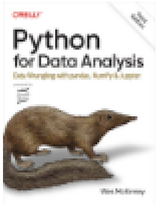

Python for Data Analysis, 3rd Edition

Wes McKinney

Published by O'Reilly Media, Inc.

https://github.com/wesm/pydata-book

Chapter 4. NumPy Basics: Arrays and Vectorized Computation

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python. Many computational packages providing scientific functionality use NumPy's array objects as one of the standard interface *lingua francas* for data exchange. Much of the knowledge about NumPy that I cover is transferable to pandas as well.

Here are some of the things you'll find in NumPy:

- ndarray, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible *broadcasting* capabilities
- 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
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN

Because NumPy provides a comprehensive and well-documented C API, it is straightforward to pass data to external libraries written in a low-level language, and 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++, or FORTRAN codebases and giving them a dynamic and accessible interface.

While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array computing semantics, like pandas, much more effectively. Since NumPy is a large topic, I will cover many advanced NumPy features like broadcasting in more depth later (see Appendix A). Many of these advanced features are not needed to follow the rest of this book, but they may help you as you go deeper into scientific computing in Python.

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

- Fast array-based operations for data munging and cleaning, subsetting and filtering, transformation, and any other kind of computation
- 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 heterogeneous datasets
- Expressing conditional logic as array expressions instead of loops with if-elif-else branches
- Group-wise data manipulations (aggregation, transformation, and function application)

While NumPy provides a computational foundation for general numerical data processing, many readers will want to use pandas as the basis for most kinds of statistics or analytics, especially on tabular data. Also, pandas provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

Note

Array-oriented computing in Python traces its roots back to 1995, when Jim Hugunin created the Numeric library. Over the next 10 years, many scientific programming communities began doing array programming in Python, but the library ecosystem had become fragmented in the early 2000s. In 2005, Travis Oliphant was able to forge the NumPy project from the then Numeric and Numarray projects to bring the community together around a single array computing framework.

One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
- NumPy operations perform complex computations on entire arrays without the need for Python for loops, which can be slow for large sequences. NumPy is faster than regular Python code because its C-based algorithms avoid overhead present with regular interpreted Python code.

To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

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


In [8]: my_arr = np.arange(1_000_000)


In [9]: my_list = list(range(1_000_000))
```

Now let’s multiply each sequence by 2:

```
In [10]: %timeit my_arr2 = my_arr * 2

721 us +- 7.49 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)


In [11]: %timeit my_list2 = [x * 2 for x in my_list]

49 ms +- 1.02 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

4.1 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 datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

To give you a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, I first import NumPy and create a small array:

```
In [12]: import numpy as np


In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])


In [14]: data

Out[14]:

array([[ 1.5, -0.1,  3. ],

       [ 0. , -3. ,  6.5]])
```

I then write mathematical operations with data:

```
In [15]: data * 10

Out[15]:
```

```
array([[ 15.,  -1.,  30.],
      [  0., -30.,  65.]])


In [16]: data + data
Out[16]:
array([[ 3. , -0.2,  6. ],
      [ 0. , -6. , 13. ]])
```

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each “cell” in the array have been added to each other.

Note

In this chapter and throughout the book, I use the standard NumPy convention of always using import numpy as np. It would be possible to put from numpy import * in your code to avoid having to write np., but I advise against making a habit of this.
The numpy namespace is large and contains a number of functions whose names conflict with built-in Python functions (like min and max). Following standard conventions like these is almost always a good idea.

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 [17]: data.shape
Out[17]: (2, 3)


In [18]: data.dtype
Out[18]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and it 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.

Note

Whenever you see “array,” “NumPy array,” or “ndarray” in the book text, in most cases they all refer to 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 [19]: data1 = [6, 7.5, 8, 0, 1]


In [20]: arr1 = np.array(data1)


In [21]: arr1

Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

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

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]


In [23]: arr2 = np.array(data2)


In [24]: arr2

Out[24]:

array([[1, 2, 3, 4],

   [5, 6, 7, 8]])
```

Since data2 was a list of lists, the NumPy array arr2 has two dimensions, with shape inferred from the data. We can confirm this by inspecting the ndim and shape attributes:

```
In [25]: arr2.ndim

Out[25]: 2
```

```
In [26]: arr2.shape

Out[26]: (2, 4)
```

Unless explicitly specified (discussed in “Data Types for ndarrays”), numpy.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype metadata object; for example, in the previous two examples we have:

```
In [27]: arr1.dtype

Out[27]: dtype('float64')


In [28]: arr2.dtype

Out[28]: dtype('int64')
```

In addition to numpy.array, there are a number of other functions for creating new arrays. As examples, numpy.zeros and numpy.ones create arrays of 0s or 1s, respectively, with a given length or shape. numpy.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 [29]: np.zeros(10)

Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])


In [30]: np.zeros((3, 6))

Out[30]:

array([[0., 0., 0., 0., 0., 0.],

    [0., 0., 0., 0., 0., 0.],

    [0., 0., 0., 0., 0., 0.]])


In [31]: np.empty((2, 3, 2))

Out[31]:
```

```
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

Caution

It’s not safe to assume that numpy.empty will return an array of all zeros. This function returns uninitialized memory and thus may contain nonzero “garbage” values. You should use this function only if you intend to populate the new array with data.

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

```
In [32]: np.arange(15)

Out[32]: 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).

Function	Description
array	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; 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

Function	Description
ones, ones_like	Produce an array of all 1s with the given shape and data type; ones_like takes another array and produces a ones array of the same shape and data type
zeros, zeros_like	Like ones and ones_like but producing arrays of 0s instead
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
full, full_like	Produce an array of the given shape and data type with all values set to the indicated “fill value”; full_like takes another array and produces a filled array of the same shape and data type
eye, identity	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Table 4-1. Some important NumPy array creation functions

Data Types for ndarrays

The *data type* or dtype is a special object containing the information (or *metadata*, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)


In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)


In [35]: arr1.dtype

Out[35]: dtype('float64')


