CHAPTER 4

NumPy Basics: Arrays and Vectorized Computation

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this book are built. Here are some of the things it provides:

- `ndarray`, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated *broadcasting* capabilities
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

The last bullet point is also one of the most important ones from an ecosystem point of view. Because NumPy provides an easy-to-use C API, it is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively. If you're new to Python and just looking to get your hands dirty working with data using pandas, feel free to give this chapter a skim. For more on advanced NumPy features like broadcasting, see Chapter 12.

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application). Much more on this in Chapter 5

While NumPy provides the computational foundation for these operations, you will likely want to use pandas as your basis for most kinds of data analysis (especially for structured or tabular data) as it provides a rich, high-level interface making most common data tasks very concise and simple. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np.`, but I would caution you against making a habit of this.

The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data
Out[8]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])
```

```
In [9]: data * 10
Out[9]:
array([[ 9.5256, -2.4601, -8.8565],
       [ 5.6385,  2.3794,  9.104 ]])
```

```
In [10]: data + data
Out[10]:
array([[ 1.9051, -0.492 , -1.7713],
       [ 1.1277,  0.4759,  1.8208]])
```

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [11]: data.shape
Out[11]: (2, 3)
```

```
In [12]: data.dtype
Out[12]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it's not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.

> Whenever you see "array", "NumPy array", or "ndarray" in the text, with few exceptions they all refer to the same thing: the ndarray object.

Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1
Out[15]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

In [19]: arr2.ndim
Out[19]: 2

In [20]: arr2.shape
Out[20]: (2, 4)
```

Unless explicitly specified (more on this later), `np.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` object; for example, in the above two examples we have:

```
In [21]: arr1.dtype
Out[21]: dtype('float64')
```

```
In [22]: arr2.dtype
Out[22]: dtype('int64')
```

In addition to `np.array`, there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0's or 1's, respectively, with a given length or shape. `empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [23]: np.zeros(10)
Out[23]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [24]: np.zeros((3, 6))
Out[24]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [25]: np.empty((2, 3, 2))
Out[25]:
array([[[  4.94065646e-324,   4.94065646e-324],
        [  3.87491056e-297,   2.46845796e-130],
        [  4.94065646e-324,   4.94065646e-324]],

       [[  1.90723115e+083,   5.73293533e-053],
        [ -2.33568637e+124,  -6.70608105e-012],
        [  4.42786966e+160,   1.27100354e+025]]])
```

It's not safe to assume that `np.empty` will return an array of all zeros. In many cases, as previously shown, it will return uninitialized garbage values.

`arange` is an array-valued version of the built-in Python `range` function:

```
In [26]: np.arange(15)
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

See Table 4-1 for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be `float64` (floating point).

Table 4-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list.
`ones, ones_like`	Produce an array of all 1's with the given shape and dtype. `ones_like` takes another array and produces a ones array of the same shape and dtype.
`zeros, zeros_like`	Like `ones` and `ones_like` but producing arrays of 0's instead

Function	Description
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`eye, identity`	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [29]: arr1.dtype           In [30]: arr2.dtype
Out[29]: dtype('float64')     Out[30]: dtype('int32')
```

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See Table 4-2 for a full listing of NumPy's supported data types.

Don't worry about memorizing the NumPy dtypes, especially if you're a new user. It's often only necessary to care about the general *kind* of data you're dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large data sets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

Type	Type Code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 32-bit integer types
`float16`	`f2`	Half-precision floating point
`float32`	`f4 or f`	Standard single-precision floating point. Compatible with C float
`float64`	`f8 or d`	Standard double-precision floating point. Compatible with C double and Python `float` object

Type	Type Code	Description
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])

In [32]: arr.dtype
Out[32]: dtype('int64')

In [33]: float_arr = arr.astype(np.float64)

In [34]: float_arr.dtype
Out[34]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [36]: arr
Out[36]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Should you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [39]: numeric_strings.astype(float)
Out[39]: array([  1.25,  -9.6 ,  42.  ])
```

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `TypeError` will be raised. See that I was a bit lazy and wrote `float` instead of `np.float64`; NumPy is smart enough to alias the Python types to the equivalent dtypes.

