Guide to NumPy

Travis E. Oliphant, PhD
Brigham Young University
Provo, UT

January 6, 2005

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. It's goal is to create 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 NumPy originated. As a graduate student studying biomedical imaging at the Mayo Clinic in Rochester, MN. I came across Python and its Numerical extension in 1998 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 Fernando Perez, Prabhu Ramachandran, Charles Harris, David Cooke, Gary Strangman, and Jean-Sebastien Roy. 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 upheveal. 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 extreme 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 is defintely the case with 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 NumPy 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 NumPy 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-long effort which culminated with the release of NumPy in early 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.core and numpy.lib namespaces.

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 (along with fft, ifft, rand, and randn from from the other standard libraries). 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
dft	discrete Fourier transforms	old FFT from Numeric
random	random number generators	old RandomArray from Numeric
distutils	enhanced build and distribution	improvements for 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`). 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 (see tip). It is also necessary to specify 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 base 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
<type 'int32_arrtype'>
```

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 32-bit integer elements (on my 32-bit platform) which consumes 4 bytes per element. 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
```

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 an integer or a floating-point number. However, this fixed number of bytes could also represent an arbitrary record made up of any collection of data types. NumPy achieves this flexibility through the use of data-type-descriptor (**dtypedescr**) objects. Every array has an associated dtypedescr object which describes the layout of the array data. Every dtypedescr object, in turn, has an associated Python type-object that determines exactly what type of Python object is returned when an element of the array is accessed. The dtypedescr objects are flexible enough to contain references to arrays of other dtypedescr objects and, therefore, can be used to define nested records. This advanced functionality will described in better detail later as it is mainly useful for the recarray (record array) subclass that will be defined later. However, all ndarrays can enjoy the flexibility provided by the dtypedescr objects. Figure 2.1 provides a conceptual diagram showing the relationship between the ndarray, its associated data-type-descriptor object, and an array-scalar that is returned when a single-element of the array is accessed. Note that the data-type

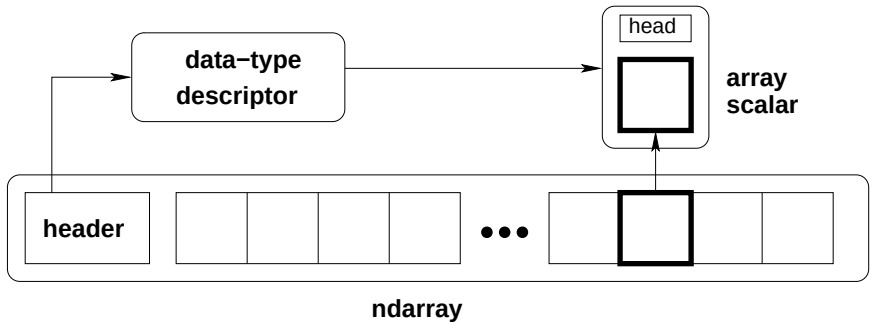

Figure 2.1: Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type-descriptor that details the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned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

descriptor 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 dtypedescr object is based on one of 21 built-in dtypedescr 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-descriptor is a Python type object whose instances are array scalars. This type-object can be obtained using the **dtype** attribute of both the dtypedescr object and the ndarray itself. 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 string, unicode, void, and object.

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. 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.

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 the type, every item in the 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.

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 dtypedescr object associated with each of these data-types). They also correspond to new

Table 2.1: Built-in data types for an ndarray. The bold-face types correspond to standard Python types

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#'

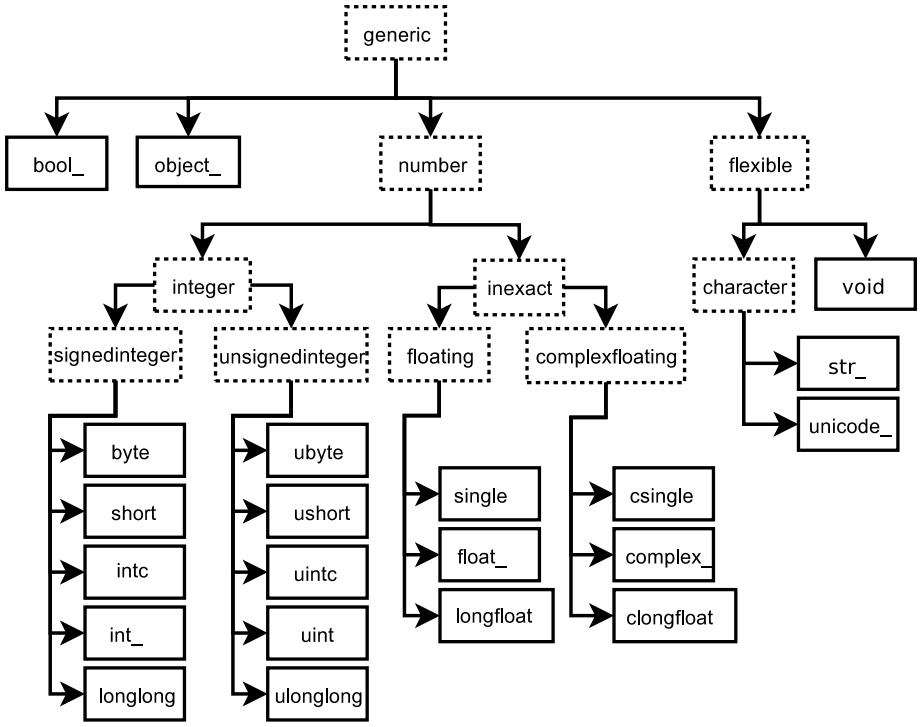

Figure 2.2: Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.