In [36]: arr2.dtype

Out[36]: dtype('int32')
```

Data types are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it possible to read and write binary streams of data to disk and to connect to code written in a low-level language like C or FORTRAN. The numerical data types 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.

Note

Don't worry about memorizing the NumPy data types, 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 is stored in memory and on disk, especially large datasets, it is good to know that you have control over the storage type.

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 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float

Type	Type code	Description
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python float object
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; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string data type with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

Table 4-2. NumPy data types

Note

There are both *signed* and *unsigned* integer types, and many readers will not be familiar with this terminology. A *signed* integer can represent both positive and negative integers, while an *unsigned* integer can only represent nonzero integers. For example, int8 (signed 8-bit integer) can represent integers from -128 to 127 (inclusive), while uint8 (unsigned 8-bit integer) can represent 0 through 255.

You can explicitly convert or *cast* an array from one data type to another using ndarray’s astype method:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])


In [38]: arr.dtype

Out[38]: dtype('int64')


In [39]: float_arr = arr.astype(np.float64)


In [40]: float_arr

Out[40]: array([1., 2., 3., 4., 5.])


In [41]: float_arr.dtype

Out[41]: dtype('float64')
```

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

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])


In [43]: arr

Out[43]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])


In [44]: arr.astype(np.int32)

Out[44]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

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

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)
In [46]: numeric_strings.astype(float)

Out[46]: array([ 1.25, -9.6 , 42.  ])
```

Caution

Be cautious when using the numpy.string_ type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.

If casting were to fail for some reason (like a string that cannot be converted to float64), a ValueError will be raised. Before, I was a bit lazy and wrote float instead of np.float64; NumPy aliases the Python types to its own equivalent data types.

You can also use another array’s dtype attribute:

```
In [47]: int_array = np.arange(10)


In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)


In [49]: int_array.astype(calibers.dtype)

Out[49]: 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 [50]: zeros_uint32 = np.zeros(8, dtype="u4")


In [51]: zeros_uint32

Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```

Note

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

Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays apply the operation element-wise:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])


In [53]: arr
Out[53]:
array([[1., 2., 3.],
    [4., 5., 6.]])


In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
    [16., 25., 36.]])


In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
    [0., 0., 0.]])
```

Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [56]: 1 / arr
Out[56]:
array([[1.    , 0.5    , 0.3333],
    [0.25  , 0.2    , 0.1667]])
```

```
In [57]: arr ** 2

Out[57]:

array([[ 1.,  4.,  9.],

      [16., 25., 36.]])
```

Comparisons between arrays of the same size yield Boolean arrays:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])


In [59]: arr2

Out[59]:

array([[ 0.,  4.,  1.],

      [ 7.,  2., 12.]])


In [60]: arr2 > arr

Out[60]:

array([[False,  True, False],

      [ True, False,  True]])
```

Evaluating operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in Appendix A. Having a deep understanding of broadcasting is not necessary for most of this book.

Basic Indexing and Slicing

NumPy array indexing is a deep 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 [61]: arr = np.arange(10)


In [62]: arr
```

```
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]
Out[63]: 5

In [64]: arr[5:8]
Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: 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 *broadcast* henceforth) to the entire selection.

Note

An important first distinction from Python’s built-in 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.

To give an example of this, I first create a slice of arr:

```
In [67]: arr_slice = arr[5:8]

In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

Now, when I change values in arr_slice, the mutations are reflected in the original array arr:

```
In [69]: arr_slice[1] = 12345
In [70]: arr

Out[70]:

array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,

           9])
```

The “bare” slice [:] will assign to all values in an array:

```
In [71]: arr_slice[:] = 64


In [72]: arr

Out[72]: 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 that copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

Caution

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(). As you will see, pandas works this way, too.

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


In [74]: arr2d[2]

Out[74]: 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 [75]: arr2d[0][2]

Out[75]: 3
```

```
In [76]: arr2d[0, 2]

Out[76]: 3
```

See Figure 4-1 for an illustration of indexing on a two-dimensional array. I find it helpful to think of axis 0 as the “rows” of the array and axis 1 as the “columns.”

