CHAPTER 2

Introduction to NumPy

This chapter, along with Chapter 3, outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images—particularly digital images—can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making them analyzable will be to transform them into arrays of numbers. (We will discuss some specific examples of this process later in "Feature Engineering" on page 375.)

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. We'll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package and the Pandas package (discussed in Chapter 3.)

This chapter will cover NumPy in detail. NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to the NumPy website and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

```
In[1]: import numpy
       numpy.__version__
```

```
Out[1]: '1.11.1'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import NumPy using `np` as an alias:

```
In[2]: import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Reminder About Built-In Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the `?` character). Refer back to "Help and Documentation in IPython" on page 3 if you need a refresher on this.

For example, to display all the contents of the `numpy` namespace, you can type this:

```
In [3]: np.<TAB>
```

And to display NumPy's built-in documentation, you can use this:

```
In [4]: np?
```

More detailed documentation, along with tutorials and other resources, can be found at *http://www.numpy.org*.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn in by its ease of use, one piece of which is dynamic typing. While a statically typed language like C or Java requires each variable to be explicitly declared, a dynamically typed language like Python skips this specification. For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

This sort of flexibility is one piece that makes Python and other dynamically typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more in the sections that follow.

A Python Integer Is More Than Just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly disguised C structure, which contains not only its value, but other information as well. For example, when we define an integer in Python, such as `x = 10000`, `x` is not just a "raw" integer. It's actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.4 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count that helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in Figure 2-1.

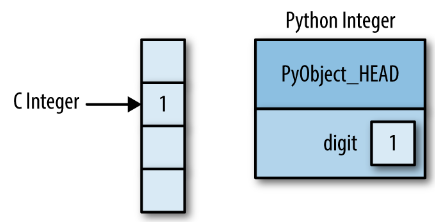

Figure 2-1. The difference between C and Python integers

Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned before.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional information in Python types comes at a cost, however, which becomes especially apparent in structures that combine many of these objects.

A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multielement container in Python is the list. We can create a list of integers as follows:

```
In[1]: L = list(range(10))
       L

Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In[2]: type(L[0])

Out[2]: int
```

Or, similarly, a list of strings:

```
In[3]: L2 = [str(c) for c in L]
       L2

Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In[4]: type(L2[0])

Out[4]: str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
In[5]: L3 = [True, "2", 3.0, 4]
       [type(item) for item in L3]

Out[5]: [bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information—that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in Figure 2-2.

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

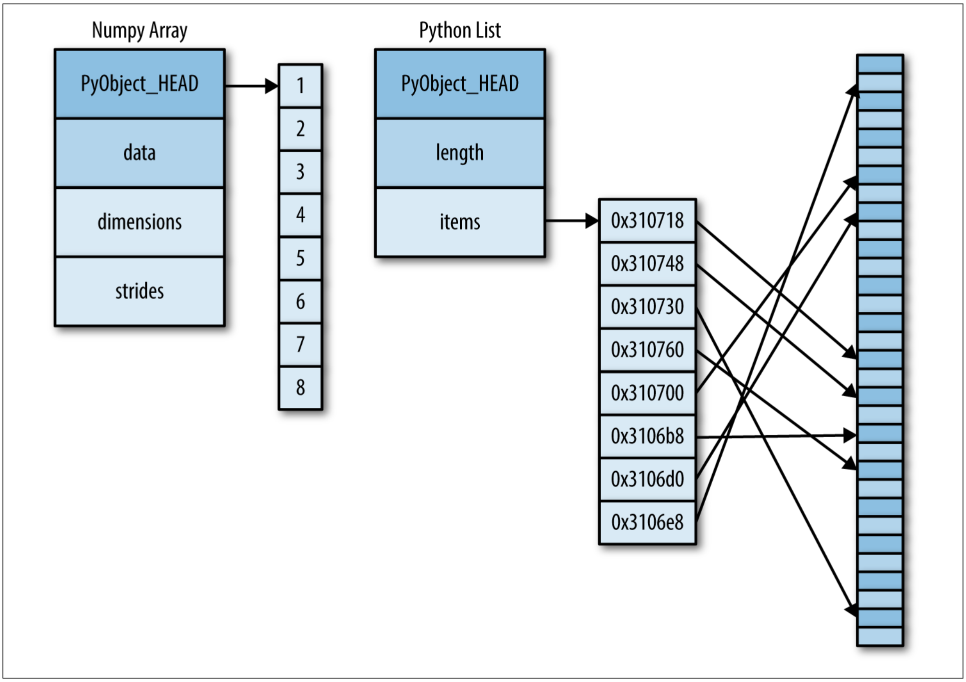

Figure 2-2. The difference between C and Python lists

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
In[6]: import array
       L = list(range(10))
       A = array.array('i', L)
       A

Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

We'll start with the standard NumPy import, under the alias `np`:

