> print a['age']
[31 15]
>>> print a['weight']
[ 260.  145.]
>>> print a[0]['name'], a[0]['age'], a[0]['weight']
Bill 31 260.0
>>> print len(a[0])
3
```

This example shows how a general array can be assigned named fields and how these fields can be accessed. In this case the a[0] object is an array-scalar of type void. The void array-scalars are unique in that they contain references to (rather than copies of) the underlying data whenever fields are defined. Therefore, the record data can be modified in place:

```
>>> a[0]['name'] = 'George'; print a
[('George', 31, 260.0) ('Fred', 15, 145.0)]
```

The recarray subclass and its accompanying record item add the ability to access named fields through attribute lookup. A quick way to get a record array is to use the view method of the ndarray.

```
>>> r = a.view(recarray)
>>> print r.name
['George' 'Fred']
```

The numpy.core.records module (aliased to nump.rec when numpy is imported) contains additional convenience functions for constructing record arrays. All of the following constructors have two different mechanisms for specifying the data-type. Either the dtype= argument can be specified or the argument formats= can be specified along with an optional set of four additional keyword arguments (names=, titles=, aligned= and byteorder=). In some cases neither dtype= nor formats= is required as the data-type can be inferred from the object passed in as the first argument.

The five argument method for specifying a data-type constructs a data-type object internally. The comma-separated formats string is used to specify the fields. The names (and optional titles) of the fields can be specified by a comma-separated string of names (or titles). The aligned flag determines whether the fields are packed (False) or padded (True) according to the platform compiler rules. The byteorder argument allows specification of the byte-order for all of the fields at once (they can also be specified individually in the formats string). The default byte-order is native to the platform.

array (obj, dtype=None, shape=None, offset=0, strides=None, formats=None, names=None, titles=None, aligned=False, byteorder=None, copy=True)

A general-purpose record array constructor that is a front-end to the other constructors If obj is None, then call the **recarray** constructor. If obj is a string, then call the **fromstring** constructor. If obj is a list or a tuple then if the first object is an ndarray, then call **fromarrays**, otherwise call **fromrecords**. If obj is a recarray, then make a copy of the data in recarray (if copy is True) and use the new formats, names, and titles. If obj is a file then call **fromfile**. Finally, if obj is an ndarray, then return obj.view(recarray) and make a copy of the data if copy is True. Otherwise, call the __array_interface__ attribute and try to convert using the information returned from that object. Either dtype or the formats argument must be given if obj is None, a string, or a file, and if obj is None so the recarray constructor will be called, then shape must be given as well.

fromarrays (array_list, dtype=None, shape=None, formats=None, names=None, titles=None, aligned=False, byteorder=None)

Create a record array from a (flat) list of ndarrays. The data from the arrays will be copied into the fields. If formats is None and dtype is None, then the formats will be determined from the arrays. The names and titles arguments can be a list, tuple or a (comma-separated) string specifying the names and/or titles to use for the fields. If aligned is True, then the structure will be padded according to the rules of the compiler that NumPy was compiled with.

```
>>> x1 = array([21,32,14])
>>> x2 = array(['my','first','name'])
>>> x3 = array([3.1, 4.5, 6.2])
>>> r = rec.fromarrays([x1,x2,x3], names='id, word, number')
>>> print r[1]
(32, 'first', 4.5)
>>> r.number
array([ 3.1,  4.5,  6.2])
>>> r.word
chararray(['my', 'first', 'name'],
dtype='|S5')
```

fromrecords (rec_list, dtype=None, shape=None, formats=None, names=None, titles=None, aligned=False, byteorder=None)

Construct a record array from a (nested) sequence of tuples that define the records. If formats are not given, they are deduced from the records, but this is slower. The field names and field titles can be specified. If aligned is non-zero, then the record array is padded so that fields are aligned as the platform compiler would do if the fields represented a C-struct.

```
>>> recs = [('Bill', 31, 260.0), ('Fred', 15, 145.0)]
>>> r = rec.fromrecords(recs, formats='S30,i2,f4', names='name, age, w
>>> print r.name
['Bill' 'Fred']
>>> print r.age
[31 15]
>>> print r.weight
[ 260.  145.]
```

fromstring (datastring, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=0, byteorder=None):

Construct a record array using the provided datastring (at the given offset) as the memory. The record array will be read-only. The byteorder argument may be used to specify the byteorder of all of the fields at the same time. A True aligned argument causes padding fields to be added as needed so that the fields are aligned on boundaries determined by the compiler. The shape of the returned array can also be specified.

fromfile (fd, dtype=None, shape=None, offset=0, formats=None, names=None, titles=None, aligned=False, byteorder=None)

Construct a record array from the (binary) data in the given file object, fd. This object may be an open file or a string to indicate a file to read from. If offset is non-zero, then data is read from the file at offset bytes from the current position.

The following classes are also available in the numpy.core (and therefore the numpy) namespace

record A subclass of the void array scalar type that allows field access using attributes.

recarray A subclass of the ndarray that allows field access using attributes

__new__ (subtype, shape, formats, names=None, titles=None, buf=None, offset=0, strides=None, byteorder=None, aligned=0)

Construct an array of the given subtype and shape with data-type (record, dtype) where dtype is constructed from formats, names, and titles. If buf is None, then create new memory. Otherwise, use the memory of buf exposed through the buffer protocol.

format_parser A class useful for creating a data-type descriptor from formats, names, titles, and aligned arguments. This is used by several of the record array constructors for consistency in behavior.

__init__ (self, formats, names, titles, aligned=False, byteorder=None)

Construct a data-type object from formats, names, titles, aligned, and byteorder arguments. Upon completion the constructed data-type object is in self._descr.

8.6 Masked Arrays (numpy.ma)

These are adapted from the masked arrays provided with Numeric. Masked Arrays do not inherit from the ndarray, they simply use two ndarray objects in their internal representation. Fortunately, as I have not used masked arrays in my work, Paul

Dubois (the original author of MA for Numeric) adapted and modified the code for use by NumPy. Alexander Belopolsky (Sasha) added additional functions and improvements

Masked arrays are created using the masked array creation function.

ma.array (data, dtype=None, copy=True, order='C', mask=ma.nomask, fill_value=None)

data Something that can be converted to an array. If data is already a masked array, then if mask is ma.nomask, the mask used be data.mask and the data used data.data.

dtype The data-type of the underlying array

copy If copy is False, then every effort will be made to not copy the data.

order Specify whether the array is in 'C', 'Fortran', or 'Any' order

mask Masked values are excluded from calculations. If this is ma.nomask, then there are no masked values. Otherwise, this should be an object that is convertible to an array of Booleans with the same shape as data.

fill_value This value is used to fill in masked values when necessary. The fill_value is not used for computation for functions within the ma module.

Masked arrays have the same methods and attributes as arrays with the addition of the mask attribute as well as the "hidden" attributes ._data and ._mask.

8.7 Standard container class

For backward compatibility and as a standard "container" class, the UserArray from Numeric has been brought over to NumPy and named **numpy.lib.user_array.container** The container class is a Python class whose self.array attribute is an ndarray. Multiple inheritance is probably easier with numpy.lib.user_array.container than with the ndarray itself and so it is included by default. It is not documented here beyond mentioning its existence because you are encouraged to use the ndarray class directly if you can.

8.8 Array Iterators

Iterators are a powerful concept for array processing. Essentially, iterators implement a generalized for-loop. If myiter is an iterator object, then the Python code

```
for val in myiter:
    ...
    some code involving val
    ...
```

calls val=myiter.next() repeatedly until StopIteration is raised by the iterator. There are several ways to iterate over an array that may be useful: default iteration, flat iteration, and N-dimensional enumeration.

8.8.1 Default iteration

The default iterator of an ndarray object is the default Python iterator of a sequence type. Thus, when the array object itself is used as an iterator. The default behavior is equivalent to:

```
for i in arr.shape[0]:
    val = arr[i]
```

This default iterator selects a sub-array of dimension $N-1$ from the array. This can be a useful construct for defining recursive algorithms. To loop over the entire array requires N for-loops.

```
>>> a = arange(24).reshape(3,2,4)+10
>>> for val in a:
...     print 'item:', val
item: [[10 11 12 13]
[14 15 16 17]]
item: [[18 19 20 21]
[22 23 24 25]]
item: [[26 27 28 29]
[30 31 32 33]]
```

8.8.2 Flat iteration

As mentioned previously, the flat attribute of ndarray objects returns an iterator that will cycle over the entire array in C-style contiguous order.

```
>>> for i, val in enumerate(a.flat):
...     if i%5 == 0: print i, val
0 10
5 15
10 20
15 25
20 30
```

Here, I've used the built-in enumerate iterator to return the iterator index as well as the value.

8.8.3 N-dimensional enumeration

Sometimes it may be useful to get the N-dimensional index while iterating. The ndenumerate iterator can achieve this.

```
>>> for i, val in ndenumerate(a):
...     if sum(i)%5 == 0: print i, val
(0, 0, 0) 10
(1, 1, 3) 25
(2, 0, 3) 29
(2, 1, 2) 32
```

8.8.4 Iterator for broadcasting

The general concept of broadcasting is also available from Python using the **broadcast** iterator. This object takes N objects as inputs and returns an iterator that returns tuples providing each of the input sequence elements in the broadcasted result.

```
>>> for val in broadcast([[1,0],[2,3]],[0,1]):
...     print val
(1, 0)
(0, 1)
(2, 0)
(3, 1)
```

The methods and attributes of the broadcast object are:

nd the number of dimensions in the broadcasted result.

shape the shape of the broadcasted result.

size the total size of the broadcasted result.

index the current (flat) index into the broadcasted array

iters a tuple of (broadcasted) NumPy.flatiter objects, one for each array.

reset ()

Reset the multiter object to the beginning.

next ()

Get the next tuple of objects from the (broadcasted) arrays

Chapter 9

Universal Functions

Computers make it easier to do a lot of things, but most of the things they make it easier to do don't need to be done.

—*Andy Rooney*

People think computers will keep them from making mistakes. They're wrong. With computers you make mistakes faster.

—*Adam Osborne*

9.1 Description

Universal functions are wrappers that provide a common interface to mathematical functions that operate on scalars, and can be made to operate on arrays in an element-by-element fashion. All `u`niversal `func`tion`s` (`ufuncs`) wrap some core function that takes n_i (scalar) inputs and produces n_o (scalar) outputs. Typically, this core function is implemented in compiled code but a Python function can also be wrapped into a universal function using the basic method `frompyfunc` in the umath module.

frompyfunc (func, nin, nout)

This function returns a new universal function wrapping a Python function func with nin inputs and nout outputs. The resulting universal function works using Object arrays for both input and output. The vectorize class makes use of frompyfunc internally. You can view the source code using numpy.source(numpy.vectorize).

9.1.1 Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs. The standard broadcasting rules are applied so that inputs without exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

1. All input arrays with ndim smaller than the input array of largest ndim have 1's pre-pended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input shapes in that dimension.
3. An input can be used in the calculation if it's shape in a particular dimension either matches the output shape or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension when otherwise needed (the stride will be 0 for that dimension).

While perhaps a bit difficult to explain, broadcasting can be quite useful and becomes second nature rather quickly. Broadcasting is used throughout NumPy to decide how to handle non equally-shaped arrays.

9.1.2 Output type determination

The output of the ufunc (and its methods) does not have to be an ndarray. All output arrays will be passed to the __array_wrap__ method of any input (besides ndarrays, and scalars) that defines it **and** has the highest __array_priority__ of any other input to the universal function. The default __array_priority__ of the ndarray is 0.0, and the default __array_priority__ of a subtype is 1.0. Matrices have __array_priority__ equal to 10.0.

The ufuncs can also all take output arguments. The output will be cast if necessary to the provided output array. If a class with an __array__ method is used for the output, results will be written to the object returned by __array__. Then, if the class also has an __array_wrap__ method, the returned `ndarray` result will be passed to that method just before passing control back to the caller.

9.1.3 Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of the internal buffers is settable on a per-thread basis. There can be up to $2(n_i + n_o)$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of the buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrect typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function

setbufsize (size)

Set the buffer size to the given number of elements in the current thread. Return the old buffer size (so that it can be reset later if desired).

9.1.4 Error handling

Universal functions can trip special floating point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. The user can determine what should be done if errors are encountered. Error handling is controlled on a per-thread basis. Four errors can be individually configured: divide-by-zero, overflow, underflow, and invalid. The errors can each be set to ignore, warn, raise, or call. The easiest way to configure the error mask is using the function

seterr (all=None, divide=None, over=None, under=None, invalid=None)

This will set the current thread so that errors can be handled if desired. If one of the errors is set to 'call', then a function must be provided using the seterrcall() routine. If any of the arguments are None, then that error mask will be unchanged. The return value of this function is a dictionary with the old error conditions. Thus, you can restore the old condition after you are finished with your function by calling seterr(**old). If all is set, then all errors will be set to the specified value.

seterrcall (callable)

This sets the function to call when an error is triggered for an error condition configured with the "call" handler. This function should take two arguments: a string showing the type of error that triggered the call, and an integer showing the state of the floating point status registers. Any return value of the call function will be ignored, but errors can be raised by the function. Only one error function handler can be specified for all the errors. The status argument shows which errors were raised. The return value of this routine is the old callable. The argument passed in to this function must be any callable object with the right signature or None.

NOTE

FPE_DIVIDEBYZERO, FPE_OVERFLOW, FPE_UNDERFLOW, and FPE_INVALID, are all defined constants in NumPy. The status flag returned for a 'call' error handling type shows which errors were raised by adding these constants together.

9.1.5 Optional keyword arguments

All ufuncs take optional keyword arguments. These represent rather advanced usage and will likely not be used by most users.

sig= either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to specify a specific signature for a the 1-d loop to use in the underlying calculation. If the loop specified does not exist for the ufunc, then a TypeError is raised. Normally a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This key-word argument lets you bypass that search and choose a loop you want. A list of available signatures is available in the **types** attribute of the ufunc object.

extobj= a list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error call-back function. Normally, these values are looked-up in a thread-specific dictionary. Passing them here bypasses that look-up and uses the low-level specification provided for the error-mode. This may be useful as an optimization for calculations requiring lots of ufuncs on small arrays in a loop.

9.2 Attributes

There are some informational attributes that universal functions possess. None of the attributes can be set.

__doc__

A docstring for each ufunc. The first part of the docstring is dynamically generated from the number of outputs, the name, and the number of inputs. The second part of the doc string is provided at creation time and stored with the ufunc.

__name__

The name of this ufunc.

nin

The number of inputs

nout

The number of outputs

nargs

The total number of inputs + outputs

ntypes

The total number of different types for which this ufunc is defined.

types

A list of length ntypes containing strings showing the types for which this ufunc is defined. Other types may still be used as inputs (and as output arrays), they will just need casting. For inputs, standard casting rules will be used to determine which of the supplied internal functions that will be used (and therefore the default type of the output). Results will always be force-cast to any array provided to hold the output.

Table 9.1: Universal function (ufunc) attributes.

Name	Description
__doc__	Dynamic docstring.
__name__	Name of ufunc
nin	Number of input arguments
nout	Number of output arguments
nargs	Total number of arguments
ntypes	Number of defined inner loops.
types	List showing types for which inner loop is defined.
identity	Identity for this ufunc.

identity

A 1, 0, or None to show the identity for this universal function. This identity is used for reduction on zero-sized arrays (arrays with a shape that includes a 0).

9.3 Casting Rules

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the `.types` attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (the character codes are used in that output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast "safely." The first one it finds in its internal list of loops is selected and performed with types cast. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer which is user settable.

NOTE

Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating point and integer values. See ldexp for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast "safely" to another data type. The answer to this question can be determined in Python with a function call: can_cast (fromtype, totype). Figure shows the results of this call for my 32-bit system on the 21 internally supported types. You can generate this table for your system with code shown in that Figure.

You should note that, while included in the table for completeness, the 'S', 'U', and 'V' types cannot be operated on by ufuncs. Also, note that on a 64-bit system the integer types may have different sizes resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot upcast an array unless the scalar is of a fundamentally different kind of data (*i.e.* under a different hierachy in the data type hierarchy) then the array. This rule enables you to use scalar constants in your code (which as Python types are interpreted accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.

9.4 Methods

All ufuncs have 4 methods. However, these methods only make sense on ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a `ValueError`. The reduce-like methods all take an axis keyword and a dtype keyword, and the arrays must all have dimension >= 1. The *axis* keyword specifies which axis of the array the reduction will take place over and may be negative, but must be an integer. The *dtype* keyword allows you to manage a very common problem that arises when naively using <op>.reduce. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The dtype keyword allows you to alter the data type that the reduction takes place over (and therefore the type of the output). Thus, you can ensure that the output is a data type with

```
>>> def print_table(ntypes):
...     print 'X',
...     for char in ntypes: print char,
...     print
...     for row in ntypes:
...         print row,
...         for col in ntypes:
...             print int(can_cast(row, col)),
...         print
>>> print_table(typecodes['All'])
X ? b h i l q p B H I L Q P f d g F D G S U V O
? 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
b 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
h 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
i 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
l 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
q 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
p 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
B 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
H 0 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
I 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1
L 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1
Q 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1
P 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1
f 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1
g 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1
F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
G 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
U 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
V 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
O 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

Figure 9.1: Code segment showing the can cast safely table for a 32-bit system.

large-enough precision to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no dtype is given for a reduction on the "add" or "multiply" operations, then if the input type is an integer (or boolean) data-type and smaller than the size of the int_ data type, it will be internally upcast to the int_ (or uint) data type.

WARNING

A reduce-like operation on an array with a data type that has range "too small" to handle the result will silently wrap. You should use dtype to increase the data type over which reduction takes place.

9.4.1 Reduce

<op>.reduce (array=, axis=0, dtype=None)

For each one-dimensional sequence along the *axis* dimension of the array, return a single number resulting from recursively applying the operation to succesive elements along that dimension. If the input array has N dimensions, then the returned array has $N-1$ dimensions. This produces the equivalent of the following Python code :

```
>>> indx = [index_exp[:]]*array.ndim
>>> indx[axis] = 0; N=array.shape[axis]
>>> result = array[indx].astype(dtype)
>>> for i in range(1,N):
...     indx[axis] = i
...     <op>(result, array[indx], result)
```

Studying the above code can also help you gain an appreciation for how to do generic indexing in Python using `index_exp`. For example, if <op> is add, then <op>.reduce produces a summation along the given axis. If <op> is prod, then a repeated multiply is performed.

9.4.2 Accumulate

<op>.accumulate (array=, axis=0, dtype=None)

This method is similar to reduce, except it returns an array of the same shape as the input, and keeps intermediate calculations. The operation is still performed along the access. This method underlies the operations of the cumsum and cumprod methods of arrays. The following Python code implements an equivalent of the accumulate method.

```
>>> i1 = [index_exp[:]]*array.ndim
>>> i2 = [index_exp[:]]*array.ndim
>>> i1[axis] = 0; N=array.shape[axis]
>>> result = array.astype(dtype)
>>> for i in range(1,N):
...     i1[axis] = i
...     i2[axis] = i-1
...     <op>(result[i1], array[i1], result[i2])
```

9.4.3 Reduceat

<op>.reduceat (array=, indices=, axis=0, dtype=None)

This method is a generalization of both reduce and accumulate. It offers the ability to reduce along an axis but only between certain indices. The indices input must be a one dimensional (index) sequence. Then, if I_k is the k^{th} element of indices, the reduceat method computes <op>.reduce(array[I_k:I_{k+1}]). This formula assumes $I_{k+1} > I_k$, and also that I_{k+1} is the length of the input array when I_k is the last element. There is no requirement that the indices be monotonic. If $I_{k+1} \leq I_k$, then reduceat simply returns array[I_k] for that particular element of indices. In these formulas, we have assumed that array is one dimensional (or axis is 0). If the array is N-dimensional and axis>0, then the index expression needs axis ':' (full slice objects) inserted (i.e. array[$\underbrace{:,\ldots,:}_{\text{axis}}, I_k : I_{k+1}$]). The effect is to slice along the axis dimension.

Equivalent Python code is

```
>>> i1 = [index_exp[:]]*array.ndim
>>> i2 = [index_exp[:]]*array.ndim
>>> outshape = list(array.shape)
>>> N = array.shape[axis]
>>> outshape[axis] = len(indices)
>>> result = zeros(outshape, dtype or array.dtype)
>>> for k,Ik in enumerate(indices):
...     i1[axis] = k
...     try:
```

```
...          Ikp1 = indices[k+1]
...      except IndexError:
...          Ikp1 = N
...      if (Ikp1 > Ik):
...          i2[axis] = index_exp[Ik:Ikp1]
...          result[i1] = <op>.reduce(array[i2],axis=axis,dtype=dtyp
...      else:
...          result[i1] = array[Ik].astype(dtype)
```

The returned array has as many dimensions as the input array, and is the same shape except for the *axis* dimension which has shape equal to the length of indices (the number of reduce operations that were performed). If you ever have a need to compute multiple reductions over portions of an array, then (if you can get your mind around what it is doing) reduceat may be just what you were looking for.

Example: Suppose a is a two-dimensional array of shape 10×20. Then, res=add.reduce (a, [0,3,1]) returns a 3×20 array with res[0,:] = add.reduce(a[:,0:3]), res[1,:] = a[:,3], and res[2,:] = add.reduce(a[:,1:]).

9.4.4 Outer

<op>.outer (a, b)

This method computes an outer operation on <op>. It computes <op>(a2, b2) where a2 is 'a' with b.ndim 1's post-pended to it's shape and b2 is 'b' with a.ndim 1's pre-pended to its shape (broadcasting takes care of this automatically in the code below). The return shape has a.ndim + b.ndim dimensions. Equivalent Python code is

```
>>> a.shape += (1,)*b.ndim
>>> <op>(a,b)
>>> a = a.squeeze()
```

Among many other uses, arithmetic tables can be conveniently built using outer:

```
>>> multiply.outer([1,7,9,12],arange(5,12))
array([[  5,   6,   7,   8,   9,  10,  11],
[ 35,  42,  49,  56,  63,  70,  77],
[ 45,  54,  63,  72,  81,  90,  99],
[ 60,  72,  84,  96, 108, 120, 132]])
```

9.5 Available ufuncs

There are currently more than 60 universal functions defined on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (i.e. add(a,b) is called internally when a + b is written and a or b is an ndarray). Nonetheless, you may still want to use the ufunc call in order to use the optional output argument(s) to place the output(s) in an object (or in objects) of your choice.

Recall that each ufunc operates element-by-element. Therefore, each ufunc will be described as if acting on a set of scalar inputs to return a set of scalar outputs.

NOTE

The ufunc still returns its output(s) even if you use the optional output argument(s).

9.5.1 Math operations

add $(x_1, x_2\ [, y])$

$y = x_1 + x_2$. Called to implement `x1+x2` for arrays

subtract $(x_1, x_2\ [, y])$

$y = x_1 - x_2$. Called to implement `x1-x2` for arrays

multiply $(x_1, x_2\ [, y])$

$y = x_1 \cdot x_2$. Called to implement `x1*x2` for arrays.

divide $(x_1, x_2\ [, y])$

$y = x_1/x_2$ Integer division results in truncation. Floating-point does not. Called to implement `x1/x2` for arrays (when __future__.division is not active).

true_divide $(x_1, x_2\ [, y])$

This version of division always returns an inexact number so that integer division returns floating point. Called with __future__.division is active to implement `x1/x2` for arrays.

floor_divide $(x_1, x_2\ [, y])$

This version of division always results in truncation of an fractional part remaining. Called to implement `x1//x2` for arrays.

negative (x [, y])

$y = -x$. Called to implement `-x` for arrays.

power (x_1, x_2 [, y])

$y = x_1^{x_2}$. There is no three-term power ufunc defined. This two-term power function is called to implement `pow(x1,x2,<any>)` or `x1**x2` for arrays. Note that the third term in `pow` is ignored for array arguments.

remainder (x_1, x_2 [, y])

Returns $x - y$*floor(x/y). Result has the sign of y. Called to implement `x1%x2`.

mod (x_1, x_2 [, y])

Same as remainder (x_1, x_2 [, y]).

fmod (x_1, x_2 [, y])

$x_1 = kx_2 + y$ where k is the largest integer satisfying this equation. Computes C-like $x_1\%x_2$ element-wise. This was the behavior of `x1%x2` in old Numeric.

absolute (x [, y])

$y = |x|$. Called to implement `abs(x)` for arrays.

rint (x, [, y])

Round x to the nearest integer. Rounds half-way cases to the nearest even integer.

sign (x [, y])

Sets y according to

$$\mathrm{sign}\,(x) = \begin{cases} 1 & x :> 0, \\ 0 & x = 0, \\ -1 & x < 0. \end{cases}$$

conj (x [, y])

conjugate (x [, y])

$y = \overline{x}$; in other words, the complex conjugate of x.

exp $(x\ [, y])$

$y = e^x$.

log $(x\ [, y])$

$y = \log(x)$. In other words, y is the number so that $e^y = x$.

expm1 $(x, [, y])$

$y = e^x - 1$. Calculated so that it is accurate for small $|x|$.

log1p $(x, [, y])$

$y = \log(1 + x)$ but accurate for small $|x|$. Returns the number y such that $e^y - 1 = x$

log10 $(x\ [, y])$

$y = \log 10(x)$. In other words, y is the number so that $10^y = x$.

sqrt $(x\ [, y])$

$y = \sqrt{x}$.

square $(x\ [,y])$

$y = x * x$

reciprocal $(x\ [, y])$

$y = 1/x$

ones_like $(x, [, y])$

$y = 1$ If an output argument is not given the returned data-type is the same as the input data type.

TIP

The optional output arguments can be used to help you save memory for large calculations. If your arrays are large, complicated expressions can take longer than absolutely necessary due to the creation and (later) destruction of temporary calculation spaces. For example, the expression 'G=a*b+c' is equivalent to t1=A*B; G=T1+C; del t1; It will be more quickly executed as G=A*B; add(G,C,G) which is the same as G=A*B; G+=C.

9.5.2 Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is $180°/\pi$.

sin (x [, y])

cos (x [, y])

tan (x [, y])

The standard trignometric functions. $y = \sin(x)$, $y = \cos(x)$, and $y = \tan(x)$.

arcsin (x [, y])

arccos (x [, y])

arctan (x [, y])

The inverse trigonometric functions: $y = \sin^{-1}(x)$, $y = \cos^{-1}(x)$, $y = \tan^{-1}(x)$. These return the value of y (in radians) such that $\sin(y) = x$ with $y \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$; $\cos(y) = x$ with $y \in [0, \pi]$; and $\tan(y) = x$ with $y \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, respectively.

arctan2 (x_1, x_2 [, y])

Returns $\tan^{-1}\left(\frac{x_1}{x_2}\right)$ but takes into account the sign on x_1 and x_2 to place the angle in the correct quadrant. The angle y is returned in the full range $-\pi < y \leq \pi$. The angle is chosen so that $\sin(y) = \frac{x_1}{\sqrt{x_1^2+x_2^2}}$, and $\cos(y) = \frac{x_2}{\sqrt{x_1^2+x_2^2}}$. Particular values are showin in the following table:

x_1	x_2	$y = \text{arctan2}(x_1, x_2)$
0	1	0
1	0	$\frac{\pi}{2}$
0	-1	π
-1	0	$-\frac{\pi}{2}$

hypot (x_1, x_2 [, y])

Returns $y = \sqrt{x_1^2 + x_2^2}$. Given a complex number in cartesian form, arctan2 and hypot can be used to compute phase and magnitude, quickly.

sinh (x [, y])

Computes $y = \sinh(x)$ which is defined as $\frac{1}{2}\left(e^x - e^{-x}\right)$.

cosh (x [, y])

Computes $y = \cosh(x)$ which is defined as $\frac{1}{2}\left(e^x + e^{-x}\right).$

tanh (x [, y])

Computes $y = \tanh(x)$ which is defined as $\left(e^x - e^{-x}\right)/\left(e^x + e^{-x}\right).$

arcsinh (x [, y])

arccosh (x [, y])

arctanh (x [, y])

These compute the inverse hyperpolic functions. $y = \text{arc}func(x)$ is the (principal) value of y such that $func(y) = x$.

9.5.3 Bit-twiddling functions

These function all need integer arguments and they maniuplate the bit-pattern of those arguments.

bitwise_and (x_1, x_2 [, y])

Each bit in y is the result of a bit-wise 'and' operation on the corresponding bits in x_1 and x_2. Called to implement `x1&x2` for arrays.

bitwise_or (x_1, x_2 [, y])

Each bit in y is the result of a bit-wise 'or' operation on the corresponding bits in x_1 and x_2. Called to implement `x1|x2` for arrays.

bitwise_xor (x_1, x_2 [, y])

Each bit in y is the result of a bit-wise 'xor' operation on the corresponding bits in x_1 and x_2. An xor operation sets the output to 1 if one and only one of the input bits is 1. Called to implement `x1^x2` for arrays. Using the bitwise_xor operation and the optional output argument you can swap the values of two integer arrays of equivalent types without using temporary arrays.

```
>>> a=arange(10)
>>> b=arange(10,20)
>>> bitwise_xor(a,b,a); bitwise_xor(a,b,b);
>>> bitwise_xor(a,b,a)
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> print a; print b
[10 11 12 13 14 15 16 17 18 19]
[0 1 2 3 4 5 6 7 8 9]
```

invert (x, [, y])

Each bit in y is the opposite of the corresponding bit in x. Called to implement `~x` for arrays.

left_shift (x_1, x_2 [, y])

Shifts the bits of x_1 to the left by x_2. Called to implement `x1<<x2` for arrays. Provided there is no overflow, the result is equal to $y = x_1 2^{x_2}$.

right_shift (x_1, x_2 [, y])

Shifts the bits of x_1 to the right by x_2. Called to implement `x1>>x2` for arrays. Absent overflow, the result is equal to $y = x_1 2^{-x_2}$.

9.5.4 Comparison functions

All of these functions (except maximum, minimum, and sign) return Boolean arrays.

greater (x_1, x_2 [, y])

greater_equal (x_1, x_2 [, y])

less (x_1, x_2 [, y])

less_equal (x_1, x_2 [, y])

not_equal (x_1, x_2 [, y])

equal (x_1, x_2 [, y])

These functions are called to implement `x1>x2`, `x1>=x2`, `x1<x2`, `x1<=x2`, `x1!=x2` (or `x1<>x2`), and `x1==x2`, respectively, for arrays.

The fact that these functions return Boolean arrays make them very useful in combination with advanced array indexing. Thus, for example, arr[arr>10] = 10 clips large values to 10. Used in conjunction with bitwise operators quite complicated expressions are possible. For example, arr[˜((arr<10)&(arr>5))] = 0 clips all values outside of the range $(5, 10)$ to 0.