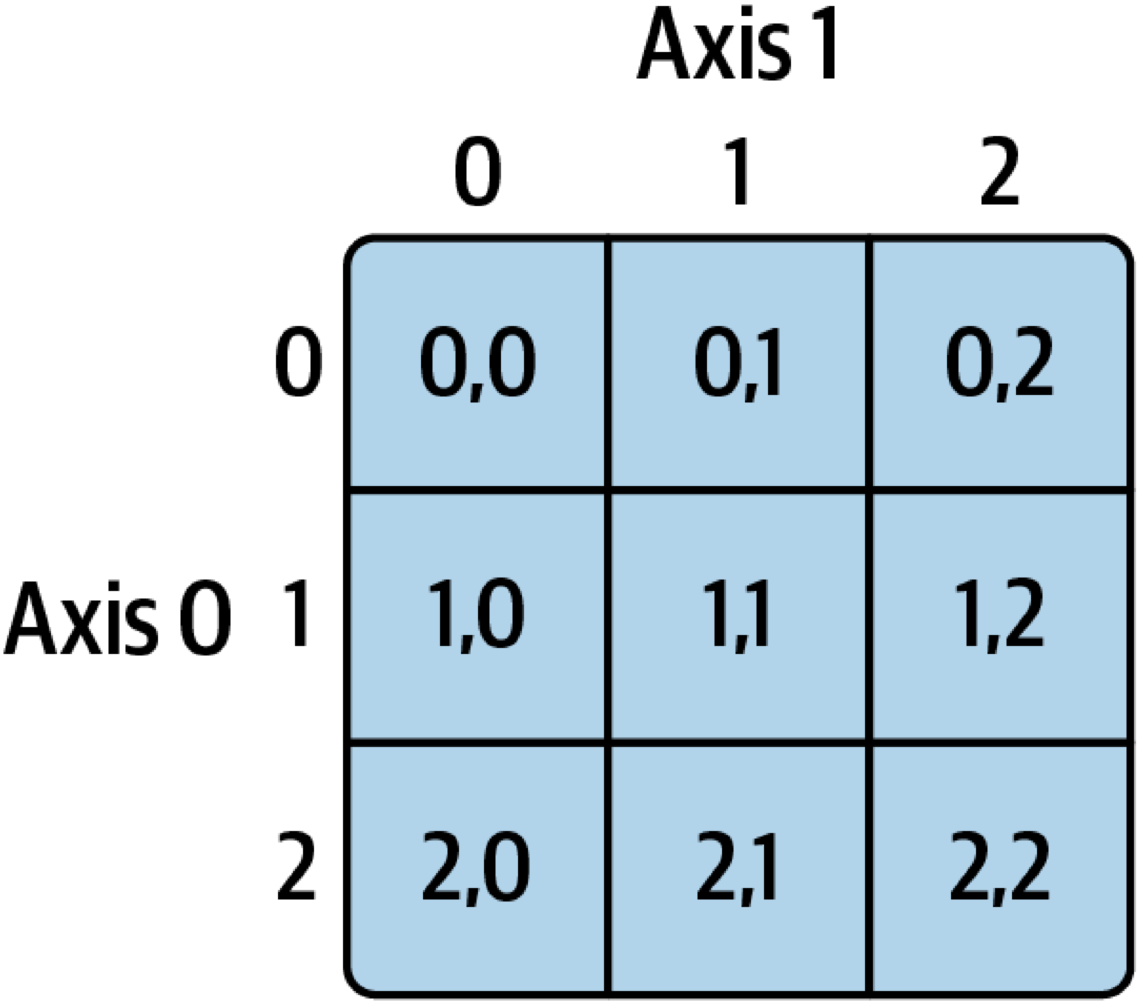

Figure 4-1. Indexing elements in a NumPy array

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 [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
In [78]: arr3d
Out[78]:
array([[[ 1,  2,  3],
       [ 4,  5,  6]],
      [[ 7,  8,  9],
       [10, 11, 12]]])
```

arr3d[0] is a 2 × 3 array:

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
      [4, 5, 6]])
```

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

```
In [80]: old_values = arr3d[0].copy()

In [81]: arr3d[0] = 42

In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
       [42, 42, 42]],
      [[ 7,  8,  9],
       [10, 11, 12]]])

In [83]: arr3d[0] = old_values
In [84]: arr3d

Out[84]:

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 one-dimensional array:

```
In [85]: arr3d[1, 0]

Out[85]: array([7, 8, 9])
```

This expression is the same as though we had indexed in two steps:

```
In [86]: x = arr3d[1]


In [87]: x

Out[87]:

array([[ 7,  8,  9],

      [10, 11, 12]])


In [88]: x[0]

Out[88]: array([7, 8, 9])
```

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

Caution

This multidimensional indexing syntax for NumPy arrays will not work with regular Python objects, such as lists of lists.

Indexing with slices

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

```
In [89]: arr

Out[89]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])


In [90]: arr[1:6]

Out[90]: array([ 1,  2,  3,  4, 64])
```

Consider the two-dimensional array from before, arr2d. Slicing this array is a bit different:

```
In [91]: arr2d

Out[91]:

array([[1, 2, 3],

    [4, 5, 6],

    [7, 8, 9]])


In [92]: arr2d[:2]

Out[92]:

array([[1, 2, 3],

    [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression arr2d[:2] as “select the first two rows of arr2d.”

You can pass multiple slices just like you can pass multiple indexes:

```
In [93]: arr2d[:2, 1:]

Out[93]:

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.

For example, I can select the second row but only the first two columns, like so:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Here, while arr2d is two-dimensional, lower_dim_slice is one-dimensional, and its shape is a tuple with one axis size:

```
In [95]: lower_dim_slice.shape

Out[95]: (2,)
```

Similarly, I can select the third column but only the first two rows, like so:

```
In [96]: arr2d[:2, 2]

Out[96]: array([3, 6])
```

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 [97]: arr2d[:, :1]

Out[97]:

array([[1],
      [4],
      [7]])
```

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

```
In [98]: arr2d[:2, 1:] = 0


In [99]: arr2d
Out[99]:
array([[1, 0, 0],
      [4, 0, 0],
      [7, 8, 9]])
```