```
In[7]: import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
In[8]: # integer array:
       np.array([1, 4, 2, 5, 3])

Out[8]: array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are upcast to floating point):

```
In[9]: np.array([3.14, 4, 2, 3])

Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')

Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multidimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In[11]: # nested lists result in multidimensional arrays
        np.array([range(i, i + 3) for i in [2, 4, 6]])

Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In[12]: # Create a length-10 integer array filled with zeros
        np.zeros(10, dtype=int)

Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Create a 3x5 floating-point array filled with 1s
        np.ones((3, 5), dtype=float)

Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In[14]: # Create a 3x5 array filled with 3.14
        np.full((3, 5), 3.14)
```

```
Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```
In[15]: # Create an array filled with a linear sequence
        # Starting at 0, ending at 20, stepping by 2
        # (this is similar to the built-in range() function)
        np.arange(0, 20, 2)
```

```
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In[16]: # Create an array of five values evenly spaced between 0 and 1
        np.linspace(0, 1, 5)
```

```
Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
In[17]: # Create a 3x3 array of uniformly distributed
        # random values between 0 and 1
        np.random.random((3, 3))
```

```
Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])
```

```
In[18]: # Create a 3x3 array of normally distributed random values
        # with mean 0 and standard deviation 1
        np.random.normal(0, 1, (3, 3))
```

```
Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
                [ 0.25671348,  0.00732722,  0.37783601],
                [ 0.68446945,  0.15926039, -0.70744073]])
```

```
In[19]: # Create a 3x3 array of random integers in the interval [0, 10)
        np.random.randint(0, 10, (3, 3))
```

```
Out[19]: array([[2, 3, 4],
                [5, 7, 8],
                [0, 5, 0]])
```

```
In[20]: # Create a 3x3 identity matrix
        np.eye(3)
```

```
Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

```
In[21]: # Create an uninitialized array of three integers
        # The values will be whatever happens to already exist at that
        # memory location
        np.empty(3)
```

```
Out[21]: array([ 1.,  1.,  1.])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in Table 2-1. Note that when constructing an array, you can specify them using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Table 2-1. Standard NumPy data types

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

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the NumPy documentation. NumPy also supports compound data types, which will be covered in "Structured Data: NumPy's Structured Arrays" on page 92.

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 3) are built around the NumPy array. This section will present several examples using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

Attributes of arrays
: Determining the size, shape, memory consumption, and data types of arrays

Indexing of arrays
: Getting and setting the value of individual array elements

Slicing of arrays
: Getting and setting smaller subarrays within a larger array

Reshaping of arrays
: Changing the shape of a given array

Joining and splitting of arrays
: Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays: a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
In[1]: import numpy as np
       np.random.seed(0)  # seed for reproducibility

       x1 = np.random.randint(10, size=6)  # One-dimensional array
       x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
       x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

```
In[2]: print("x3 ndim: ", x3.ndim)
       print("x3 shape:", x3.shape)
       print("x3 size: ", x3.size)

x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the dtype, the data type of the array (which we discussed previously in "Understanding Data Types in Python" on page 34):

```
In[3]: print("dtype:", x3.dtype)
```

```
dtype: int64
```

Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array:

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
       print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 480 bytes
```

In general, we expect that nbytes is equal to itemsize times size.

Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, you can access the i^{th} value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1
```

```
Out[5]: array([5, 0, 3, 3, 7, 9])
```

```
In[6]: x1[0]
```

```
Out[6]: 5
```

```
In[7]: x1[4]
```

```
Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
In[8]: x1[-1]
```

```
Out[8]: 9
```

```
In[9]: x1[-2]
```

```
Out[9]: 7
```

In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[10]: x2
```

```
Out[10]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])
```

```
In[11]: x2[0, 0]
```

```
Out[11]: 3
```

```
In[12]: x2[2, 0]
```

```
Out[12]: 1
```

```
In[13]: x2[2, -1]
```

```
Out[13]: 7
```

You can also modify values using any of the above index notation:

```
In[14]: x2[0, 0] = 12
        x2
```

```
Out[14]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
In[15]: x1[0] = 3.14159  # this will be truncated!
        x1
```

```
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=`*`size of dimension`*, `step=1`. We'll take a look at accessing subarrays in one dimension and in multiple dimensions.

One-dimensional subarrays

```
In[16]: x = np.arange(10)
        x
```

```
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[17]: x[:5]  # first five elements
```

```
Out[17]: array([0, 1, 2, 3, 4])
```

```
In[18]: x[5:]  # elements after index 5
```

```
Out[18]: array([5, 6, 7, 8, 9])
```

```
In[19]: x[4:7]  # middle subarray
```

```
Out[19]: array([4, 5, 6])
```

```
In[20]: x[::2]  # every other element
Out[20]: array([0, 2, 4, 6, 8])
In[21]: x[1::2]  # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
In[22]: x[::-1]  # all elements, reversed
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
In[23]: x[5::-2]  # reversed every other from index 5
Out[23]: array([5, 3, 1])
```

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas. For example:

```
In[24]: x2
Out[24]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
In[25]: x2[:2, :3]  # two rows, three columns
Out[25]: array([[12,  5,  2],
                [ 7,  6,  8]])
In[26]: x2[:3, ::2]  # all rows, every other column
Out[26]: array([[12,  2],
                [ 7,  8],
                [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]
Out[27]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Accessing array rows and columns. One commonly needed routine is accessing single rows or columns of an array. You can do this by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

```
In[28]: print(x2[:, 0])  # first column of x2
[12  7  1]
```

```
In[29]: print(x2[0, :])  # first row of x2
```

```
[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
In[30]: print(x2[0])  # equivalent to x2[0, :]
```

```
[12  5  2  4]
```

Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

```
In[31]: print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
        print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

```
In[34]: print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)

[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)

[[42  5]
 [ 7  6]]

In[37]: print(x2)

[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with noncontiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the `reshape` method, or more easily by making use of the `newaxis` keyword within a slice operation:

```
In[39]: x = np.array([1, 2, 3])

        # row vector via reshape
        x.reshape((1, 3))

Out[39]: array([[1, 2, 3]])

In[40]: # row vector via newaxis
        x[np.newaxis, :]

Out[40]: array([[1, 2, 3]])
```

```
In[41]: # column vector via reshape
        x.reshape((3, 1))

Out[41]: array([[1],
                [2],
                [3]])

In[42]: # column vector via newaxis
        x[:, np.newaxis]

Out[42]: array([[1],
                [2],
                [3]])
```

We will see this type of transformation often throughout the remainder of the book.

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
In[43]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])

Out[43]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In[44]: z = [99, 99, 99]
        print(np.concatenate([x, y, z]))

[ 1  2  3  3  2  1 99 99 99]
```

`np.concatenate` can also be used for two-dimensional arrays:

```
In[45]: grid = np.array([[1, 2, 3],
                         [4, 5, 6]])

In[46]: # concatenate along the first axis
        np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In[47]: # concatenate along the second axis (zero-indexed)
        np.concatenate([grid, grid], axis=1)
```

```
Out[47]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```
In[48]: x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
                         [6, 5, 4]])

        # vertically stack the arrays
        np.vstack([x, grid])
```

```
Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])
```

```
In[49]: # horizontally stack the arrays
        y = np.array([[99],
                      [99]])
        np.hstack([grid, y])
```

```
Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

Similarly, `np.dstack` will stack arrays along the third axis.