WARNING

Do not use the Python keywords `and` and `or` to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators: & and | instead.

logical_and (x_1, x_2 [, y])

The output is the truth value of x_1 **and** x_2. Numbers equal to 0 are considered False. Nonzero numbers are True.

logical_or (x_1, x_2 [, y])

The output, y, is the truth value of x_1 **or** x_2 .

logical_xor (x_1, x_2 [, y])

The output, y, is the truth value of x_1 **xor** x_2, which is the same as (x_1 and not x_2) or (not x_1 and x_2).

logical_not (x [, y])

The output, y is the truth value of **not** x.

WARNING

The Bitwise operators (& and |) are the proper way to combine element-by-element array comparisons. Be sure to understand the operator precedence: (a>2) & (a<5) is the proper syntax because a>2 & a<5 will result in an error due to the fact that 2 & a is evaluated first.

maximum (x_1, x_2 [, y])

The output, y, is the larger of x_1 and x_2.

```
>>> maximum([1,0,5,10],[3,2,4,5])
array([ 3,  2,  5, 10])
>>> max([1,0,5,10],[3,2,4,5])
[3, 2, 4, 5]
```

TIP

The Python function max() will find the maximum over a one-dimensional array, but it will do so using a slower sequence interface. The reduce method of the maximum ufunc is much faster. Also, the max() method will not give answers you might expect for arrays with greater than one dimension. The reduce method of minimum also allows you to compute a total minimum over an array.

minimum (x_1, x_2 [, y])

The output, y, is the smaller of x_1 and x_2.

```
>>> minimum([1,0,5,10],[3,2,4,5])
array([1, 0, 4, 5])
>>> min([1,0,5,10],[3,2,4,5])
[1, 0, 5, 10]
```

WARNING

the behavior of maximum(a,b) is than that of max(a,b). As a ufunc, maximum(a,b) performs an element-by-element comparison of a and b and chooses each element of the result according to which element in the two arrays is larger. In contrast, max(a,b) treats the objects a and b as a whole, looks at the (total) truth value of a>b and uses it to return either a or b (as a whole). A similar difference exists between minimum(a,b) and min(a,b).

9.5.5 Floating functions

Recall that all of these functions work element-by-element over an array, returning an array output. The description details only a single operation.

isreal (x)

True if x has an imaginary part that is 0.

iscomplex (x)

True if x has an imaginary part that is non-zero.

isfinite (x)

True if x is a finite floating point number (not a NaN or an Inf).

isinf (x)

True if x is $\pm\infty$.

isnan (x)

True if x is Not-a-Number. This represents invalid results. When a NaN is created, the invalid flag is set. If you set the error mode of invalid to 'warn', 'raise', or 'call', then the appropriate action will be performed on NaN creation.

signbit (x)

True where the sign bit of the floating point number is set. This should correspond to $x > 0$ when x is a finite number. When, x is NaN or infinite, then this tests the actual signbit.

modf (x [, y_1, y_2])

Breaks up the floating point value x into its fractional, y_1, and integral, y_2, parts. Thus, $x = y_1 + y_2$ with `floor(y2)==y2`.

ldexp (x, n [, y])

Fast multiply of a floating point number by an integral power of 2: $y = 2^n x$.

frexp (x [, y, n])

Breaks up the floating point value x into a normalized fraction, y and an exponent, n which corresponds to how the value is represented in the computer. The results are such that $x = y2^n$. Effectively, the inverse of ldexp.

fmod (x_1, x_2 [, y])

Computes the remainder of dividing x_1 by x_2. The result, y, is $x_1 - nx_2$ where n is the quotient (rounded towards zero to an integer) of x_1/x_2.

floor (x [,y])

Return $y = \lfloor x \rfloor$ where y is the nearest integer smaller-than or equal to x. Thus, $\lfloor x \rfloor \le x < \lfloor x \rfloor + 1$.

ceil (x [,y])

Return $y = \lceil x \rceil$ where y is the nearest integer greater-than or equal to x. Thus, $x \le \lceil x \rceil < x + 1$.

Chapter 10

Basic Modules

The NumPy distribution contains some basic functionality equivalent to what was available in the Numeric packages previously. This section documents the new interfaces. These are sub-packages of the NumPy namespace. The linalg and fft capabilities are useful but limited. You should install the full SciPy package to access more functionality. The numpy.dual module contains functions that are defined in both SciPy and NumPy. If SciPy defines func, then numpy.dual.func will point to the SciPy version, otherwise it will point to the NumPy version. It must be imported specifically to be used. Table 10.1 shows the functions defined in numpy.dual that are in both NumPy and SciPy.

10.1 Linear Algebra (`linalg`)

These functions are in the numpy.linalg sub-package.

inv (A)

Table 10.1: Functions in numpy.dual (both in NumPy and SciPy)

Family	Functions
Fourier Transforms	fft, ifft, fft2, ifft2, fftn, ifftn
Linear Algebra	norm, det, inv, pinv, solve, eig, eigh, eigvals, eigvalsh, lstsq, cholesky, svd
Special Functions	i0

Return the (matrix) inverse of the 2-d array A. The result, X, is such that dot(A,X) is equal to eye(*A.shape) (to within machine precision).

solve (A,b)

Find the solution to the linear equation $\mathbf{Ax} = \mathbf{b}$, where A is a 2-d array and b is a 1-d or 2-d array.

tensorsolve (A, b, axes=None)

Find the solution, x_{kl}, to the multi-index linear equation

$$\sum_{kl} A_{ijkl} x_{kl} = b_{ij}.$$

The axes argument specifies which dimensions of A are summed over. If it is None, then the last A.ndim - b.ndim dimensions are summed over. The result, therefore, has dimension x.ndim = A.ndim-b.ndim.

tensorinv (A, ind=2)

Find the tensor inverse of A, defined to be the tensor such that tensordot (Ainv, A) is an identity operator.

cholesky (A)

Return, $\mathbf{L}$, the Cholesky decomposition of $\mathbf{A}$. Cholesky decomposition is applicable to a Hermitian, positive definite matrices. When $\mathbf{A} = \mathbf{A}^H$ and $\mathbf{x}^H \mathbf{Ax} \geq 0$ for all $\mathbf{x}$, then decompositions of $\mathbf{A}$ can be found so that $\mathbf{A} = \mathbf{LL}^H$, where $\mathbf{L}$ is lower-triangular.

eigvals (A)

Return all solutions (λ) to the equation $\mathbf{Ax} = \lambda\mathbf{x}$.

eig (A)

Return all solutions $(\lambda, \mathbf{x})$ to the equation $\mathbf{Ax} = \lambda\mathbf{x}$. The first element of the return tuple contains all the eigenvalues. The second element of the return tuple contains the eigenvectors in the columns (x[:,i] is the ith eigenvector).

eigvalsh (U)

eigh (U)

These functions are identical to eigvals and eig except they only work with Hermitian matrices where $\mathbf{U}^H = \mathbf{U}$ (only the lower-triangular part of the array is used).

svd (A)

Compute the singular value decomposition of the 2-d array $\mathbf{A}$. Every $m \times n$ matrix can be decomposed into a pair of unitary matrices, $\mathbf{U} = \mathbf{U}^H$ $(m \times m)$ and $\mathbf{V} = \mathbf{V}^H$ $(n \times n)$ and an $m \times n$ "diagonal" matrix $\mathbf{\Sigma}$, such that $\mathbf{A} = \mathbf{U\Sigma V}^H$. The only non-zero portion of $\mathbf{\Sigma}$ is the upper $r \times r$ block where $r = \min(m, n)$. The svd function returns three arrays as a tuple: ($\mathbf{U}$, $\boldsymbol{\sigma}$, $\mathbf{V}^H$). The singular values are returned in the 1-d array $\boldsymbol{\sigma}$. If needed, the array $\mathbf{\Sigma}$ can be found (if really needed) using the command diag($\boldsymbol{\sigma}$) which creates the $r \times r$ diagonal block and then inserting this into a zeros matrix:

```
>>> A = random.rand(3,5)
>>> from numpy.dual import svd; U,s,Vh = svd(A)
>>> r=min(*A.shape); Sig = zeros_like(A);
>>> Sig[:r,:r] = diag(s); print Sig
[[ 2.1634  0.      0.      0.      0.    ]
[ 0.      0.7076  0.      0.      0.    ]
[ 0.      0.      0.2098  0.      0.    ]]
```

pinv (A, rcond=10^{-10})

Return the generalized, pseudo inverse, of A. For invertible matrices, this is the same as the inverse.

det (A)

Return the determinant of the array. The determinant of an array is the product of its singular values.

lstsq (A, b, rcond=10^{-10})

Return (x, resids, rank, s) where x minimizes resids=$\|\mathbf{Ax} - \mathbf{b}\|_2$. The output rank is the rank of A and s is the singular values of a in descending order. Singular values less than s[0]*rcond are treated as 0. If the rank of A is less than the number of columns of A or greater than the number of rows, resids will be returned as an empty array.

10.2 Discrete Fourier Transforms (`fft`)

All of the algorithms here are most efficient if the length of the data is a power of 2 (or decomposable into low prime factors).

fft (x, n=None, axis=-1)

Return, X, the N-point Discrete Fourier Transform (DFT) of x along the given axis using a fast algorithm. If N is larger than x.shape[axis], then x will be zero-padded. If N is smaller than x.shape[axis], then the first N items will be used. The result is computed for $k = 0 \ldots n-1$ from the formula:

$$X[k] = \sum_{m=0}^{n-1} x[m] \exp\left(-j\frac{2\pi km}{n}\right).$$

TIP

The fft returns values for $k = 0 \ldots N-1$. Because $X[N-k] = X[-k]$ in the FFT formula, larger values of k correspond also to negative frequencies.

ifft (X, n=None, axis=-1)

Return the inverse of the fft so that (ifft(fft(a)) == a within numerical precision. The order of frequencies must be the same as returned by fft. The result is computed (using a fast algorithm) for $m = 0 \ldots n-1$ from the formula:

$$x[m] = \sum_{k=0}^{n-1} X[k] \exp\left(j\frac{2\pi km}{n}\right).$$

Sometimes having the "negative" frequencies at the end of the output returned by fft can be a little confusing. There are two ways to deal with this confusion. In my opinion, the most useful way is to get a collection of DFT sample frequencies and use them to keep track of where each frequency is. The function `fftfreq` provides these sample frequencies. Making an x-y plot, where the sample frequencies are along the "x"-axis and the result of the DFT is along the "y"-axis provides a useful visualization with the zero-frequency at the center of the plot. The advantage of this approach is that your data is still in proper order for using the `ifft` function. Some people, however, prefer to simply swap one-half of the output with the other. This is exactly what the function `fftshift` does. Of course, now the data is not in the proper order for the ifft function, but to each his own.

The reason that the “negative” frequencies are in the upper part of the return signal was given in the description of the DFT. The reason is that the output of the DFT is just one period of a periodic function (with period n). The traditional output of the FFT algorithm is to provide the portion of the function from from $k = 0$ to $k = n - 1$.

fftshift (x, axes=None)

Shift zero-frequency component to the center of the spectrum. This function swaps half-spaces for all axes listed (defaults to all of them).

ifftshift (x, axes=None)

Reverse the effect of the fftshift operation. Thus, it takes zero-centered data and shifts it into the correct order for the ifft operation.

fftfreq (n, d=1.0)

Provide the DFT sample frequencies. The returned float array contains the frequency bins in the order returned from the fft function. If given, d represents the sample-spacing. The units on the frequency bins are cycles / unit. For example, the following example computes the output frequencies (in Hz) of the fft of 256 samples of a voice signal sampled at 20000Hz.

```
>>> from numpy.fft import fftfreq; f=fftfreq(256,d=1./20e3)
>>> print f[0], f[1], f[2], f[128]
0.0 78.125 156.25 -10000.0
```

fft2 (x, s=None, axes=(-2,-1))

Return the two-dimensional fft of the array x for each 2-d array formed by axes. The 2-d fft is computed as

$$X[k_0, k_1] = \sum_{m_0=0}^{s[0]-1} \sum_{m_1=0}^{s[1]-1} x[m_0, m_1] \exp\left(-j\frac{2\pi k_0 m_0}{s[0]}\right) \exp\left[-j\frac{2\pi k_1 m_1}{s[1]}\right]$$

and can be realized by repeated application of the 1-d fft (first over the axes[0] and then over axes[1]). In other-words fft2(x,s,axes) is equivalent to fft(fft(x, s[0], axes[0]), s[1], axes[1]). The 2-d fft is returned for $k_0 = 0 \ldots s[0] - 1$ and $k_1 = 0 \ldots s[1] - 1$. Symmetry ($X[s[0] - k_0, s[1] - k_1] = X[-k_0, -k_1]$) ensures that higher values of k_i correspond to negative frequencies.

ifft2 (X, s=None, axes=(-2,-1))

Return the inverse of the two-dimension fft. Thus, ifft2(fft2(x)) == x to within numerical precision. Note that the "negative frequencies" must be

fftn (x, s=None, axes=None)

Return the N-dimensional fft of x. If s is not given, then if axes is given, then N=len(axes), otherwise N=x.ndim. If s is given, then N=len(s). Results are computed using a similar formula as for the 1- and 2-d FFT with N summations.

ifftn (X, s=None, axes=None)

Return the N-dimensional inverse fft of X. Note that ifftn(fftn(x)) == x to within machine precision.

The Discrete Fourier transform returns complex-valued data (even for real-valued input). If the data was originally real-valued, then much of the output of the full DFT is redundant. Notice that if $x\left[m\right]$ is real, then

$$\begin{aligned} X\left[n-k\right] &= \sum_{m=0}^{n-1} x[m]\exp\left(-j\frac{2\pi\left(n-k\right)m}{n}\right) \\ &= \left[\sum_{m=0}^{n-1} x\left[m\right]\exp\left(-j\frac{2\pi km}{n}\right)\right]^{*} \\ &= X^{*}\left[k\right], \end{aligned}$$

where a^{*} denotes the complex-conjugate of a. So, the upper half of the fft output (the negative frequencies) is determined exactly by the lower half of the output when the input is purely real. This kind of symmetry is called Hermitian symmetry. The real-valued Fourier transforms described next take advantage of Hermitian symmetry to compute only the unique outputs more quickly.

The symmetry in higher dimensions is always about the origin. If N is the number of dimensions, then:

$$X[n_1-k_1, n_2-k_2, \ldots n_N-k_N] = X^{*}\left[k_1, k_2, \ldots, k_N\right].$$

Thus, the data-savings remains constant at about 1/2 for higher dimensions as well.

rfft (x, n=None, axis=-1)

Compute the first n//2+1 points of the n-point discrete Fourier transform of the real valued data along the given axis. The returned array will be just the first half of the `fft`, corresponding to positive frequencies: rfft(x) == fft(x)[:n//2+1]

irfft (X, n=None, axis=-1)

Compute the inverse n-point discrete Fourier transform along the given axis using the first n//2+1 points. To within numerical precision, irfft(rfft(x))==x.

rfft2 (x, s=None, axes=(-2, -1))

Compute only the first half-plane of the two-dimensional discrete Fourier transform corresponding to unique values. s[0] and s[1]-point DFTs will be computed along axes[0] and axes[1] dimensions, respectively. Requires a real array. If s is None it defaults to the shape of x. The real fft will be computed along the last axis specified in axes while a full fft will be computed in the other dimension.

irfft2 (X, s=None, axes=(-2, -1))

Compute the inverse of the 2-d DFT using only the first quadrant. Returns a real array such that to within numerical precision irfft2(rfft2(x))==x.

rfftn (x, s=None, axes=None)

Compute only the unique part of the N-dimensional DFT from a real-valued array. If s is None it defaults to the shape of x. If axes is not given it defaults to all the axes (-n,. . ., -1). The length of axes provides the dimensionality of the DFT. The unique part of the real N-dimensional DFT is obtained by slicing the full fft along the last axis specified and taking n//2+1 slices. rfftn(x) == fft(x)[sliceobj] where sliceobj[axes[-1]] = slice(None,s[-1]//2+1,None).

irfftn (X, s=None, axes=None)

Compute the inverse DFT from the unique portions of the N-dimensional DFT provided by `rfftn`.

Occasionally, the situation may arise where you have complex-valued data with Hermitian symmetry (or real-valued symmetric data). This ensures that the Fourier transform will be real. The two functions below can calculate it without wasting extra space for the zero-valued imaginary entries of the Discrete Fourier transform, or the entire signal.

hfft (x, n=None, axis=-1)

Calculate the n-point real-valued Fourier transform from (the first half of Hermitian-symmetric data, x.

ihfft (X, n=None, axis=-1)

Return (the first half-of) Hermitian-symmetric data from the real-valued Fourier transform, X.

10.3 Random Numbers (`random`)

The random number capabilities surpass those that were available in Numeric. The random number facilities were generously contributed by Robert Kern, who has been a dedicated and patient help to many mailing list questioners. Robert built the random package using Pyrex to build on top of his own code as well as that of randomkit by Jean-Sebastien Roy as well as code by Ivan Frohne. The fundamental random number generator is the Mersenne Twister based on code written by Makoto Matsumoto and Takuji Nishimura (and modified for Python by Raymond Hettinger). Random numbers from discrete and continuous distributions are available, as well as some useful random-number-related utilities. Many of the random number generators are based on algorithms published by Luc Devroye in "Non-Uniform Random Variate Generation" available electronically at `http://cgm.cs.mcgill.ca/˜luc/rnbookindex.html`

Each of the discrete and continuous random number generators take a size keyword. If this is None (default), then the size is determined from the additional inputs (using ufunc-like broadcasting). If no additional inputs are needed, or if these additional inputs are scalars, then a single number is generated from the selected distribution. If size is an integer, then a 1-d array of that size is generated filled with random numbers from the selected distribution. Finally, if size is a tuple, then an array of that shape is returned filled with random numbers.

Many distributions take additional inputs as parameters. These additional inputs must be broadcastable to each other (and to the size parameter if it is not None). The broadcasting behavior of the additional inputs is ufunc-like.

10.3.1 Discrete Distributions

Discrete random numbers take on only a countable number of values (typically integers). Each distribution has associated with it a probability mass function (pmf), $p_m(k;\cdot)$, that is defined as the probability that the returned random number is k. The arguments after k represent possible parameters to the distribution. Thus, let $X(\cdot)$ represent the random number generator for a particular distribution. Then,

$$p_m(k;\cdot) = \text{Probability}\left\{X(\cdot) = k\right\}.$$

It will be useful to define the discrete indicator function, $I_S(k)$, where S is a set of integers (often represented by an interval). $I_S(k) = 1$ if $k \in S$, otherwise $I_S(k) = 0$. This convenient notation isolates the relevance of a particular functional form to a certain range. Also, the formulas below make use of the following definition:

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n!}{k!\,(n-k)!}$$

where $k! = k \cdot (k-1) \cdot \cdots \cdot 2 \cdot 1$.

binomial (n, p, size=None)

This random number models the number of successes in n independent trials of a random experiment where the probability of success in each experiment is p.

$$p_m(k) = \left(\begin{array}{c} n \\ k \end{array}\right) p^k (1-p)^{n-k} I_{[0,n]}(k).$$

geometric (p, size=None)

This random number models the number of (independent) attempts required to obtain a success where the probability of success on each attempt is p.

$$p_m(k;p) = (1-p)^{k-1} p\, I_{[1,\infty)}(k).$$

hypergeometric (ngood, nbad, nsample, size=None)

Imagine a probability theorists favorite urn filled with n_g"good" objects and n_b "bad" objects. In other words there are two types of objects in a jar. The hypergeometric random number models how many "good" objects will be present when N items are taken out of the urn without replacement.

$$p(k;n_g,n_b,N) = \frac{\left(\begin{array}{c} n_g \\ k \end{array}\right)\left(\begin{array}{c} n_b \\ N-k \end{array}\right)}{\left(\begin{array}{c} n_g+n_b \\ N \end{array}\right)} I_{[N-n_b,\min(n,N)]}(k).$$

logseries (p, size=None)

A random number whose pmf with terms proportional to the Taylor series expansion of $\log(1-p)$. It has been used in biological studies to model the species abundance distribution.

$$p_m(k;p) = -\frac{p^k}{k\log(1-p)} I_{[1,\infty)}(k).$$

multinomial (n, pvals, size=None)

This generator produces random vectors of length N where $N = \mathrm{len}(pvals)$. The shape of the returned array is always the shape indicated by size + $(N,)$. The multinomial distribution is a generalization of the binomial distribution. This time, n trials of an experiment are independently repeated but each trial results in N possible integers $k_1, k_2, \ldots, k_N$ with $\sum_{i=1}^{N} k_i = n$.

$$\begin{aligned} p_m(k_1, k_2, \ldots, k_N; \cdot) &= \text{Probability}\{X(\cdot) = [k_1, k_2, \cdots, k_N]\} \\ &= \frac{n!}{k_1!k_2!\cdots k_N!} p_1^{k_1} p_2^{k_2} \cdots p_N^{k_N} \end{aligned}$$

where $pvals = [p_1, p_2, \ldots, p_N]$. It must be true that $\sum_{i=1}^{N} p_i = 1$. Therefore, as long as $\sum_{i=1}^{N-1} p_i \leq 1$, the last entry in *pvals* is computed as $1 - \sum_{i=1}^{N-1} p_i$.

negative_binomial (n, p, size=None)

Models the number of extra independent trials (beyond n) required to accumulate a total of n successes where the probability of success on each trial is p. Equivalently, this random number models the number of failures encountered while accumulating n successes during independent trials of the experiment that succeeds with probability, p.

$$p_m(k;n,p) = \left(\begin{array}{c} k+n-1 \\ n-1 \end{array}\right) p^n (1-p)^k \, I_{[0,\infty)}(k).$$

poisson (lam=1.0, size=None)

This random number counts the number of successes in n independent experiments (where the probability of success in each experiment is p) in the limit as $n \to \infty$ and $p \to 0$ gets very small such that $\lambda = np \geq 0$ is a constant. It can be used, for example, to model how many typographical errors are on each page of a book.

$$p(k;\lambda) = e^{-\lambda}\frac{\lambda^k}{k!} I_{[0,\infty)}(k).$$

zipf (a, size=None)

The probability mass function of this random number (also called the zeta distribution) is

$$p_m(k;a) = \frac{1}{\zeta(a) k^a} I_{[1,\infty)}(k),$$

where

$$\zeta(a) = \sum_{n=1}^{\infty} \frac{1}{n^a}$$

is the Riemann zeta function. Zipf distributions have been shown to characterize use of words in a natural language (like English), the popularity of library books, and even the use of the web. The Zipf distribution describes collections that have a few items whose probability of selection is very high, a medium number of items whose probability of selection is medium, and a huge number of items whose probability of selection is very low.

10.3.2 Continuous Distributions

Continuous random numbers can take on an uncountable number of values. Therefore, the value returned by a continuous distribution is denoted x. Because there is an uncountable number of possibilities for the random number[1], a continuous distribution is modeled by a probability density function, $f(x;\cdot)$. To obtain the probability that the random number generated by $X(\cdot)$ is in a certain interval, we integrate this density function:

$$\int_{-\infty}^{b} f(x)\, dx = \text{Probability}\{X(\cdot) \leq b\}.$$

To obtain a probability, we have to integrate $f(x)$ which is why it is called a density function. Most continuous distributions are defined by their probability density functions (pdf). Some have basic origins, a few are derived from other distributions, and some are used mainly for modelling unknown distributions.

Some of the parameters of the distributions are labeled as location (*loc*) and *scale* parameters. These parameters are not shown in the equation for the pdf.

[1]A computer actually always generates a random number from a discrete distribution because there are only a finite set of numbers that can be represented by a computer. However, for continuous random number generators, the resulting random numbers usually approximate the continuous distribution well enough to ignore the subtlety.

because they affect the distribution in a known way. This is due to the fact that if X is a number drawn from a distribution with pdf $f_X(x)$, then $Y = Sx + L$ is a number drawn from a distribution with pdf

$$f_Y(y) = \frac{1}{S} f_X\left(\frac{y-L}{S}\right).$$

Thus, from the standard from provided, the pdf of the actual random numbers generated by fixing the location and scale parameters can be quickly found.

In this section, the indicator function $I_A(x)$ will be used where A is a set defined over all the real numbers. For clarity,

$$I_A(x) = \begin{cases} 1 & x \in A, \\ 0 & x \notin A. \end{cases}$$

Also, the following functions are used in the definitions:

$$\begin{aligned} \Gamma(x) &= \int_0^\infty t^{x-1} e^{-t} dt = (x-1)\Gamma(x-1), \\ B(a,b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}. \end{aligned}$$

beta (a, b, size=None)

$$f(x; a, b) = \frac{1}{B(a,b)} x^{a-1} (1-x)^{b-1} I_{(0,1)}(x).$$

chisquare (ν, size=None)

If $Z_1, \ldots, Z_\nu$ are random numbers from standard normal distributions, then $W = \sum_{k=1}^{\nu} Z_k^2$ is a random number from the chi-square (χ^2) distribution with ν degrees of freedom.

$$f(x; \nu) = \frac{1}{2\Gamma\left(\frac{\nu}{2}\right)} \left(\frac{x}{2}\right)^{\nu/2-1} e^{-x/2} I_{[0,\infty)}(x).$$

exponential (scale=1.0, size=None)

$$f(x) = e^{-x} I_{[0,\infty)}(x).$$

f (ν_1, ν_2, size=None)

The distribution of $\frac{X_1/\nu_1}{X_2/\nu_2}$ where X_i is chi-squared with ν_i degrees of freedom.

$$f\left(x;\nu_1,\nu_2\right)=\frac{\nu_2^{\nu_2/2}\nu_1^{\nu_1/2}x^{\nu_1/2-1}}{\left(\nu_2+\nu_1 x\right)^{(\nu_1+\nu_2)/2}B\left(\frac{\nu_1}{2},\frac{\nu_2}{2}\right)}I_{[0,\infty)}\left(x\right).$$

gamma (a, scale=1.0, size=None)

$$f\left(x;a\right)=\frac{1}{\Gamma\left(a\right)}x^{a-1}e^{-x}I_{[0,\infty)}\left(x\right).$$

gumbel (loc=0.0, scale=1.0, size=None)

A right-skewed extreme value distribution.

$$f\left(x\right)=\exp\left[-x-e^{-x}\right].$$

laplace (loc=0.0, scale=1.0, size=None)

$$f\left(x\right)=\frac{1}{2}e^{-|x|}.$$

lognormal (μ=0.0, σ=1.0, size=None)

$$f\left(x;\mu,\sigma\right)=\frac{1}{\sigma x\sqrt{2\pi}}\exp\left[-\frac{1}{2}\left(\frac{\log x-\mu}{\sigma}\right)^2\right]I_{[0,\infty)}\left(x\right),$$

The parameters, μ and σ are not the mean and variance of this distribution, but the parameters of the underlying normal distribution. Random numbers from this distribution are generated as $e^{\sigma Z+\mu}$ where Z is a standard normal random number.

logistic (loc=0.0, scale=1.0, size=None)

$$f\left(x\right)=\frac{e^{-x}}{\left[1+e^{-x}\right]^2}I_{[0,\infty)}\left(x\right)$$

multivariate_normal ($\boldsymbol{\mu}$, $\mathbf{C}$, size=None)

Returns a vector of random numbers which are jointly drawn from a multivariate normal distribution. The last-dimension of the output array contains the sample vector, which is of length $N=\operatorname{len}\left(mean\right)$. The covariance matrix must be $N\times N$. If $\boldsymbol{\mu}\equiv mean$ and $\mathbf{C}=cov$, then the joint-pdf representing the returned random vector(s) is

$$f\left(\mathbf{x}\right)=\frac{1}{\sqrt{\left(2\pi\right)^N\left|\mathbf{C}\right|}}\exp\left[-\frac{1}{2}\left(\mathbf{x}-\boldsymbol{\mu}\right)^T\mathbf{C}^{-1}\left(\mathbf{x}-\boldsymbol{\mu}\right)\right].$$

noncentral_chisquare (ν, λ, size=None)

This is the distribution of $\sum_{i=1}^{\nu}\left(Z_i+\delta_i\right)^2$ where Z_i are independent standard normal random numbers and δ_i are constants. It is a a generalized Rayleigh-Rice distribution:

$$f\left(x;\nu,\lambda\right)=e^{-\left(\lambda+x\right)/2}\frac{1}{2}\left(\frac{x}{\lambda}\right)^{\left(\nu-2\right)/4}I_{\left(\nu-2\right)/2}\left(\sqrt{\lambda x}\right)I_{\left(0,\infty\right)}\left(x\right),$$

where $I_{\nu}\left(z\right)$ (a real-number in the subscript, not an interval) is the modified Bessel Function of the first kind.

noncentral_f (ν_1, ν_2, λ, size=None)

The pdf of this distribution is

$$\begin{aligned}f\left(x;\nu_1,\nu_2,\lambda\right) &= \exp\left[\frac{\lambda}{2}+\frac{\lambda v_1 x}{2\left(\nu_1 x+\nu_2\right)}\right]\nu_1^{\nu_1/2}\nu_2^{\nu_2/2}x^{\nu_1/2-1}\\ &\quad\times\left(\nu_2+\nu_1 x\right)^{-\left(\nu_1+\nu_2\right)/2}\\ &\quad\times\frac{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(1+\frac{\nu_2}{2}\right)L_{n_2/2}^{n_1/2-1}\left(-\frac{\lambda\nu_1 x}{2(\nu_1 x+\nu_2)}\right)}{B\left(\frac{\nu_1}{2},\frac{\nu_2}{2}\right)\Gamma\left(\frac{\nu_1+\nu_2}{2}\right)}.\end{aligned}$$

normal (loc=0.0, scale=1.0, size=None)

The normal, or Gaussian, distribution is the limiting distribution of independent samples from any sufficiently well-behaved distributions (this is the content of the celebrated central limit theorem). The normal distribution is also the distribution of maximum entropy when the mean and variance alone are fixed. These two facts account for its name as well as the wide variety of situations that can be usefully modelled using the normal distribution. Because it is so widely used, the full pdf with the location (μ) and scale (σ) parameters is provided:

$$f\left(x\right)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{\left(x-\mu\right)^2}{2\sigma^2}\right].$$

pareto (a, size=None)

$$f\left(x;a\right)=\frac{a}{x^{a+1}}I_{\left[1,\infty\right)}\left(x\right).$$

power (a, size=None)

A special case of the beta distribution with $b=1$.

$$f\left(x;a\right)=ax^{a-1}I_{\left[0,1\right]}\left(x\right).$$

rand (d_1,d_2, …,d_n)

A convenient interface to obtain an array of shape $(d_1, d_2, \ldots, d_n)$ of uniform random numbers in the interval $[0, 1)$. Notice the different convention for passing in the shape (as separate arguments instead of a tuple). The standard convention is used in the function numpy.random.random(shape) for which this function is merely a convenient short-hand. If you have a tuple named shape, then rand(*shape) will work correctly.

randint (low, high=None, size=None)

Equally probably random integers in the range $low \leq x < high$. If $high$ is None, then the range is $0 \leq x < low$. Similar to random_integers, but check the difference on the bounds.

randn (d_1,d_2, …,d_n)

A convenient interface to obtain an array of shape $(d_1, d_2, \ldots, d_n)$ of standard normal ($\mu = 0$, $\sigma = 1$) random numbers. Notice the different convention for passing in the shape (as separate arguments intead of a tuple). The standard convention is used in the function numpy.random.standard_normal(shape) for which this function is merely a convenient short-hand. If you have a tuple named shape, then randn(*shape) will work correctly.

random_integers (low, high=None, size=None)

Equally probably random integers in the range $low \leq x \leq high$. If high is None, then the range is $1 \leq x \leq low$. Similar to randint, but check the difference on the bounds.

rayleigh (scale=1.0, size=None)

Rayleigh-distributed random numbers can be obtained as $X = \sqrt{Z_1^2 + Z_2^2}$ where Z_i are independent standard normal random numbers. The scale parameter is also the mode of the distribution (the value of X with highest probability).

$$f(x) = xe^{-x^2/2} I_{[0,\infty)}(x)$$

standard_cauchy (size=None)

A Cauchy distribution is a heavy-tailed distribution with no variance. It's distribution is that of the ratio of two standard normal distributions Z_1/Z_2.

$$f(x) = \frac{1}{\pi(1+x^2)}.$$

standard_exponential (size=None)

A standard exponetial random number with scale=1.0. The pdf was given under the description of `random.exponential`.

standard_gamma (a, size=None)

A standard gamma random number with scale=1.0. The pdf was given under the description of `random.gamma`.

standard_normal (size=None)

A zero-mean, unit-variance, normally distributed random number often denoted Z.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

standard_t (ν, size=None)

Often called Student's t distribution, this random number distribution arises in the problem of estimating the mean of normally distributed samples when the sample-size is small. The first parameter, ν, is the number of degrees of freedom of the distribution.

$$f(x;\nu) \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi\nu}\Gamma\left(\frac{\nu}{2}\right)\left[1+\frac{x^2}{\nu}\right]^{\frac{\nu+1}{2}}}.$$

triangular (left, mode, right, size=None)

Returns random numbers according to a triangularly-shaped density that starts at left, peaks at mode, and ends at right.

uniform (low=0.0, high=1.0, size=None)

Returns random numbers that are equally probable over the range $[low, high)$.

vonmises (μ, κ, size=None)

A continuous distribution that is well suited for circular attributes such as angles, time of day, day of the year, etc. The mean direction is μ and concentration (or dispersion) parameter is κ. For small κ the distribution tends towards a uniform distribution over $[-\pi,\pi]$. For large κ, the distribution tends towards a normal distribution with mean μ and variance $1/\kappa$.

$$f(x)=\frac{e^{\kappa\cos(x-\mu)}}{2\pi I_0(\kappa)}I_{[-\pi,\pi]}(x).$$

wald (μ, λ, size=None)

This distribution is also called the inverse Gaussian distribution (and the Wald distribution considered as a special case when $\mu=\lambda$). It can be generated by noticing that if X is a wald random number then $\frac{\lambda(X-\mu)^2}{\mu^2 X}$ is the square of a standard normal random number (i.e. it is chi-square with one degree of freedom). The pdf is

$$f(x)=\sqrt{\frac{\lambda}{2\pi x^3}}e^{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}}.$$

weibull (a, size=None)

An extreme-value distribution:

$$f(x;c)=ax^{a-1}\exp(-x^a)\,I_{(0,\infty)}(x).$$

10.3.3 Miscellaneous utilities

bytes (length)

Return a string of random bytes of the provided length.

get_state ()

Return an object that holds the state of the random number generator (allows you to restart simulations where you left off).

set_state (state)

Set the state of the random number generator. The argument should be the returned object of a previous get_state command.

shuffle (sequence)

Randomly permute the items of any sequence. If sequence is an array, then it must be 1-d.

permutation (n)

Return a permutation of the integers from 0 to n-1.