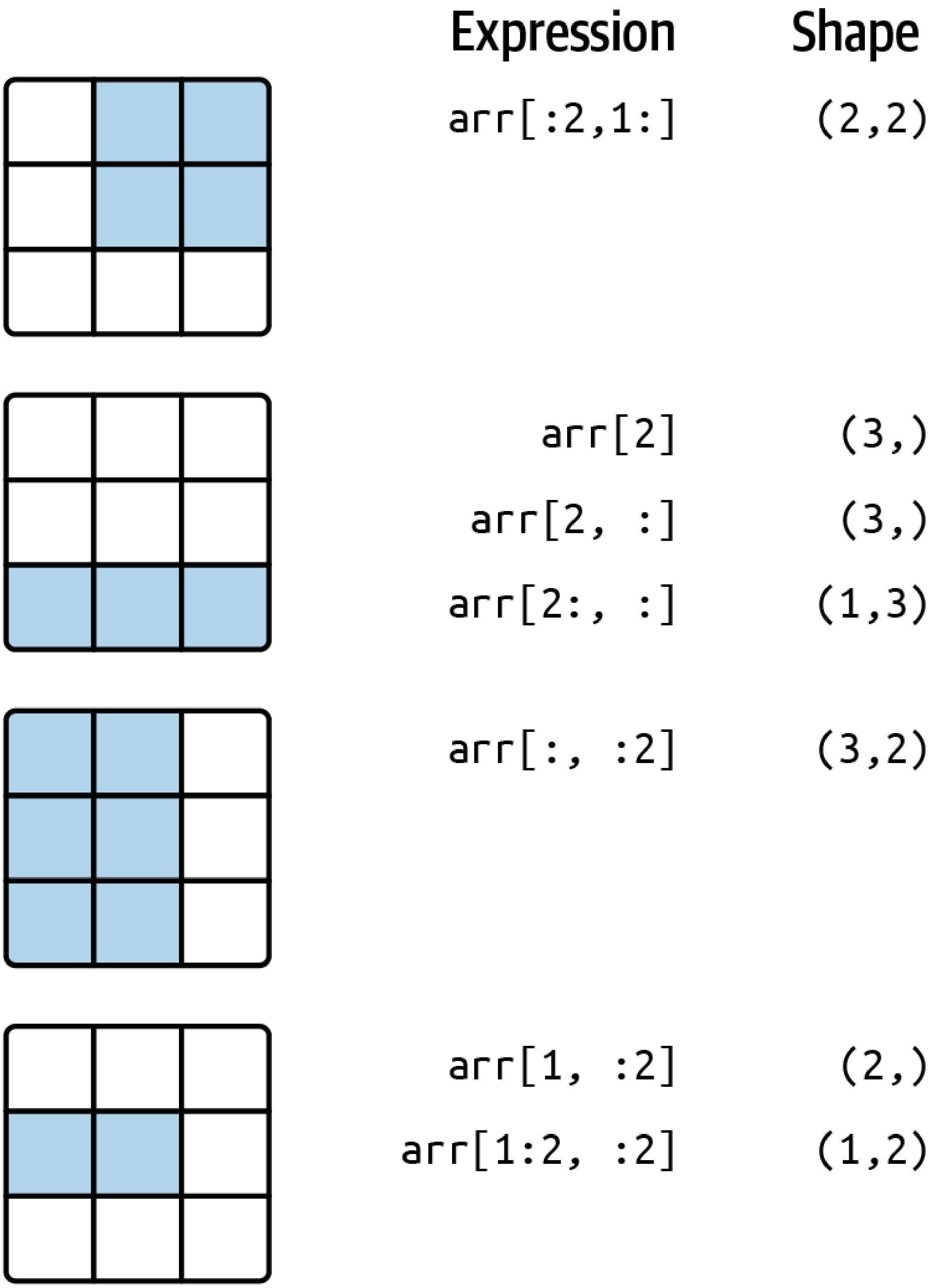

Figure 4-2. Two-dimensional array slicing

Boolean Indexing

Let’s consider an example where we have some data in an array and an array of names with duplicates:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2],
   .....:              [-12, -4], [3, 4]])

In [102]: names
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [103]: data
Out[103]:
array([[  4,   7],
     [  0,   2],
     [ -5,   6],
     [  0,   0],
     [  1,   2],
     [-12,  -4],
     [  3,   4]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with the 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 [104]: names == "Bob"
Out[104]: array([ True, False, False,  True, False, False, False])
```

This Boolean array can be passed when indexing the array:

```
In [105]: data[names == "Bob"]
Out[105]:
array([[4, 7],
   [0, 0]])
```

The Boolean array must be of the same length as the array 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 these examples, I select from the rows where names == "Bob" and index the columns, too:

```
In [106]: data[names == "Bob", 1:]

Out[106]:

array([[7],

   [0]])


In [107]: data[names == "Bob", 1]

Out[107]: array([7, 0])
```

To select everything but "Bob" you can either use != or negate the condition using ~:

```
In [108]: names != "Bob"

Out[108]: array([False,  True,  True, False,  True,  True,  True])


In [109]: ~(names == "Bob")

Out[109]: array([False,  True,  True, False,  True,  True,  True])


In [110]: data[~(names == "Bob")]

Out[110]:

array([[  0,   2],

   [ -5,   6],

   [  1,   2],

   [-12,  -4],
       [  3,   4]])
```

The ~ operator can be useful when you want to invert a Boolean array referenced by a variable:

```
In [111]: cond = names == "Bob"


In [112]: data[~cond]

Out[112]:

array([[  0,   2],

       [ -5,   6],

       [  1,   2],

       [-12,  -4],

       [  3,   4]])
```

To select two of the three names to combine multiple Boolean conditions, use Boolean arithmetic operators like & (and) and | (or):

```
In [113]: mask = (names == "Bob") | (names == "Will")


In [114]: mask

Out[114]: array([ True, False,  True,  True,  True, False, False])


In [115]: data[mask]

Out[115]:

array([[ 4,  7],

       [-5,  6],

       [ 0,  0],

       [ 1,  2]])
```

Selecting data from an array by Boolean indexing and assigning the result to a new variable *always* creates a copy of the data, even if the returned array is unchanged.

Caution

The Python keywords and and or do not work with Boolean arrays. Use & (and) and | (or) instead.

Setting values with Boolean arrays works by substituting the value or values on the righthand side into the locations where the Boolean array's values are True. To set all of the negative values in data to 0, we need only do:

```
In [116]: data[data < 0] = 0

In [117]: data
Out[117]:
array([[4, 7],
   [0, 2],
   [0, 6],
   [0, 0],
   [1, 2],
   [0, 0],
   [3, 4]])
```

You can also set whole rows or columns using a one-dimensional Boolean array:

```
In [118]: data[names != "Joe"] = 7

In [119]: data
Out[119]:
array([[7, 7],
   [0, 2],
   [7, 7],
     [7, 7],
     [7, 7],
     [0, 0],
     [3, 4]])
```

As we will see later, these types of operations on two-dimensional data are convenient to do with pandas.

Fancy Indexing

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

```
In [120]: arr = np.zeros((8, 4))


In [121]: for i in range(8):
  .....:     arr[i] = i


In [122]: arr
Out[122]:
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 a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [123]: arr[[4, 3, 0, 6]]

Out[123]:

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 selects rows from the end:

```
In [124]: arr[[-3, -5, -7]]

Out[124]:

array([[5., 5., 5., 5.],

    [3., 3., 3., 3.],

    [1., 1., 1., 1.]])
```

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

```
In [125]: arr = np.arange(32).reshape((8, 4))


In [126]: arr

Out[126]:

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 [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[127]: array([ 4, 23, 29, 10])
```

To learn more about the reshape method, have a look at Appendix A.

Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. The result of fancy indexing with as many integer arrays as there are axes is always one-dimensional.

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 [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[128]:
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 when assigning the result to a new variable. If you assign values with fancy indexing, the indexed values will be modified:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[129]: array([ 4, 23, 29, 10])


In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0


In [131]: arr
Out[131]:
array([[ 0,  1,  2,  3],
      [ 0,  5,  6,  7],
      [ 8,  9,  0, 11],
      [12, 13, 14, 15],
      [16, 17, 18, 19],
      [20, 21, 22,  0],
      [24, 25, 26, 27],
      [28,  0, 30, 31]])
```

Transposing Arrays and Swapping Axes

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

```
In [132]: arr = np.arange(15).reshape((3, 5))


In [133]: arr
Out[133]:
array([[ 0,  1,  2,  3,  4],
      [ 5,  6,  7,  8,  9],
      [10, 11, 12, 13, 14]])


In [134]: arr.T
Out[134]:
array([[ 0,  5, 10],
      [ 1,  6, 11],
      [ 2,  7, 12],
      [ 3,  8, 13],
      [ 4,  9, 14]])
```

When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using numpy.dot:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1
]])

In [136]: arr
Out[136]:
array([[ 0,  1,  0],
     [ 1,  2, -2],
     [ 6,  3,  2],
     [-1,  0, -1],
     [ 1,  0,  1]])

In [137]: np.dot(arr.T, arr)
Out[137]:
array([[39, 20, 12],
     [20, 14,  2],
     [12,  2, 10]])
```