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
        x1, x2, x3 = np.split(x, [3, 5])
        print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split points lead to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
In[51]: grid = np.arange(16).reshape((4, 4))
        grid
```

```
Out[51]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In[52]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
```

```
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
In[53]: left, right = np.hsplit(grid, [2])
        print(left)
        print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

Computation on NumPy Arrays: Universal Functions

Up until now, we have been discussing some of the basic nuts and bolts of NumPy; in the next few sections, we will dive into the reasons that NumPy is so important in the Python data science world. Namely, it provides an easy and flexible interface to optimized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the PyPy project, a just-in-time compiled implementation of Python; the Cython project, which converts Python code to compilable C code; and the Numba project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated—for instance, looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocal of each. A straightforward approach might look like this:

```
In[1]: import numpy as np
       np.random.seed(0)

       def compute_reciprocals(values):
           output = np.empty(len(values))
           for i in range(len(values)):
               output[i] = 1.0 / values[i]
           return output

       values = np.random.randint(1, 10, size=5)
       compute_reciprocals(values)
```

```
Out[1]: array([ 0.16666667,  1.        ,  0.25      ,  0.25      ,  0.125     ])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! We'll benchmark this with IPython's `%timeit` magic (discussed in "Profiling and Timing Code" on page 25):

```
In[2]: big_array = np.random.randint(1, 100, size=1000000)
       %timeit compute_reciprocals(big_array)
```

```
1 loop, best of 3: 2.91 s per loop
```

It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. You can accomplish this by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Compare the results of the following two:

```
In[3]: print(compute_reciprocals(values))
       print(1.0 / values)
```

```
[ 0.16666667  1.          0.25        0.25        0.125     ]
[ 0.16666667  1.          0.25        0.25        0.125     ]
```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
In[4]: %timeit (1.0 / big_array)

100 loops, best of 3: 4.6 ms per loop
```

Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible—before we saw an operation between a scalar and an array, but we can also operate between two arrays:

```
In[5]: np.arange(5) / np.arange(1, 6)

Out[5]: array([ 0.        ,  0.5       ,  0.66666667,  0.75      ,  0.8       ])
```

And ufunc operations are not limited to one-dimensional arrays—they can act on multidimensional arrays as well:

```
In[6]: x = np.arange(9).reshape((3, 3))
       2 ** x

Out[6]: array([[  1,   2,   4],
               [  8,  16,  32],
               [ 64, 128, 256]])
```

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented through Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
In[7]: x = np.arange(4)
       print("x     =", x)
       print("x + 5 =", x + 5)
       print("x - 5 =", x - 5)
       print("x * 2 =", x * 2)
```

```
        print("x / 2 =", x / 2)
        print("x // 2 =", x // 2)  # floor division
```

```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, a `**` operator for exponentiation, and a `%` operator for modulus:

```
In[8]: print("-x     = ", -x)
       print("x ** 2 = ", x ** 2)
       print("x % 2  = ", x % 2)
```

```
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

```
In[9]: -(0.5*x + 1) ** 2
```

```
Out[9]: array([-1.  , -2.25, -4.  , -6.25])
```

All of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the `+` operator is a wrapper for the `add` function:

```
In[10]: np.add(x, 2)
```

```
Out[10]: array([2, 3, 4, 5])
```

Table 2-2 lists the arithmetic operators implemented in NumPy.

Table 2-2. Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Additionally there are Boolean/bitwise operators; we will explore these in "Comparisons, Masks, and Boolean Logic" on page 70.

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

```
In[11]: x = np.array([-2, -1, 0, 1, 2])
        abs(x)

Out[11]: array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available under the alias `np.abs`:

```
In[12]: np.absolute(x)

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

```
In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
        np.abs(x)

Out[14]: array([ 5.,  5.,  2.,  1.])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
In[16]: print("theta      = ", theta)
        print("sin(theta) = ", np.sin(theta))
        print("cos(theta) = ", np.cos(theta))
        print("tan(theta) = ", np.tan(theta))

theta      =  [ 0.          1.57079633  3.14159265]
sin(theta) =  [  0.00000000e+00   1.00000000e+00   1.22464680e-16]
cos(theta) =  [  1.00000000e+00   6.12323400e-17  -1.00000000e+00]
tan(theta) =  [  0.00000000e+00   1.63312394e+16  -1.22464680e-16]
```

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```
In[17]: x = [-1, 0, 1]
        print("x         = ", x)
        print("arcsin(x) = ", np.arcsin(x))
        print("arccos(x) = ", np.arccos(x))
        print("arctan(x) = ", np.arctan(x))

x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [ 3.14159265  1.57079633  0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```
In[18]: x = [1, 2, 3]
        print("x     =", x)
        print("e^x   =", np.exp(x))
        print("2^x   =", np.exp2(x))
        print("3^x   =", np.power(3, x))

x     = [1, 2, 3]
e^x   = [  2.71828183   7.3890561   20.08553692]
2^x   = [ 2.   4.   8.]
3^x   = [ 3  9 27]
```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```
In[19]: x = [1, 2, 4, 10]
        print("x        =", x)
        print("ln(x)    =", np.log(x))
        print("log2(x)  =", np.log2(x))
        print("log10(x) =", np.log10(x))