10.4 Matrix-specific functions (matlib)

This module contains functions that are geared specifically toward matrix objects. In particular it includes the functions **empty**, **ones**, **zeros**, **identity**, **eye**, **rand**, and **randn** each of which returns a matrix object by default instead of an ndarray object.

10.5 Ctypes utiltity functions (ctypeslib)

This module contains utility functions that make it easier to work with the ctypes module.

load_library (name, path)

Load a shared library named "name" (use the full name including any prefix but excluding the extension) located in the directory indicated by path and return a ctypes library object whose attributes are the functions in the library. If ctypes is not available, this function will raise an ImportError. If there is an error loading the library, ctypes raises an OSError. The extension is appended to the library name (on a platform-dependent basis) unless the name includes the "." character in which case name is assumed to be the "full-name" of the library.

ndpointer (dtype=None, ndim=None, shape=None, flags=None)

Create a class object that can be used in the argtypes list of a ctypes function that will do basic type, number-of-dimensions, shape, and flags checking on input array objects. Setting an argtypes entry with the result of this function allows passing arrays directly to ctypes-wrapped functions. The returned class object will contain a from_param method as required by ctypes. This from_param method takes the array object, does data-type, number-of-dimensions, shape, and flags checking on the object and if all tests pass returns an object that ctypes can use as the data area of the array. Checking is not performed for any entries which are None in this class creation function.

Chapter 11

Testing and Packaging

There are two additional sub-packages distributed with NumPy that simplify the process of distributing and testing code based on NumPy. The numpy.distutils sub-package extends the standard distutils package to handle Fortran code along with providing support for the auto-generated code in NumPy. The numpy.testing sub-package defines a few functions and classes for standardizing unit-tests in NumPy. These facilities can be used in your own packages that build on top of NumPy.

11.1 Testing

In this sub-package are two classes and some useful utilities for writing unit-tests

NumpyTestCase a subclass of unittest.TestCase which adds a measure method that can determine the elasped time to execute a code string and enhances the __call__ method

NumpyTest the test manager for NumPy which was extracted originally from the SciPy code base. This test manager makes it easy to add unit-tests to a package simply by creating a tests sub-directory with files named test_<module>.py. These test files should then define sub-classes of NumpyTestCase (or unittest.TestCase) named "test*". These classes should then define functions named "test*" or "bench*" or "check*" that contain the actual unit-tests. The first keyword argument should specify the level above which this test should be run.

To run the tests excecute NumpyTest(<package>).test(level=1, verbosity=1) which will run all tests above the given level using the given verbosity.

Here <package> can be either a string or a previously imported module. You can get the level and verbosity arguments from sys.argv using NumpyTest(<package>).run() with -v or –verbosity and -l or –level as command-line arguments.

set_local_path (reldir='', level=1)

prepend local directory (+ reldir) to sys.path. The caller is responsible for removing this path using restore_path().

set_package_path (level=1)

prepend package directory to sys.path. This should be called from a test_file.py that satisfies the tree structure: <somepath>/<somedir>/test_file.py. The, the first existing path name from the list <somepath>/build/lib.<platform>-<version>, <somepath>/.. is pre-pended to sys.path. The caller is responsible for removing this path using restore_path().

restore_path ()

Remove the first entry from sys.path.

assert_equal (actual, desired, err_msg='', verbose=1)

Raise an assertion error if the two items are not equal. Automatically calls assert_array_equal if actual or desired is an ndarray.

assert_almost_equal (actual, desired, decimal=7, err_msg='', verbose=1)

Raise an assertion error if the two items are not equal within decimal places. Automatically calls assert_array_almost_equal if actual or desired is an ndarray.

assert_approx_equal (actual, desired, significant=7, err_msg='', verbose=1)

Raise an assertion error if the two items are not equal to within the given significant digits. Does not work on arrays.

assert_array_equal (x, y, err_msg='')

Raise an error if the two arrays x and y are not equal at every element.

assert_array_less (x, y, err_msg='')

Raise an error if the two arrays x and y have different shapes or if x is not less than y at every element.

assert_array_almost_equal (x, y, decimal=6, err_msg='')

Raise an error if x and y are not equal to decimal places at every element.

jiffies ()

Return a number of 1/100ths of a second that this process has been scheduled in user mode. Implemented using time.time() unless on Linux where the special /proc directory filesystem is used.

memusage ()

Return the virtual memory size in bytes of the running python. If the operation is not supported on the platform, then return None. This works only on linux for now.

rand (*args)

Return an array of random numbers with the given shape using only the standard library random number generator.

runstring (astr, dict)

Run the given string in the dictionary provided. Functional form for (exec astr in dict) that is useful for the failUnlessRaises method of unittest.TestCase class.

11.2 NumPy Distutils

NumPy provides enhanced distutils functionality to make it easier to build and install sub-packages, auto-generate code, and extension modules that use Fortran-compiled libraries. To use features of numpy distutils use the setup command from numpy.distutils.core. A useful Configuration class is also provided in numpy.distutils.misc_util that can make it easier to construct keyword arguments to pass to the setup function (by passing the dictionary obtained from the todict() method of the class). More information is available in the NumPy Distutils Users Guide in <site-packages>/numpy/doc/DISTUTILS.txt.

11.2.1 misc_util

Configuration (package_name=None, parent_name=None, top_path=None, package_path=None, **attrs)

Construct a configuration instance for the given package name. If parent_name is not None, then construct the package as a sub-package of the parent_name package. If top_path and package_path are None then they are assumed equal to the path of the file this instance was created in. The setup.py files in the numpy distribution are good examples of how to use the Configuration instance.

self.todict ()

Return a dictionary compatible with the keyword arguments of distutils setup function. Thus, this method may be used as setup(**config.todict()).

self.get_distribution ()

Return the distutils distribution object for self.

self.get_subpackage (subpackage_name, subpackage_path=None)

Return a Configuration instance for the sub-package given. If subpackage_path is None then the path is assumed to be the local path plus the subpackage_name. If a setup.py file is not found in the subpackage_path, then a default configuration is used.

self.add_subpackage (subpackage_name, subpackage_path=None)

Add a sub-package to the current Configuration instance. This is useful in a setup.py script for adding sub-packages to a package. The sub-package is contained in subpackage_path / subpackage_name and this directory may contain a setup.py script or else a default setup (suitable for Python-code-only subpackages) is assumed. If the subpackage_path is None, then it is assumed to be located in the local path / subpackage_name.

self.add_data_files (*files)

Add files to the list of data_files to be included with the package. The form of each element of the files sequence is very flexible allowing many combinations of where to get the files from the package and where they should ultimately be installed on the system. The most basic usage is for an element of the files argument sequence to be a simple filename. This will cause that file from the local path to be installed to the installation path of the self.name package (package path). The file argument can also be a relative path in which case the entire relative path will be installed into the package directory. Finally, the file can be an absolute path name in which case the file will be found at the absolute path name but installed to the package path.

This basic behavior can be augmented by passing a 2-tuple in as the file argument. The first element of the tuple should specify the relative path (under the package install directory) where the remaining sequence of files should be installed to (it has nothing to do with the file-names in the source distribution). The second element of the tuple is the sequence of files that should be installed. The files in this sequence can be filenames, relative paths, or absolute paths. For absolute paths the file will be installed in the top-level package installation directory (regardless of the first argument). Filenames and relative path names will be installed in the package install directory under the path name given as the first element of the tuple. An example may clarify:

```
self.add_data_files('foo.dat',
('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
'bar/cat.dat',
'/full/path/to/can.dat')
```

will install these data files to:

```
<package install directory>/
 foo.dat
 fun/
   gun.dat
   nun/
     pun.dat
 sun.dat
 bar/
   car.dat
 can.dat
```

where <package install directory> is the package (or sub-package) directory such as '/usr/lib/python2.4/site-packages/mypackage' ('C:\\Python2.4\\Lib\\site-packages\\mypackage') or '/usr/lib/python2.4/site-packages/mypackage/mysubpackage' ('C:\\Python2.4\\Lib\\site-packages\\mypackage\\mysubpackage').

An additional feature is that the path to a data-file can actually be a function that takes no arguments and returns the actual path(s) to the data-files. This is useful when the data files are generated while building the package.

self.add_data_dir (data_path)

Recursively add files under data_path to the list of data_files to be installed (and distributed). The data_path can be either a relative path-name, or an absolute path-name, or a 2-tuple where the first argument shows where in the install directory the data directory should be installed to. For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat

```
self.add_data_dir('fun')
self.add_data_dir(('sun', 'fun'))
self.add_data_dir(('gun', '/full/path/to/fun'))
```

Will install data-files to the locations

```
<package install directory>/
  fun/
    foo.dat
    bar/
      car.dat
  sun/
    foo.dat
    bar/
      car.dat
  gun/
    foo.dat
    car.dat
```

self.add_include_dirs (*paths)

Add the given sequence of paths to the beginning of the include_dirs list. This list will be visible to all extension modules of the current package.

self.add_headers (*files)

Add the given sequence of files to the beginning of the headers list. By default, headers will be installed under <python-include>/<self.name.replace('.','/')>/ directory. If an item of files is a tuple, then its first argument specifies the actual installation location relative to the <python-include> path.

self.add_extension (name, sources, **kw)

Create and add an Extension instance to the ext_modules list. The first argument defines the name of the extension module that will be installed under the self.name package. The second argument is a list of sources. This method also takes the following optional keyword arguments that are passed on to the Extension constructor: include_dirs, define_macros, undef_macros, library_dirs, libraries, runtime_library_dirs, extra_objects, swig_opts, depends, language, f2py_options, module_dirs, and extra_info.

The self.paths(...) method is applied to all lists that may contain paths. The extra_info is a dictionary or a list of dictionaries whose content will be appended to the keyword arguments. The depends list contains paths to files or directories that the sources of the extension module depend on. If any path in the depends list is newer than the extension module, then the module will be rebuilt.

The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

self.add_library (name, sources, **build_info)

Add a library to the list of libraries. Allowed keyword arguments are depends, macros, include_dirs, extra_compiler_args, and f2py_options. The name is the name of the library to be built and sources is a list of sources (or source generating functions) to add to the library.

self.add_scripts (*files)

Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/ directory.

self.paths (*paths)

Applies glob.glob(...) to each path in the sequence (if needed) and pre-pends the local_path if needed. Because this is called on all source lists, this allows wildcard characters to be specified in lists of sources for extension modules and libraries and scripts and allows path-names be relative to the source directory.

self.get_config_cmd ()

Returns the numpy.distutils config command instance.

self.get_build_temp_dir ()

Return a path to a temporary directory where temporary files should be placed.

self.have_f77c ()

True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).

self.have_f90c ()

True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

self.get_version ()

Return a version string of the current package or None if the version information could not be detected. This method scans files named __version__.py, <packagename>_version.py, version.py, and __svn_version__.py for string variables version, __version__, and <packagename>_version, until a version number is found.

self.make_svn_version_py ()

Appends a data function to the data_files list that will generate __svn_version__.py file to the current package directory. This file will be removed from the source directory when Python exits (so that it can be re-generated next time the package is built). This is intended for working with source directories that are in an SVN repository.

self.make_config_py ()

Generate a package __config__.py file containing system information used during the building of the package. This file is installed to the package installation directory.

self.get_info (*names)

Return information (from system_info.get_info) for all of the names in the argument list in a single dictionary.

get_numpy_include_dirs ()

Return the include directory where the numpy/arrayobject.h and numpy/ufuncobject.h files are found. This should be added to the include_dirs of any extension module built using NumPy. If numpy.distutils is used to build the extension, then this directory is added automatically.

get_numarray_include_dirs ()

Return the include directory where the numpy/libnumarray.h file is found. This should be added to the include_dirs of any extension module that relies on the Numarray-compatible C-API.

dict_append (d, **kwds)

Add the keyword arguments given as entries in the dictionary provided as the first argument. If the entry is already present, then assume it is a list and extend the list with the keyword value.

appendpath (prefix, path)

Platform-independent intelligence for appending path to prefix. It replaces '/' in the prefix and the path with the correct path-separator on the platform ad returns a full path name that will be valid for the platform.

allpath (name)

Convert a '/' separated pathname to one using the platform's path separator.

dot_join (*args)

Converts a sequence of string arguments to a string joined by '.' (removing any empty strings).

generate_config_py (extension, build_dir)

A suitable function that can be used in a source list. This constructs a python file that contains system_info information used during building the package. Generally easier to use a Configuration instance and the config.make_config_py() method.

get_cmd (cmdname, _cache={})

Returns an instance of the distutils command object named cmdname if the setup distribution instance has been initialized. Caches the result in _cache[cmdname] and gets it from there if present.

terminal_has_colors ()

Tries to determine if the stdout terminal can be written to using ANSI colors. Returns 1 if it can be determined that ANSI colors are acceptable or 0 if not.

red_text (s)

green_text (s)

yellow_text (s)

blue_text (s)

cyan_text (s)

If terminal_has_colors() is true, then these commands return a string with the necessary codes prepended to display the given string argument in the specified color on an ANSI terminal. If terminal_has_colors() is false, then these functions simply return the input argument.

cyg2win32 (path)

Convert a cygwin path beginning with /cygdrive to a standard win32 path name.

all_strings (lst)

Return True if all items in the input list are string objects otherwise return False.

has_f_sources (sources)

Return True if any of the source files listed in the input argument are Fortran files because its name matches against the compiled regular expression **fortran_ext_match**.

has_cxx_sources (sources)

Return True if any of the source files listed in the input argument are C++ files because its name matches against the compiled regular expression **cxx_ext_match**.

filter_sources (sources)

From the provided list of sources, return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources respectively. The compiled regular expressions used in this search (which are also available in the misc_util module) are cxx_ext_match, fortran_ext_match, f90_ext_match, and f90_module_name_match.

get_dependencies (sources)

Scan the files in the sources list for include statements.

is_local_src_dir (directory)

Return True if the provided directory is the local current working directory.

get_ext_source_files (ext)

Get sources and any include files in the same directory from an Extension instance.

get_script_files (scripts)

Returns the list scripts with all non-string arguments removed.

11.2.2 Other modules

system_info.get_info (name)

For the given string representing a particular resource, return a dictionary that is compatible with the distutils.setup keyword arguments. If this is an empty dictionary, then the requested resource is not available. Some of the names that can be checked are 'lapack_opt', 'blas_opt', 'fft_opt', 'fftw', 'fftw3', 'fftw2', 'djbfft', 'numpy', 'numarray', 'boost_python', 'agg2', 'wx', 'gdk', 'xft', 'freetype2'.

system_info.get_standard_file (filename)

Return a list of length 0 to 3 containing the full-path filenames for the filename provided. The filename is searched for in three places in the following order 1) the system-wide location which is the directory that the system_info file is located in; 2) the directory specified by the environment variable HOME; and 3) the current local directory.

cpuinfo.cpu an instance of a cpuinfo class that defines methods for checking various aspects of the cpu. The info attribute is a list of length (# of CPUs). Each entry is a dictionary providing technical information about that CPU.

log.set_verbosity (level)

Set the distutils logging threshold and return the previously stored value. The level is an integer that corresponds to distutils.log thresholds: -1 <–> ERROR, 0 <–> WARN, 1 <–> INFO, and 2 <–> DEBUG.

exec_command

exec_command (command, execute_in=", use_shell=None, use_tee=None, _with_python=1, **env)

Return (status, output) of the executed command. The command input is a string of executable and arguments. The output contains both stderr and stdout messages. If execute_in is given, then change to the provided directory prior to executing the command and afterwords restore to the current directory. On NT, and DOS systems the returned status is correct for external commands. However, wild cards will not work for non-posix systems.

splitcmdline (line)

Inverse of ' '.join(sys.argv)

find_executable (exe, path=None)

Return full path of an executable using information from the PATH environment variable. Equivalent to the POSIX 'which' command.

get_pythonexe ()

Return the full path to the python executable with some fixes for nt and dos to replace pythonw with python if it is encountered. A basic wrapper around sys.executable.

11.3 Conversion of .src files

NumPy distutils supports automatic conversion of source files named <somefile>.src. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named <somefile>.src is encountered, a new file named <somefile> is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named named <file>.ext.src where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases.

11.3.1 Fortran files

This template converter will replicate all **function** and **subroutine** blocks in the file with names that contain '<...>' according to the rules in '<...>'. The number of comma-separated words in '<...>' determines the number of times the block is repeated. What these words are indicates what that repeat rule, '<...>', should be replaced with in each block. All of the repeat rules in a block must contain the same number of comma-separated words indicating the number of times that block should be repeated. If the word in the repeat rule needs a comma, leftarrow, or rightarrow, then prepend it with a backslash '\'. If a word in the repeat rule matches '\\<index>' then it will be replaced with the <index>-th word in the same repeat specification. There are two forms for the repeat rule: named and short.

11.3.1.1 Named repeat rule

A named repeat rule is useful when the same set of repeats must be used several times in a block. It is specified using <rule1=item1, item2, item3,..., itemN>, where N is the number of times the block should be repeated. On each repeat of the block, the entire expression, '<...>' will be replaced first with item1, and then with item2, and so forth until N repeats are accomplished. Once a named repeat specification has been introduced, the same repeat rule may be used **in the current block** by referring only to the name (i.e. <rule1>.

11.3.1.2 Short repeat rule

A short repeat rule looks like <item1, item2, item3, ..., itemN>. The rule specifies that the entire expression, '<...>' should be replaced first with item1, and then with item2, and so forth until N repeats are accomplished.

11.3.1.3 Pre-defined names

The following predefined named repeat rules are available:

- <prefix=s,d,c,z>
- <_c=s,d,c,z>
- <_t=real, double precision, complex, double complex>
- <ftype=real, double precision, complex, double complex>
- <ctype=float, double, complex_float, complex_double>
- <ftypereal=float, double precision, \\0, \\1>
- <ctypereal=float, double, \\0, \\1>

11.3.2 Other files

Non-Fortran files use a separate syntax for defining template blocks that should be repeated using a variable expansion similar to the named repeat rules of the Fortran-specific repeats. The template rules for these files are:

1. "/**begin repeat" on a line by itself marks the beginning of a segment that should be repeated.

2. Named variable expansions are defined using #name=item1, item2, item3, ..., itemN# and placed on successive lines. These variables are replaced in each repeat block with corresponding word. All named variables in the same repeat block must define the same number of words.

3. In specifying the repeat rule for a named variable, item*N is short-hand for item, item, ..., item repeated N times. In addition, parenthesis in combination with *N can be used for grouping several items that should be repeated. Thus, #name=(item1, item2)*4# is equivalent to #name=item1, item2, item1, item2, item1, item2, item1, item2#

4. "*/" on a line by itself marks the end of the the variable expansion naming. The next line is the first line that will be repeated using the named rules.

5. Inside the block to be repeated, the variables that should be expanded are specified as @name@.

6. "/**end repeat**/" on a line by itself marks the previous line as the last line of the block to be repeated.

Part II

C-API

Chapter 12

New Python Types and C-Structures

NumPy provides a C-API to enable users to extend the system and get access to the array object for use in other routines. The best way to truly understand the C-API is to read the source code. If you are unfamiliar with (C) source code, however, this can be a daunting experience at first. Be assured that the task becomes easier with practice, and you may be surprised at how simple the C-code can be to understand. Even if you don't think you can write C-code from scratch, it is much easier to understand and modify already-written source code then create it *de novo*.

Python extensions are especially straightforward to understand because they all have a very similar structure. Admittedly, NumPy is not a trivial extension to Python, and may take a little more snooping to grasp. This is especially true because of the code-generation techniques, which simplify maintenance of very similar code, but can make the code a little less readable to beginners. Still, with a little persistence, the code can be opened to your understanding. It is my hope, that this guide to the C-API can assist in the process of becoming familiar with the compiled-level work that can be done with NumPy in order to squeeze that last bit of necessary speed out of your code.

Several new types are defined in the C-code. Most of these are accessible from Python, but a few are not exposed due to their limited use. Every new Python type has an associated PyObject * with an internal structure that includes a pointer to a "method table" that defines how the new object behaves in Python. When you receive a Python object into C code, you always get a pointer to a

`PyObject` structure. Because a `PyObject` structure is very generic and defines only `PyObject_HEAD`, by itself it is not very interesting. However, different objects contain more details after the `PyObject_HEAD` (but you have to cast to the correct type to access them — or use accessor functions or macros).

12.1 New Python Types Defined

Python types are the functional equivalent in C of classes in Python. By constructing a new Python type you make available a new object for Python. The ndarray object is an example of a new type defined in C. New types are defined in C by two basic steps:

1. creating a C-structure (usually named Py<Name>Object) that is binary-compatible with the `PyObject` structure itself but holds the additional information needed for that particular object;

2. populating the `PyTypeObject` table (pointed to by the ob_type member of the `PyObject` structure) with pointers to functions that implement the desired behavior for the type.

Instead of special method names which define behavior for Python classes, there are "function tables" which point to functions that implement the desired results. Since Python 2.2, the PyTypeObject itself has become dynamic which allows C types that can be "sub-typed" from other C-types in C, and sub-classed in Python. The children types inherit the attributes and methods from their parent(s).

There are two major new types: the ndarray (`PyArray_Type`) and the ufunc (`PyUFunc_Type`). Additional types play a supportive role: the `PyArrayIter_Type`, the `PyArrayMultiIter_Type`, and the `PyArrayDescr_Type`. The `PyArrayIter_Type` is the type for a flat iterator for an ndarray (the object that is returned when getting the flat attribute). The `PyArrayMultiIter_Type` is the type of the object returned when calling `broadcast`(). It handles iteration and broadcasting over a collection of nested sequences. Also, the `PyArrayDescr_Type` is the data-type-descriptor type whose instances describe the data. Finally, there are 21 new scalar-array types which are new Python scalars corresponding to each of the fundamental data types available for arrays. An additional 10 other types are place holders that allow the array scalars to fit into a hierarchy of actual Python types.

12.1.1 PyArray_Type

The Python type of the ndarray is `PyArray_Type`. In C, every ndarray is a pointer to a `PyArrayObject` structure. The ob_type member of this structure contains a pointer to the `PyArray_Type` typeobject.

The `PyArrayObject` C-structure contains all of the required information for an array. All instances of an ndarray (and its subclasses) will have this structure. For future compatibility, these structure members should normally be accessed using the provided macros. If you need a shorter name, then you can make use of `NPY_AO` which is defined to be equivalent to `PyArrayObject`.

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;
```

PyObject_HEAD This is needed by all Python objects. It consists of (at least) a reference count member (`ob_refcnt`) and a pointer to the typeobject (`ob_type`). (Other elements may also be present if Python was compiled with special options see Include/object.h in the Python source tree for more information). The ob_type member points to a Python type object.

data A pointer to the first element of the array. This pointer can (and normally should) be recast to the data type of the array.

nd An integer providing the number of dimensions for this array. When nd is 0, the array is sometimes called a rank-0 array. Such arrays have undefined dimensions and strides and cannot be accessed. `NPY_MAXDIMS` is the largest number of dimensions for any array.

dimensions An array of integers providing the shape in each dimension as long as nd$\geq$1. The integer is always large enough to hold a pointer on the platform, so the dimension size is only limited by memory.

strides An array of integers providing for each dimension the number of bytes that must be skipped to get to the next element in that dimension.

base This member is used to hold a pointer to another Python object that is related to this array. There are two use cases: 1) If this array does not own its own memory, then base points to the Python object that owns it (perhaps another array object), 2) If this array has the `NPY_UPDATEIFCOPY` flag set, then this array is a working copy of a "misbehaved" array. As soon as this array is deleted, the array pointed to by base will be updated with the contents of this array.

descr A pointer to a data-type descriptor object (see below). The data-type descriptor object is an instance of a new built-in type which allows a generic description of memory. There is a descriptor structure for each data type supported. This descriptor structure contains useful information about the type as well as a pointer to a table of function pointers to implement specific functionality.

flags Flags indicating how the memory pointed to by data is to be interpreted. Possible flags are `NPY_C_CONTIGUOUS`, `NPY_F_CONTIGUOUS`, `NPY_OWNDATA`, `NPY_ALIGNED`, `NPY_WRITEABLE`, and `NPY_UPDATEIFCOPY`.

weakreflist This member allows array objects to have weak references (using the weakref module).

12.1.2 PyArrayDescr_Type

The `PyArrayDescr_Type` is the built-in type of the data-type-descriptor objects used to describe how the bytes comprising the array are to be interpreted. There are 21 statically-defined `PyArray_Descr` objects for the built-in data-types. While these participate in reference counting, their reference count should never reach zero. There is also a dynamic table of user-defined `PyArray_Descr` objects that is also maintained. Once a data-type-descriptor object is "registered" it should never be deallocated either. The function `PyArray_DescrFromType`(...) can be used to retrieve a `PyArray_Descr` object from an enumerated type-number (either built-in or user-defined). The format of the structure that lies at the heart of the `PyArrayDescr_Type` is.