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 a data-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 called slicing in Python, and slicing

[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

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 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]]]
```

slicing (wherein a step can be provided) was also a feature added to Python because of Numeric.

```
>>> 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, an understanding of memory layout will help to explain the use of certain array attributes that may otherwise be mysterious.

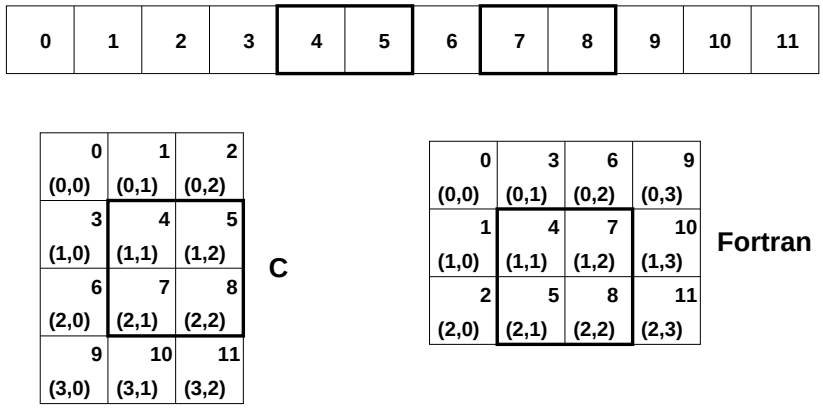

Figure 2.3: Options for memory layout of a 2-dimensional array.

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.

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 reset to zero. While NumPy can be compiled without the use of a Fortran compiler, several modules of the full installation of NumPy (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 Discontiguous 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 strides 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

[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.

constant jumps in each dimension. The `ndarray` object of NumPy uses this strides information and therefore the underlying memory of an `ndarray` can be layed 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 aglorithms, a copy is automatically made.

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 arbitrary array objects. 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 still 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. If it is desired, instead, to add 1's to the end of the array shape, then dimensions can always be added using the `newaxis` name in NumPy.

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. data types are true Python types contained in a hierarchy of types;
4. data descriptors define the data-type with support for data-descriptors with fields and subarrays which allows 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;
10. arrays can be quickly written to files in text and/or binary mode;
11. arrays inherit from big arrays which do not define the sequence, or buffer protocol and can therefore be very large on 64-bit platforms.
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 function / module / builtin 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. arbitrary classes can be passed through ufuncs (__array_wrap__ and __array_priority__);

20. ufuncs can be easily created from Python functions;

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

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

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

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

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

26. new iterator objects created for easy handling in C of discontiguous arrays;

27. 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 attempt 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.

TIP

There is a module called convertcode.py that is distributed with NumPy. This script takes a Python filename <name>.py as an argument, saves a copy <name>.orig, and makes any needed changes to the script. This script only makes the necessary name replacement changes, and should handle many needs. The script is also availabel as a module NumPy.convertcode.

Throughout this book, warnings are inserted when compatibility issues with old Numeric are raised. Here you can find a summary of all the differences that may need changing in your code to work with the new NumPy.base ndarray object. 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.base. Note that Numeric and NumPy can both be loaded together, however, so you can use both simultaneously while you make the transition. If you have Numeric 24.0, they should even be able to use each other's memory.

2.6.1 The list of necessary changes:

1. Importing
   (a) import Numeric –> import numpy as Numeric
   (b) import Numeric as XX –> import numpy as XX
   (c) from Numeric import <name1>,...<nameN> –> from numpy import <name1>,...,<nameN>
   (d) from Numeric 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 Numeric import * to one of the previous three approaches and then continue.
   (e) Similar name changes need to be made for MLab (numpy.lib.mlab), LinearAlgebra (numpy.linalg), RandomArray (numpy.random), RNG (numpy.random), and FFT (numpy.dft).
   (f) multiarray and umath (if you used them directly) are now numpy.core.multiarray and numpy.core.umath.
2. The old names under LinearAlgebra, RandomArray, and FFT are still there but they are not advertised in this book. The old interfaces for RNG are gone, for now. The functionality is available under NumPy.basic.random.

3. Method name changes and methods converted to attributes

   (a) *arr*.typecode() –> *arr*.dtypechar

   (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

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() or *arr*.flatten().

5. If you used the construct `arr.shape=<tuple>`, this will not work for array scalars. 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.

6. Some of the typecode characters have changed to be more consistent with other Python modules (array and struct). Numeric –> numpy

   (a) 'c' –> 'S1', 'B'

   (b) 'b' –> 'B'

   (c) '1' –> 'b'

   (d) 's' –> 'h'

   (e) 'w' –> 'H'

   (f) 'u' –> 'I'

7. UserArray is no longer available because the `ndarray` can be sub-classed without the extra help.

8. Keyword and argument changes

   (a) All `typecode=` keywords have been 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.

9. Character arrays work differently now (there are no character arrays but only string arrays whose basic element size can be any size). An 'S1' array is similar to a Character array in many ways but depending on how you were using character arrays you may want to use a uint8 array.

2.6.2 Recommended changes

1. Convert typecharacters to bitwidth type names or c-type names.
2. Convert use of uppercase Int32, Float, etc., to lower case int32, float, etc.
3. Convert use of functions to method calls where appropriate (but notice the possibly different default arguments).
4. Look for ways to take advantage of advanced slicing.
5. Remove any kludges you inserted to eliminate problems with Numeric that are now gone.
6. Look for ways to take advantage of new features.