The @ infix operator is another way to do matrix multiplication:

```
In [138]: arr.T @ arr
Out[138]:
array([[39, 20, 12],
     [20, 14,  2],
     [12,  2, 10]])
```

Simple transposing with .T is a special case of swapping axes. ndarray has the method swapaxes, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [139]: arr
Out[139]:
array([[ 0,  1,  0],
    [ 1,  2, -2],
    [ 6,  3,  2],
    [-1,  0, -1],
    [ 1,  0,  1]])

In [140]: arr.swapaxes(0, 1)
Out[140]:
array([[ 0,  1,  6, -1,  1],
    [ 1,  2,  3,  0,  0],
    [ 0, -2,  2, -1,  1]])
```

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

4.2 Pseudorandom Number Generation

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

```
In [141]: samples = np.random.standard_normal(size=(4, 4))

In [142]: samples
Out[142]:
array([[-0.2047,  0.4789, -0.5194, -0.5557],
    [ 1.9658,  1.3934,  0.0929,  0.2817],
    [ 0.769 ,  1.2464,  1.0072, -1.2962],
    [ 0.275 ,  0.2289,  1.3529,  0.8864]])
```

Python’s built-in random module, by contrast, samples only 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 [143]: from random import normalvariate

In [144]: N = 1_000_000

In [145]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.05 s +- 14.5 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [146]: %timeit np.random.standard_normal(N)
21.8 ms +- 212 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

These random numbers are not truly random (rather, *pseudorandom*) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like numpy.random.standard_normal use the numpy.random module’s default random number generator, but your code can be configured to use an explicit generator:

```
In [147]: rng = np.random.default_rng(seed=12345)

In [148]: data = rng.standard_normal((2, 3))
```

The seed argument is what determines the initial state of the generator, and the state changes each time the rng object is used to generate data. The generator object rng is also isolated from other code which might use the numpy.random module:

```
In [149]: type(rng)
Out[149]: numpy.random._generator.Generator
```

See Table 4-3 for a partial list of methods available on random generator objects like rng. I will use the rng object I created above to generate random data throughout the rest of the chapter.

Method	Description
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in place
uniform	Draw samples from a uniform distribution
integers	Draw random integers from a given low-to-high range
standard_normal	Draw samples from a normal distribution with mean 0 and standard deviation 1
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

Method	Description

Table 4-3. NumPy random number generator methods

4.3 Universal Functions: Fast Element-Wise Array Functions

A universal function, or *ufunc*, is a function that performs element-wise 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 element-wise transformations, like numpy.sqrt or numpy.exp:

```
In [150]: arr = np.arange(10)


In [151]: arr

Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])


In [152]: np.sqrt(arr)

Out[152]:

array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458,

       2.8284, 3.    ])


In [153]: np.exp(arr)

Out[153]:

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 numpy.add or numpy.maximum, take two arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [154]: x = rng.standard_normal(8)


In [155]: y = rng.standard_normal(8)
In [156]: x

Out[156]:

array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,

       0.9022])


In [157]: y

Out[157]:

array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  1.3223,

      -0.2997])


In [158]: np.maximum(x, y)

Out[158]:

array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  1.3223,

       0.9022])
```

In this example, numpy.maximum computed the element-wise maximum of the elements in x and y.

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

```
In [159]: arr = rng.standard_normal(7) * 5


In [160]: arr

Out[160]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])


In [161]: remainder, whole_part = np.modf(arr)


In [162]: remainder

Out[162]: array([ 0.5146, -0.1079, -0.7909,  0.2474, -0.718 , -0.4084,  0.6237])
```

```
In [163]: whole_part

Out[163]: array([ 4., -8., -0.,  2., -6., -0.,  8.])
```

Ufuncs accept an optional out argument that allows them to assign their results into an existing array rather than create a new one:

```
In [164]: arr

Out[164]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])


In [165]: out = np.zeros_like(arr)


In [166]: np.add(arr, 1)

Out[166]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])


In [167]: np.add(arr, 1, out=out)

Out[167]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])


In [168]: out

Out[168]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])
```

See Tables 4-4 and 4-5 for a listing of some of NumPy's ufuncs. New ufuncs continue to be added to NumPy, so consulting the online NumPy documentation is the best way to get a comprehensive listing and stay up to date.

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values

Function	**Description**
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 that number)
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 arrays
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

Function	Description
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. Some unary 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)

Function	Description
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding Boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and	Compute element-wise truth value of AND (&) logical operation
logical_or	Compute element-wise truth value of OR (|) logical operation
logical_xor	Compute element-wise truth value of XOR (^) logical operation

Table 4-5. Some binary universal functions

4.4 Array-Oriented Programming with 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 referred to by some people as *vectorization*. In general, vectorized array operations will usually be significantly faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Appendix A, I 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 numpy.meshgrid function takes two one-dimensional arrays and produces two two-dimensional matrices corresponding to all pairs of (x, y) in the two arrays:

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 equally spaced points


In [170]: xs, ys = np.meshgrid(points, points)


In [171]: ys
```

```
Out[171]:
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 matter of writing the same expression you would write with two points:

```
In [172]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [173]: z
Out[173]:
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]])
```

As a preview of Chapter 9, I use matplotlib to create visualizations of this two-dimensional array:

```
In [174]: import matplotlib.pyplot as plt
```

```
In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
```

```
Out[175]: <matplotlib.image.AxesImage at 0x7f7132db3ac0>


In [176]: plt.colorbar()

Out[176]: <matplotlib.colorbar.Colorbar at 0x7f713a5833a0>


In [177]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")

Out[177]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values'