```
typedef struct {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char hasobject;
    int type_num;
    int elsize;
    int alignment;
    PyArray_ArrayDescr *subarray;
    PyObject *fields;
    PyArray_ArrFuncs *f;
} PyArray_Descr;
```

typeobj Pointer to a typeobject that is the corresponding Python type for the elements of this array. For the builtin types, this points to the corresponding array scalar. For user-defined types, this should point to a user-defined typeobject. This typeobject can either inherit from array scalars or not. If it does not inherit from array scalars, then the `NPY_USE_GETITEM` and `NPY_USE_SETITEM` flags should be set in the `hasobject` flag.

kind A character code indicating the kind of array (using the array interface typestring notation). A 'b' represents Boolean, a 'i' represents signed integer, a 'u' represents unsigned integer, 'f' represents floating point, 'c' represents complex floating point, 'S' represents 8-bit character string, 'U' represents 32-bit/character unicode string, and 'V' repesents arbitrary.

type A traditional character code indicating the data type.

byteorder A character indicating the byte-order: '>' (big-endian), '<' (little-endian), '=' (native), '|' (irrelevant, ignore). All builtin data-types have byteorder '='.

hasobject A data-type bit-flag that determines if the data-type exhibits object-array like behavior. Each bit in this member is a flag which are named as:

NPY_ITEM_REFCOUNT (NPY_ITEM_HASOBJECT) Indicates that items of this data-type must be reference counted (using `Py_INCREF` and `Py_DECREF`).

NPY_ITEM_LISTPICKLE Indicates arrays of this data-type must be converted to a list before pickling.

NPY_ITEM_IS_POINTER Indicates the item is a pointer to some other data-type

NPY_NEEDS_INIT Indicates memory for this data-type must be initialized (set to 0) on creation.

NPY_NEEDS_PYAPI Indicates this data-type requires the Python C-API during access (so don't give up the GIL if array access is going to be needed).

NPY_USE_GETITEM On array access use the `f->getitem` function pointer instead of the standard conversion to an array scalar. Must use if you don't define an array scalar to go along with the data-type.

NPY_USE_SETITEM When creating a 0-d array from an array scalar use `f->setitem` instead of the standard copy from an array scalar. Must use if you don't define an array scalar to go along with the data-type.

NPY_FROM_FIELDS The bits that are inherited for the parent data-type if these bits are set in any field of the data-type. Currently (`NPY_NEEDS_INIT` | `NPY_LIST_PICKLE` | `NPY_ITEM_REFCOUNT` | `NPY_NEEDS_PYAPI`).

NPY_OBJECT_DTYPE_FLAGS Bits set for the object data-type: (`NPY_LIST_PICKLE` | `NPY_USE_GETITEM` | `NPY_ITEM_IS_POINTER` | `NPY_REFCOUNT` | `NPY_NEEDS_INIT` | `NPY_NEEDS_PYAPI`).

PyDataType_FLAGCHK (`PyArray_Descr*` dtype, `int` flags) Return true if all the given flags are set for the data-type object.

PyDataType_REFCHK (`PyArray_Descr*` dtype) Equivalent to `PyDataType_FLAGCHK(`*dtype*, `NPY_ITEM_REFCOUNT)`.

type_num A number that uniquely identifies the data type. For new data-types, this number is assigned when the data-type is registered.

elsize For data types that are always the same size (such as long), this holds the size of the data type. For flexible data types where different arrays can have a different elementsize, this should be 0.

alignment A number providing alignment information for this data type. Specifically, it shows how far from the start of a 2-element structure (whose first element is a `char`), the compiler places an item of this type: `offsetof(struct {char c; type v;}, v)`

subarray If this is non-NULL, then this data-type descriptor is a C-style contiguous array of another data-type descriptor. In other-words, each element that this descriptor describes is actually an array of some other base descriptor. This is most useful as the data-type descriptor for a field in another data-type descriptor. The fields member should be NULL if this is non-NULL (the fields member of the base descriptor can be non-NULL however). The PyArray_ArrayDescr structure is defined using

```
typedef struct {
    PyArray_Descr *base;
    PyObject *shape;
} PyArray_ArrayDescr;
```

The elements of this structure are:

base The data-type-descriptor object of the base-type.

shape The shape (always C-style contiguous) of the sub-array as a Python tuple.

fields If this is non-NULL, then this data-type-descriptor has fields described by a Python dictionary whose keys are names (and also titles if given) and whose values are tuples that describe the fields. Recall that a data-type-descriptor always describes a fixed-length set of bytes. A field is a named sub-region of that total, fixed-length collection. A field is described by a tuple composed of another data-type-descriptor and a byte offset. Optionally, the tuple may contain a title which is normally a Python string. These tuples are placed in this dictionary keyed by name (and also title if given).

f A pointer to a structure containing functions that the type needs to implement internal features. These functions are not the same thing as the universal functions (ufuncs) described later. Their signatures can vary arbitrarily. Not all of these function pointers must be defined for a given type. The required members are `nonzero`, `copyswap`, `copyswapn`, `setitem`, `getitem`, and `cast`. These are assumed to be non-NULL and NULL entries will cause a program crash. The other functions may be NULL which will just mean reduced functionality for that data-type. (Also, the nonzero function will be filled in with a default function if it is NULL when you register a user-defined data-type).

```
typedef struct {
    PyArray_VectorUnaryFunc *cast[PyArray_NTYPES];
    PyArray_GetItemFunc *getitem;
    PyArray_SetItemFunc *setitem;
    PyArray_CopySwapNFunc *copyswapn;
    PyArray_CopySwapFunc *copyswap;
    PyArray_CompareFunc *compare;
    PyArray_ArgFunc *argmax;
    PyArray_DotFunc *dotfunc;
    PyArray_ScanFunc *scanfunc;
    PyArray_FromStrFunc *fromstr;
    PyArray_NonzeroFunc *nonzero;
    PyArray_FillFunc *fill;
    PyArray_FillWithScalarFunc *fillwithscalar;
    PyArray_SortFunc *sort[PyArray_NSORTS];
    PyArray_ArgSortFunc *argsort[PyArray_NSORTS];
    PyObject *castdict;
    PyArray_ScalarKindFunc *scalarkind;
    int **cancastscalarkindto;
    int *cancastto;
    int listpickle
} PyArray_ArrFuncs;
```

The concept of a behaved segment is used in the description of the function pointers. A behaved segment is one that is aligned and in native machine byte-order for the data-type. The `nonzero`, `copyswap`, `copyswapn`, `getitem`, and `setitem` functions can (and must) deal with mis-behaved arrays. The other functions require behaved memory segments.

cast (`void`) (`void*` from, `void*` to, `npy_intp` n, `void*` fromarr, `void*` toarr)

An array of function pointers to cast from the current type to all of the other builtin types. Each function casts a contiguous, aligned, and notswapped buffer pointed at by *from* to a contiguous, aligned, and notswapped buffer pointed at by *to* The number of items to cast is given by *n*, and the arguments *fromarr* and *toarr* are interpreted as PyArrayObjects for flexible arrays to get itemsize information.

getitem (`PyObject*`) (`void*` data, `void*` arr)

A pointer to a function that returns a standard Python object from a single element of the array object *arr* pointed to by *data*. This function must be able to deal with “misbehaved” (misaligned and/or swapped) arrays correctly.

setitem (`int`) (`PyObject*` item, `void*` data, `void*` arr)

A pointer to a function that sets the Python object *item* into the array, *arr*, at the position pointed to by *data*. This function deals with “misbehaved” arrays. If successful, a zero is returned, otherwise, a negative one is returned (and a Python error set).

copyswapn (`void`) (`void*` dest, `npy_intp` dstride, `void*` src, `npy_intp` sstride, `npy_intp` n, `int` swap, void *arr)

copyswap (`void`) (`void*` dest, `void*` src, `int` swap, void *arr)

These members are both pointers to functions to copy data from *src* to *dest* and *swap* if indicated. The value of arr is only used for flexible (`NPY_STRING`, `NPY_UNICODE`, and `NPY_VOID`) arrays (and is obtained from `arr->descr->elsize`). The second function copies a single value, while the first loops over n values with the provided strides. These functions can deal with misbehaved *src* data. If *src* is NULL then no copy is performed. If *swap* is 0, then no byteswapping occurs. It is assumed that *dest* and *src* do not overlap. If they overlap, then use `memmove`(...) first followed by `copyswap(n)` with NULL valued `src`.

compare (`int`) (`const void*` d1, `const void*` d2, `void*` arr)

A pointer to a function that compares two elements of the array, `arr`, pointed to by `d1` and `d2`. This function requires behaved arrays. The return value is 1 if *`d1` > *`d2`, 0 if *`d1` == *`d2`, and -1 if *`d1` < *`d2`. The array object arr is used to retrieve itemsize and field information for flexible arrays.

argmax (`int`) (`void*` data, `npy_intp` n, `npy_intp*` max_ind, `void*` arr)

A pointer to a function that retrieves the index of the largest of `n` elements in `arr` beginning at the element pointed to by `data`. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the largest element is returned in `max_ind`.

dotfunc (`void`) (`void*` ip1, `npy_intp` is1, `void*` ip2, `npy_intp` is2, `void*` op, `npy_intp` n, `void*` arr)

A pointer to a function that multiplies two `n`-length sequences together, adds them, and places the result in element pointed to by `op` of `arr`. The start of the two sequences are pointed to by `ip1` and `ip2`. To get to the next element in each sequence requires a jump of `is1` and `is2` *bytes*, respectively. This function requires behaved (though not necessarily contiguous) memory.

scanfunc (`int`) (`FILE*` fd, `void*` ip ,`void*` sep ,`void*` arr)

A pointer to a function that scans (scanf style) one element of the corresponding type from the file descriptor `fd` into the array memory pointed to by `ip`. The array is assumed to be behaved. If `sep` is not NULL, then a separator string is also scanned from the file before returning. The last argument `arr` is the array to be scanned into. A 0 is returned if the scan is successful. A negative number indicates something went wrong: -1 means the end of file was reached before the separator string could be scanned, -4 means that the end of file was reached before the element could be scanned, and -3 means that the element could not be interpreted from the format string. Requires a behaved array.

fromstr (`int`) (`char*` str, `void*` ip, `char**` endptr, `void*` arr)

A pointer to a function that converts the string pointed to by `str` to one element of the corresponding type and places it in the memory location pointed to by `ip`. After the conversion is completed, `*endptr` points to the rest of the string. The last argument `arr` is the array into which ip points (needed for variable-size data-types). Returns 0 on success or -1 on failure. Requires a behaved array.

nonzero (`Bool`) (`void*` data, `void*` arr)

A pointer to a function that returns TRUE if the item of `arr` pointed to by `data` is nonzero. This function can deal with misbehaved arrays.

fill (`void`) (`void*` data, `npy_intp` length, `void*` arr)

A pointer to a function that fills a contiguous array of given length with data. The first two elements of the array must already be filled-in. From these two values, a delta will be computed and the values from item 3 to the end will be computed by repeatedly adding this computed delta. The data buffer must be well-behaved.

fillwithscalar (`void`)(`void*` buffer, `npy_intp` length, `void*` value, `void*` arr)

A pointer to a function that fills a contiguous `buffer` of the given `length` with a single scalar `value` whose address is given. The final argument is the array which is needed to get the itemsize for variable-length arrays.

sort (`int`) (`void*` start, `npy_intp` length, `void*` arr)

An array of function pointers to a particular sorting algorithms. A particular sorting algorithm is obtained using a key (so far `PyArray_QUICKSORT`, `PyArray_HEAPSORT`, and `PyArray_MERGESORT` are defined). These sorts are done in-place assuming contiguous and aligned data.

argsort (`int`) (`void*` start, `npy_intp*` result, `npy_intp` length, void *arr)

An array of function pointers to sorting algorithms for this data type. The same sorting algorithms as for sort are available. The indices producing the sort are returned in result (which must be initialized with indices 0 to length-1 inclusive).

castdict

Either `NULL` or a dictionary containing low-level casting functions for user-defined data-types. Each function is wrapped in a `PyCObject*` and keyed by the data-type number.

scalarkind (`PyArray_SCALARKIND`) (`PyArrayObject*` arr)

A function to determine how scalars of this type should be interpreted. The argument is `NULL` or a 0-dimensional array containing the data (if that is needed to determine the kind of scalar). The return value must be of type `PyArray_SCALARKIND`.

cancastscalarkindto

Either `NULL` or an array of `PyArray_NSCALARKINDS` pointers. These pointers should each be either `NULL` or a pointer to an array of integers (terminated by `PyArray_NOTYPE`) indicating data-types that a scalar of this data-type of the specified kind can be cast to safely (this usually means without losing precision).

cancastto

Either `NULL` or an array of integers (terminated by `PyArray_NOTYPE`) indicated data-types that this data-type can be cast to safely (this usually means without losing precision).

listpickle

Unused.

The `PyArray_Type` typeobject implements many of the features of Python objects including the tp_as_number, tp_as_sequence, tp_as_mapping, and tp_as_buffer interfaces. The rich comparison (tp_richcompare) is also used along with new-style attribute lookup for methods (tp_methods) and properties (tp_getset). The `PyArray_Type` can also be sub-typed.

TIP

The tp_as_number methods use a generic approach to call whatever function has been registered for handling the operation. The function PyNumeric_SetOps(..) can be used to register functions to handle particular mathematical operations (for all arrays). When the umath module is imported, it sets the numeric operations for all arrays to the corresponding ufuncs.
The tp_str and tp_repr methods can also be altered using PyString_SetStringFunction(...).

12.1.3 PyUFunc_Type

The ufunc object is implemented by creation of the `PyUFunc_Type`. It is a very simple type that implements only basic getattribute behavior, printing behavior, and has call behavior which allows these objects to act like functions. The basic idea behind the ufunc is to hold a reference to fast 1-dimensional (vector) loops for each data type that supports the operation. These one-dimensional loops all have the same signature and are the key to creating a new ufunc. They are called by the generic looping code as appropriate to implement the N-dimensional function. There are also some generic 1-d loops defined for floating and complexfloating arrays that allow you to define a ufunc using a single scalar function (*e.g.* atanh).

The core of the ufunc is the `PyUFuncObject` which contains all the information needed to call the underlying C-code loops that perform the actual work. It has the following structure.

```
typedef struct {
    PyObject_HEAD
    int nin;
    int nout;
    int nargs;
```

```
        int identity;
        PyUFuncGenericFunction *functions;
        void **data;
        int ntypes;
        int check_return;
        char *name;
        char *types;
        char *doc;
        void *ptr;
        PyObject *obj;
        PyObject *userloops;
    } PyUFuncObject;
```

PyObject_HEAD required for all Python objects.

nin The number of input arguments.

nout The number of output arguments.

nargs The total number of arguments (*nin*+*nout*). This must be less than `NPY_MAXARGS`.

identity Either `PyUFunc_One`, `PyUFunc_Zero`, or `PyUFunc_None` to indicate the identity for this operation. It is only used for a reduce-like call on an empty array.

functions (`void`) (`char**` args, `npy_intp*` dims, `npy_intp*` steps, `void*` extradata)

An array of function pointers — one for each data type supported by the ufunc. This is the vector loop that is called to implement the underlying function *dims*[0] times. The first argument, *args*, is an array of *nargs* pointers to behaved memory. Pointers to the data for the input arguments are first, followed by the pointers to the data for the output arguments. How many bytes must be skipped to get to the next element in the sequence is specified by the corresponding entry in the *steps* array. The last argument allows the loop to receive extra information. This is commonly used so that a single, generic vector loop can be used for multiple functions. In this case, the actual scalar function to call is passed in as *extradata*. The size of this function pointer array is ntypes.

data Extra data to be passed to the 1-d vector loops or `NULL` if no extra-data is needed. This C-array must be the same size (*i.e.* ntypes) as the functions array. `NULL` is used if extra_data is not needed. Several C-API calls for UFuncs are just 1-d vector loops that make use of this extra data to receive a pointer to the actual function to call.

ntypes The number of supported data types for the ufunc. This number specifies how many different 1-d loops (of the builtin data types) are available.

check_return Obsolete and unused. However, it is set by the corresponding entry in the main ufunc creation routine: `PyUFunc_FromFuncAndData`(...).

name A string name for the ufunc. This is used dynamically to build the __doc__ attribute of ufuncs.

types An array of *nargs*×*ntypes* 8-bit type_numbers which contains the type signature for the function for each of the supported (builtin) data types. For each of the *ntypes* functions, the corresponding set of type numbers in this array shows how the *args* argument should be interpreted in the 1-d vector loop. These type numbers do not have to be the same type and mixed-type ufuncs are supported.

doc Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when __doc__ is retrieved.

ptr Any dynamically allocated memory. Currently, this is used for dynamic ufuncs created from a python function to store room for the types, data, and name members.

obj For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.

userloops A dictionary of user-defined 1-d vector loops (stored as CObject ptrs) for user-defined types. A loop may be registered by the user for any user-defined type. It is retrieved by type number. User defined type numbers are always larger than `NPY_USERDEF`.

12.1.4 PyArrayIter_Type

This is an iterator object that makes it easy to loop over an N-dimensional array. It is the object returned from the flat attribute of an ndarray. It is also used extensively throughout the implementation internals to loop over an N-dimensional array. The tp_as_mapping interface is implemented so that the iterator object can be indexed (using 1-d indexing), and a few methods are implemented through the tp_methods table. This object implements the next method and can be used anywhere an iterator can be used in Python.

The C-structure corresponding to an object of `PyArrayIter_Type` is the `PyArrayIterObject`. The `PyArrayIterObject` is used to keep track of a pointer into an N-dimensional array. It contains associated information used to quickly march through the array. The pointer can be adjusted in three basic ways: 1) advance to the "next" position in the array in a C-style contiguous fashion, 2) advance to an arbitrary N-dimensional coordinate in the array, and 3) advance to an arbitrary one-dimensional index into the array. The members of the `PyArrayIterObject` structure are used in these calculations. Iterator objects keep their own dimension and strides information about an array. This can be adjusted as needed for "broadcasting," or to loop over only specific dimensions.

```
typedef struct {
    PyObject_HEAD
    int   nd_m1;
    npy_intp  index;
    npy_intp  size;
    npy_intp  coordinates[NPY_MAXDIMS];
    npy_intp  dims_m1[NPY_MAXDIMS];
    npy_intp  strides[NPY_MAXDIMS];
    npy_intp  backstrides[NPY_MAXDIMS];
    npy_intp  factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char  *dataptr;
    Bool  contiguous;
} PyArrayIterObject;
```

nd_m1 $N-1$ where N is the number of dimensions in the underlying array.

index The current 1-d index into the array.

size The total size of the underlying array.

coordinates An N-dimensional index into the array.

dims_m1 The size of the array minus 1 in each dimension.

strides The strides of the array. How many bytes needed to jump to the next element in each dimension.

backstrides How many bytes needed to jump from the end of a dimension back to its beginning. Note that *backstrides*[k]=*strides*[k]*d*ims_m1*[k], but it is stored here as an optimization.

factors This array is used in computing an N-d index from a 1-d index. It contains needed products of the dimensions.

ao A pointer to the underlying ndarray this iterator was created to represent.

dataptr This member points to an element in the ndarray indicated by the index.

contiguous This flag is true if the underlying array is `NPY_C_CONTIGUOUS`. It is used to simplify calculations when possible.

How to use an array iterator on a C-level is explained more fully in later sections. Typically, you do not need to concern yourself with the internal structure of the iterator object, and merely interact with it through the use of the macros `PyArray_ITER_NEXT`(it), `PyArray_ITER_GOTO`(it, dest), or `PyArray_ITER_GOTO1D`(it, index). All of these macros require the argument *it* to be a `PyArrayIterObject*`.

12.1.5 PyArrayMultiIter_Type

This type provides an iterator that encapsulates the concept of broadcasting. It allows N arrays to be broadcast together so that the loop progresses in C-style contiguous fashion over the broadcasted array. The corresponding C-structure is the `PyArrayMultiIterObject` whose memory layout must begin any object, *obj*, passed in to the `PyArray_Broadcast`(obj) function. Broadcasting is performed by adjusting array iterators so that each iterator represents the broadcasted shape and size, but has its strides adjusted so that the correct element from the array is used at each iteration.

```
typedef struct {
    PyObject_HEAD
    int numiter;
    npy_intp size;
    npy_intp index;
    int nd;
```

```
        npy_intp dimensions[NPY_MAXDIMS];
        PyArrayIterObject *iters[NPY_MAXDIMS];
    } PyArrayMultiIterObject;
```

PyObject_HEAD Needed at the start of every Python object (holds reference count and type identification).

numiter The number of arrays that need to be broadcast to the same shape.

size The total broadcasted size.

index The current (1-d) index into the broadcasted result.

nd The number of dimensions in the broadcasted result.

dimensions The shape of the broadcasted result (only `nd` slots are used).

iters An array of iterator objects that holds the iterators for the arrays to be broadcast together. On return, the iterators are adjusted for broadcasting.

12.1.6 PyArrayFlags_Type

When the flags attribute is retrieved from Python, a special builtin object of this type is constructed. This special type makes it easier to work with the different flags by accessing them as attributes or by accessing them as if the object were a dictionary with the flag names as entries.

12.1.7 ScalarArrayTypes

There is a Python type for each of the different built-in data types that can be present in the array Most of these are simple wrappers around the corresponding data type in C. The C-names for these types are **Py**<TYPE>**ArrType_Type** where <TYPE> can be

> **Bool**, **Byte**, **Short**, **Int**, **Long**, **LongLong**, **UByte**, **UShort**, **UInt**, **ULong**, **ULongLong**, **Float**, **Double**, **LongDouble**, **CFloat**, **CDouble**, **CLongDouble**, **String**, **Unicode**, **Void**, and **Object**.

These type names are part of the C-API and can therefore be created in extension C-code. There is also a `PyIntpArrType_Type` and a `PyUIntpArrType_Type` that are simple substitutes for one of the integer types that can hold a pointer on the platform. The structure of these scalar objects is not exposed to C-code.

The function PyArray_ScalarAsCtype(..) can be used to extract the C-type value from the array scalar and the function PyArray_Scalar(...) can be used to construct an array scalar from a C-value.

12.2 Other C-Structures

A few new C-structures were found to be useful in the development of NumPy. These C-structures are used in at least one C-API call and are therefore documented here. The main reason these structures were defined is to make it easy to use the Python ParseTuple C-API to convert from Python objects to a useful C-Object.

12.2.1 PyArray_Dims

This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is

```
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```

The members of this structure are

ptr A pointer to a list of (npy_intp) integers which usually represent array shape or array strides.

len The length of the list of integers. It is assumed safe to access *ptr*[0] to *ptr*[len-1].

12.2.2 PyArray_Chunk

This is equivalent to the buffer object structure in Python up to the ptr member. On 32-bit platforms (*i.e.* if NPY_SIZEOF_INT==NPY_SIZEOF_INTP) or in Python 2.5, the len member also matches an equivalent member of the buffer object. It is useful to represent a generic single-segment chunk of memory.

```
typedef struct {
    PyObject_HEAD
    PyObject *base;
    void *ptr;
    npy_intp len;
```

```
        int flags;
    } PyArray_Chunk;
```

The members are

PyObject_HEAD Necessary for all Python objects. Included here so that the `PyArray_Chunk` structure matches that of the buffer object (at least to the len member).

base The Python object this chunk of memory comes from. Needed so that memory can be accounted for properly.

ptr A pointer to the start of the single-segment chunk of memory.

len The length of the segment in bytes.

flags Any data flags (*e.g.* `NPY_WRITEABLE`) that should be used to interpret the memory.

12.2.3 PyArrayInterface

The `PyArrayInterface` structure is defined so that NumPy and other extension modules can use the rapid array interface protocol. The **__array_struct__** method of an object that supports the rapid array interface protocol should return a `PyCObject` that contains a pointer to a `PyArrayInterface` structure with the relevant details of the array. After the new array is created, the attribute should be `DECREF`'d which will free the `PyArrayInterface` structure. Remember to `INCREF` the object (whose **__array_struct__** attribute was retrieved) and point the base member of the new `PyArrayObject` to this same object. In this way the memory for the array will be managed correctly.

```
    typedef struct {
        int two;
        int nd;
        char typekind;
        int itemsize;
        int flags;
        npy_intp *shape;
        npy_intp *strides;
        void *data;
        PyObject *descr;
    } PyArrayInterface;
```

two the integer 2 as a sanity check.

nd the number of dimensions in the array.

typekind A character indicating what kind of array is present according to the typestring convention with 't' -> bitfield, 'b' -> Boolean, 'i' -> signed integer, 'u' -> unsigned integer, 'f' -> floating point, 'c' -> complex floating point, 'O' -> object, 'S' -> string, 'U' -> unicode, 'V' -> void.

itemsize the number of bytes each item in the array requires.

flags any of the bits `NPY_C_CONTIGUOUS` (1), `NPY_F_CONTIGUOUS` (2), `NPY_ALIGNED` (0x100), `NPY_NOTSWAPPED` (0x200), or `NPY_WRITEABLE` (0x400) to indicate something about the data. The `NPY_ALIGNED`, `NPY_C_CONTIGUOUS`, and `NPY_F_CONTIGUOUS` flags can actually be determined from the other parameters. The flag `NPY_ARR_HAS_DESCR` (0x800) can also be set to indicate to objects consuming the version 3 array interface that the descr member of the structure is present (it will be ignored by objects consuming version 2 of the array interface).

shape An array containing the size of the array in each dimension.

strides An array containing the number of bytes to jump to get to the next element in each dimension.

data A pointer *to* the first element of the array.

descr A Python object describing the data-type in more detail (currently an array_description list of tuples). This can be `NULL` if *typekind* and *itemsize* provide enough information.

12.2.4 Internally used structures

Internally, the code uses some additional Python objects primarily for memory management. These types are not accessible directly from Python, and are not exposed to the C-API. They are included here only for completeness and assistance in understanding the code.

12.2.4.1 PyUFuncLoopObject

A loose wrapper for a C-structure that contains the information needed for looping. This is useful if you are trying to understand the ufunc looping code. The `PyUFuncLoopObject` is the associated C-structure. It is defined in the `ufuncobject.h` header.

12.2.4.2 PyUFuncReduceObject

A loose wrapper for the C-structure that contains the information needed for reduce-like methods of ufuncs. This is useful if you are trying to understand the reduce, accumulate, and reduce-at code. The `PyUFuncReduceObject` is the associated C-structure. It is defined in the `ufuncobject.h` header.

12.2.4.3 PyUFunc_Loop1d

A simple linked-list of C-structures containing the information needed to define a 1-d loop for a ufunc for every defined signature of a user-defined data-type.

12.2.4.4 PyArrayMapIter_Type

Advanced indexing is handled with this Python type. It is simply a loose wrapper around the C-structure containing the variables needed for advanced array indexing. The associated C-structure, `PyArrayMapIterObject`, is useful if you are trying to understand the advanced-index mapping code. It is defined in the `arrayobject.h` header. This type is not exposed to Python and could be replaced with a C-structure. As a Python type it takes advantage of reference-counted memory management.

Chapter 13

Complete API

13.1 Configuration defines

When NumPy is built, a configuration file is constructed and placed as config.h in the NumPy include directory. This configuration file ensures that specific macros are defined and defines other macros based on whether or not your system has certain features. It is included by the arrayobject.h file.

13.1.1 Guaranteed to be defined

The **SIZEOF_**<CTYPE> constants are defined so that sizeof information is available to the pre-processor.

CHAR_BIT The number of bits of a char. The char is the unit of all sizeof definitions

SIZEOF_SHORT sizeof(short)

SIZEOF_INT sizeof(int)

SIZEOF_LONG sizeof(long)

SIZEOF_LONG_LONG sizeof(longlong) where longlong is defined appropriately on the platform (A macro defines **SIZEOF_LONGLONG** as well.)

SIZEOF_PY_LONG_LONG

SIZEOF_FLOAT sizeof(float)

SIZEOF_DOUBLE sizeof(double)

SIZEOF_LONG_DOUBLE sizeof(longdouble) (A macro defines **SIZEOF_LONGDOUBLE** as well.)

SIZEOF_PY_INTPTR_T Size of a pointer on this platform (sizeof(void *)) (A macro defines SIZEOF_INTP as well.)

13.1.2 Possible defines

These defines will cause the compilation to ignore compatibility code that is placed in NumPy and use the system code instead. If they are not defined, then the system does not have that capability.

HAVE_LONGDOUBLE_FUNCS System has C99 long double math functions.

HAVE_FLOAT_FUNCS System has C99 float math functions.

HAVE_INVERSE_HYPERBOLIC System has inverse hyperbolic functions: asinh, acosh, and atanh.

HAVE_INVERSE_HYPERBOLIC_FLOAT System has C99 float extensions to inverse hyperbolic functions: asinhf, acoshf, atanhf

HAVE_INVERSE_HYPERBOLIC_LONGDOUBLE System has C99 long double extensions to inverse hyperbolic functions: asinhl, acoshl, atanhl.

HAVE_ISNAN System has an isnan function.

HAVE_ISINF System has an isinf function.

HAVE_LOG1P System has the log1p function: $\log(x+1)$.

HAVE_EXPM1 System has the expm1 function: $\exp(x)-1$.

HAVE_RINT System has the rint function.

13.2 Array Data Types

The standard array can have 21 different data types (and has some support for adding your own types). These data types all have an enumerated type, an enumerated type-character, and a corresponding array scalar Python type object (placed in a hierarchy). There are also standard C typedefs to make it easier to manipulate elements of the given data type. For the numeric types, there are also bit-width equivalent C typedefs and named typenumbers that make it easier to select the precision desired.

WARNING

The names for the types in c code follows c naming conventions more closely. The Python names for these types follow Python conventions. Thus, NPY_FLOAT picks up a 32-bit float in C, but "float_" in python corresponds to a 64-bit double. The bit-width names can be used in both Python and C for clarity.

13.2.1 Enumerated Types

There is a list of enumerated types defined providing the basic 21 data types plus some useful generic names. Whenever the code requires a type number, one of these enumerated types is requested. The types are all called **NPY_**<NAME> where <NAME> can be

> **BOOL**, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **OBJECT**, **STRING**, **UNICODE**, **VOID**
>
> **NTYPES**, **NOTYPE**, **USERDEF**, **DEFAULT_TYPE**

The various character codes indicating certain types are also part of an enumerated list. References to type characters (should they be needed at all) should always use these enumerations. The form of them is **NPY_**<NAME>**LTR** where <NAME> can be

> **BOOL**, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **OBJECT**, **STRING**, **VOID**
>
> **INTP**, **UINTP**
>
> **GENBOOL**, **SIGNED**, **UNSIGNED**, **FLOATING**, **COMPLEX**

The latter group of <NAME>s corresponds to letters used in the array interface typestring specification.

13.2.2 Defines

13.2.2.1 Max and min values for integers

NPY_MAX_INT<bits>

NPY_MAX_UINT<bits>

NPY_MIN_INT<bits>

These are defined for <bits> = 8, 16, 32, 64, 128, and 256 and provide the maximum (minimum) value of the corresponding (unsigned) integer type. Note: the actual integer type may not be available on all platforms (i.e. 128-bit and 256-bit integers are rare).

NPY_MIN_<type>

This is defined for <type> = **BYTE**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **INTP**

NPY_MAX_<type>

This is defined for all defined for <type> = **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **INTP**, **UINTP**

13.2.2.2 Number of bits in data types

All **NPY_SIZEOF_**<CTYPE> constants have corresponding **NPY_BITSOF_**<CTYPE> constants defined. The **NPY_BITSOF_**<CTYPE> constants provide the number of bits in the data type. Specifically, the available <CTYPE>s are

> **BOOL**, **CHAR**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**

13.2.2.3 Bit-width references to enumerated typenums

All of the numeric data types (integer, floating point, and complex) have constants that are defined to be a specific enumerated type number. Exactly which enumerated type a bit-width type refers to is platform dependent. In particular, the constants available are **PyArray_**<NAME><BITS> where <NAME> is **INT**, **UINT**, **FLOAT**, **COMPLEX** and <BITS> can be 8, 16, 32, 64, 80, 96, 128, 160, 192, 256, and 512. Obviously not all bit-widths are available on all platforms for all the kinds of numeric types. Commonly 8-, 16-, 32-, 64-bit integers; 32-, 64-bit floats; and 64-, 128-bit complex types are available.