You can also use another array's dtype attribute:

```
In [40]: int_array = np.arange(10)
```

```
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')

In [44]: empty_uint32
Out[44]:
array([       0,        0, 65904672,        0, 64856792,        0,
       39438163,        0], dtype=uint32)
```

Calling `astype` *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

It's worth keeping in mind that floating point numbers, such as those in `float64` and `float32` arrays, are only capable of approximating fractional quantities. In complex computations, you may accrue some *floating point error*, making comparisons only valid up to a certain number of decimal places.

Operations between Arrays and Scalars

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. This is usually called *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise:

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

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

In [47]: arr * arr
Out[47]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [48]: arr - arr
Out[48]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [49]: 1 / arr
Out[49]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [50]: arr ** 0.5
Out[50]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in Chapter 12. Having a deep understanding of broadcasting is not necessary for most of this book.

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
Out[56]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [57]: arr_slice = arr[5:8]

In [58]: arr_slice[1] = 12345

In [59]: arr
Out[59]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages which copy data more zealously. As NumPy has been designed with large data use cases in mind, you could imagine performance and memory problems if NumPy insisted on copying data left and right.

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example `arr[5:8].copy()`.

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [63]: arr2d[2]
Out[63]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [64]: arr2d[0][2]
Out[64]: 3

In [65]: arr2d[0, 2]
Out[65]: 3
```

See Figure 4-1 for an illustration of indexing on a 2D array.

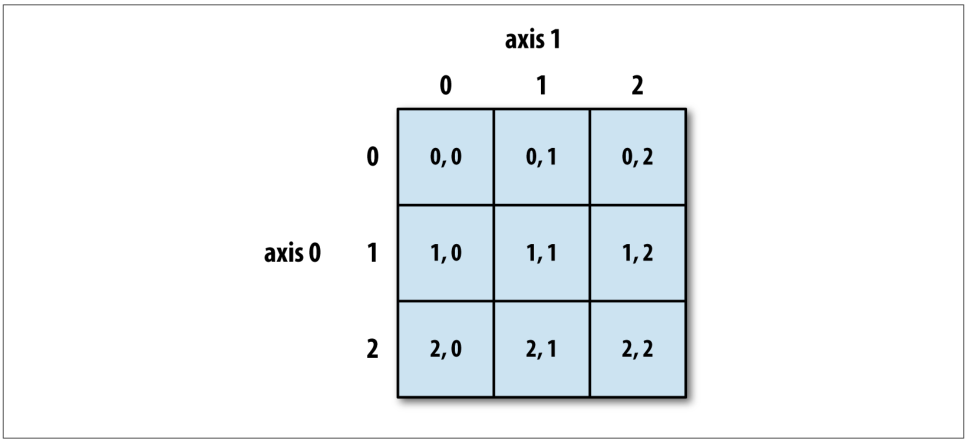

Figure 4-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array `arr3d`

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [67]: arr3d
Out[67]:
array([[[ 1,  2,  3],
```

```
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

arr3d[0] is a 2 × 3 array:

```
In [68]: arr3d[0]
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Both scalar values and arrays can be assigned to arr3d[0]:

```
In [69]: old_values = arr3d[0].copy()

In [70]: arr3d[0] = 42

In [71]: arr3d
Out[71]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [72]: arr3d[0] = old_values

In [73]: arr3d
Out[73]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Similarly, arr3d[1, 0] gives you all of the values whose indices start with (1, 0), forming a 1-dimensional array:

```
In [74]: arr3d[1, 0]
Out[74]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
In [75]: arr[1:6]
Out[75]: array([ 1,  2,  3,  4, 64])
```

Higher dimensional objects give you more options as you can slice one or more axes and also mix integers. Consider the 2D array above, arr2d. Slicing this array is a bit different:

```
In [76]: arr2d            In [77]: arr2d[:2]
Out[76]:                  Out[77]:
```

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

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [79]: arr2d[1, :2]         In [80]: arr2d[2, :1]
Out[79]: array([4, 5])        Out[80]: array([7])
```

See Figure 4-2 for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [81]: arr2d[:, :1]
Out[81]:
array([[1],
       [4],
       [7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [82]: arr2d[:2, 1:] = 0
```

Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the `randn` function in `numpy.random` to generate some random normally distributed data:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [84]: data = randn(7, 4)

In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='|S4')

In [86]: data
Out[86]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289],
```

```
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

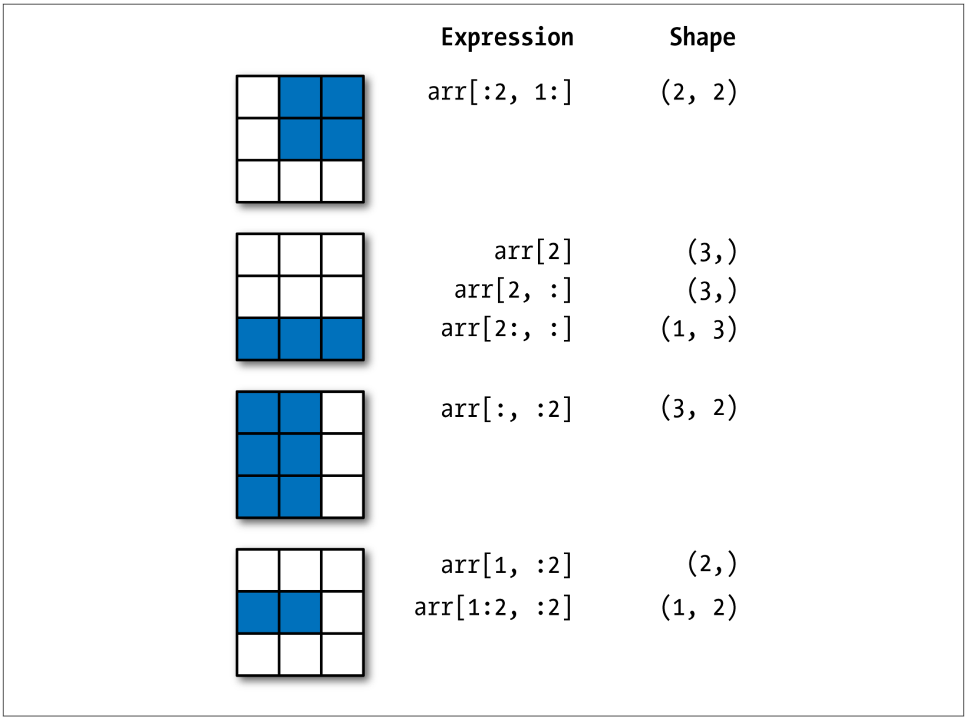

Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the `data` array and we wanted to select all the rows with corresponding name `'Bob'`. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing `names` with the string `'Bob'` yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]: 
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]: 
array([[-0.2349,  1.2792],
```

```
       [-0.0523,  0.0672]])

In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792,  0.0672])
```

To select everything but `'Bob'`, you can either use `!=` or negate the condition using `-`:

```
In [91]: names != 'Bob'
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)

In [92]: data[-(names == 'Bob')]
Out[92]:
array([[-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([True, False, True, True, True, False, False], dtype=bool)

In [95]: data[mask]
Out[95]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

> The Python keywords `and` and `or` do not work with boolean arrays.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in `data` to 0 we need only do:

```
In [96]: data[data < 0] = 0

In [97]: data
Out[97]:
array([[ 0.    ,  0.5433,  0.    ,  1.2792],
       [ 0.    ,  0.5465,  0.0939,  0.    ],
       [ 0.    ,  0.    ,  0.7719,  0.3103],
       [ 2.1452,  0.8799,  0.    ,  0.0672],
       [ 0.    ,  0.    ,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.    ,  0.    ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.    ,  0.5465,  0.0939,  0.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.    ,  0.    ]])
```

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had a 8 × 4 array:

```
In [100]: arr = np.empty((8, 4))

In [101]: for i in range(8):
   .....:     arr[i] = i

In [102]: arr
Out[102]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from the end:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Passing multiple index arrays does something slightly different; it selects a 1D array of elements corresponding to each tuple of indices:

```
# more on reshape in Chapter 12
In [105]: arr = np.arange(32).reshape((8, 4))

In [106]: arr
Out[106]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Take a moment to understand what just happened: the elements `(1, 0)`, `(5, 3)`, `(7, 1)`, and `(2, 2)` were selected. The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Another way is to use the `np.ix_` function, which converts two 1D integer arrays to an indexer that selects the square region:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and also the special `T` attribute:

```
In [110]: arr = np.arange(15).reshape((3, 5))

In [111]: arr                          In [112]: arr.T
```

```
Out[111]:                               Out[112]:
array([[ 0,  1,  2,  3,  4],            array([[ 0,  5, 10],
       [ 5,  6,  7,  8,  9],                   [ 1,  6, 11],
       [10, 11, 12, 13, 14]])                  [ 2,  7, 12],
                                               [ 3,  8, 13],
                                               [ 4,  9, 14]])
```

When doing matrix computations, you will do this very often, like for example computing the inner matrix product X^TX using `np.dot`:

```
In [113]: arr = np.random.randn(6, 3)

In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])
```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))

In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Simple transposing with `.T` is just a special case of swapping axes. ndarray has the method `swapaxes` which takes a pair of axis numbers:

```
In [118]: arr                         In [119]: arr.swapaxes(1, 2)
Out[118]:                             Out[119]:
array([[[ 0,  1,  2,  3],             array([[[ 0,  4],
        [ 4,  5,  6,  7]],                    [ 1,  5],
                                              [ 2,  6],
       [[ 8,  9, 10, 11],                     [ 3,  7]],
        [12, 13, 14, 15]]])
                                             [[ 8, 12],
                                              [ 9, 13],
                                              [10, 14],
                                              [11, 15]]])
```

`swapaxes` similarly returns a view on the data without making a copy.

Universal Functions: Fast Element-wise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple elementwise transformations, like `sqrt` or `exp`:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [122]: np.exp(arr)
Out[122]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

These are referred to as *unary* ufuncs. Others, such as `add` or `maximum`, take 2 arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [123]: x = randn(8)

In [124]: y = randn(8)

In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

While not common, a ufunc can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`: it returns the fractional and integral parts of a floating point array:

```
In [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

See Table 4-3 and Table 4-4 for a listing of available ufuncs.

Table 4-3. Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`sqrt`	Compute the square root of each element. Equivalent to `arr ** 0.5`
`square`	Compute the square of each element. Equivalent to `arr ** 2`
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
`ceil`	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
`floor`	Compute the floor of each element, i.e. the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is NaN (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-NaN) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise. Equivalent to `-arr`.

Table 4-4. Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum. `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum. `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument

Function	Description
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `& |, ^`

Data Processing Using Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Chapter 12, I will explain *broadcasting*, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values. The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of `(x, y)` in the two arrays:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [131]: xs, ys = np.meshgrid(points, points)

In [132]: ys
Out[132]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Now, evaluating the function is a simple matter of writing the same expression you would write with two points:

```
In [134]: import matplotlib.pyplot as plt

In [135]: z = np.sqrt(xs ** 2 + ys ** 2)

In [136]: z
Out[136]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