)
```

In Figure 4-3, I used the matplotlib function imshow to create an image plot from a two-dimensional array of function values.

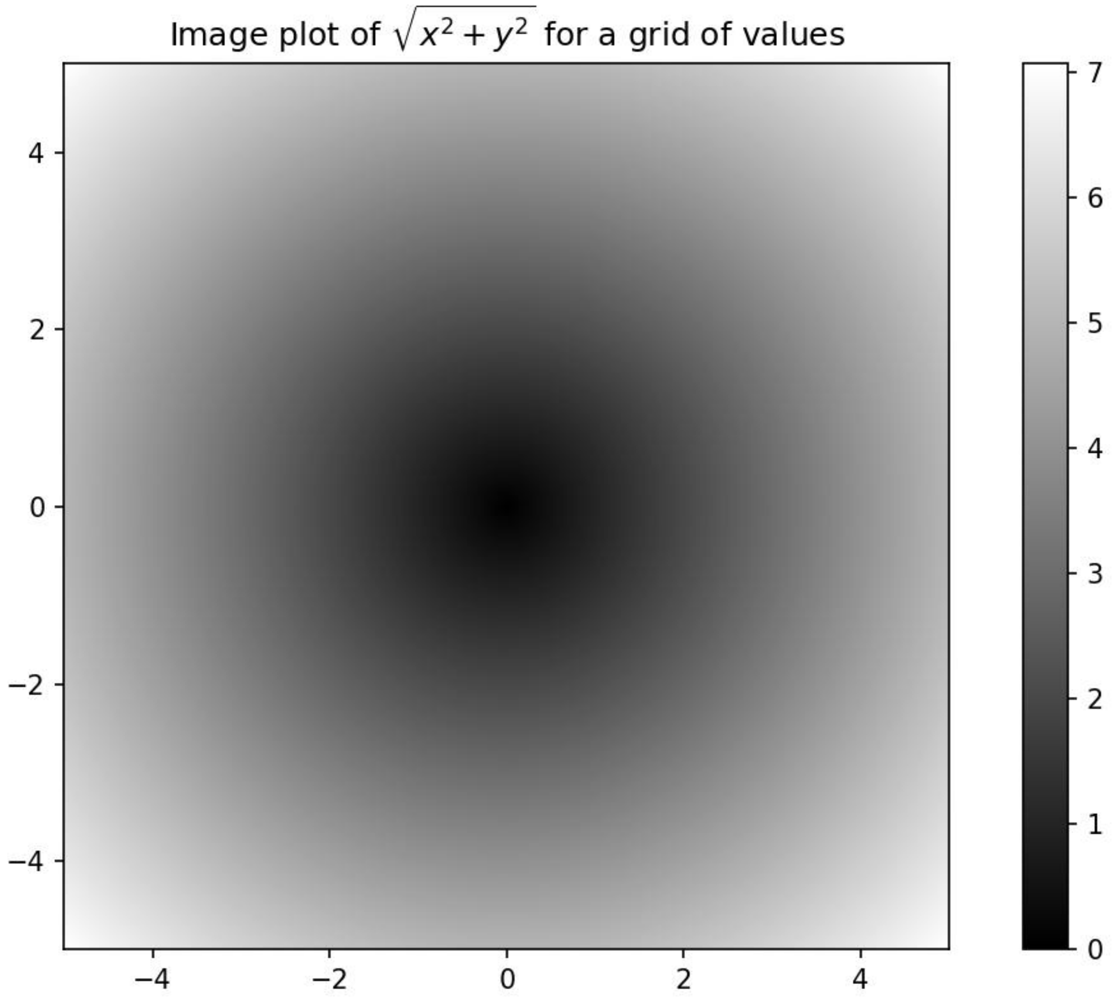

Figure 4-3. Plot of function evaluated on a grid

If you’re working in IPython, you can close all open plot windows by executing plt.close("all"):

```
In [179]: plt.close("all")
```

Note

The term *vectorization* is used to describe some other computer science concepts, but in this book I use it to describe operations on whole arrays of data at once rather than going value by value using a Python for loop.

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


In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])


In [182]: 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, and otherwise take the value from yarr. A list comprehension doing this might look like:

```
In [183]: result = [(x if c else y)

   .....:         for x, y, c in zip(xarr, yarr, cond)]


In [184]: result

Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With numpy.where you can do this with a single function call:

```
In [185]: result = np.where(cond, xarr, yarr)


In [186]: result

Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

The second and third arguments to numpy.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 possible to do with numpy.where:

```
In [187]: arr = rng.standard_normal((4, 4))


In [188]: arr

Out[188]:

array([[ 2.6182,  0.7774,  0.8286, -0.959 ],

    [-1.2094, -1.4123,  0.5415,  0.7519],

    [-0.6588, -1.2287,  0.2576,  0.3129],

    [-0.1308,  1.27  , -0.093 , -0.0662]])


In [189]: arr > 0

Out[189]:

array([[ True,  True,  True, False],

    [False, False,  True,  True],

    [False, False,  True,  True],

    [False,  True, False, False]])


In [190]: np.where(arr > 0, 2, -2)

Out[190]:
```

```
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

You can combine scalars and arrays when using numpy.where. For example, I can replace all positive values in arr with the constant 2, like so:

```
In [191]: np.where(arr > 0, 2, arr) # set only positive values to 2

Out[191]:

array([[ 2.    ,  2.    ,  2.    , -0.959 ],
       [-1.2094, -1.4123,  2.    ,  2.    ],
       [-0.6588, -1.2287,  2.    ,  2.    ],
       [-0.1308,  2.    , -0.093 , -0.0662]])
```

Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (sometimes called *reductions*) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function. When you use the NumPy function, like numpy.sum, you have to pass the array you want to aggregate as the first argument.

Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [192]: arr = rng.standard_normal((5, 4))


In [193]: arr

Out[193]:

array([[-1.1082,  0.136 ,  1.3471,  0.0611],
       [ 0.0709,  0.4337,  0.2775,  0.5303],
       [ 0.5367,  0.6184, -0.795 ,  0.3   ],
       [-1.6027,  0.2668, -1.2616, -0.0713],
       [ 0.474 , -0.4149,  0.0977, -1.6404]])

In [194]: arr.mean()
Out[194]: -0.08719744457434529

In [195]: np.mean(arr)
Out[195]: -0.08719744457434529

In [196]: arr.sum()
Out[196]: -1.743948891486906
```

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

```
In [197]: arr.mean(axis=1)
Out[197]: array([ 0.109 ,  0.3281,  0.165 , -0.6672, -0.3709])

In [198]: arr.sum(axis=0)
Out[198]: array([-1.6292,  1.0399, -0.3344, -0.8203])
```

Here, arr.mean(axis=1) means “compute mean across the columns,”
where arr.sum(axis=0) means “compute sum down the rows.”

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

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [200]: arr.cumsum()
Out[200]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like cumsum return an array of the same size but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])


In [202]: arr

Out[202]:

array([[0, 1, 2],

    [3, 4, 5],

    [6, 7, 8]])
```

The expression arr.cumsum(axis=0) computes the cumulative sum along the rows, while arr.cumsum(axis=1) computes the sums along the columns:

```
In [203]: arr.cumsum(axis=0)

Out[203]:

array([[ 0,  1,  2],

    [ 3,  5,  7],

    [ 9, 12, 15]])


In [204]: arr.cumsum(axis=1)

Out[204]:

array([[ 0,  1,  3],

    [ 3,  7, 12],

    [ 6, 13, 21]])
```

See Table 4-6 for a full listing. We'll see many examples of these methods in action in later chapters.