13.2.2.4 Integer that can hold a pointer

The constants **PyArray_INTP** and **PyArray_UINTP** refer to an enumerated integer type that is large enough to hold a pointer on the platform. Index arrays should always be converted to **PyArray_INTP**, because the dimension of the array is of type npy_intp.

13.2.3 C-type names

There are standard variable types for each of the numeric data types and the bool data type. Some of these are already available in the C-specification. You can create variables in extension code with these types.

13.2.3.1 Boolean

npy_bool unsigned char; The constants NPY_FALSE and NPY_TRUE are also defined.

13.2.3.2 (Un)Signed Integer

Unsigned versions of the integers can be defined by pre-pending a 'u' to the front of the integer name.

npy_(u)byte (unsigned) char

npy_(u)short (unsigned) short

npy_(u)int (unsigned) int

npy_(u)long (unsigned) long int

npy_(u)longlong (unsigned long long int)

npy_(u)intp (unsigned) Py_intptr_t (an integer that is the size of a pointer on the platform).

13.2.3.3 (Complex) Floating point

npy_(c)float float

npy_(c)double double

npy_(c)longdouble long double

complex types are structures with **.real** and **.imag** members (in that order).

13.2.3.4 Bit-width names

There are also typedefs for signed integers, unsigned integers, floating point, and complex floating point types of specific bit-widths. The available type names are

> **npy_int**<bits>, **npy_uint**<bits>, **npy_float**<bits>, and **npy_complex**<bits>

where <bits> is the number of bits in the type and can be **8**, **16**, **32**, **64**, 128, and 256 for integer types; 16, **32**, **64**, 80, 96, 128, and 256 for floating-point types; and 32, **64**, **128**, 160, 192, and 512 for complex-valued types. Which bit-widths are available is platform dependent. The bolded bit-widths are usually available on all platforms.

13.2.4 Printf Formatting

For help in printing, the following strings are defined as the correct format specifier in printf and related commands.

> **NPY_LONGLONG_FMT**, **NPY_ULONGLONG_FMT**, **NPY_INTP_FMT**, **NPY_UINTP_FMT**, **NPY_LONGDOUBLE_FMT**

13.3 Array API

13.3.1 Array structure and data access

These macros all access the PyArrayObject structure members. The input argument, obj, can be any `PyObject*` that is directly interpretable as a `PyArrayObject*` (any instance of the **PyArray_Type** and its sub-types).

PyArray_DATA (`void*`) (`PyObject*` obj)

PyArray_BYTES (`char*`) (`PyObject*` obj)

These two macros are similar and obtain the pointer to the data-buffer for the array. The first macro can (and should be) assigned to a particular pointer where the second is for generic processing. If you have not guaranteed a contiguous and/or aligned array then be sure you understand how to access the data in the array to avoid memory and/or alignment problems.

PyArray_DIMS (`npy_intp*`) (`PyObject*` arr)

PyArray_STRIDES (`npy_intp*`) (`PyObject*` arr)

PyArray_DIM (`npy_intp`) (`PyObject*` arr, `int` n)

Return the shape in the n^{th} dimension.

PyArray_STRIDE (`npy_intp`) (`PyObject*` arr, `int` n)

Return the stride in the n^{th} dimension.

PyArray_BASE (`PyObject*`) (`PyObject*` arr)

PyArray_DESCR (`PyArray_Descr*`) (`PyObject*` arr)

PyArray_FLAGS (`int`) (`PyObject*` arr)

PyArray_ITEMSIZE (`int`) (`PyObject*` arr)

Return the itemsize for the elements of this array.

PyArray_TYPE (`int`) (`PyObject*` arr)

Return the (builtin) typenumber for the elements of this array.

PyArray_GETITEM (`PyObject` *) (`PyObject*` arr, `void*` itemptr)

Get a Python object from the ndarray, *arr*, at the location pointed to by itemptr. Return `NULL` on failure.

PyArray_SETITEM (`int`) (`PyObject*` arr, `void*` itemptr, `PyObject*` obj)

Convert obj and place it in the ndarray, *arr*, at the place pointed to by itemptr. Return -1 if an error occurs or 0 on success.

PyArray_SIZE (`npy_intp`) (`PyObject*` arr)

Returns the total size (in number of elements) of the array.

PyArray_Size (`npy_intp`) (`PyObject*` obj)

Returns 0 if *obj* is not a sub-class of bigndarray. Otherwise, returns the total number of elements in the array. Safer version of `PyArray_SIZE`(*obj*).

PyArray_NBYTES (`npy_intp`) (`PyObject*` arr)

Returns the total number of bytes consumed by the array.

13.3.1.1 Data access

These functions and macros provide easy access to elements of the ndarray from C. These work for all arrays. You may need to take care when accessing the data in the array, however, if it is not in machine byte-order, misaligned, or not writeable. In other words, be sure to respect the state of the flags unless you know what you are doing, or have previously guaranteed an array that is writeable, aligned, and in machine byte-order using PyArray_FromAny. If you wish to handle all types of arrays, the copyswap function for each type is useful for handling misbehaved arrays. Some platforms (e.g. Solaris) do not like misaligned data and will crash if you de-reference a misaligned pointer. Other platforms (e.g. x86 Linux) will just work more slowly with misaligned data.

PyArray_GetPtr (`void*`) (`PyArrayObject*` aobj, `npy_intp*` ind)

Return a pointer to the data of the ndarray, *aobj*, at the N-dimensional index given by the c-array, *ind*, (which must be at least *aobj*->nd in size). You may want to typecast the returned pointer to the data type of the ndarray.

PyArray_GETPTR1 (`void*`) (`PyObject*` obj, `<npy_intp>` i)

PyArray_GETPTR2 (`void*`) (`PyObject*` obj, `<npy_intp>` i, `<npy_intp>` j)

PyArray_GETPTR3 (`void*`) (`PyObject*` obj, `<npy_intp>` i, `<npy_intp>` j, `<npy_intp>` k)

PyArray_GETPTR4 (`void*`) (`PyObject*` obj, `<npy_intp>` i, `<npy_intp>` j, `<npy_intp>` k, `<npy_intp>` l)

Quick, inline access to the element at the given coordinates in the ndarray, *obj*, which must have respectively 1, 2, 3, or 4 dimensions (this is not checked). The corresponding *i*, *j*, *k*, and *l* coordinates can be any integer but will be interpreted as `npy_intp`. You may want to typecast the returned pointer to the data type of the ndarray.

13.3.2 Creating arrays

13.3.2.1 From scratch

PyArray_NewFromDescr (`PyObject*`) (`PyTypeObject*` subtype, `PyArray_Descr*` descr, `int` nd, `npy_intp*` dims, `npy_intp*` strides, `void*` data, `int` flags, `PyObject*` obj)

This is the main array creation function. Most new arrays are created with this flexible function. The returned object is an object of Python-type *subtype*, which must be a subtype of `PyArray_Type`. The array has *nd* dimensions, described by *dims*. The data-type descriptor of the new array is *descr*. If *subtype* is not `&PyArray_Type` (*e.g.* a Python subclass of the ndarray), then *obj* is the object to pass to the **__array_finalize__** method of the subclass. If *data* is `NULL`, then new memory will be allocated and *flags* can be non-zero to indicate a Fortran-style contiguous array. If *data* is not `NULL`, then it is assumed to point to the memory to be used for the array and the *flags* argument is used as the new flags for the array (except the state of `NPY_OWNDATA` and `UPDATEIFCOPY` flags of the new array will be reset). In addition, if *data* is non-NULL, then *strides* can also be provided. If *strides* is `NULL`, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (*flags* is nonzero for *data*=`NULL` or *flags* & `NPY_F_CONTIGUOUS` is nonzero non-NULL *data*). Any provided *dims* and *strides* are copied into newly allocated dimension and strides arrays for the new array object.

PyArray_New (`PyObject*`) (`PyTypeObject*` subtype, `int` nd, `npy_intp*` dims, `int` type_num, `npy_intp*` strides, `void*` data, `int` itemsize, `int` flags, `PyObject*` obj)

This is similar to `PyArray_DescrNew`(...) except you specify the data-type descriptor with *type_num* and *itemsize*, where *type_num* corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then itemsize is ignored. Otherwise, itemsize specifies the particular size of this array.

WARNING

If data is passed to `PyArray_NewFromDescr` or `PyArray_New`, this memory must not be deallocated until the new array is deleted. If this data came from another Python object, this can be accomplished using `Py_INCREF` on that object and setting the base member of the new array to point to that object. If strides are passed in they must be consistent with the dimensions, the itemsize, and the data of the array.

PyArray_SimpleNew (`PyObject*`) (`int` nd, `npy_intp*` dims, `int` typenum)

Create a new unitialized array of type, *typenum*, whose size in each of *nd* dimensions is given by the integer array, *dims*. This function cannot be used to create a flexible-type array (no itemsize given).

PyArray_SimpleNewFromData (`PyObject*`) (`int` nd, `npy_intp*` dims, `int` typenum, `void*` data)

Create an array wrapper around *data* pointed to by the given pointer. The array flags will have a default that the data area is well-behaved and C-style contiguous. The shape of the array is given by the *dims* c-array of length *nd*. The data-type of the array is indicated by *typenum*.

PyArray_SimpleNewFromDescr (`PyObject*`) (`int` nd, `npy_intp*` dims, `PyArray_Descr*` descr)

Create a new array with the provided data-type descriptor, *descr*, of the shape deteremined by *nd* and *dims*.

PyArray_FILLWBYTE (`PyObject*` obj, `int` val)

Fill the array pointed to by *obj*—which must be a (subclass of) bigndarray—with the contents of *val* (evaluated as a byte).

PyArray_Zeros (`PyObject*`) (`int` nd, `npy_intp*` dims, `PyArray_Descr*` dtype, `int` fortran)

Construct a new *nd*-dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. Fill the memory with zeros (or the 0 object if *dtype* corresponds to `PyArray_OBJECT`).

PyArray_ZEROS (`PyObject*`) (`int` nd, `npy_intp*` dims, `int` type_num, `int` fortran)

Macro form of `PyArray_Zeros` which takes a type-number instead of a data-type object.

PyArray_Empty (`PyObject*`) (`int` nd, `npy_intp*` dims, `PyArray_Descr*` dtype, `int` fortran)

Construct a new *nd*-dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. The array is uninitialized unless the data type corresponds to `PyArray_OBJECT` in which case the array is filled with `Py_None`.

PyArray_EMPTY (`PyObject*`) (`int` nd, `npy_intp*` dims, `int` typenum, `int` fortran)

Macro form of `PyArray_Empty` which takes a type-number, *typenum*, instead of a data-type object.

PyArray_Arange (`PyObject*`) (`double` start, `double` stop, `double` step, `int` typenum)

Construct a new 1-dimensional array of data-type, *typenum*, that ranges from *start* to *stop* (exclusive) in increments of *step*. Equivalent to **arange**(*start*, *stop*, *step*, dtype).

PyArray_ArangeObj (`PyObject*`) (`PyObject*` start, `PyObject*` stop, `PyObject*` step, `PyArray_Descr*` descr)

Construct a new 1-dimensional array of data-type determined by `descr`, that ranges from `start` to `stop` (exclusive) in increments of `step`. Equivalent to arange(`start`, `stop`, `step`, `typenum`).

13.3.2.2 From other objects

PyArray_FromAny (`PyObject*`) (`PyObject*` op, `PyArray_Descr*` dtype, `int` min_depth, `int` max_depth, `int` requirements, `PyObject*` context)

This is the main function used to obtain an array from any nested sequence, or object that exposes the array interface, `op`. The parameters allow specification of the required *type*, the minimum (*min_depth*) and maximum (*max_depth*) number of dimensions acceptable, and other *requirements* for the array. The *dtype* argument needs to be a `PyArray_Descr` structure indicating the desired data-type (including required byteorder). The *dtype* argument may be NULL, indicating that any data-type (and byteorder) is acceptable. If you want to use `NULL` for the *dtype* and ensure the array is notswapped then use `PyArray_CheckFromAny`. A value of 0 for either of the depth parameters causes the parameter to be ignored. Any of the following array flags can be added (*e.g.* using |) to get the *requirements* argument. If your code can handle general (*e.g.* strided, byte-swapped, or unaligned arrays) then *requirements* may be 0. Also, if *op* is not already an array (or does not expose the array interface), then a new array will be created (and filled from *op* using the sequence protocol). The new array will have `NPY_DEFAULT` as its flags member. The *context* argument is passed to the **__array__** method of *op* and is only used if the array is constructed that way.

NPY_C_CONTIGUOUS Make sure the returned array is C-style contiguous

NPY_F_CONTIGUOUS Make sure the returned array is Fortran-style contiguous.

NPY_ALIGNED Make sure the returned array is aligned on proper boundaries for its data type. An aligned array has the data pointer and every strides factor as a multiple of the alignment factor for the data-type-descriptor.

NPY_WRITEABLE Make sure the returned array can be written to.

NPY_ENSURECOPY Make sure a copy is made of *op*. If this flag is not present, data is not copied if it can be avoided.

NPY_ENSUREARRAY Make sure the result is a base-class ndarray or bigndarray. By default, if *op* is an instance of a subclass of the bigndarray, an instance of that same subclass is returned. If this flag is set, an ndarray object will be returned instead.

NPY_FORCECAST Force a cast to the output type even if it cannot be done safely. Without this flag, a data cast will occur only if it can be done safely, otherwise an error is reaised.

NPY_UPDATEIFCOPY If *op* is already an array, but does not satisfy the requirements, then a copy is made (which will satisfy the requirements).

If this flag is present and a copy (of an object that is already an array) must be made, then the corresponding NPY_UPDATEIFCOPY flag is set in the returned copy and *op* is made to be read-only. When the returned copy is deleted (presumably after your calculations are complete), its contents will be copied back into *op* and the *op* array will be made writeable again. If *op* is not writeable to begin with, then an error is raised. If *op* is not already an array, then this flag has no effect.

NPY_BEHAVED NPY_ALIGNED | NPY_WRITEABLE

NPY_CARRAY NPY_C_CONTIGUOUS | NPY_BEHAVED

NPY_CARRAY_RO NPY_C_CONTIGUOUS | NPY_ALIGNED

NPY_FARRAY NPY_F_CONTIGUOUS | NPY_BEHAVED

NPY_FARRAY_RO NPY_F_CONTIGUOUS | NPY_ALIGNED

NPY_DEFAULT NPY_CARRAY

NPY_IN_ARRAY NPY_CONTIGUOUS | NPY_ALIGNED

NPY_IN_FARRAY NPY_F_CONTIGUOUS | NPY_ALIGNED

NPY_INOUT_ARRAY NPY_C_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED

NPY_INOUT_FARRAY NPY_F_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED

NPY_OUT_ARRAY NPY_C_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED | NPY_UPDATEIFCOPY

NPY_OUT_FARRAY NPY_F_CONTIGUOUS | NPY_WRITEABLE | NPY_ALIGNED | UPDATEIFCOPY

PyArray_CheckFromAny (PyObject*) (PyObject* op, PyArray_Descr* dtype, int min_depth, int max_depth, int requirements, PyObject* context)

Nearly identical to PyArray_FromAny(...) except *requirements* can contain NPY_NOTSWAPPED (over-riding the specification in *dtype*) and NPY_ELEMENTSTRIDES which indicates that the array should be aligned in the sense that the strides are multiples of the element size.

NPY_NOTSWAPPED Make sure the returned array has a data-type descriptor that is in machine byte-order, over-riding any specification in the *dtype* argument. Normally, the byte-order requirement is determined by the *dtype* argument. If this flag is set and the dtype argument does not indicate a machine byte-order descriptor (or is NULL and the object is already an array with a data-type descriptor that is not in machine byte-order), then a new data-type descriptor is created and used with its byte-order field set to native.

NPY_BEHAVED_NS `NPY_ALIGNED` | `NPY_WRITEABLE` | `NPY_NOTSWAPPED`

NPY_ELEMENTSTRIDES Make sure the returned array has strides that are multiples of the element size.

PyArray_FromArray (`PyObject*`) (`PyArrayObject*` op, `PyArray_Descr*` newtype, `int` requirements)

Special case of `PyArray_FromAny` for when *op* is already an array but it needs to be of a specific *newtype* (including byte-order) or has certain *requirements*.

PyArray_FromStructInterface (`PyObject*`) (`PyObject*` op)

Returns an ndarray object from a Python object that exposes the **__array_struct__** method and follows the array interface protocol. If the object does not contain this method then a borrowed reference to `Py_NotImplemented` is returned.

PyArray_FromInterface (`PyObject*`) (`PyObject*` op)

Returns an ndarray object from a Python object that exposes the **__array_shape__** and **__array_typestr__** methods following the array interface protocol. If the object does not contain one of these method then a borrowed reference to `Py_NotImplemented` is returned.

PyArray_FromArrayAttr (`PyObject*`) (`PyObject*` op, `PyArray_Descr*` dtype, `PyObject*` context)

Return an ndarray object from a Python object that exposes the **__array__** method. The **__array__** method can take 0, 1, or 2 arguments ([dtype, context]) where *context* is used to pass information about where the **__array__** method is being called from (currently only used in ufuncs).

PyArray_ContiguousFromAny (`PyObject*`) (`PyObject*` op, `int` typenum, `int` min_depth, `int` max_depth)

This function returns a (C-style) contiguous and behaved function array from any nested sequence or array interface exporting object, *op*, of (non-flexible) type given by the enumerated *typenum*, of minimum depth *min_depth*, and of maximum depth *max_depth*. Equivalent to a call to `PyArray_FromAny` with requirements set to `NPY_DEFAULT` and the type_num member of the type argument set to *typenum*.

PyArray_FromObject (PyObject *) (PyObject * op, int typenum, int min_depth, int max_depth)

Return an aligned and in native-byteorder array from any nested sequence or array-interface exporting object, op, of a type given by the enumerated typenum. The minimum number of dimensions the array can have is given by min_depth while the maximum is max_depth. This is equivalent to a call to PyArray_FromAny with requirements set to BEHAVED.

PyArray_EnsureArray (`PyObject*`) (`PyObject*` op)

This function **steals a reference** to `op` and makes sure that `op` is a base-class ndarray. It special cases array scalars, but otherwise calls **PyArray_FromAny**(`op`, NULL, 0, 0, NPY_ENSUREARRAY).

PyArray_FromString (`PyObject*`) (`char*` string, `npy_intp` slen, `PyArray_Descr*` dtype, `npy_intp` num, `char*` sep)

Construct a one-dimensional ndarray of a single type from a binary or (ASCII) text `string` of length `slen`. The data-type of the array to-be-created is given by `dtype`. If num is -1, then **copy** the entire string and return an appropriately sized array, otherwise, `num` is the number of items to **copy** from the string. If `sep` is NULL (or “”), then interpret the string as bytes of binary data, otherwise convert the sub-strings separated by `sep` to items of data-type `dtype`. Some data-types may not be readable in text mode and an error will be raised if that occurs. All errors return NULL.

PyArray_FromFile (`PyObject*`) (`FILE*` fp, `PyArray_Descr*` dtype, `npy_intp` num, `char*` sep)

Construct a one-dimensional ndarray of a single type from a binary or text file. The open file pointer is `fp`, the data-type of the array to be created is given by `dtype`. This must match the data in the file. If `num` is -1, then read until the end of the file and return an appropriately sized array, otherwise, `num` is the number of items to read. If `sep` is NULL (or “”), then read from the file in binary mode, otherwise read from the file in text mode with `sep` providing the item separator. Some array types cannot be read in text mode in which case an error is raised.

PyArray_FromBuffer (`PyObject*`) (`PyObject*` buf, `PyArray_Descr*` dtype, `npy_intp` count, `npy_intp` offset)

Construct a one-dimensional ndarray of a single type from an object, `buf`, that exports the (single-segment) buffer protocol (or has an attribute __buffer__ that returns an object that exports the buffer protocol). A writeable buffer will be tried first followed by a read-only buffer. The NPY_WRITEABLE flag of the returned array will reflect which one was successful. The data is assumed to start at `offset` bytes from the start of the memory location for the object. The type of the data in the buffer will be interpreted depending on the data-type descriptor, `dtype.` If `count` is negative then it will be determined from the size of the buffer and the requested itemsize, otherwise, `count` represents how many elements should be converted from the buffer.

PyArray_CopyInto (`int`) (`PyArrayObject*` dest, `PyArrayObject*` src)

Copy from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src must not overlap.

PyArray_MoveInto (`int`) (`PyArrayObject*` dest, `PyArrayObject*` src)

Move data from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src may overlap.

PyArray_GETCONTIGUOUS (`PyArrayObject*`) (`PyObject*` op)

If `op` is already (C-style) contiguous and well-behaved then just return a reference, otherwise return a (contiguous and well-behaved) copy of the array. The parameter op must be a (sub-class of an) ndarray and no checking for that is done.

PyArray_FROM_O (`PyObject*`) (`PyObject*` obj)

Convert `obj` to an ndarray. The argument can be any nested sequence or object that exports the array interface. This is a macro form of `PyArray_FromAny` using `NULL`, 0, 0, 0 for the other arguments. Your code must be able to handle any data-type descriptor and any combination of data-flags to use this macro.

PyArray_FROM_OF (`PyObject*`) (`PyObject*` obj, `int` requirements)

Similar to `PyArray_FROM_O` except it can take an argument of *requirements* indicating properties the resulting array must have. Available requirements that can be enforced are `NPY_CONTIGUOUS`, `NPY_F_CONTIGUOUS`, `NPY_ALIGNED`, `NPY_WRITEABLE`, `NPY_NOTSWAPPED`, `NPY_ENSURECOPY`, `NPY_UPDATEIFCOPY`, `NPY_FORCECAST`, and `NPY_ENSUREARRAY`. Standard combinations of flags can also be used:

PyArray_FROM_OT (`PyObject*`) (`PyObject*` obj, `int` typenum)

Similar to `PyArray_FROM_O` except it can take an argument of *typenum* specifying the type-number the returned array.

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

Combination of `PyArray_FROM_OF` and `PyArray_FROM_OT` allowing both a *typenum* and a *flags* argument to be provided..

PyArray_FROMANY (`PyObject*`) (`PyObject*` obj, `int` typenum, `int` min, `int` max, `int` requirements)

Similar to `PyArray_FromAny` except the data-type is specified using a typenumber. `PyArray_DescrFromType`(*typenum*) is passed directly to `PyArray_FromAny`. This macro also adds `NPY_DEFAULT` to requirements if `NPY_ENSURECOPY` is passed in as requirements.

13.3.3 Dealing with types

13.3.3.1 General check of Python Type

PyArray_Check (op)

Evaluates true if *op* is a Python object whose type is a sub-type of `PyArray_Type`.

PyArray_CheckExact (op)

Evaluates true if *op* is a Python object with type `PyArray_Type`.

PyArray_HasArrayInterface (op, out)

If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to `Py_NotImplemented` and no error condition is set.

PyArray_HasArrayInterfaceType (op, type, context, out)

If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to Py_NotImplemented and no error condition is set. This version allows setting of the type and context in the part of the array interface that looks for the **__array__** attribute.

PyArray_IsZeroDim (op)

Evaluates true if *op* is an instance of (a subclass of) `PyArray_Type` and has 0 dimensions.

PyArray_IsScalar (op, cls)

Evaluates true if *op* is an instance of `Py<cls>ArrType_Type`.

PyArray_CheckScalar (op)

Evaluates true if *op* is either an array scalar (an instance of a sub-type of `PyGenericArr_Type`), or an instance of (a sub-class of) `PyArray_Type` whose dimensionality is 0.

PyArray_IsPythonScalar (op)

Evaluates true if *op* is a builtin Python "scalar" object (int, float, complex, str, unicode, long, bool).

PyArray_IsAnyScalar (op)

Evaluates true if *op* is either a Python scalar or an array scalar (an instance of a sub-type of `PyGenericArr_Type`).

13.3.3.2 Data-type checking

For the typenum macros, the argument is an integer representing an enumerated array data type. For the array type checking macros the argument must be a `PyObject*` that can be directly interpreted as a `PyArrayObject*`.

PyTypeNum_ISUNSIGNED (num)

PyDataType_ISUNSIGNED (descr)

PyArray_ISUNSIGNED (obj)

Type represents an unsigned integer.

PyTypeNum_ISSIGNED (num)

PyDataType_ISSIGNED (descr)

PyArray_ISSIGNED (obj)

Type represents a signed integer.

PyTypeNum_ISINTEGER (num)

PyDataType_ISINTEGER (descr)

PyArray_ISINTEGER (obj)

Type represents any integer.

PyTypeNum_ISFLOAT (num)

PyDataType_ISFLOAT (descr)

PyArray_ISFLOAT (obj)

Type represents any floating point number.

PyTypeNum_ISCOMPLEX (num)

PyDataType_ISCOMPLEX (descr)

PyArray_ISCOMPLEX (obj)

Type represents any complex floating point number.

PyTypeNum_ISNUMBER (num)

PyDataType_ISNUMBER (descr)

PyArray_ISNUMBER (obj)

Type represents any integer, floating point, or complex floating point number.

PyTypeNum_ISSTRING (num)

PyDataType_ISSTRING (descr)

PyArray_ISSTRING (obj)

Type represents a string data type.

PyTypeNum_ISPYTHON (num)

PyDataType_ISPYTHON (descr)

PyArray_ISPYTHON (obj)

Type represents an enumerated type corresponding to one of the standard Python scalar (bool, int, float, or complex).

PyTypeNum_ISFLEXIBLE (num)

PyDataType_ISFLEXIBLE (descr)

PyArray_ISFLEXIBLE (obj)

Type represents one of the flexible array types (`NPY_STRING`, `NPY_UNICODE`, or `NPY_VOID`).

PyTypeNum_ISUSERDEF (num)

PyDataType_ISUSERDEF (descr)

PyArray_ISUSERDEF (obj)

Type represents a user-defined type.

PyTypeNum_ISEXTENDED (num)

PyDataType_ISEXTENDED (descr)

PyArray_ISEXTENDED (obj)

Type is either flexible or user-defined.

PyTypeNum_ISOBJECT (num)

PyDataType_ISOBJECT (descr)

PyArray_ISOBJECT (obj)

Type represents object data type.

PyTypeNum_ISBOOL (num)

PyDataType_ISBOOL (descr)

PyArray_ISBOOL (obj)

Type represents Boolean data type.

PyDataType_HASFIELDS (descr)

PyArray_HASFIELDS (obj)

Type has fields associated with it.

PyArray_ISNOTSWAPPED (m)

Evaluates true if the data area of the ndarray *m* is in machine byte-order according to the array's data-type descriptor.

PyArray_ISBYTESWAPPED (m)

Evaluates true if the data area of the ndarray *m* is **not** in machine byte-order according to the array's data-type descriptor.

PyArray_EquivTypes (`Bool`) (`PyArray_Descr*` type1, `PyArray_Descr*` type2)

Return `NPY_TRUE` if *type1* and *type2* actually represent equivalent types for this platform (the fortran member of each type is ignored). For example, on 32-bit platforms, `NPY_LONG` and `NPY_INT` are equivalent. Otherwise return `NPY_FALSE`.

PyArray_EquivArrTypes (`Bool`) (`PyArrayObject*` a1, `PyArrayObject*` a2)

Return `NPY_TRUE` if *a1* and *a2* are arrays with equivalent types for this platform.

PyArray_EquivTypenums (`Bool`) (`int` typenum1, `int` typenum2)

Special case of `PyArray_EquivTypes`(...) that does not accept flexible data types but may be easier to call.

PyArray_EquivByteorders (int) (<byteorder> b1, <byteorder> b2)

True if byteorder characters (`NPY_LITTLE`, `NPY_BIG`, `NPY_NATIVE`, `NPY_IGNORE`) are either equal or equivalent as to their specification of a native byte order. Thus, on a little-endian machine `NPY_LITTLE` and `NPY_NATIVE` are equivalent where they are not equivalent on a big-endian machine.

13.3.3.3 Converting data types

PyArray_Cast (`PyObject*`) (`PyArrayObject*` arr, `int` typenum)

Mainly for backwards compatibility to the Numeric C-API and for simple casts to non-flexible types. Return a new array object with the elements of *arr* cast to the data-type *typenum* which must be one of the enumerated types and not a flexible type.

PyArray_CastToType (`PyObject*`) (`PyArrayObject*` arr, `PyArray_Descr*` type)

Return a new array of the *type* specified, casting the elements of *arr* as appropriate. The fortran member of the type structure will be respected in producing the strides of the output array.

PyArray_CastTo (`int`) (`PyArrayObject*` out, `PyArrayObject*` in)

Cast the elements of the array *in* into the array *out*. The output array should be writeable, have an integer-multiple of the number of elements in the input array (more than one copy can be placed in out), and have a data type that is one of the builtin types. Returns 0 on success and -1 if an error occurs.

PyArray_GetCastFunc (`PyArray_VectorUnaryFunc*`) (`PyArray_Descr*` from, `int` totype)

Return the low-level casting function to cast from the given descriptor to the builtin type number. If no casting function exists return `NULL` and set an error. Using this function instead of direct access to *from*->f->cast will allow support of any user-defined casting functions added to a descriptors casting dictionary.

PyArray_CanCastSafely (`int`) (`int` fromtype, `int` totype)

Returns non-zero if an array of data type *fromtype* can be cast to an array of data type *totype* without losing information. An exception is that 64-bit integers are allowed to be cast to 64-bit floating point values even though this can lose precision on large integers so as not to proliferate the use of long doubles without explict requests. Flexible array types are not checked according to their lengths with this function.

PyArray_CanCastTo (`int`) (`PyArray_Descr*` fromtype, `PyArray_Descr*` totype)

Returns non-zero if an array of data type *fromtype* (which can include flexible types) can be cast safely to an array of data type *totype* (which can include flexible types). This is basically a wrapper around `PyArray_CanCastSafely` with additional support for size checking if *fromtype* and *totype* are `NPY_STRING` or `NPY_UNICODE`.

PyArray_ObjectType (`int`) (`PyObject*` op, `int` mintype)

This function is useful for determining a common type that two or more arrays can be converted to. It only works for non-flexible array types as no itemsize information is passed. The *mintype* argument represents the minimum type acceptable, and *op* represents the object that will be converted to an array. The return value is the enumerated typenumber that represents the data-type that *op* should have.

PyArray_ArrayType (`void`) (`PyObject*` op, `PyArray_Descr*` mintype, `PyArray_Descr*` outtype)

This function works similarly to `PyArray_ObjectType`(...) except it handles flexible arrays. The *mintype* argument can have an itemsize member and the *outtype* argument will have an itemsize member at least as big but perhaps bigger depending on the object *op*.