```
In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[137]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [138]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[138]: <matplotlib.text.Text at 0x4565790>
```

See Figure 4-3. Here I used the matplotlib function `imshow` to create an image plot from a 2D array of function values.

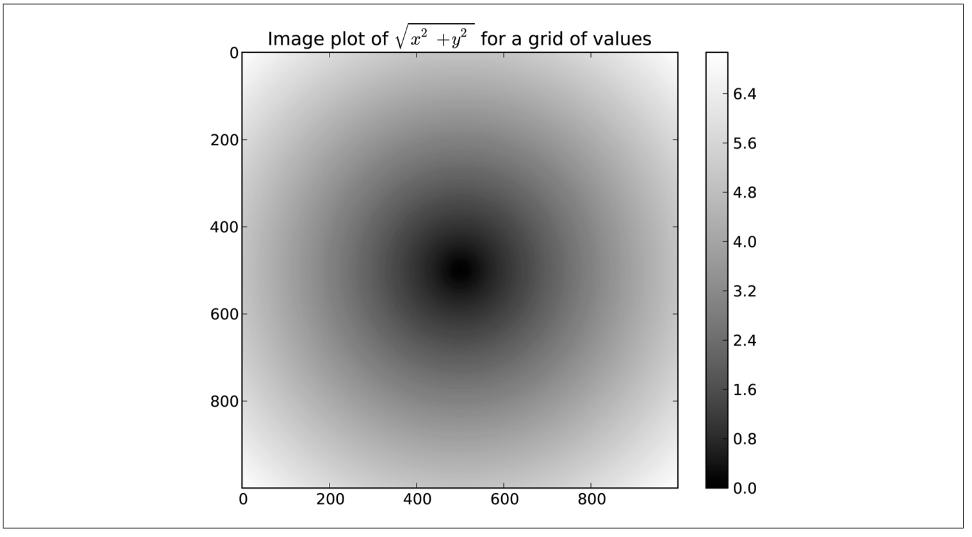

Figure 4-3. Plot of function evaluated on grid

Expressing Conditional Logic as Array Operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a boolean array and two arrays of values:

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [142]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True` otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [143]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in pure Python). Secondly, it will not work with multidimensional arrays. With `np.where` you can write this very concisely:

```
In [145]: result = np.where(cond, xarr, yarr)

In [146]: result
Out[146]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with `np.where`:

```
In [147]: arr = randn(4, 4)

In [148]: arr
Out[148]:
array([[ 0.6372,  2.2043,  1.7904,  0.0752],
       [-1.5926, -1.1536,  0.4413,  0.3483],
       [-0.1798,  0.3299,  0.7827, -0.7585],
       [ 0.5857,  0.1619,  1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)
Out[149]:
array([[ 2,  2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2,  2, -2],
       [ 2,  2,  2, -2]])

In [150]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[150]:
array([[ 2.    ,  2.    ,  2.    ,  2.    ],
       [-1.5926, -1.1536,  2.    ,  2.    ],
       [-0.1798,  2.    ,  2.    , -0.7585],
       [ 2.    ,  2.    ,  2.    , -1.3865]])
```

The arrays passed to `where` can be more than just equal sizes array or scalars.

With some cleverness you can use `where` to express more complicated logic; consider this example where I have two boolean arrays, `cond1` and `cond2`, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

While perhaps not immediately obvious, this `for` loop can be converted into a nested `where` expression:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

In this particular example, we can also take advantage of the fact that boolean values are treated as 0 or 1 in calculations, so this could alternatively be expressed (though a bit more cryptically) as an arithmetic operation:

```
result = 1 * (cond1 & -cond2) + 2 * (cond2 & -cond1) + 3 * -(cond1 | cond2)
```

Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called *reductions*) like `sum`, `mean`, and standard deviation `std` can either be used by calling the array instance method or using the top level NumPy function:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data

In [152]: arr.mean()
Out[152]: 0.062814911084854597

In [153]: np.mean(arr)
Out[153]: 0.062814911084854597

In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Functions like `mean` and `sum` take an optional `axis` argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833,  0.2844,  0.6574,  0.6743, -0.0187])

In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189,  1.4044,  4.5712])
```

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [158]: arr.cumsum(0)        In [159]: arr.cumprod(1)
Out[158]:                      Out[159]:
array([[ 0,  1,  2],           array([[  0,   0,   0],
       [ 3,  5,  7],                  [  3,  12,  60],
       [ 9, 12, 15]])                 [  6,  42, 336]])
```

See Table 4-5 for a full listing. We'll see many examples of these methods in action in later chapters.