Method	Description
sum	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
mean	Arithmetic mean; invalid (returns NaN) on zero-length arrays
std, var	Standard deviation and variance, respectively
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

Table 4-6. Basic array statistical methods

Methods for Boolean Arrays

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

```
In [205]: arr = rng.standard_normal(100)


In [206]: (arr > 0).sum() # Number of positive values

Out[206]: 48


In [207]: (arr <= 0).sum() # Number of non-positive values

Out[207]: 52
```

The parentheses here in the expression (arr > 0).sum() are necessary to be able to call sum() on the temporary result of arr > 0.

Two additional methods, any and all, are 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 [208]: bools = np.array([False, False, True, False])


In [209]: bools.any()

Out[209]: True


In [210]: bools.all()

Out[210]: False
```

These methods also work with non-Boolean arrays, where nonzero elements are treated as True.

Sorting

Like Python’s built-in list type, NumPy arrays can be sorted in place with the sort method:

```
In [211]: arr = rng.standard_normal(6)


In [212]: arr

Out[212]: array([ 0.0773, -0.6839, -0.7208,  1.1206, -0.0548, -0.0824])


In [213]: arr.sort()


In [214]: arr

Out[214]: array([-0.7208, -0.6839, -0.0824, -0.0548,  0.0773,  1.1206])
```

You can sort each one-dimensional section of values in a multidimensional array in place along an axis by passing the axis number to sort. In this example data:

```
In [215]: arr = rng.standard_normal((5, 3))


In [216]: arr

Out[216]:

array([[ 0.936 ,  1.2385,  1.2728],

     [ 0.4059, -0.0503,  0.2893],

     [ 0.1793,  1.3975,  0.292 ],

     [ 0.6384, -0.0279,  1.3711],

     [-2.0528,  0.3805,  0.7554]])
```

arr.sort(axis=0) sorts the values within each column, while arr.sort(axis=1) sorts across each row:

```
In [217]: arr.sort(axis=0)


In [218]: arr

Out[218]:

array([[-2.0528, -0.0503,  0.2893],

     [ 0.1793, -0.0279,  0.292 ],

     [ 0.4059,  0.3805,  0.7554],

     [ 0.6384,  1.2385,  1.2728],

     [ 0.936 ,  1.3975,  1.3711]])


In [219]: arr.sort(axis=1)


In [220]: arr

Out[220]:

array([[-2.0528, -0.0503,  0.2893],

     [-0.0279,  0.1793,  0.292 ],
     [ 0.3805,  0.4059,  0.7554],
     [ 0.6384,  1.2385,  1.2728],
     [ 0.936 ,  1.3711,  1.3975]])
```

The top-level method numpy.sort returns a sorted copy of an array (like the Python built-in function sorted) instead of modifying the array in place. For example:

```
In [221]: arr2 = np.array([5, -10, 7, 1, 0, -3])

In [222]: sorted_arr2 = np.sort(arr2)

In [223]: sorted_arr2
Out[223]: array([-10,  -3,   0,   1,   5,   7])
```

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

Unique and Other Set Logic

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

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])

In [225]: np.unique(names)
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [227]: np.unique(ints)
Out[227]: array([1, 2, 3, 4])
```

Contrast numpy.unique with the pure Python alternative:

```
In [228]: sorted(set(names))

Out[228]: ['Bob', 'Joe', 'Will']
```

In many cases, the NumPy version is faster and returns a NumPy array rather than a Python list.

Another function, numpy.in1d, tests membership of the values in one array in another, returning a Boolean array:

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])


In [230]: np.in1d(values, [2, 3, 6])

Out[230]: array([ True, False, False,  True,  True, False,  True])
```

See Table 4-7 for a listing of array set operations in NumPy.

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

Method	Description
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

Table 4-7. Array set operations

4.5 File Input and Output with Arrays

NumPy is able to save and load data to and from disk in some text or binary formats. In this section I discuss only NumPy’s built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data (see Chapter 6 for much more).

numpy.save and numpy.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 [231]: arr = np.arange(10)


In [232]: 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 with numpy.load:

```
In [233]: np.load("some_array.npy")

Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You can save multiple arrays in an uncompressed archive using numpy.savez and passing the arrays as keyword arguments:

```
In [234]: np.savez("array_archive.npz", a=arr, b=arr)
```

When loading an *.npz* file, you get back a dictionary-like object that loads the individual arrays lazily:

```
In [235]: arch = np.load("array_archive.npz")

In [236]: arch["b"]

Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use numpy.savez_compressed instead:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

4.6 Linear Algebra

Linear algebra operations, like matrix multiplication, decompositions, determinants, and other square matrix math, are an important part of many array libraries. Multiplying two two-dimensional arrays with * is an element-wise product, while matrix multiplications require using a function. Thus, there is a function dot, both an array method and a function in the numpy namespace, for matrix multiplication:

```
In [241]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [242]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [243]: x

Out[243]:

array([[1., 2., 3.],

   [4., 5., 6.]])
```

```
In [244]: y

Out[244]:

array([[ 6., 23.],

    [-1.,  7.],

    [ 8.,  9.]])


In [245]: x.dot(y)

Out[245]:

array([[ 28.,  64.],

    [ 67., 181.]])
```

x.dot(y) is equivalent to np.dot(x, y):

```
In [246]: np.dot(x, y)

Out[246]:

array([[ 28.,  64.],

    [ 67., 181.]])
```

A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

```
In [247]: x @ np.ones(3)

Out[247]: array([ 6., 15.])
```

numpy.linalg has a standard set of matrix decompositions and things like inverse and determinant:

```
In [248]: from numpy.linalg import inv, qr
In [249]: X = rng.standard_normal((5, 5))

In [250]: mat = X.T @ X

In [251]: inv(mat)
Out[251]:
array([[  3.4993,   2.8444,   3.5956, -16.5538,   4.4733],
       [  2.8444,   2.5667,   2.9002, -13.5774,   3.7678],
       [  3.5956,   2.9002,   4.4823, -18.3453,   4.7066],
       [-16.5538, -13.5774, -18.3453,  84.0102, -22.0484],
       [  4.4733,   3.7678,   4.7066, -22.0484,   6.0525]])