PyArray_ConvertToCommonType (`PyArrayObject**`) (`PyObject*` op, `int*` n)

Convert a sequence of Python objects contained in *op* to an array of ndarrays each having the same data type. The type is selected based on the typenumber (larger type number is chosen over a smaller one) ignoring objects that are only scalars. The length of the sequence is returned in n, and an n-length array of `PyArrayObject` pointers is the return value (or `NULL` if an error occurs). The returned array must be freed by the caller of this routine (using `PyDataMem_FREE`) and all the array objects in it `DECREF`'d or a memory-leak will occur. The example template-code below shows a typically usage.

```
mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
<code>
<before return>
for (i=0; i<n; i++) Py_DECREF(mps[i]);
PyDataMem_FREE(mps);
<return>
```

PyArray_Zero (`char*`) (`PyArrayObject*` arr)

A pointer to newly created memory of size *arr*->itemsize that holds the representation of 0 for that type. The returned pointer, *ret*, **must be freed** using *`PyDataMem_FREE`*(ret) when it is not needed anymore.

PyArray_One (`char*`) (`PyArrayObject*` arr)

A pointer to newly created memory of size *arr*->itemsize that holds the representation of 1 for that type. The returned pointer, *ret*, **must be freed** using `PyDataMem_FREE`(ret) when it is not needed anymore.

PyArray_ValidType (`int`) (`int` typenum)

Returns `NPY_TRUE` if *`typenum`* represents a valid type-number (builtin or user-defined or character code). Otherwise, this function returns `NPY_FALSE`.

13.3.3.4 New data types

PyArray_InitArrFuncs (`void`) (`PyArray_ArrFuncs*` f)

Initialize all function pointers and members to `NULL`.

PyArray_RegisterDataType (`int`) (`PyArray_Descr*` dtype)

Register a data-type as a new user-defined data type for arrays. The type must have most of its entries filled in. This is not always checked and errors can produce segfaults. In particular, the typeobj member of the `dtype` structure must be filled with a Python type that has a fixed-size element-size that corresponds to the elsize member of *dtype*. Also the `f` member must have the required functions: nonzero, copyswap, copyswapn, getitem, setitem, and cast (some of the cast functions may be `NULL` if no support is desired). To avoid confusion, you should choose a unique character typecode but this is not enforced and not relied on internally.

A user-defined type number is returned that uniquely identifies the type. A pointer to the new structure can then be obtained from `PyArray_DescrFromType` using the returned type number. A -1 is returned if an error occurs. If this ***`dtype`*** has already been registered (checked only by the address of the pointer), then return the previously-assigned type-number.

PyArray_RegisterCastFunc (`int`) (`PyArray_Descr*` descr, `int` totype, `PyArray_VectorUnaryFunc*` castfunc)

Register a low-level casting function, *castfunc*, to convert from the data-type, *descr*, to the given data-type number, *totype*. Any old casting function is over-written. A `0` is returned on success or a `-1` on failure.

PyArray_RegisterCanCast (`int`) (`PyArray_Descr*` descr, `int` totype, `PyArray_SCALARKIND` scalar)

Register the data-type number, *totype*, as castable from data-type object, *descr*, of the given *scalar* kind. Use *scalar* = `NPY_NOSCALAR` to register that an array of data-type *descr* can be cast safely to a data-type whose type_number is *totype*.

13.3.3.5 Special functions for PyArray_OBJECT

PyArray_INCREF (`int`) (`PyArrayObject*` op)

Used for an array, *op*, that contains any Python objects. It increments the reference count of every object in the array according to the data-type of *op*. A -1 is returned if an error occurs, otherwise 0 is returned.

PyArray_Item_INCREF (`void`) (`char*` ptr, `PyArray_Descr*` dtype)

A function to INCREF all the objects at the location *ptr* according to the data-type *dtype*. If *ptr* is the start of a record with an object at any offset, then this will (recursively) increment the reference count of all object-like items in the record.

PyArray_XDECREF (`int`) (`PyArrayObject*` op)

Used for an array, *op*, that contains any Python objects. It decrements the reference count of every object in the array according to the data-type of *op*. Normal return value is 0. A -1 is returned if an error occurs.

PyArray_Item_XDECREF (`void`) (`char*` ptr, `PyArray_Descr*` dtype)

A function to XDECREF all the object-like items at the loacation *ptr* as recorded in the data-type, *dtype*. This works recursively so that if `dtype` itself has fields with data-types that contain object-like items, all the object-like fields will be XDECREF`'d`.

PyArray_FillObjectArray (`void`) (`PyArrayObject*` arr, `PyObject*` obj)

Fill a newly created array with a single value obj at all locations in the structure with object data-types. No checking is performed but *arr* must be of data-type `PyArray_OBJECT` and be single-segment and uninitialized (no previous objects in position). Use `PyArray_DECREF`(*arr*) if you need to decrement all the items in the object array prior to calling this function.

13.3.4 Array flags

13.3.4.1 Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manipulate. It may not be aligned with word boundaries (very important on some platforms). It might have its data in a different byte-order than the machine recognizes. It might not be writeable. It might be in Fortan-contiguous order. The array flags are used to indicate what can be said about data associated with an array.

NPY_C_CONTIGUOUS The data area is in C-style contiguous order (last index varies the fastest).

NPY_F_CONTIGUOUS The data area is in Fortran-style contiguous order (first index varies the fastest).

NPY_OWNDATA The data area is owned by this array.

NPY_ALIGNED The data area is aligned appropriately (for all strides).

NPY_WRITEABLE The data area can be written to.

Notice that the above 3 flags are are defined so that a new, well-behaved array has these flags defined as true.

NPY_UPDATEIFCOPY The data area represents a (well-behaved) copy whose information should be transferred back to the original when this array is deleted.

13.3.4.2 Combinations of array flags

NPY_BEHAVED `NPY_ALIGNED` | `NPY_WRITEABLE`

NPY_CARRAY `NPY_C_CONTIGUOUS` | `NPY_BEHAVED`

NPY_CARRAY_RO `NPY_C_CONTIGUOUS` | `NPY_ALIGNED`

NPY_FARRAY `NPY_F_CONTIGUOUS` | `NPY_BEHAVED`

NPY_FARRAY_RO `NPY_F_CONTIGUOUS` | `NPY_ALIGNED`

NPY_DEFAULT `NPY_CARRAY`

NPY_UPDATE_ALL `NPY_C_CONTIGUOUS` | `NPY_F_CONTIGUOUS` | `NPY_ALIGNED`

13.3.4.3 Flag-like constants

These constants are used in PyArray_FromAny (and its macro forms) to specify desired properties of the new array.

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 (or bigndarray), and not a sub-class.

NPY_NOTSWAPPED Only used in `PyArray_CheckFromAny` to over-ride the byteorder of the data-type object passed in.

NPY_BEHAVED_NS `NPY_ALIGNED` | `NPY_WRITEABLE` | `NPY_NOTSWAPPED`

13.3.4.4 Flag checking

For all of these macros *arr* must be an instance of a (subclass of) `PyArray_Type`, but no checking is done.

PyArray_CHKFLAGS (arr, flags)

The first parameter, arr, must be an ndarray or subclass. The parameter, *flags*, should be an integer consisting of bitwise combinations of the possible flags an array can have: `NPY_C_CONTIGUOUS`, `NPY_F_CONTIGUOUS`, `NPY_OWNDATA`, `NPY_ALIGNED`, `NPY_WRITEABLE`, `NPY_UPDATEIFCOPY`.

PyArray_ISCONTIGUOUS (arr)

Evaluates true if *arr* is C-style contiguous.

PyArray_ISFORTRAN (arr)

Evaluates true if *arr* is Fortran-style contiguous.

PyArray_ISWRITEABLE (arr)

Evaluates true if the data area of *arr* can be written to

PyArray_ISALIGNED (arr)

Evaluates true if the data area of *arr* is properly aligned on the machine.

PyArray_ISBEHAVED (arr)

Evalutes true if the data area of *arr* is aligned and writeable and in machine byte-order according to its descriptor.

PyArray_ISBEHAVED_RO (arr)

Evaluates true if the data area of *arr* is aligned and in machine byte-order.

PyArray_ISCARRAY (arr)

Evaluates true if the data area of *arr* is C-style contiguous, and `PyArray_ISBEHAVED`(*arr*) is true.

PyArray_ISFARRAY (arr)

Evaluates true if the data area of *arr* is Fortran-style contiguous and `PyArray_ISBEHAVED`(*arr*) is true.

PyArray_ISCARRAY_RO (arr)

Evaluates true if the data area of *arr* is C-style contiguous, aligned, and in machine byte-order.

PyArray_ISFARRAY_RO (arr)

Evaluates true if the data area of *arr* is Fortran-style contiguous, aligned, and in machine byte-order.

PyArray_ISONESEGMENT (arr)

Evaluates true if the data area of *arr* consists of a single (C-style or Fortran-style) contiguous segment.

PyArray_UpdateFlags (`void`) (`PyArrayObject*` arr, `int` flagmask)

The `NPY_C_CONTIGUOUS`, `NPY_ALIGNED`, and `NPY_F_CONTIGUOUS` array flags can be "calculated" from the array object itself. This routine updates one or more of these flags of *arr* as specified in *flagmask* by performing the required calculation.

WARNING

It is important to keep the flags updated (using PyArray_UpdateFlags can help) whenever a manipulation with an array is performed that might cause them to change. Later calculations in NumPy that rely on the state of these flags do not repeat the calculation to update them.

13.3.5 Array method alternative API

13.3.5.1 Conversion

PyArray_GetField (`PyObject*`) (`PyArrayObject*` self, `PyArray_Descr*` dtype, `int` offset)

Equivalent to *self*.**getfield**(*dtype*, *offset*). Return a new array of the given *dtype* using the data in the current array at a specified *offset* in bytes. The *offset* plus the itemsize of the new array type must be less than *self*->descr->elsize or an error is raised. The same shape and strides as the original array are used. Therefore, this function has the effect of returning a field from a record array. But, it can also be used to select specific bytes or groups of bytes from any array type.

PyArray_SetField (`int`) (`PyArrayObject*` self, `PyArray_Descr*` dtype, `int` offset, `PyObject*` val)

Equivalent to *self*.**setfield**(*val*, *dtype*, *offset*). Set the field starting at *offset* in bytes and of the given *dtype* to *val*. The *offset* plus *dtype*->elsize must be less than *self*->descr->elsize or an error is raised. Otherwise, the *val* argument is converted to an array and copied into the field pointed to. If necessary, the elements of *val* are repeated to fill the destination array, But, the number of elements in the destination must be an integer multiple of the number of elements in *val*.

PyArray_Byteswap (`PyObject*`) (`PyArrayObject*` self, `Bool` inplace)

Equivalent to *self*.**byteswap**(*inplace*). Return an array whose data area is byteswapped. If *inplace* is non-zero, then do the byteswap inplace and return a reference to self. Otherwise, create a byteswapped copy and leave self unchanged.

PyArray_NewCopy (`PyObject*`) (`PyArrayObject*` old, `NPY_ORDER` order)

Equivalent to *self*.**copy**(*fortran*). Make a copy of the *old* array. The returned array is always aligned and writeable with data interpreted the same as the old array. If *order* is `NPY_CORDER`, then a C-style contiguous array is returned. If *order* is `NPY_FORTRANORDER`, then a Fortran-style contiguous array is returned. If *order is* `NPY_ANYORDER`, then the array returned is Fortran-style contiguous only if the old one is; otherwise, it is C-style contiguous.

PyArray_ToList (`PyObject*`) (`PyArrayObject*` self)

Equivalent to *self*.**tolist**(). Return a nested Python list from *self*.

PyArray_ToString (PyObject*) (PyArrayObject* self, NPY_ORDER order)

Equivalent to *self*.**tostring**(*order*). Return the bytes of this array in a Python string.

PyArray_ToFile (`PyObject*`) (`PyArrayObject*` self, `FILE*` fp, `char*` sep, `char*` format)

Write the contents of *self* to the file pointer *fp* in C-style contiguous fashion. Write the data as binary bytes if `sep` is the string “” or `NULL`. Otherwise, write the contents of *self* as text using the `sep` string as the item separator. Each item will be printed to the file. If the *format* string is not `NULL` or “”, then it is a Python print statement format string showing how the items are to be written.

PyArray_Dump (`int`) (`PyObject*` self, `PyObject*` file, `int` protocol)

Pickle the object in *self* to the given *file* (either a string or a Python file object). If *file* is a Python string it is considered to be the name of a file which is then opened in binary mode. The given *protocol* is used (if *protocol* is negative, or the highest available is used). This is a simple wrapper around cPickle.dump(*self*, *file*, *protocol*).

PyArray_Dumps (`PyObject*`) (`PyObject*` self, `int` protocol)

Pickle the object in *self* to a Python string and return it. Use the Pickle *protocol* provided (or the highest available if *protocol* is negative).

PyArray_FillWithScalar (`int`) (`PyArrayObject*` arr, `PyObject*` obj)

Fill the array, *arr*, with the given scalar object, *obj*. The object is first converted to the data type of *arr*, and then copied into every location. A -1 is returned if an error occurs, otherwise 0 is returned.

PyArray_View (`PyObject*`) (`PyArrayObject*` self, `PyArray_Descr*` dtype)

Equivalent to *self*.**view**(*dtype*). Return a new view of the array *self* as possibly a different data-type, *dtype*. If *dtype* is `NULL`, then the returned array will have the same data type as *self*. The new data-type must be consistent with the size of *self*. Either the itemsizes must be identical, or *self* must be single-segment and the total number of bytes must be the same. In the latter case the dimensions of the returned array will be altered in the last (or first for Fortran-style contiguous arrays) dimension. The data area of the returned array and self is exactly the same.

13.3.5.2 Shape Manipulation

PyArray_Newshape (`PyObject*`) (`PyArrayObject*` self, `PyArray_Dims*` newshape)

Result will be a new array (pointing to the same memory location as *self* if possible), but having a shape given by *newshape*. If the new shape is not compatible with the strides of *self*, then a copy of the array with the new specified shape will be returned.

PyArray_Reshape (`PyObject*`) (`PyArrayObject*` self, `PyObject*` shape)

Equivalent to `self`.**reshape**(`shape`) where *shape* is a sequence. Converts *shape* to a `PyArray_Dims` structure and calls `PyArray_Newshape` internally.

PyArray_Squeeze (`PyObject*`) (`PyArrayObject*` self)

Equivalent to *self*.**squeeze**(). Return a new view of *self* with all of the dimensions of length 1 removed from the shape.

WARNING

matrix objects are always 2-dimensional. Therefore, `PyArray_Squeeze` has no effect on arrays of matrix sub-class.

PyArray_SwapAxes (`PyObject*`) (`PyArrayObject*` self, `int` a1, `int` a2)

Equivalent to *self*.**swapaxes**(*a1*, *a2*). The returned array is a new view of the data in *self* with the given axes, *a1* and *a2*, swapped.

PyArray_Resize (`PyObject*`) (`PyArrayObject*` self, `PyArray_Dims*` newshape, `int` refcheck)

Equivalent to *self*.**resize**(*newshape*, refcheck=*refcheck*). This function only works on single-segment arrays. It changes the shape of *self* inplace and will reallocate the memory for *self* if *newshape* has a different total number of elements then the old shape. If reallocation is necessary, then *self* must own its data, have *self*`->base==NULL`, have *self*`->weakrefs==NULL`, and (unless refcheck is 0) not be referenced by any other array. A reference to the new array is returned.

PyArray_Transpose (`PyObject*`) (`PyArrayObject*` self, `PyArray_Dims*` permute)

Equivalent to *self*.**transpose**(*permute*). Permute the axes of the ndarray object *self* according to the data structure *permute* and return the result. If *permute* is `NULL`, then the resulting array has its axes reversed. For example if *self* has shape $10 \times 20 \times 30$, and *permute*`.ptr` is (0,2,1) the shape of the result is $10 \times 30 \times 20$. If *permute* is `NULL`, the shape of the result is $30 \times 20 \times 10$.

PyArray_Flatten (`PyObject*`) (`PyArrayObject*` self, `NPY_ORDER` order)

Equivalent to *self*.**flatten**(*order*). Return a 1-d copy of the array. If *order* is `NPY_FORTRANORDER` the elements are scanned out in Fortran order (first-dimension varies the fastest). If *order* is `NPY_CORDER`, the elements of `self` are scanned in C-order (last dimension varies the fastest). If *order* `NPY_ANYORDER`, then the result of `PyArray_ISFORTRAN`(*self*) is used to determine which order to flatten.

PyArray_Ravel (`PyObject*`) (`PyArrayObject*` self, `NPY_ORDER` order)

Equivalent to *self*.ravel(*order*). Same basic functionality as `PyArray_Flatten`(*self*, *order*) except if *order* is 0 and *self* is C-style contiguous, the shape is altered but no copy is performed.

13.3.5.3 Item selection and manipulation

PyArray_TakeFrom (`PyObject*`) (`PyArrayObject*` self, `PyObject*` indices, `int` axis, `PyArrayObject*` ret, `NPY_CLIPMODE` clipmode)

Equivalent to *self*.**take**(*indices*, *axis*, *ret*, *clipmode*) except *axis*=None in Python is obtained by setting *axis*=`NPY_MAXDIMS` in C. Extract the items from self indicated by the integer-valued *indices* along the given *axis.* The clipmode argument can be `NPY_RAISE`, `NPY_WRAP`, or `NPY_CLIP` to indicate what to do with out-of-bound indices. The *ret* argument can specify an output array rather than having one created internally.

PyArray_PutTo (`PyObject*`) (`PyArrayObject*` self, `PyObject*` values, `PyObject*` indices, `NPY_CLIPMODE` clipmode)

Equivalent to *self*.put(*values*, *indices*, *clipmode*). Put *values* into *self* at the corresponding (flattened) *indices.* If *values* is too small it will be repeated as necessary.

PyArray_PutMask (`PyObject*`) (`PyArrayObject*` self, `PyObject*` values, `PyObject*` mask)

Place the *values* in *self* wherever corresponding positions (using a flattened context) in *mask* are true. The *mask* and *self* arrays must have the same total number of elements. If *values* is too small, it will be repeated as necessary.

PyArray_Repeat (`PyObject*`) (`PyArrayObject*` self, `PyObject*` op, `int` axis)

Equivalent to *self*.**repeat**(*op*, *axis*). Copy the elements of *self*, *op* times along the given *axis*. Either *op* is a scalar integer or a sequence of length *self*->dimensions[*axis*] indicating how many times to repeat each item along the axis.

PyArray_Choose (`PyObject*`) (`PyArrayObject*` self, `PyObject*` op, `PyArrayObject*` ret, `NPY_CLIPMODE` clipmode)

Equivalent to *self*.**choose**(*op*, *ret*, *clipmode*). Create a new array by selecting elements from the sequence of arrays in *op* based on the integer values in *self*. The arrays must all be broadcastable to the same shape and the entries in *self* should be between 0 and len(*op*). The output is placed in *ret* unless it is `NULL` in which case a new output is created. The *clipmode* argument determines behavior for when entries in *self* are not between 0 and len(*op*).

NPY_RAISE raise a ValueError;

NPY_WRAP wrap values <0 by adding len(*op*) and values >=len(*op*) by subtracting len(*op*) until they are in range;

NPY_CLIP all values are clipped to the region [0, len(*op*)).

PyArray_Sort (`PyObject*`) (`PyArrayObject*` self, `int` axis)

Equivalent to *self*.**sort**(*axis*). Return an array with the items of *self* sorted along *axis*.

PyArray_ArgSort (`PyObject*`) (`PyArrayObject*` self, `int` axis)

Equivalent to *self*.**argsort**(*axis*). Return an array of indices such that selection of these indices along the given `axis` would return a sorted version of *self*. If *self*->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a record array, create a new data-type with a different order of names and construct a view of the array with that new data-type.

PyArray_LexSort (`PyObject*`) (`PyObject*` sort_keys, `int` axis)

Given a sequence of arrays (*sort_keys*) of the same shape, return an array of indices (similar to `PyArray_ArgSort`(...)) that would sort the arrays lexicographically. A lexicographic sort specifies that when two keys are found to be equal, the order is based on comparison of subsequent keys. A merge sort (which leaves equal entries unmoved) is required to be defined for the types. The sort is accomplished by sorting the indices first using the first *sort_key* and then using the second *sort_key* and so forth. This is equivalent to the lexsort(*sort_keys*, *axis*) Python command. Because of the way the merge-sort works, be sure to understand the order the *sort_keys* must be in (reversed from the order you would use when comparing two elements).

If these arrays are all collected in a record array, then `PyArray_Sort`(...) can also be used to sort the array directly.

PyArray_SearchSorted (`PyObject*`) (`PyArrayObject*` self, `PyObject*` values)

Equivalent to *self*.**searchsorted**(*values*). Assuming *self* is a 1-d array in ascending order representing bin boundaries then the output is an array the same shape as *values* of bin numbers, giving the bin into which each item in *values* would be placed. No checking is done on whether or not self is in ascending order.

PyArray_Diagonal (`PyObject*`) (`PyArrayObject*` self, `int` offset, `int` axis1, `int` axis2)

Equivalent to *self*.**diagonal**(*offset*, *axis1*, *axis2*). Return the *offset* diagonals of the 2-d arrays defined by *axis1* and *axis2*.

PyArray_Nonzero (`PyObject*`) (`PyArrayObject*` self)

Equivalent to *self*.**nonzero**(). Returns a tuple of index arrays that select elements of *self* that are nonzero. If (nd=`PyArray_NDIM`(`self`))==1, then a single index array is returned. The index arrays have data type `NPY_INTP`. If a tuple is returned (nd$\neq$1), then its length is nd.

PyArray_Compress (`PyObject*`) (`PyArrayObject*` self, `PyObject*` condition, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**compress**(*condition*, *axis*). Return the elements along *axis* corresponding to elements of *condition* that are true.

13.3.5.4 Calculation

TIP

Pass in NPY_MAXDIMS for axis in order to achieve the same effect that is obtained by passing in axis = None in Python (treating the array as a 1-d array).

PyArray_ArgMax (`PyObject*`) (`PyArrayObject*` self, `int` axis)

Equivalent to *self*.**argmax**(*axis*). Return the index of the largest element of *self* along *axis*.

PyArray_ArgMin (`PyObject*`) (`PyArrayObject*` self, `int` axis)

Equivalent to *self*.**argmin**(*axis*). Return the index of the smallest element of *self* along *axis*.

PyArray_Max (`PyObject*`) (`PyArrayObject*` self, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**max**(*axis*). Return the largest element of *self* along the given *axis*.

PyArray_Min (`PyObject*`) (`PyArrayObject*` self, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**min**(*axis*). Return the smallest element of *self* along the given *axis*.

PyArray_Ptp (`PyObject*`) (`PyArrayObject*` self, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**ptp**(*axis*). Return the difference between the largest element of *self* along *axis* and the smallest element of *self* along *axis*.

NOTE

The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not "large" enough to handle the output. By default, all integer data-types are made at least as large as NPY_LONG for the "add" and "multiply" ufuncs (which form the basis for mean, sum, cumsum, prod, and cumprod functions).

PyArray_Mean (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**mean**(*axis*, *rtype*). Returns the mean of the elements along the given *axis*, using the enumerated type *rtype* as the data type to sum in. Default sum behavior is obtained using `PyArray_NOTYPE` for *rtype*.

PyArray_Trace (`PyObject*`) (`PyArrayObject*` self, `int` offset, `int` axis1, `int` axis2, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**trace**(*offset*, *axis1*, *axis2*, *rtype*). Return the sum (using *rtype* as the data type of summation) over the *offset* diagonal elements of the 2-d arrays defined by *axis1* and *axis2* variables. A positive offset chooses diagonals above the main diagonal. A negative offset selects diagonals below the main diagonal.

PyArray_Clip (`PyObject*`) (`PyArrayObject*` self, `PyObject*` min, `PyObject*` max)

Equivalent to *self*.**clip**(*min*, *max*). Clip an array, *self*, so that values larger than *max* are fixed to *max* and values less than *min* are fixed to *min*.

PyArray_Conjugate (`PyObject*`) (`PyArrayObject*` self)

Equivalent to *self*.**conjugate**() and *self*.**conj**() Return the complex conjugate of *self*. If *self* is not of complex data type, then return *self* with an reference.

PyArray_Round (`PyObject*`) (`PyArrayObject*` self, `int` decimals, `PyArrayObject*` out)

Equivalent to *self*.**round**(*decimals*, *out*). Returns the array with elements rounded to the nearest decimal place. The decimal place is defined as the $10^{-\text{decimals}}$ digit so that negative *decimals* cause rounding to the nearest 10's, 100's, etc. If out is `NULL`, then the output array is created, otherwise the output is placed in *`out`* which must be the correct size and type.

PyArray_Std (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**std**(*axis*, *rtype*). Return the standard deviation using data along *axis* converted to data type *rtype*.

PyArray_Sum (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**sum**(*axis*, *rtype*). Return 1-d vector sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyArray_CumSum (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**cumsum**(*axis*, *rtype*). Return cumulative 1-d sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyArray_Prod (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**prod**(*axis*, *rtype*). Return 1-d products of elements in *self* along *axis*. Perform the product after converting data to data type *rtype*.

PyArray_CumProd (`PyObject*`) (`PyArrayObject*` self, `int` axis, `int` rtype, `PyArrayObject*` out)

Equivalent to *self*.**cumprod**(*axis*, *rtype*). Return 1-d cumulative products of elements in `self` along `axis`. Perform the product after converting data to data type `rtype`.

PyArray_All (`PyObject*`) (`PyArrayObject*` self, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**all**(*axis*). Return an array with True elements for every 1-d sub-array of `self` defined by `axis` in which all the elements are True.

PyArray_Any (`PyObject*`) (`PyArrayObject*` self, `int` axis, `PyArrayObject*` out)

Equivalent to *self*.**any**(*axis*). Return an array with True elements for every 1-d sub-array of *self* defined by *axis* in which any of the elements are True.

13.3.6 Functions

13.3.6.1 Array Functions

PyArray_AsCArray (`int`) (`PyObject**` op, `void*` ptr, `npy_intp*` dims, `int` nd, `int` typenum, `int` itemsize)

Sometimes it is useful to access a multidimensional array as a C-style multidimensional array so that algorithms can be implemented using C's a[i][j][k] syntax. This routine returns a pointer, *ptr*, that simulates this kind of C-style array, for 1-, 2-, and 3-d ndarrays.

op The address to any Python object. This Python object will be replaced with an equivalent well-behaved, C-style contiguous, ndarray of the given data type specifice by the last two arguments. Be sure that stealing a reference in this way to the input object is justified.

ptr The address to a (ctype* for 1-d, ctype** for 2-d or ctype*** for 3-d) variable where ctype is the equivalent C-type for the data type. On return, *ptr* will be addressable as a 1-d, 2-d, or 3-d array.

dims An output array that contains the shape of the array object. This array gives boundaries on any looping that will take place.

nd The dimensionality of the array (1, 2, or 3).

typenum The expected data type of the array.

itemsize This argument is only needed when *typenum* represents a flexible array. Otherwise it should be 0.

The simulation of a C-style array is not complete for 2-d and 3-d arrays. For example, the simulated arrays of pointers cannot be passed to subroutines expecting specific, statically-defined 2-d and 3-d arrays. To pass to functions requiring those kind of inputs, you must statically define the required array and copy data.

PyArray_Free (`int`) (`PyObject*` op, `void*` ptr)

Must be called with the same objects and memory locations returned from `PyArray_AsCArray`(...). This function cleans up memory that otherwise would get leaked.

PyArray_Concatenate (`PyObject*`) (`PyObject*` obj, `int` axis)

Join the sequence of objects in *obj* together along *axis* into a single array. If the dimensions or types are not compatible an error is raised.

PyArray_InnerProduct (`PyObject*`) (`PyObject*` obj1, `PyObject*` obj2)

Compute a product-sum over the last dimensions of *obj1* and *obj2*. Neither array is conjugated.

PyArray_MatrixProduct (`PyObject*`) (`PyObject*` obj1, `PyObject*` obj)

Compute a product-sum over the last dimension of *obj1* and the second-to-last dimension of *obj2*. For 2-d arrays this is a matrix-product. Neither array is conjugated.

PyArray_CopyAndTranspose (`PyObject*`) (`PyObject*` op)

A specialized copy and transpose function that works only for 2-d arrays. The returned array is a transposed copy of *op*.

PyArray_Correlate (`PyObject*`) (`PyObject*` op1, `PyObject*` op2, `int` mode)

Compute the 1-d correlation of the 1-d arrays *op1* and *op2*. The correlation is computed at each output point by multiplying *op1* by a shifted version of *op2* and summing the result. As a result of the shift, needed values outside of the defined range of *op1* and *op2* are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero-values; 1 - return an object that is the same size as *op1*, 2 - return all possible shifts (any overlap at all is accepted).

PyArray_Where (`PyObject*`) (`PyObject*` condition, `PyObject*` x, `PyObject*` y)

If both `x` and `y` are `NULL`, then return `PyArray_Nonzero(`*`condition`*`)`. Otherwise, both *x* and *y* must be given and the object returned is shaped like *condition* and has elements of *x* and *y* where *condition* is respectively True or False.

13.3.6.2 Other functions

PyArray_CheckStrides (`Bool`) (`int` elsize, `int` nd, `npy_intp` numbytes, `npy_intp*` dims, `npy_intp*` newstrides)

Determine if *newstrides* is a strides array consistent with the memory of an *nd*-dimensional array with shape `dims` and element-size, *elsize*. The *newstrides* array is checked to see if jumping by the provided number of bytes in each direction will ever mean jumping more than *numbytes* which is the assumed size of the available memory segment. If *numbytes* is 0, then an equivalent *numbytes* is computed assuming *nd*, *dims*, and *elsize* refer to a single-segment array. Return `NPY_TRUE` if *newstrides* is acceptable, otherwise return `NPY_FALSE`.

PyArray_MultiplyList (`npy_intp`) (`npy_intp*` seq, `int` n)

PyArray_MultiplyIntList (`int`) (`int*` seq, `int` n)

Both of these routines multiply an n-length array, *seq*, of integers and return the result. No overflow checking is performed.

PyArray_CompareLists (`int`) (`npy_intp*` l1, `npy_intp*` l2, `int` n)

Given two n-length arrays of integers, *l1*, and *l2*, return 1 if the lists are identical; otherwise, return 0.

13.3.7 Array Iterators

An array iterator is a simple way to access the elements of an N-dimensional array quickly and efficiently. Section 15.1.1 provides more description and examples of this useful approach to looping over an array.

PyArray_IterNew (`PyObject*`) (`PyObject*` arr)

Return an array iterator object from the array, *arr*. This is equivalent to *arr*.**flat**. The array iterator object makes it easy to loop over an N-dimensional non-contiguous array in C-style contiguous fashion.