Table 4-5. Basic array statistical methods

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

Methods for Boolean Arrays

Boolean values are coerced to 1 (`True`) and 0 (`False`) in the above methods. Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [160]: arr = randn(100)

In [161]: (arr > 0).sum() # Number of positive values
Out[161]: 44
```

There are two additional methods, `any` and `all`, useful especially for boolean arrays. `any` tests whether one or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [162]: bools = np.array([False, False, True, False])

In [163]: bools.any()
Out[163]: True

In [164]: bools.all()
Out[164]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to `True`.

Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place using the `sort` method:

```
In [165]: arr = randn(8)

In [166]: arr
Out[166]:
array([ 0.6903,  0.4678,  0.0968, -0.1349,  0.9879,  0.0185, -1.3147,
       -0.5425])

In [167]: arr.sort()
```

```
In [168]: arr
Out[168]:
array([-1.3147, -0.5425, -0.1349,  0.0185,  0.0968,  0.4678,  0.6903,
        0.9879])
```

Multidimensional arrays can have each 1D section of values sorted in-place along an axis by passing the axis number to `sort`:

```
In [169]: arr = randn(5, 3)

In [170]: arr
Out[170]:
array([[-0.7139, -1.6331, -0.4959],
       [ 0.8236, -1.3132, -0.1935],
       [-1.6748,  3.0336, -0.863 ],
       [-0.3161,  0.5362, -2.468 ],
       [ 0.9058,  1.1184, -1.0516]])

In [171]: arr.sort(1)

In [172]: arr
Out[172]:
array([[-1.6331, -0.7139, -0.4959],
       [-1.3132, -0.1935,  0.8236],
       [-1.6748, -0.863 ,  3.0336],
       [-2.468 , -0.3161,  0.5362],
       [-1.0516,  0.9058,  1.1184]])
```

The top level method `np.sort` returns a sorted copy of an array instead of modifying the array in place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [173]: large_arr = randn(1000)

In [174]: large_arr.sort()

In [175]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[175]: -1.5791023260896004
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see Chapter 12. Several other kinds of data manipulations related to sorting (for example, sorting a table of data by one or more columns) are also to be found in pandas.

Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. Probably the most commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
```

```
array(['Bob', 'Joe', 'Will'],
      dtype='|S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

Contrast `np.unique` with the pure Python alternative:

```
In [180]: sorted(set(names))
Out[180]: ['Bob', 'Joe', 'Will']
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

See Table 4-6 for a listing of set functions in NumPy.

Table 4-6. Array set operations

Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. In later chapters you will learn about tools in pandas for reading tabular data into memory.

Storing Arrays on Disk in Binary Format

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`.

```
In [183]: arr = np.arange(10)

In [184]: np.save('some_array', arr)
```

If the file path does not already end in `.npy`, the extension will be appended. The array on disk can then be loaded using `np.load`:

```
In [185]: np.load('some_array.npy')
Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in a zip archive using `np.savez` and passing the arrays as keyword arguments:

```
In [186]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an `.npz` file, you get back a dict-like object which loads the individual arrays lazily:

```
In [187]: arch = np.load('array_archive.npz')

In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Saving and Loading Text Files

Loading text from files is a fairly standard task. The landscape of file reading and writing functions in Python can be a bit confusing for a newcomer, so I will focus mainly on the `read_csv` and `read_table` functions in pandas. It will at times be useful to load data into vanilla NumPy arrays using `np.loadtxt` or the more specialized `np.genfromtxt`.

These functions have many options allowing you to specify different delimiters, converter functions for certain columns, skipping rows, and other things. Take a simple case of a comma-separated file (CSV) like this:

```
In [191]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

This can be loaded into a 2D array like so:

```
In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')

In [193]: arr
Out[193]:
array([[ 0.5801,  0.1867,  1.0407,  1.1344],
       [ 0.1942, -0.6369, -0.9387,  0.1241],
       [-0.1264,  0.2686, -0.6957,  0.0474],
       [-1.4844,  0.0042, -0.7442,  0.0055],
       [ 2.3029,  0.2001,  1.6702, -1.8811],
       [-0.1932,  1.0472,  0.4828,  0.9603]])
```

`np.savetxt` performs the inverse operation: writing an array to a delimited text file. `genfromtxt` is similar to `loadtxt` but is geared for structured arrays and missing data handling; see Chapter 12 for more on structured arrays.