x        = [1, 2, 4, 10]
ln(x)    = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x)  = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103     0.60205999  1.        ]
```

There are also some specialized versions that are useful for maintaining precision with very small input:

```
In[20]: x = [0, 0.001, 0.01, 0.1]
        print("exp(x) - 1 =", np.expm1(x))
        print("log(1 + x) =", np.log1p(x))

exp(x) - 1 = [ 0.          0.0010005   0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995   0.00995033  0.09531018]
```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were used.

Specialized ufuncs

NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the submodule `scipy.special`. If you want to compute some obscure mathematical function on your data, chances are it is implemented in `scipy.special`. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:

```
In[21]: from scipy import special
```

```
In[22]: # Gamma functions (generalized factorials) and related functions
        x = [1, 5, 10]
        print("gamma(x)     =", special.gamma(x))
        print("ln|gamma(x)| =", special.gammaln(x))
        print("beta(x, 2)   =", special.beta(x, 2))

gamma(x)     = [  1.00000000e+00   2.40000000e+01   3.62880000e+05]
ln|gamma(x)| = [  0.           3.17805383  12.80182748]
beta(x, 2)   = [ 0.5         0.03333333  0.00909091]
```

```
In[23]: # Error function (integral of Gaussian)
        # its complement, and its inverse
        x = np.array([0, 0.3, 0.7, 1.0])
        print("erf(x)  =", special.erf(x))
        print("erfc(x) =", special.erfc(x))
        print("erfinv(x) =", special.erfinv(x))

erf(x)  = [ 0.          0.32862676  0.67780119  0.84270079]
erfc(x) = [ 1.          0.67137324  0.32219881  0.15729921]
erfinv(x) = [ 0.          0.27246271  0.73286908         inf]
```

There are many, many more ufuncs available in both NumPy and `scipy.special`. Because the documentation of these packages is available online, a web search along the lines of "gamma function python" will generally find the relevant information.

Advanced Ufunc Features

Many NumPy users make use of ufuncs without ever learning their full set of features. We'll outline a few specialized features of ufuncs here.

Specifying output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you'd like them to be. For all ufuncs, you can do this using the `out` argument of the function:

```
In[24]: x = np.arange(5)
        y = np.empty(5)
        np.multiply(x, 10, out=y)
        print(y)

[  0.  10.  20.  30.  40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```
In[25]: y = np.zeros(10)
        np.power(2, x, out=y[::2])
        print(y)

[  1.   0.   2.   0.   4.   0.   8.   0.  16.   0.]
```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
In[26]: x = np.arange(1, 6)
        np.add.reduce(x)

Out[26]: 15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
In[27]: np.multiply.reduce(x)

Out[27]: 120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```
In[28]: np.add.accumulate(x)

Out[28]: array([ 1,  3,  6, 10, 15])
```

```
In[29]: np.multiply.accumulate(x)

Out[29]: array([  1,   2,   6,  24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in "Aggregations: Min, Max, and Everything in Between" on page 58.

Outer products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

```
In[30]: x = np.arange(1, 6)
        np.multiply.outer(x, x)

Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods, which we'll explore in "Fancy Indexing" on page 78, are very helpful as well.

Another extremely useful feature of ufuncs is the ability to operate between arrays of different sizes and shapes, a set of operations known as *broadcasting*. This subject is important enough that we will devote a whole section to it (see "Computation on Arrays: Broadcasting" on page 63).

Ufuncs: Learning More

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites.

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (`?`) functionality, as described in "Help and Documentation in IPython" on page 3.

Aggregations: Min, Max, and Everything in Between

Often when you are faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).