PyArray_IterAllButAxis (`PyObject*`) (`PyObject*` arr, `int` *axis)

Return an array iterator that will iterate over all axes but the one provided in **axis*. The returned iterator cannot be used with `PyArray_ITER_GOTO1D`. This iterator could be used to write something similar to what ufuncs do wherein the loop over the largest axis is done by a separate sub-routine. If **axis* is negative then **axis* will be set to the axis having the smallest stride and that axis will be used.

PyArray_BroadcastToShape (`PyObject*`)(`PyObject*` arr, `npy_intp` *dimensions, `int` nd)

Return an array iterator that is broadcast to iterate as an array of the shape provided by *dimensions* and *nd*.

PyArrayIter_Check (`int`) (`PyObject*` op)

Evaluates true if *op* is an array iterator (or instance of a subclass of the array iterator type).

PyArray_ITER_RESET (`void`) (`PyArrayIterObject*` iterator)

Reset an *iterator* to the beginning of the array.

PyArray_ITER_NEXT (`void`) (`PyArrayIterObject*` iterator)

Incremement the index and the dataptr members of the *iterator* to point to the next element of the array. If the array is not (C-style) contiguous, also increment the N-dimensional coordinates array.

PyArray_ITER_GOTO (`void`) (`PyArrayIterObject*` iterator, `npy_intp*` destination)

Set the *iterator* index, dataptr, and coordinates members to the location in the array indicated by the N-dimensional c-array, *destination*, which must have size at least *iterator*->nd_m1+1.

PyArray_ITER_GOTO1D (`PyArrayIterObject*` iterator, `npy_intp` index)

Set the *iterator* index and dataptr to the location in the array indicated by the integer *index* which points to an element in the C-styled flattened array.

13.3.8 Broadcasting (multi-iterators)

PyArray_MultiIterNew (`PyObject*`) (`int` num, ...)

A simplified interface to broadcasting. This function takes the number of arrays to broadcast and then *num* extra (`PyObject*`) arguments. These arguments are converted to arrays and iterators are created. `PyArray_Broadcast` is then called on the resulting multi-iterator object. The resulting, broadcasted mult-iterator object is then returned. A broadcasted operation can then be performed using a single loop and using `PyArray_MultiIter_NEXT`(..)

PyArray_MultiIter_RESET (`void`) (`PyObject*` multi)

Reset all the iterators to the beginning in a multi-iterator object, *multi*.

PyArray_MultiIter_NEXT (`void`) (`PyObject*` multi)

Advance each iterator in a multi-iterator object, *multi*, to its next (broadcasted) element.

PyArray_MultiIter_DATA (`char*`)(`PyObject*` multi, `int` i)

Return the data-pointer of the i^{th} iterator in a multi-iterator object.

PyArray_MultiIter_NEXTi (`void`) (`PyObject*` multi, `int` i)

Advance the pointer of only the i^{th} iterator.

PyArray_MultiIter_GOTO (`void`) (`PyObject*` multi, `npy_intp*` destination)

Advance each iterator in a multi-iterator object, *multi*, to the given N-dimensional *destination* where N is the number of dimensions in the broadcasted array.

PyArray_MultiIter_GOTO1D (void) (`PyObject*` multi, `npy_intp` index)

Advance each iterator in a multi-iterator object, *multi*, to the corresponding location of the *index* into the flattened broadcasted array.

PyArray_Broadcast (`int`) (`PyArrayMultiIterObject*` mit)

This function encapsulates the broadcasting rules. The *mit* container should already contain iterators for all the arrays that need to be broadcast. On return, these iterators will be adjusted so that iteration over each simultaneously will accomplish the broadcasting. A negative number is returned if an error occurs.

PyArray_RemoveSmallest (`int`) (`PyArrayMultiIterObject*` mit)

This function takes a multi-iterator object that has been previously "broadcasted," finds the dimension with the smallest "sum of strides" in the broadcasted result and adapts all the iterators so as not to iterate over that dimension (by effectively making them of length-1 in that dimension). The corresponding dimension is returned unless *mit*->nd is 0, then -1 is returned. This function is useful for constructing ufunc-like routines that broadcast their inputs correctly and then call a strided 1-d version of the routine as the inner-loop. This 1-d version is usually optimized for speed and for this reason the loop should be performed over the axis that won't require large stride jumps.

13.3.9 Array Scalars

PyArray_Return (`PyObject*`) (`PyArrayObject*` arr)

This function checks to see if *arr* is a 0-dimensional array and, if so, returns the appropriate array scalar. It should be used whenever 0-dimensional arrays could be returned to Python.

PyArray_Scalar (`PyObject*`) (`void*` data, `PyArray_Descr*` dtype, `PyObject*` itemsize)

Return an array scalar object of the given enumerated *typenum* and *itemsize* by **copying** from memory pointed to by *data*. If *swap* is nonzero then this function will byteswap the data if appropriate to the data-type because array scalars are always in correct machine-byte order.

PyArray_ToScalar (`PyObject*`) (`void*` data, `PyArrayObject*` arr)

Return an array scalar object of the type and itemsize indicated by the array object *arr* copied from the memory pointed to by *data* and swapping if the data in *arr* is not in machine byte-order.

PyArray_FromScalar (`PyObject*`) (`PyObject*` scalar, `PyArray_Descr*` outcode)

Return a 0-dimensional array of type determined by *outcode* from *scalar* which should be an array-scalar object. If *outcode* is NULL, then the type is determined from *`scalar`*.

PyArray_ScalarAsCtype (`void`) (`PyObject*` scalar, `void*` ctypeptr)

Return in *ctypeptr* a pointer to the actual value in an array scalar. There is no error checking so *scalar* must be an array-scalar object, and ctypeptr must have enough space to hold the correct type. For flexible-sized types, a pointer to the data is copied into the memory of *ctypeptr*, for all other types, the actual data is copied into the address pointed to by *ctypeptr*.

PyArray_CastScalarToCtype (`void`) (`PyObject*` scalar, `void*` ctypeptr, `PyArray_Descr*` outcode)

Return the data (cast to the data type indicated by *outcode*) from the array-scalar, *scalar*, into the memory pointed to by *ctypeptr* (which must be large enough to handle the incoming memory).

PyArray_TypeObjectFromType (`PyObject*`) (`int` type)

Returns a scalar type-object from a type-number, *type*. Equivalent to `PyArray_DescrFromType`(*type*)->typeobj except for reference counting and error-checking. Returns a new reference to the typeobject on success or `NULL` on failure.

PyArray_ScalarKind (`NPY_SCALARKIND`) (`int` typenum, `PyArrayObject**` arr)

Return the kind of scalar represented by *typenum* and the array in **arr* (if *arr* is not `NULL`). The array is assumed to be rank-0 and only used if *typenum* represents a signed integer. If *arr* is not `NULL` and the first element is negative then `NPY_INTNEG_SCALAR` is returned, otherwise `NPY_INTPOS_SCALAR` is returned. The possible return values are `NPY_`<kind>`_SCALAR` where <kind> can be **INTPOS**, **INTNEG**, **FLOAT**, **COMPLEX**, **BOOL**, or **OBJECT**. `NPY_NOSCALAR` is also an enumerated value `NPY_SCALARKIND` variables can take on.

PyArray_CanCoerceScalar (`int`) (`char` thistype, `char` neededtype, `NPY_SCALARKIND` scalar)

Implements the rules for scalar coercion. Scalars are only silently coerced from thistype to neededtype if this function returns nonzero. If scalar is `NPY_NOSCALAR`, then this function is equivalent to `PyArray_CanCastSafely`. The rule is that scalars of the same KIND can be coerced into arrays of the same KIND. This rule means that high-precision scalars will never cause low-precision arrays of the same KIND to be upcast.

13.3.10 Data-type descriptors

WARNING

Data-type objects must be reference counted so be aware of the action on the data-type reference of different C-API calls. The Rule is that when a data-type descriptor object is returned it is a new reference. Functions that take `PyArray_Descr*` objects and return arrays steal references to their inputs unless otherwise noted. Unless you just created the data-type object you must usually increase the reference count of an object passed in describing the data-type.

PyArrayDescr_Check (`int`) (`PyObject*` obj)

Evaluates as true if *obj* is a data-type object (`PyArray_Descr*`).

PyArray_DescrNew (`PyArray_Descr*`) (`PyArray_Descr*` obj)

Return a new data-type object copied from *obj* (the fields reference is just updated so that the new object points to the same fields dictionary if any).

PyArray_DescrNewFromType (`PyArray_Descr*`) (`int` typenum)

Create a new data-type object from the built-in (or user-registered) data-type indicated by *typenum*. All builtin types should not have any of their fields changed. This creates a new copy of the `PyArray_Descr` structure so that you can fill it in as appropriate. This function is especially needed for flexible data-types which need to have a new elsize member in order to be meaningful in array construction.

PyArray_DescrNewByteorder (`PyArray_Descr*`) (`PyArray_Descr*` obj, `char` newendian)

Create a new data-type object with the byteorder set according to *newendian*. All referenced data-type objects (in subdescr and fields members of the data-type object) are also changed (recursively). If a byteorder of `NPY_IGNORE` is encountered it is left alone. If newendian is `NPY_SWAP`, then all byte-orders are swapped. Other valid newendian values are `NPY_NATIVE`, `NPY_LITTLE`, and `NPY_BIG` which all cause the returned data-typed descriptor (and all it's referenced data-type descriptors) to have the corresponding byte-order.

PyArray_DescrFromObject (`PyArray_Descr*`) (`PyObject*` op, `PyArray_Descr*` mintype)

Determine an appropriate data-type object from the object *op* (which should be a "nested" sequence object) and the minimum data-type descriptor mintype (which can be `NULL`). Similar in behavior to array(*op*).dtype. Don't confuse this function with `PyArray_DescrConverter`. This function essentially looks at all the objects in the (nested) sequence and determines the data-type from the elements it finds.

PyArray_DescrFromScalar (`PyArray_Descr*`) (`PyObject*` scalar)

Return a data-type object from an array-scalar object. No checking is done to be sure that *scalar* is an array scalar. If no suitable data-type can be determined, then a data-type of NPY_OBJECT is returned by default.

PyArray_DescrFromType (`PyArray_Descr*`) (`int` typenum)

Returns a data-type object corresponding to *typenum*. The *typenum* can be one of the enumerated types, a character code for one of the enumerated types, or a user-defined type.

PyArray_DescrConverter (`int`) (`PyObject*` obj, `PyArray_Descr**` dtype)

Convert any compatible Python object, *obj*, to a data-type object in *dtype*. A large number of Python objects can be converted to data-type objects. See Chapter 7 for a complete description. This version of the converter converts None objects to a `NPY_DEFAULT_TYPE` data-type object. This function can be used with the "O&" character code in PyArg_ParseTuple processing.

PyArray_DescrConverter2 (`int`) (`PyObject*` obj, `PyArray_Descr**` dtype)

Convert any compatible Python object, *obj*, to a data-type object in *dtype*. This version of the converter converts None objects so that the returned data-type is `NULL`. This function can also be used with the "O&" character in PyArg_ParseTuple processing.

Pyarray_DescrAlignConverter (`int`) (`PyObject*` obj, `PyArray_Descr**` dtype)

Like `PyArray_DescrConverter` except it aligns C-struct-like objects on word-boundaries as the compiler would.

Pyarray_DescrAlignConverter2 (`int`) (`PyObject*` obj, `PyArray_Descr**` dtype)

Like `PyArray_DescrConverter2` except it aligns C-struct-like objects on word-boundaries as the compiler would.

PyArray_FieldNames (`PyObject*`)(`PyObject*` dict)

Take the fields dictionary, ***`dict`***, such as the one attached to a data-type object and construct an ordered-list of field names such as is stored in the names field of the `PyArray_Descr` object.

13.3.11 Conversion Utilities

13.3.11.1 For use with `PyArg_ParseTuple`

All of these functions can be used in `PyArg_ParseTuple`(...) with the "O&" format specifier to automatically convert any Python object to the required C-object. All of these functions return `NPY_SUCCEED` if successful and `NPY_FAIL` if not. The first argument to all of these function is a Python object. The second argument is the **address** of the C-type to convert the Python object to.

WARNING

Be sure to understand what steps you should take to manage the memory when using these conversion functions. These functions can require freeing memory, incrementing or decrementing reference counts of specific objects based on your use.

PyArray_Converter (`int`) (`PyObject*` obj, `PyObject**` address)

Convert any Python object to a `PyArrayObject`. If `PyArray_Check`(*`obj`*) is TRUE then its reference count is incremented and a reference placed in *address*. If *obj* is not an array, then convert it to an array using `PyArray_FromAny`. No matter what is returned, you must DECREF the object returned by this routine in *address* when you are done with it.

PyArray_OutputConverter (`int`)(`PyObject*` obj, `PyArrayObject**` address)

This is a default converter for output arrays given to functions. If *obj* is `Py_None` or `NULL`, then **address* will be `NULL` but the call will succeed.

If `PyArray_Check`(*obj*) is TRUE then it is returned in **address* without incrementing its reference count.

PyArray_IntpConverter (`int`) (`PyObject*` obj, `PyArray_Dims*` seq)

Convert any Python sequence, *obj*, smaller than `NPY_MAXDIMS` to an array of `npy_intp's`. The Python sequence could also be a single number. The *seq* variable is a pointer to a structure with members ptr and len. On successful return, *seq*->ptr contains a pointer to memory that must be freed to avoid a memory leak. The restriction on memory size allows this converter to be conveniently used for sequences intended to be interpreted as array shapes.

PyArray_BufferConverter (`int`) (`PyObject*` obj, `PyArray_Chunk*` buf)

Convert any Python object, *obj*, with a (single-segment) buffer interface to a variable with members that detail the object's use of its chunk of memory. The *buf* variable is a pointer to a structure with base, ptr, len, and flags members. The `PyArray_Chunk` structure is binary compatibile with the Python's buffer object (through its len member on 32-bit platforms and its ptr member on 64-bit platforms or in Python 2.5). On return, the base member is set to *obj* (or its base if *obj* is already a buffer object pointing to another object). If you need to hold on to the memory be sure to INCREF the base member. The chunk of memory is pointed to by *buf*->ptr member and has length *buf*->len. The flags member of *buf* is `NPY_BEHAVED_RO` with the `NPY_WRITEABLE` flag set if *obj* has a writeable buffer interface.

PyArray_AxisConverter (`int`) (`PyObject*` obj, `int*` axis)

Convert a Python object, *obj*, representing an axis argument to the proper value for passing to the functions that take an integer axis. Specifically, if *obj* is None, set *axis* to `NPY_MAXDIMS` which is interpreted by all these C-API functions correctly.

PyArray_BoolConverter (`int`) (`PyObject*` obj, `Bool*` value)

Convert any Python object, *obj*, to `NPY_TRUE` or `NPY_FALSE`, and place the result in *value*.

PyArray_ByteorderConverter (`int`) (`PyObject*` obj, `char*` endian)

Convert Python strings into the corresponding byte-order character: '>', '<', 's', '=', or '|'.

PyArray_SortkindConverter (`int`) (`PyObject*` obj, `NPY_SORTKIND*` sort)

Convert Python strings into one of `NPY_QUICKSORT` (starts with 'q' or 'Q') , `NPY_HEAPSORT` (starts with 'h' or 'H'), or `NPY_MERGESORT` (starts with 'm' or 'M').

PyArray_SearchsideConverter (`int`) (`PyObject*` obj, `NPY_SEARCHSIDE*` side)

Convert Python strings into one of `NPY_SEARCHLEFT` (starts with 'l' or 'L'), or `NPY_SEARCHRIGHT` (starts with 'r' or 'R').

13.3.11.2 Other conversions

PyArray_PyIntAsInt (`int`) (`PyObject*` op)

Convert all kinds of Python objects (including arrays and array scalars) to a standard integer. On error, -1 is returned and an exception set. You may find useful the macro:

```
#define error_converting(x) (((x) == -1) && PyErr_Occurred()
```

PyArray_PyIntAsIntp (`npy_intp`) (`PyObject*` op)

Convert all kinds of Python objects (including arrays and array scalars) to a (platform-pointer-sized) integer. On error, -1 is returned and an exception set.

PyArray_IntpFromSequence (`int`) (`PyObject*` seq, `npy_intp*` vals, `int` maxvals)

Convert any Python sequence (or single Python number) passed in as *seq* to (up to) *maxvals* pointer-sized integers and place them in the *vals* array. The sequence can be smaller then *maxvals* as the number of converted objects is returned.

PyArray_TypestrConvert (`int`) (`int` itemsize, `int` gentype)

Convert typestring characters (with *itemsize*) to basic enumerated data types. The typestring character corresponding to signed and unsigned integers, floating point numbers, and complex-floating point numbers are recognized and converted. Other values of gentype are returned. This function can be used to convert, for example, the string 'f4' to `NPY_FLOAT32`.

13.3.12 Miscellaneous

13.3.12.1 Importing the API

In order to make use of the C-API from another extension module, the `import_array`() command must be used. If the extension module is self-contained in a single .c file, then that is all that needs to be done. If however, the extension module involve multiple files where the C-API is needed then some additional steps must be taken.

import_array (void) (void)

This function must be called in the initialization section of a module that will make use of the C-API. It imports the module where the function-pointer table is stored and points the correct variable to it.

PY_ARRAY_UNIQUE_SYMBOL

NO_IMPORT_ARRAY

Using these #defines you can use the C-API in multiple files for a single extension module. In each file you must define `PY_ARRAY_UNIQUE_SYMBOL` to some name that will hold the C-API (*e.g.* myextension_ARRAY_API). This must be done **before** including the numpy/arrayobject.h file. In the module intialization routine you call `import_array`(). In addition, in the files that do not have the module initialization sub_routine define `NO_IMPORT_ARRAY` prior to including numpy/arrayobject.h.

Suppose I have two files coolmodule.c and coolhelper.c which need to be compiled and linked into a single extension module. Suppose coolmodule.c contains the required initcool module initialization function (with the import_array() function called). Then, coolmodule.c would have at the top:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#define NO_IMPORT_ARRAY
#include numpy/arrayobject.h
```

PyArray_GetNDArrayCVersion (`unsigned int`) (`void`)

This just returns the value `NPY_VERSION`. Because it is in the C-API, however, comparing the output of this function from the value defined in the current header gives a way to test if the C-API has changed thus requiring a re-compilation of extension modules that use the C-API.

13.3.12.2 Internal Flexibility

PyArray_SetNumericOps (`int`) (`PyObject*` dict)

NumPy stores an internal table of Python callable objects that are used to implement arithmetic operations for arrays as well as certain array calculation methods. This function allows the user to replace any or all of these Python objects with their own versions. The keys of the dictionary, *dict*, are the named functions to replace and the paired value is the Python callable object to use. Care should be taken that the function used to replace an internal array operation does not itself call back to that internal array operation (unless you have designed the function to handle that), or an unchecked infinite recursion can result (possibly causing program crash). The key names that represent operations that can be replaced are:

add, **subtract**, **multiply**, **divide**, **remainder**, **power**, **sqrt**, **negative**, **absolute**, **invert**, **left_shift**, **right_shift**, **bitwise_and**, **bitwise_xor**, **bitwise_or**, **less**, **less_equal**, **equal**, **not_equal**, **greater**, **greater_equal**, **floor_divide**, **true_divide**, **logical_or**, **logical_and**, **floor**, **ceil**, **maximum**, **minimum**.

These functions are included here because they are used at least once in the array object's methods. The function returns -1 (without setting a Python Error) if one of the objects being assigned is not callable.

PyArray_GetNumericOps (`PyObject*`) (`void`)

Return a Python dictionary containing the callable Python objects stored in the the internal arithmetic operation table. The keys of this dictionary are given in the explanation for `PyArray_SetNumericOps`.

PyArray_SetStringFunction (`void`) (`PyObject*` op, `int` repr)

This function allows you to alter the tp_str and tp_repr methods of the array object to any Python function. Thus you can alter what happens for all arrays when str(arr) or repr(arr) is called from Python. The function to be called is passed in as *op*. If *repr* is non-zero, then this function will be called in response to repr(arr), otherwise the function will be called in response to str(arr). No check on whether or not *op* is callable is performed. The callable passed in to *op* should expect an array argument and should return a string to be printed.

13.3.12.3 Memory management

PyDataMem_NEW (`char*`) (`size_t` nbytes)

PyDataMem_FREE (`char*` ptr)

PyDataMem_RENEW (`char*`) (`void *` ptr, `size_t` newbytes)

Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.

PyDimMem_NEW (`npy_intp*`) (nd)

PyDimMem_FREE (`npy_intp*` ptr)

PyDimMem_RENEW (`npy_intp*`) (`npy_intp*` ptr, `npy_intp` newnd)

Macros to allocate, free, and reallocate dimension and strides memory.

PyArray_malloc (nbytes)

PyArray_free (ptr)

PyArray_realloc (ptr, nbytes)

These macros use different memory allocators, depending on the constant `NPY_USE_PYMEM`. Currently, the system malloc or the Python Object allocator can be used.

13.3.12.4 Threading support

These macros are only meaningful if `NPY_ALLOW_THREADS` evaluates True during compilation of the extension module. Otherwise, these macros are equivalent to whitespace. Python uses a single Global Interpreter Lock (GIL) for each Python process so that only a single thread may excecute at a time (even on multi-cpu machines). When calling out to a compiled function that may take time to compute (and does not have side-effects for other threads like updated global variables), the GIL should be released so that other Python threads can run while the time-consuming calculations are performed. This can be accomplished using two groups of macros. Typically, if one macro in a group is used in a code block, all of them must be used in the same code block. Currently, `NPY_ALLOW_THREADS` is defined to the python-defined `WITH_THREADS` constant unless the environment variable `NPY_NOSMP` is set in which case `NPY_ALLOW_THREADS` is defined to be 0.

Group 1 This group is used to call code that may take some time but does not use any Python C-API calls. Thus, the GIL should be released during its calculation.

NPY_BEGIN_ALLOW_THREADS Equivalent to `Py_BEGIN_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

NPY_END_ALLOW_THREADS Equivalent to `Py_END_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

NPY_BEGIN_THREADS_DEF Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

NPY_BEGIN_THREADS Place right before code that does not need the Python interpreter (no Python C-API calls). This macro saves the Python state and releases the GIL.

NPY_END_THREADS Place right after code that does not need the Python interpreter. This macro acquires the GIL and restores the Python state from the saved variable.

NPY_BEGIN_THREADS_DESCR (`PyArray_Descr*` dtype) Useful to release the GIL only if *dtype* does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop. Equivalent to

NPY_END_THREADS_DESCR (`PyArray_Descr*` dtype) Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

Group 2 This group is used to re-acquire the Python GIL after it has been released. For example, suppose the GIL has been released (using the previous calls), and then some path in the code (perhaps in a different subroutine) requires use of the Python C-API, then these macros are useful to acquire the GIL.

These macros accomplish essentially a reverse of the previous three (acquire the LOCK saving what state it had) and then re-release it with the saved state.

NPY_ALLOW_C_API_DEF Place in the variable declaration area to set up the necessary variable.

NPY_ALLOW_C_API Place before code that needs to call the Python C-API (when it is known that the GIL has already been released).

NPY_DISABLE_C_API Place after code that needs to call the Python C-API (to re-release the GIL).

TIP

Never use semicolons after the threading support macros.

13.3.12.5 Priority

NPY_PRIOIRTY Default priority for arrays.

NPY_SUBTYPE_PRIORITY Default subtype priority.

NPY_SCALAR_PRIORITY Default scalar priority (very small)

PyArray_GetPriority (`double`) (`PyObject*` obj, `double` def)

Return the **__array_priority__** attribute (converted to a double) of *obj* or *def* if no attribute of that name exists. Fast returns that avoid the attribute lookup are provided for objects of type `PyArray_Type`.

13.3.12.6 Default buffers

NPY_BUFSIZE Default size of the user-settable internal buffers.

NPY_MIN_BUFSIZE Smallest size of user-settable internal buffers.

NPY_MAX_BUFSIZE Largest size allowed for the user-settable buffers.

13.3.12.7 Other constants

NPY_NUM_FLOATTYPE The number of floating-point types

NPY_MAXDIMS The maximum number of dimensions allowed in arrays.

NPY_VERSION The current version of the ndarray object (check to see if this variable is defined to guarantee the numpy/arrayobject.h header is being used).

NPY_FALSE Defined as 0 for use with Bool.

NPY_TRUE Defined as 1 for use with Bool.

NPY_FAIL The return value of failed converter functions which are called using the "O&" syntax in PyArg_ParseTuple-like functions.

NPY_SUCCEED The return value of successful converter functions which are called using the "O&" syntax in PyArg_ParseTuple-like functions.

13.3.12.8 Miscellaneous Macros

PyArray_SAMESHAPE (a1, a2)

Evaluates as True if arrays *a1* and *a2* have the same shape.

PyArray_MAX (a,b)

Returns the maximum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

PyArray_MIN (a,b)

Returns the minimum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

PyArray_REFCOUNT (`PyObject*` op)

Returns the reference count of any Python object.

PyArray_XDECREF_ERR (PyObject *obj)

DECREF's an array object which may have the `NPY_UPDATEIFCOPY` flag set without causing the contents to be copied back into the original array. Resets the `NPY_WRITEABLE` flag on the base object. This is useful for recovering from an error condition when `NPY_UPDATEIFCOPY` is used.

13.3.12.9 Enumerated Types

NPY_SORTKIND A special variable-type which can take on the values **NPY_**<KIND> where <KIND> is

QUICKSORT, **HEAPSORT**, **MERGESORT**

NPY_NSORTS is defined to be the number of sorts.

NPY_SCALARKIND A special variable type indicating the number of "kinds" of scalars distinguished in determining scalar-coercion rules. This variable can take on the values NPY_<KIND> where <KIND> can be

NOSCALAR, **BOOL_SCALAR**, **INTPOS_SCALAR**, **INTNEG_SCALAR**, **FLOAT_SCALAR**, **COMPLEX_SCALAR**, **OBJECT_SCALAR**

NPY_NSCALARKINDS is defined to be the number of scalar kinds (not including `NPY_NOSCALAR`).

NPY_ORDER A variable type indicating the order that an array should be interpreted in. The value of a variable of this type can be **NPY_**<ORDER> where <ORDER> is

ANYORDER, **CORDER**, **FORTRANORDER**

NPY_CLIPMODE A variable type indicating the kind of clipping that should be applied in certain functions. The value of a variable of this type can be **NPY_**<MODE> where <MODE> is

CLIP, **WRAP**, **RAISE**

13.4 UFunc API

13.4.1 Constants

UFUNC_ERR_<HANDLER>

<HANDLER> can be **IGNORE**, **WARN**, **RAISE**, or **CALL**

UFUNC_<THING>_<ERR>

<THING> can be **MASK**, **SHIFT**, or **FPE**, and <ERR> can be **DIVIDEBYZERO**, **OVERFLOW**, **UNDERFLOW**, and **INVALID**.

PyUFunc_<VALUE> <VALUE> can be **One** (1), **Zero** (0), or **None** (-1)

13.4.2 Macros

NPY_LOOP_BEGIN_THREADS

Used in universal function code to only release the Python GIL if loop->obj is not true (*i.e.* this is not an OBJECT array loop). Requires use of `NPY_BEGIN_THREADS_DEF` in variable declaration area.

NPY_LOOP_END_THREADS

Used in universal function code to re-acquire the Python GIL if it was released (because loop->obj was not true).

UFUNC_CHECK_ERROR (loop)

A macro used internally to check for errors and goto fail if found. This macro requires a fail label in the current code block. The *loop* variable must have at least members (obj, errormask, and errorobj). If *loop*->obj is nonzero, then `PyErr_Occurred`() is called (meaning the GIL must be held). If *loop*->obj is zero, then if *loop*->errormask is nonzero, `PyUFunc_checkfperr` is called with arguments *loop*->errormask and *loop*->errobj. If the result of this check of the IEEE floating point registers is true then the code redirects to the fail label which must be defined.

UFUNC_CHECK_STATUS (*ret*)

A macro that expands to platform-dependent code. The *ret* variable can can be any integer. The `UFUNC_FPE_`<ERR> bits are set in *ret* according to the status of the corresponding error flags of the floating point processor.

13.4.3 Functions

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

Create a new broadcasting universal function from required variables. Each ufunc builds around the notion of an element-by-element operation. Each ufunc object contains pointers to 1-d loops implementing the basic functionality for each supported type.

nin The number of inputs to this operation.

nout The number of outputs

ntypes How many different data-type "signatures" the ufunc has implemented.

func Must to an array of length *ntypes* containing `PyUFuncGenericFunction` items. These items are pointers to functions that acutally implement the underlying (element-by-element) function N times. T

types Must be of length (*nin*+*nout*)***ntypes*, and it contains the data-types (built-in only) that the corresponding function in the *func* array can deal with.

data Should be `NULL` or a pointer to an array of size *ntypes*. This array may contain arbitrary extra-data to be passed to the corresponding 1-d loop function in the func array.

name The name for the ufunc.

doc Allows passing in a documentation string to be stored with the ufunc. The documentation string should not contain the name of the function or the calling signature as that will be dynamically determined from the object and available when accessing the **__doc__** attribute of the ufunc.

check_return Unused and present for backwards compatibility of the C-API. A corresponding *check_return* integer does exist in the ufunc structure and it does get set with this value when the ufunc object is created.

PyUFunc_RegisterLoopForType (`int`) (`PyUFuncObject*` ufunc, `int` usertype, `PyUFuncGenericFunction` function, `int*` arg_types, `void*` data)

This function allows the user to register a 1-d loop with an already-created ufunc to be used whenever the ufunc is called with any of its input arguments as the user-defined data-type. This is needed in order to make ufuncs work with built-in data-types. The data-type must have been previously registered with the numpy system. The loop is passed in as *function*. This loop can take arbitrary data which should be passed in as *data*. The data-types the loop requires are passed in as *arg_types* which must be a pointer to memory at least as large as ufunc->nargs.

PyUFunc_ReplaceLoopBySignature (`int`) (`PyUFuncObject*` ufunc, `PyUFuncGenericFunction` newfunc, `int*` signature, `PyUFuncGenericFunction*` oldfunc)

Replace a 1-d loop matching the given *signature* in the already-created *ufunc* with the new 1-d loop newfunc. Return the old 1-d loop function in *oldfunc*. Return 0 on success and -1 on failure. This function works only with built-in types (use `PyUFunc_RegisterLoopForType` for user-defined types). A signature is an array of data-type numbers indicating the inputs followed by the outputs assumed by the 1-d loop.

PyUFunc_GenericFunction (`int`) (`PyUFuncObject*` self, `PyObject*` args, `PyArrayObject**` mps)

A generic ufunc call. The ufunc is passed in as *self*, the arguments to the ufunc as *args*. The *mps* argument is an array of `PyArrayObject` pointers containing the converted input arguments as well as the ufunc outputs on return. The user is responsible for managing this array and receives a new reference for each array in *mps*. The total number of arrays in *mps* is given by *self*->nin + *self*->nout.

PyUFunc_checkfperr (`int`) (`int` errmask, `PyObject*` errobj)

A simple interface to the IEEE error-flag checking support. The *errmask* argument is a mask of `UFUNC_MASK_<ERR>` bitmasks indicating which errors to check for (and how to check for them). The *errobj* must be a Python tuple with two elements: a string containing the name which will be used in any communication of error and either a callable Python object (call-back function) or `Py_None`. The callable object will only be used if `UFUNC_ERR_CALL` is set as the desired error checking method. This routine manages the GIL and is safe to call even after releasing the GIL. If an error in the IEEE-compatibile hardware is determined a -1 is returned, otherwise a 0 is returned.

PyUFunc_clearfperr (`void`) ()

Clear the IEEE error flags.

PyUFunc_GetPyValues (`void`) (`char*` name, `int*` bufsize, `int*` errmask, `PyObject**` errobj)

Get the Python values used for ufunc processing from the thread-local storage area unless the defaults have been set in which case the name lookup is bypassed. The name is placed as a string in the first element of **errobj*. The second element is the looked-up function to call on error callback. The value of the looked-up buffer-size to use is passed into *bufsize*, and the value of the error mask is placed into *errmask*.

13.4.4 Generic functions

At the core of every ufunc is a collection of type-specific functions that defines the basic functionality for each of the supported types. These functions must evaluate the underlying function $N \geq 1$ times. Extra-data may be passed in that may be used during the calculation. This feature allows some general functions to be used as these basic looping functions. The general function has all the code needed to point variables to the right place and set up a function call. The general function assumes that the actual function to call is passed in as the extra data and calls it with the correct values. All of these functions are suitable for placing directly in the array of functions stored in the functions member of the PyUFuncObject structure.

PyUFunc_f_f_As_d_d (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_d_d (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_f_f (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_g_g (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_F_F_As_D_D (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_F_F (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_D_D (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_G_G (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking one input argument and returning one output. This function is passed in `func`. The letters correspond to dtypechar's of the supported data types (`f` - float, `d` - double, `g` - long double, `F` - cfloat, `D` - cdouble, `G` - clongdouble). The argument *func* must support the same signature. The _As_X_X variants assume ndarray's of one data type but cast the values to use an underlying function that takes a different data type. Thus, `PyUFunc_f_f_As_d_d` uses ndarrays of data type `NPY_FLOAT` but calls out to a C-function that takes double and returns double.

PyUFunc_ff_f_As_dd_d (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_ff_f (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_dd_d (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_gg_g (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_FF_F_As_DD_D (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_DD_D (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_FF_F (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_GG_G (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking two input arguments and returning one output. The underlying function to call is passed in as *func*. The letters correspond to dtypechar's of the specific data type supported by the general-purpose function. The argument `func` must support the corresponding signature. The `_As_XX_X` variants assume ndarrays of one data type but cast the values at each iteration of the loop to use the underlying function that takes a different data type.

PyUFunc_O_O (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

PyUFunc_OO_O (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

One-input, one-output, and two-input, one-output core 1-d functions for the `NPY_OBJECT` data type. These functions handle reference count issues and return early on error. The actual function to call is *func* and it must accept calls with the signature (`PyObject*`) (`PyObject*`) for `PyUFunc_O_O` or (`PyObject*`) (`PyObject` *, `PyObject` *) for `PyUFunc_OO_O`.

PyUFunc_O_O_method (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object. For each iteration of the loop, the Python obejct is extracted from the array and its *func* method is called returning the result to the output array.

PyUFunc_OO_O_method (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object that takes one argument. The first argument in *args* is the method whose function is called, the second argument in *args* is the argument passed to the function. The output of the function is stored in the third entry of *args*.

PyUFunc_On_Om (`void`) (`char**` args, `npy_intp*` dimensions, `npy_intp*` steps, `void*` func)

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case *func* is a pointer to a `PyUFunc_PyFuncData` structure which has definition {`int` nin; `int` nout; `PyObject*` callable}. At each iteration of the loop, the *nin* input objects are exctracted from their object arrays and placed into an argument tuple, the Python *callable* is called with the input arguments, and the nout outputs are placed into their object arrays.

13.5 Importing the API

PY_UFUNC_UNIQUE_SYMBOL

NO_IMPORT_UFUNC

import_ufunc (`void`) (`void`)

These are the constants and functions for accessing the ufunc C-API from extension modules in precisely the same way as the array C-API can be accessed. The `import_ufunc`() function must always be called (in the initialization subroutine of the extension module). If your extension module is in one file then that is all that is required. The other two constants are useful if your extension module makes use of multiple files. In that case, define `PY_UFUNC_UNIQUE_SYMBOL` to something unique to your code and then in source files that do not contain the module initialization function but still need access to the UFUNC API, define `PY_UFUNC_UNIQUE_SYMBOL` to the same name used previously and also define `NO_IMPORT_UFUNC`.

The C-API is actually an array of function pointers. This array is created (and pointed to by a global variable) by import_ufunc. The global variable is either statically defined or allowed to be seen by other files depending on the state of `Py_UFUNC_UNIQUE_SYMBOL` and `NO_IMPORT_UFUNC`.

Chapter 14

How to extend NumPy

14.1 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 `http://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.

14.2 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:

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

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

PyModule_AddStringConstant (`int`) (`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.

14.3 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.

14.3.1 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.

14.3.2 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.

14.3.3 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 mishandle 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.

14.4 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.

14.4.1 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

PyArray_FROM_OTF (`PyObject*`) (`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.

14.4.2 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.

PyArray_SimpleNew (`PyObject*`)(`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).

PyArray_SimpleNewFromData (`PyObject*`) (`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.

14.4.3 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 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.

14.5 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;
}
```