For more on file reading and writing, especially tabular or spreadsheet-like data, see the later chapters involving pandas and DataFrame objects.

Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with * is an element-wise product instead of a matrix dot product. As such, there is a function `dot`, both an array method, and a function in the `numpy` namespace, for matrix multiplication:

```
In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [196]: x                 In [197]: y
Out[196]:                   Out[197]:
array([[ 1.,  2.,  3.],     array([[  6.,  23.],
       [ 4.,  5.,  6.]])           [ -1.,   7.],
                                   [  8.,   9.]])

In [198]: x.dot(y)  # equivalently np.dot(x, y)
Out[198]:
array([[  28.,   64.],
       [  67.,  181.]])
```

A matrix product between a 2D array and a suitably sized 1D array results in a 1D array:

```
In [199]: np.dot(x, np.ones(3))
Out[199]: array([  6.,  15.])
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood using the same industry-standard Fortran libraries used in other languages like MATLAB and R, such as like BLAS, LAPACK, or possibly (depending on your NumPy build) the Intel MKL:

```
In [201]: from numpy.linalg import inv, qr

In [202]: X = randn(5, 5)

In [203]: mat = X.T.dot(X)

In [204]: inv(mat)
Out[204]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])

In [205]: mat.dot(inv(mat))
```

```
Out[205]:
array([[ 1.,  0.,  0.,  0., -0.],
       [ 0.,  1., -0.,  0.,  0.],
       [ 0., -0.,  1.,  0.,  0.],
       [ 0., -0., -0.,  1., -0.],
       [ 0.,  0.,  0.,  0.,  1.]])

In [206]: q, r = qr(mat)

In [207]: r
Out[207]:
array([[ -6.9271,   7.389 ,   6.1227,  -7.1163,  -4.9215],
       [  0.    ,  -3.9735,  -0.8671,   2.9747,  -5.7402],
       [  0.    ,   0.    , -10.2681,   1.8909,   1.6079],
       [  0.    ,   0.    ,   0.    ,  -1.2996,   3.3577],
       [  0.    ,   0.    ,   0.    ,   0.    ,   0.5571]])
```

See Table 4-7 for a list of some of the most commonly-used linear algebra functions.

The scientific Python community is hopeful that there may be a matrix multiplication infix operator implemented someday, providing syntactically nicer alternative to using `np.dot`. But for now this is the way.

Table 4-7. Commonly-used numpy.linalg functions

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system $Ax = b$ for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to $Ax = b$

Random Number Generation

The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 by 4 array of samples from the standard normal distribution using `normal`:

```
In [208]: samples = np.random.normal(size=(4, 4))

In [209]: samples
Out[209]:
array([[ 0.1241,  0.3026,  0.5238,  0.0009],
       [ 1.3438, -0.7135, -0.8312, -2.3702],
       [-1.8608, -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

Python's built-in `random` module, by contrast, only samples one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [210]: from random import normalvariate

In [211]: N = 1000000

In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

See Table 4-8 for a partial list of functions available in `numpy.random`. I'll give some examples of leveraging these functions' ability to generate large arrays of samples all at once in the next section.

Table 4-8. Partial list of numpy.random functions

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

Example: Random Walks

An illustrative application of utilizing array operations is in the simulation of random walks. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability. A pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

See Figure 4-4 for an example plot of the first 100 values on one of these random walks.