In [252]: mat @ inv(mat)
Out[252]:
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.]])
```

The expression X.T.dot(X) computes the dot product of X with its transpose X.T.

See Table 4-8 for a list of some of the most commonly used linear algebra 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

Function	Description
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 pseudoinverse 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

Table 4-8. Commonly used numpy.linalg functions

4.7 Example: Random Walks

The simulation of *random walks* provides an illustrative application of utilizing array operations. Let’s first consider a simple random walk starting at 0 with steps of 1 and –1 occurring with equal probability.

Here is a pure Python way to implement a single random walk with 1,000 steps using the built-in random module:

```
#! blockstart

import random

position = 0

walk = [position]

nsteps = 1000
```

```
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
#! blockend
```

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

```
In [255]: plt.plot(walk[:100])
```

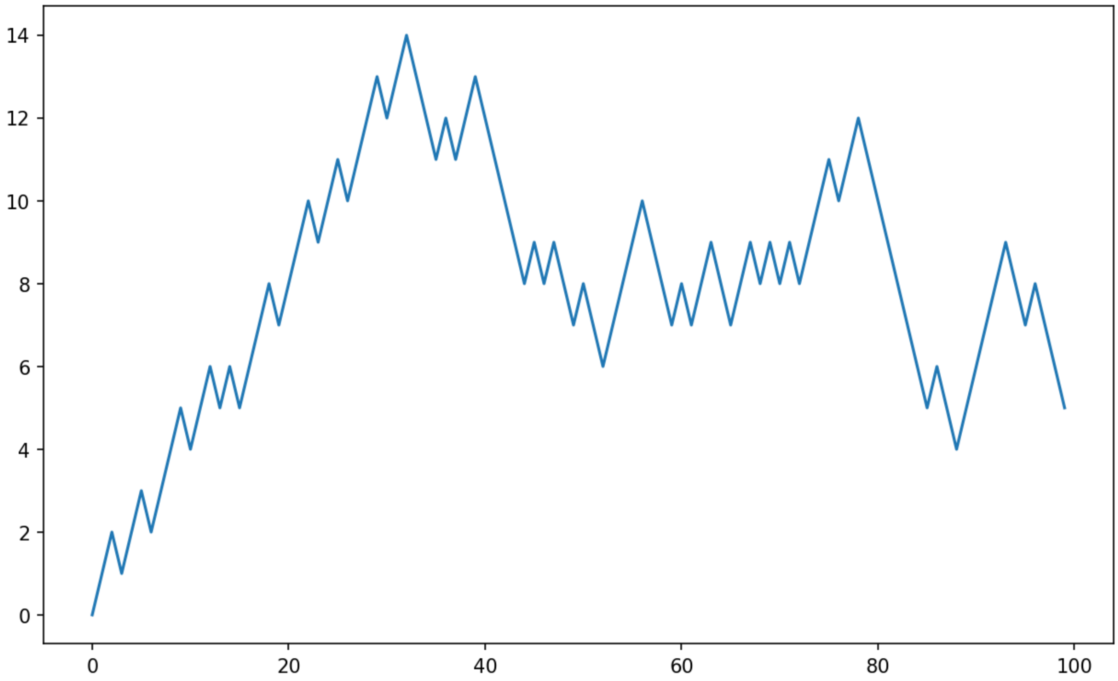

Figure 4-4. A simple random walk

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

```
In [256]: nsteps = 1000

In [257]: rng = np.random.default_rng(seed=12345)  # fresh random generator
In [258]: draws = rng.integers(0, 2, size=nsteps)
```

```
In [259]: steps = np.where(draws == 0, 1, -1)
```

```
In [260]: walk = steps.cumsum()
```

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

```
In [261]: walk.min()

Out[261]: -8
```

```
In [262]: walk.max()

Out[262]: 50
```

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, we can compute this using argmax, which returns the first index of the maximum value in the Boolean array (True is the maximum value):

```
In [263]: (np.abs(walk) >= 10).argmax()

Out[263]: 155
```

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 five thousand of them, you can generate all of the random walks with minor modifications to the preceding code. If passed a 2-tuple, the numpy.random functions will generate a two-dimensional array of draws, and we can compute the cumulative sum for each row to compute all five thousand random walks in one shot:

```
In [264]: nwalks = 5000


In [265]: nsteps = 1000


In [266]: draws = rng.integers(0, 2, size=(nwalks, nsteps)) # 0 or 1


In [267]: steps = np.where(draws > 0, 1, -1)


In [268]: walks = steps.cumsum(axis=1)


In [269]: walks

Out[269]:

array([[  1,   2,   3, ...,  22,  23,  22],

       [  1,   0,  -1, ..., -50, -49, -48],

       [  1,   2,   3, ...,  50,  49,  48],

       ...,

       [ -1,  -2,  -1, ..., -10,  -9, -10],

       [ -1,  -2,  -3, ...,   8,   9,   8],

       [ -1,   0,   1, ...,  -4,  -3,  -2]])
```

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

```
In [270]: walks.max()

Out[270]: 114


In [271]: walks.min()
```

```
Out[271]: -120
```

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 [272]: hits30 = (np.abs(walks) >= 30).any(axis=1)

In [273]: hits30
Out[273]: array([False,  True,  True, ...,  True, False,  True])

In [274]: hits30.sum() # Number that hit 30 or -30
Out[274]: 3395
```

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

```
In [275]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(axis=1)

In [276]: crossing_times
Out[276]: array([201, 491, 283, ..., 219, 259, 541])
```

Lastly, we compute the average minimum crossing time:

```
In [277]: crossing_times.mean()
Out[277]: 500.5699558173785
```

Feel free to experiment with other distributions for the steps other than equal-sized coin flips. You need only use a different random generator method, like standard_normal to generate normally distributed steps with some mean and standard deviation:

```
In [278]: draws = 0.25 * rng.standard_normal((nwalks, nsteps))
```

Note

Keep in mind that this vectorized approach requires creating an array with nwalks * nsteps elements, which may use a large amount of memory for large simulations. If memory is more constrained, then a different approach will be required.

4.8 Conclusion

While much of the rest of the book will focus on building data wrangling skills with pandas, we will continue to work in a similar array-based style. In Appendix A, we will dig deeper into NumPy features to help you further develop your array computing skills.