Chapter 15

Beyond the Basics

15.1 Iterating over elements in the array

15.1.1 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++;
}
```

15.1.2 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 over-head 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$^{\text{th}}$ dimension. If dim is negative, then the dimension with the largest axis is found and used.

15.1.3 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);
}
```

15.1.4 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.

15.2 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.

PyUFunc_FromFuncAndData (`PyObject*`) (`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, voi
{
    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

```
char my_sigs[] = \
{NPY_INT, NPY_INT, NPY_CDOUBLE,
NPY_DOUBLE, NPY_DOUBLE, NPY_CDOUBLE};
```

The bit-width names can also be used (e.g. `NPY_INT32`, `NPY_COMPLEX128`) if desired.

ntypes The number of data-types supported. This is equal to the number of 1-d loops provided.

nin The number of input arguments.

nout The number of output arguments.

identity Either **PyUFunc_One**, **PyUFunc_Zero**, **PyUFunc_None**. This specifies what should be returned when an empty array is passed to the reduce method of the ufunc.

name A `NULL`-terminated string providing the name of this ufunc (should be the Python name it will be called).

doc A documentation string for this ufunc (will be used in generating the response to <ufunc_name>.__doc__). Do not include the function signature or the name as this is generated automatically.

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);
    ...
```

15.3 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.

15.3.1 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**).

15.3.2 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.

castfunc (`void`) (`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);
```

15.3.3 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**.

15.3.4 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.

PyUFunc_RegisterLoopForType (`int`) (`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 15.2.

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.

15.4 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.

15.4.1 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).

15.4.2 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.

15.4.2.1 The __array_finalize__ method

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.

15.4.2.2 The __array_priority__ attribute

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_prioirty__** 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.

15.4.2.3 The __array_wrap__ method

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.

Chapter 16

Using Python as glue

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.

16.1 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.

16.2 Hand-generated wrappers

Extension modules were discussed in Chapter 14.1. 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.

16.3 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).

16.3.1 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.

16.3.2 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
```

16.3.3 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.

16.3.4 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.

16.3.5 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.

16.3.6 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.

16.3.7 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.

16.3.8 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.

16.4 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.

16.4.1 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.

16.4.2 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.

16.4.3 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

16.4.4 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.

16.5 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.

16.5.1 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.

16.5.2 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)
```

16.5.3 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).

16.6 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.

16.6.1 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.

16.6.2 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.

16.6.3 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).

16.6.4 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.

16.6.5 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
```

16.6.6 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.

16.7 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.

16.7.1 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.

16.7.2 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.

16.7.3 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.

16.7.4 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"],
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.

16.7.5 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.

16.7.6 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.

Chapter 17

Code Explanations

This Chapter attempts to explain the logic behind some of the new pieces of code. The purpose behind these explanations is to enable somebody to be able to understand the ideas behind the implementation somewhat more easily than just staring at the code. Perhaps in this way, the algorithms can be improved on, borrowed from, and/or optimized.

17.1 Memory model

One fundamental aspect of the ndarray is that an array is seen as a “chunk” of memory starting at some location. The interpretation of this memory depends on the stride information. For each dimension in an N-dimensional array, an integer (stride) dictates how many bytes must be skipped to get to the next element in that dimension. Unless you have a single-segment array, this stride information must be consulted when traversing through an array. It is not difficult to write code that accepts strides, you just have to use (char *) pointers because strides are in units of bytes. Keep in mind also that strides do not have to be unit-multiples of the element size. Also, remember that if the number of dimensions of the array is 0 (sometimes called a rank-0 array), then the strides and dimensions variables are NULL.

Besides the structural information contained in the strides and dimensions members of the PyArrayObject, the flags contain important information about how the data may be accessed. In particular, the NPY_ALIGNED flag is set when the memory is on a suitable boundary according to the data-type array. Even if you have a contiguous chunk of memory, you cannot just assume it is safe to dereference a data-type-specific pointer to an element. Only if the NPY_ALIGNED flag is set is this a safe operation (on some platforms it will work but on others, like Solaris, it will cause a bus error). The NPY_WRITEABLE should also be ensured if you plan on writing to the memory area of the array. It is also possible to obtain a pointer to an unwriteable memory area. Sometimes, writing to the memory area when the NPY_WRITEABLE flag is not set will just be rude. Other times it can cause program crashes (*e.g.* a data-area that is a read-only memory-mapped file).

17.2 Data-type encapsulation

The data-type is an important abstraction of the ndarray. Operations will look to the data-type to provide the key functionality that is needed to operate on the array. This functionality is provided in the list of function pointers pointed to by the 'f' member of the PyArray_Descr structure. In this way, the number of data-types can be extended simply by providing a PyArray_Descr structure with suitable function pointers in the 'f' member. For built-in types there are some optimizations that by-pass this mechanism, but the point of the data-type abstraction is to allow new data-types to be added.

One of the built-in data-types, the void data-type allows for arbitrary records containing 1 or more fields as elements of the array. A field is simply another data-type object along with an offset into the current record. In order to support arbitrarily nested fields, several recursive implementations of data-type access are implemented for the void type. A common idiom is to cycle through the elements of the dictionary and perform a specific operation based on the data-type object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering mis-aligned data must be recognized and taken into account if necessary.

17.3 N-D Iterators

A very common operation in much of NumPy code is the need to iterate over all the elements of a general, strided, N-dimensional array. This operation of a general-purpose N-dimensional loop is abstracted in the notion of an iterator object. To write an N-dimensional loop, you only have to create an iterator object from an ndarray, work with the dataptr member of the iterator object structure and call the macro PyArray_ITER_NEXT(it) on the iterator object to move to the next element. The "next" element is always in C-contiguous order. The macro works by first special casing the C-contiguous, 1-d, and 2-d cases which work very simply.

For the general case, the iteration works by keeping track of a list of coordinate counters in the iterator object. At each iteration, the last coordinate counter is increased (starting from 0). If this counter is smaller then one less than the size of the array in that dimension (a pre-computed and stored value), then the counter is increased and the dataptr member is increased by the strides in that dimension and the macro ends. If the end of a dimension is reached, the counter for the last dimension is reset to zero and the dataptr is moved back to the beginning of that dimension by subtracting the strides value times one less than the number of elements in that dimension (this is also pre-computed and stored in the backstrides member of the iterator object). In this case, the macro does not end, but a local dimension counter is decremented so that the next-to-last dimension replaces the role that the last dimension played and the previously-described tests are executed again on the next-to-last dimension. In this way, the dataptr is adjusted appropriately for arbitrary striding.

The coordinates member of the PyArrayIterObject structure maintains the current N-d counter unless the underlying array is C-contiguous in which case the coordinate counting is by-passed. The index member of the PyArrayIterObject keeps track of the current flat index of the iterator. It is updated by the PyArray_ITER_NEXT macro.

17.4 Broadcasting

In Numeric, broadcasting was implemented in several lines of code buried deep in ufuncobject.c. In NumPy, the notion of broadcasting has been abstracted so that it can be performed in multiple places. Broadcasting is handled by the function PyArray_Broadcast. This function requires a PyArrayMultiIterObject (or something that is a binary equivalent) to be passed in. The PyArrayMultiIterObject keeps track of the broadcasted number of dimensions and size in each dimension along with the total size of the broadcasted result. It also keeps track of the number of arrays being broadcast and a pointer to an iterator for each of the arrays being broadcasted.

The PyArray_Broadcast function takes the iterators that have already been defined and uses them to determine the broadcast shape in each dimension (to create the iterators at the same time that broadcasting occurs then use the PyMuliIter_New function). Then, the iterators are adjusted so that each iterator thinks it is iterating over an array with the broadcasted size. This is done by adjusting the iterators number of dimensions, and the shape in each dimension. This works because the iterator strides are also adjusted. Broadcasting only adjusts (or adds) length-1 dimensions. For these dimensions, the strides variable is simply set to 0 so that the data-pointer for the iterator over that array doesn't move as the broadcasting operation operates over the extended dimension.

Broadcasting was always implemented in Numeric using 0-valued strides for the extended dimensions. It is done in exactly the same way in NumPy. The big difference is that now the array of strides is kept track of in a PyArrayIterObject, the iterators involved in a broadcasted result are kept track of in a PyArrayMultiIterObject, and the PyArray_BroadCast call implements the broad-casting rules.

17.5 Array Scalars

The array scalars offer a hierarchy of Python types that allow a one-to-one correspondence between the data-type stored in an array and the Python-type that is returned when an element is extracted from the array. An exception to this rule was made with object arrays. Object arrays are heterogeneous collections of arbitrary Python objects. When you select an item from an object array, you get back the original Python object (and not an object array scalar which does exist but is rarely used for practical purposes).

The array scalars also offer the same methods and attributes as arrays with the intent that the same code can be used to support arbitrary dimensions (including 0-dimensions). The array scalars are read-only (immutable) with the exception of the void scalar which can also be written to so that record-array field setting works more naturally (a[0]['f1'] = `value`).

17.6 Advanced ("Fancy") Indexing

The implementation of advanced indexing represents some of the most difficult code to write and explain. In fact, there are two implementations of advanced indexing. The first works only with 1-d arrays and is implemented to handle expressions involving a.flat[obj]. The second is general-purpose that works for arrays of "arbitrary dimension" (up to a fixed maximum). The one-dimensional indexing approaches were implemented in a rather straightforward fashion, and so it is the general-purpose indexing code that will be the focus of this section.

There is a multi-layer approach to indexing because the indexing code can at times return an array scalar and at other times return an array. The functions with "_nice" appended to their name do this special handling while the function without the _nice appendage always return an array (perhaps a 0-dimensional array). Some special-case optimizations (the index being an integer scalar, and the index being a tuple with as many dimensions as the array) are handled in array_subscript_nice function which is what Python calls when presented with the code "a[obj]." These optimizations allow fast single-integer indexing, and also ensure that a 0-dimensional array is not created only to be discarded as the array scalar is returned instead. This provides significant speed-up for code that is selecting many scalars out of an array (such as in a loop). However, it is still not faster than simply using a list to store standard Python scalars, because that is optimized by the Python interpreter itself.

After these optimizations, the array_subscript function itself is called. This function first checks for field selection which occurs when a string is passed as the indexing object. Then, 0-d arrays are given special-case consideration. Finally, the code determines whether or not advanced, or fancy, indexing needs to be performed. If fancy indexing is not needed, then standard view-based indexing is performed using code borrowed from Numeric which parses the indexing object and returns the offset into the data-buffer and the dimensions necessary to create a new view of the array. The strides are also changed by multiplying each stride by the step-size requested along the corresponding dimension.

17.6.1 Fancy-indexing check

The fancy_indexing_check routine determines whether or not to use standard view-based indexing or new copy-based indexing. If the indexing object is a tuple, then view-based indexing is assumed by default. Only if the tuple contains an array object or a sequence object is fancy-indexing assumed. If the indexing object is an array, then fancy indexing is automatically assumed. If the indexing object is any other kind of sequence, then fancy-indexing is assumed by default. This is over-ridden to simple indexing if the sequence contains any slice, newaxis, or Ellipsis objects, and no arrays or additional sequences are also contained in the sequence. The purpose of this is to allow the construction of "slicing" sequences which is a common technique for building up code that works in arbitrary numbers of dimensions.

17.6.2 Fancy-indexing implementation

The concept of indexing was also abstracted using the idea of an iterator. If fancy indexing is performed, then a PyArrayMapIterObject is created. This internal object is not exposed to Python. It is created in order to handle the fancy-indexing at a high-level. Both get and set fancy-indexing operations are implemented using this object. Fancy indexing is abstracted into three separate operations: (1) creating the PyArrayMapIterObject from the indexing object, (2) binding the PyArrayMapIterObject to the array being indexed, and (3) getting (or setting) the items determined by the indexing object. There is an optimization implemented so that the PyArrayIterObject (which has it's own less complicated fancy-indexing) is used for indexing when possible.

17.6.2.1 Creating the mapping object

The first step is to convert the indexing objects into a standard form where iterators are created for all of the index array inputs and all Boolean arrays are converted to equivalent integer index arrays (as if nonzero(arr) had been called). Finally, all integer arrays are replaced with the integer 0 in the indexing object and all of the index-array iterators are "broadcast" to the same shape.

17.6.2.2 Binding the mapping object

When the mapping object is created it does not know which array it will be used with so once the index iterators are constructed during mapping-object creation, the next step is to associate these iterators with a particular ndarray. This process interprets any ellipsis and slice objects so that the index arrays are associated with the appropriate axis (the axis indicated by the iteraxis entry corresponding to the iterator for the integer index array). This information is then used to check the indices to be sure they are within range of the shape of the array being indexed. The presence of ellipsis and/or slice objects implies a sub-space iteration that is accomplished by extracting a sub-space view of the array (using the index object resulting from replacing all the integer index arrays with 0) and storing the information about where this sub-space starts in the mapping object. This is used later during mapping-object iteration to select the correct elements from the underlying array.

17.6.2.3 Getting (or Setting)

After the mapping object is successfully bound to a particular array, the mapping object contains the shape of the resulting item as well as iterator objects that will walk through the currently-bound array and either get or set its elements as needed. The walk is implemented using the PyArray_MapIterNext function. This function sets the coordinates of an iterator object into the current array to be the next coordinate location indicated by all of the indexing-object iterators while adjusting, if necessary, for the presence of a sub-space. The result of this function is that the dataptr member of the mapping object structure is pointed to the next position in the array that needs to be copied out or set to some value.

When advanced indexing is used to extract an array, an iterator for the new array is constructed and advanced in phase with the mapping object iterator. When advanced indexing is used to place values in an array, a special "broadcasted" iterator is constructed from the object being placed into the array so that it will only work if the values used for setting have a shape that is "broadcastable" to the shape implied by the indexing object.

17.7 Universal Functions

Universal functions are callable objects that take N inputs and produce M outputs by wrapping basic 1-d loops that work element-by-element into full easy-to use functions that seamlessly implement broadcasting, type-checking and buffered coercion, and output-argument handling. New universal functions are normally created in C, although there is a mechanism for creating ufuncs from Python functions (**frompyfunc**). The user must supply a 1-d loop that implements the basic function taking the input scalar values and placing the resulting scalars into the appropriate output slots as explaine n implementation.

17.7.1 Setup

Every ufunc calculation involves some overhead related to setting up the calculation. The practical significance of this overhead is that even though the actual calculation of the ufunc is very fast, you will be able to write array and type-specific code that will work faster for small arrays than the ufunc. In particular, using ufuncs to perform many calculations on 0-d arrays will be slower than other Python-based solutions (the silently-imported scalarmath module exists precisely to give array scalars the look-and-feel of ufunc-based calculations with significantly reduced overhead).

When a ufunc is called, many things must be done. The information collected from these setup operations is stored in a loop-object. This loop object is a C-structure (that could become a Python object but is not initialized as such because it is only used internally). This loop object has the layout needed to be used with PyArray_Broadcast so that the broadcasting can be handled in the same way as it is handled in other sections of code.

The first thing done is to look-up in the thread-specific global dictionary the current values for the buffer-size, the error mask, and the associated error object. The state of the error mask controls what happens when an error-condiction is found. It should be noted that checking of the hardware error flags is only performed after each 1-d loop is executed. This means that if the input and output arrays are contiguous and of the correct type so that a single 1-d loop is performed, then the flags may not be checked until all elements of the array have been calcluated. Looking up these values in a thread-specific dictionary takes time which is easily ignored for all but very small arrays.

After checking, the thread-specific global variables, the inputs are evaluated to determine how the ufunc should proceed and the input and output arrays are constructed if necessary. Any inputs which are not arrays are converted to arrays (using context if necessary). Which of the inputs are scalars (and therefore converted to 0-d arrays) is noted.

Next, an appropriate 1-d loop is selected from the 1-d loops available to the ufunc based on the input array types. This 1-d loop is selected by trying to match the signature of the data-types of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the types member of the ufunc structure. The signatures corresponding to user-defined types are stored in a linked-list of function-information with the head element stored as a `CObject` in the userloops dictionary keyed by the data-type number (the first user-defined type in the argument list is used as the key). The signatures are searched until a signature is found to which the input arrays can all be cast safely (ignoring any scalar arguments which are not allowed to determine the type of the result). The implication of this search procedure is that "lesser types" should be placed below "larger types" when the signatures are stored. If no 1-d loop is found, then an error is reported. Otherwise, the argument_list is updated with the stored signature — in case casting is necessary and to fix the output types assumed by the 1-d loop.

If the ufunc has 2 inputs and 1 output and the second input is an Object array then a special-case check is performed so that NotImplemented is returned if the second input is not an ndarray, has the __array_priority__ attribute, and has an __r<op>__ special method. In this way, Python is signaled to give the other object a chance to complete the operation instead of using generic object-array calculations. This allows (for example) sparse matrices to override the multiplication operator 1-d loop.

For input arrays that are smaller than the specified buffer size, copies are made of all non-contiguous, mis-aligned, or out-of-byteorder arrays to ensure that for small arrays, a single-loop is used. Then, array iterators are created for all the input arrays and the resulting collection of iterators is broadcast to a single shape.

The output arguments (if any) are then processed and any missing return arrays are constructed. If any provided output array doesn't have the correct type (or is mis-aligned) and is smaller than the buffer size, then a new output array is constructed with the special UPDATEIFCOPY flag set so that when it is DECREF'd on completion of the function, it's contents will be copied back into the output array. Iterators for the output arguments are then processed.

Finally, the decision is made about how to execute the looping mechanism to ensure that all elements of the input arrays are combined to produce the output arrays of the correct type. The options for loop execution are one-loop (for contiguous, aligned, and correct data-type), strided-loop (for non-contiguous but still aligned and correct data-type), and a buffered loop (for mis-aligned or incorrect data-type situations). Depending on which execution method is called for, the loop is then setup and computed.

17.7.2 Function call

This section describes how the basic universal function computation loop is setup and executed for each of the three different kinds of execution possibilities. If NPY_ALLOW_THREADS is defined during compilation, then the Python Global Interpreter Lock (GIL) is released prior to calling all of these loops (as long as they don't involve object arrays). It is re-acquired if necessary to handle error conditions. The hardware error flags are checked only after the 1-d loop is calcluated.

17.7.2.1 One Loop

This is the simplest case of all. The ufunc is executed by calling the underlying 1-d loop exactly once. This is possible only when we have aligned data of the correct type (including byte-order) for both input and output and all arrays have uniform strides (either contiguous, 0-d, or 1-d). In this case, the 1-d computational loop is called once to compute the calculation for the entire array. Note that the hardware error flags are only checked after the entire calculation is complete.

17.7.2.2 Strided Loop

When the input and output arrays are aligned and of the correct type, but the striding is not uniform (non-contiguous and 2-d or larger), then a second looping structure is employed for the calculation. This approach converts all of the iterators for the input and output arguments to iterate over all but the largest dimension. The inner loop is then handled by the underlying 1-d computational loop. The outer loop is a standard iterator loop on the converted iterators. The hardware error flags are checked after each 1-d loop is completed.

17.7.2.3 Buffered Loop

This is the code that handles the situation whenever the input and/or output arrays are either misaligned or of the wrong data-type (including being byte-swapped) from what the underlying 1-d loop expects. The arrays are also assumed to be non-contiguous. The code works very much like the strided loop except for the inner 1-d loop is modified so that pre-processing is performed on the inputs and post-processing is performed on the outputs in bufsize chunks (where bufsize is a user-settable parameter). The underlying 1-d computational loop is called on data that is copied over (if it needs to be). The setup code and the loop code is considerably more complicated in this case because it has to handle:

- memory allocation of the temporary buffers
- deciding whether or not to use buffers on the input and output data (misaligned and/or wrong data-type)
- copying and possibly casting data for any inputs or outputs for which buffers are necessary.
- special-casing Object arrays so that reference counts are properly handled when copies and/or casts are necessary.
- breaking up the inner 1-d loop into bufsize chunks (with a possible remainder).

Again, the hardware error flags are checked at the end of each 1-d loop.

17.7.3 Final output manipulation

Ufuncs allow other array-like classes to be passed seamlessly through the interface in that inputs of a particular class will induce the outputs to be of that same class. The mechanism by which this works is the following. If any of the inputs are not ndarrays and define the **__array_wrap__** method, then the class with the largest **__array_priority__** attribute determines the type of all the outputs (with the exception of any output arrays passed in). The **__array_wrap__** method of the input array will be called with the ndarray being returned from the ufunc as it's input. There are two calling styles of the **__array_wrap__** function supported. The first takes the ndarray as the first argument and a tuple of "context" as the second argument. The context is (ufunc, arguments, output argument number). This is the first call tried. If a TypeError occurs, then the function is called with just the ndarray as the first argument.

17.7.4 Methods

Their are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are reduce, accumulate, and reduceat. Each of these methods requires a setup command followed by a loop. There are four loop styles possible for the methods corresponding to no-elements, one-element, strided-loop, and buffered-loop. These are the same basic loop styles as implemented for the general purpose function call except for the no-element and one-element cases which are special-cases occurring when the input array objects have 0 and 1 elements respectively.

17.7.4.1 Setup

The setup function for all three methods is `construct_reduce`. This function creates a reducing loop object and fills it with parameters needed to complete the loop. All of the methods only work on ufuncs that take 2-inputs and return 1 output. Therefore, the underlying 1-d loop is selected assuming a signature of [`otype`, `otype`, `otype`] where `otype` is the requested reduction data-type. The buffer size and error handling is then retrieved from (per-thread) global storage. For small arrays that are mis-aligned or have incorrect data-type, a copy is made so that the un-buffered section of code is used. Then, the looping strategy is selected. If there is 1 element or 0 elements in the array, then a simple looping method is selected. If the array is not mis-aligned and has the correct data-type, then strided looping is selected. Otherwise, buffered looping must be performed. Looping parameters are then established, and the return array is constructed. The output array is of a different shape depending on whether the method is reduce, accumulate, or reduceat. If an output array is already provided, then it's shape is checked. If the output array is not C-contiguous, aligned, and of the correct data type, then a temporary copy is made with the UPDATEIFCOPY flag set. In this way, the methods will be able to work with a well-behaved output array but the result will be copied back into the true output array when the method computation is complete. Finally, iterators are set up to loop over the correct axis (depending on the value of axis provided to the method) and the setup routine returns to the actual computation routine.

17.7.4.2 Reduce

All of the ufunc methods use the same underlying 1-d computational loops with input and output arguments adjusted so that the appropriate reduction takes place. For example, the key to the functioning of reduce is that the 1-d loop is called with the output and the second input pointing to the same position in memory and both having a step-size of 0. The first input is pointing to the input array with a step-size given by the appropriate stride for the selected axis. In this way, the operation performed is

$$\begin{aligned} o &= i[0] \\ o &= i[k] \text{ ¡op¿ } o \quad k = 1 \ldots N \end{aligned}$$

where $N+1$ is the number of elements in the input, i, o is the output, and $i[k]$ is the k^{th} element of i along the selected axis. This basic operations is repeated for arrays with greater than 1 dimension so that the reduction takes place for every 1-d sub-array along the selected axis. An iterator with the selected dimension removed handles this looping.

For buffered loops, care must be taken to copy and cast data before the loop function is called because the underlying loop expects aligned data of the correct data-type (including byte-order). The buffered loop must handle this copying and casting prior to calling the loop function on chunks no greater than the user-specified bufsize.

17.7.4.3 Accumulate

The accumulate function is very similar to the reduce function in that the output and the second input both point to the output. The difference is that the second input points to memory one stride behind the current output pointer. Thus, the operation performed is

$$\begin{aligned} o[0] &= i[0] \\ o[k] &= i[k] \text{ ¡op¿ } o[k-1] \quad k = 1 \ldots N. \end{aligned}$$

The output has the same shape as the input and each 1-d loop operates over N elements when the shape in the selected axis is $N+1$. Again, buffered loops take care to copy and cast the data before calling the underlying 1-d computational loop.

17.7.4.4 Reduceat

The reduceat function is a generalization of both the reduce and accumulate functions. It implements a reduce over ranges of the input array specified by indices. The extra indices argument is checked to be sure that every input is not too large for the input array along the selected dimension before the loop calculations take place. The loop implementation is handled using code that is very similar to the reduce code repeated as many times as there are elements in the indices input. In particular: the first input pointer passed to the underlying 1-d computational loop points to the input array at the correct location indicated by the index array. In addition, the output pointer and the second input pointer passed to the underlying 1-d loop point to the same position in memory. The size of the 1-d computational loop is fixed to be the difference between the current index and the next index (when the current index is the last index, then the next index is assumed to be the length of the array along the selected dimension). In this way, the 1-d loop will implement a reduce over the specified indices.

Mis-aligned or a loop data-type that does not match the input and/or output data-type is handled using buffered code where-in data is copied to a temporary buffer and cast to the correct data-type if necessary prior to calling the underlying 1-d function. The temporary buffers are created in (element) sizes no bigger than the user settable buffer-size value. Thus, the loop must be flexible enough to call the underlying 1-d computational loop enough times to complete the total calculation in chunks no bigger than the buffer-size.

Index