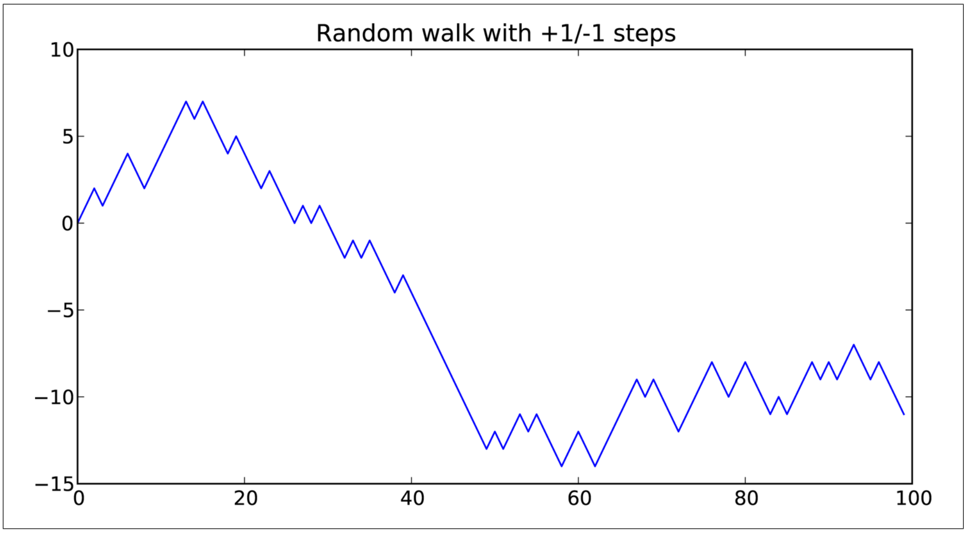

Figure 4-4. A simple random walk

You might make the observation that `walk` is simply the cumulative sum of the random steps and could be evaluated as an array expression. Thus, I use the `np.random` module to draw 1,000 coin flips at once, set these to 1 and -1, and compute the cumulative sum:

```
In [215]: nsteps = 1000

In [216]: draws = np.random.randint(0, 2, size=nsteps)

In [217]: steps = np.where(draws > 0, 1, -1)

In [218]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [219]: walk.min()        In [220]: walk.max()
Out[219]: -3                Out[220]: 31
```

A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the *first* 10 or -10. Turns out this can be computed using `argmax`, which returns the first index of the maximum value in the boolean array (`True` is the maximum value):

```
In [221]: (np.abs(walk) >= 10).argmax()
Out[221]: 37
```

Note that using `argmax` here is not always efficient because it always makes a full scan of the array. In this special case once a `True` is observed we know it to be the maximum value.

Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the above code. The `numpy.random` functions if passed a 2-tuple will generate a 2D array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot:

```
In [222]: nwalks = 5000

In [223]: nsteps = 1000

In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [225]: steps = np.where(draws > 0, 1, -1)

In [226]: walks = steps.cumsum(1)

In [227]: walks
Out[227]:
array([[  1,   0,   1, ...,   8,   7,   8],
       [  1,   0,  -1, ...,  34,  33,  32],
       [  1,   0,  -1, ...,   4,   5,   4],
       ...,
       [  1,   2,   1, ...,  24,  25,  26],
       [  1,   2,   3, ...,  14,  13,  14],
       [ -1,  -2,  -3, ..., -24, -23, -22]])
```

Now, we can compute the maximum and minimum values obtained over all of the walks:

```
In [228]: walks.max()        In [229]: walks.min()
Out[228]: 138                Out[229]: -133
```

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the `any` method:

```
In [230]: hits30 = (np.abs(walks) >= 30).any(1)

In [231]: hits30
Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)

In [232]: hits30.sum() # Number that hit 30 or -30
Out[232]: 3410
```

We can use this boolean array to select out the rows of `walks` that actually cross the absolute 30 level and call `argmax` across axis 1 to get the crossing times:

```
In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [234]: crossing_times.mean()
Out[234]: 498.88973607038122
```

Feel free to experiment with other distributions for the steps other than equal sized coin flips. You need only use a different random number generation function, like `normal` to generate normally distributed steps with some mean and standard deviation:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```