O'REILLY®

Second Edition

Python Data Science Handbook

Essential Tools for Working with Data

Jake VanderPlas

Table of Contents

Part V. Machine Learning

PART II
Introduction to NumPy

This part of the book, along with Part III, outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and in a wide range of formats, including collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, many datasets can be represented fundamentally as arrays of numbers.

For example, images—particularly digital images—can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, such as binary digits representing the frequency of certain words or pairs of words. No matter what the data is, the first step in making it analyzable will be to transform it into arrays of numbers. (We will discuss some specific examples of this process in Chapter 40.)

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

This part of the book will cover NumPy in detail. NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in

Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

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

```
In [1]: import numpy
        numpy.__version__
Out[1]: '1.21.2'
```

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

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

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

Reminder About Built-in Documentation

As you read through this part of the book, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the Tab completion feature), as well as the documentation of various functions (using the `?` character). For a refresher on these, revisit Chapter 1.

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

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

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

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

Numpy offers more detailed documentation, along with tutorials and other resources.

CHAPTER 4

Understanding Data Types in Python

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

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

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

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

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

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

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

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

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

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

A Python Integer Is More Than Just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly disguised C structure, which contains not only its value, but other information as well. For example, when we define an integer in Python, such as `x = 10000`, `x` is not just a "raw" integer. It's actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.10 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

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

A single integer in Python 3.10 actually contains four pieces:

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

This means that there is some overhead involved in storing an integer in Python as compared to a compiled language like C, as illustrated in Figure 4-1.

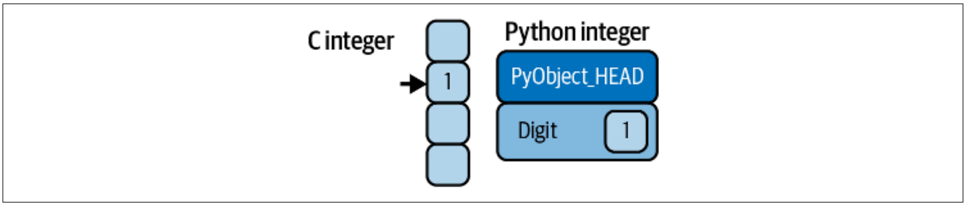

Figure 4-1. The difference between C and Python integers

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

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

A Python List Is More Than Just a List

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

```
In [1]: L = list(range(10))
        L
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: type(L[0])
Out[2]: int
```

Or, similarly, a list of strings:

```
In [3]: L2 = [str(c) for c in L]
        L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [4]: type(L2[0])
Out[4]: str
```

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

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

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type, reference count, and other information. That is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant, so it can be much more efficient to store the data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in Figure 4-2.

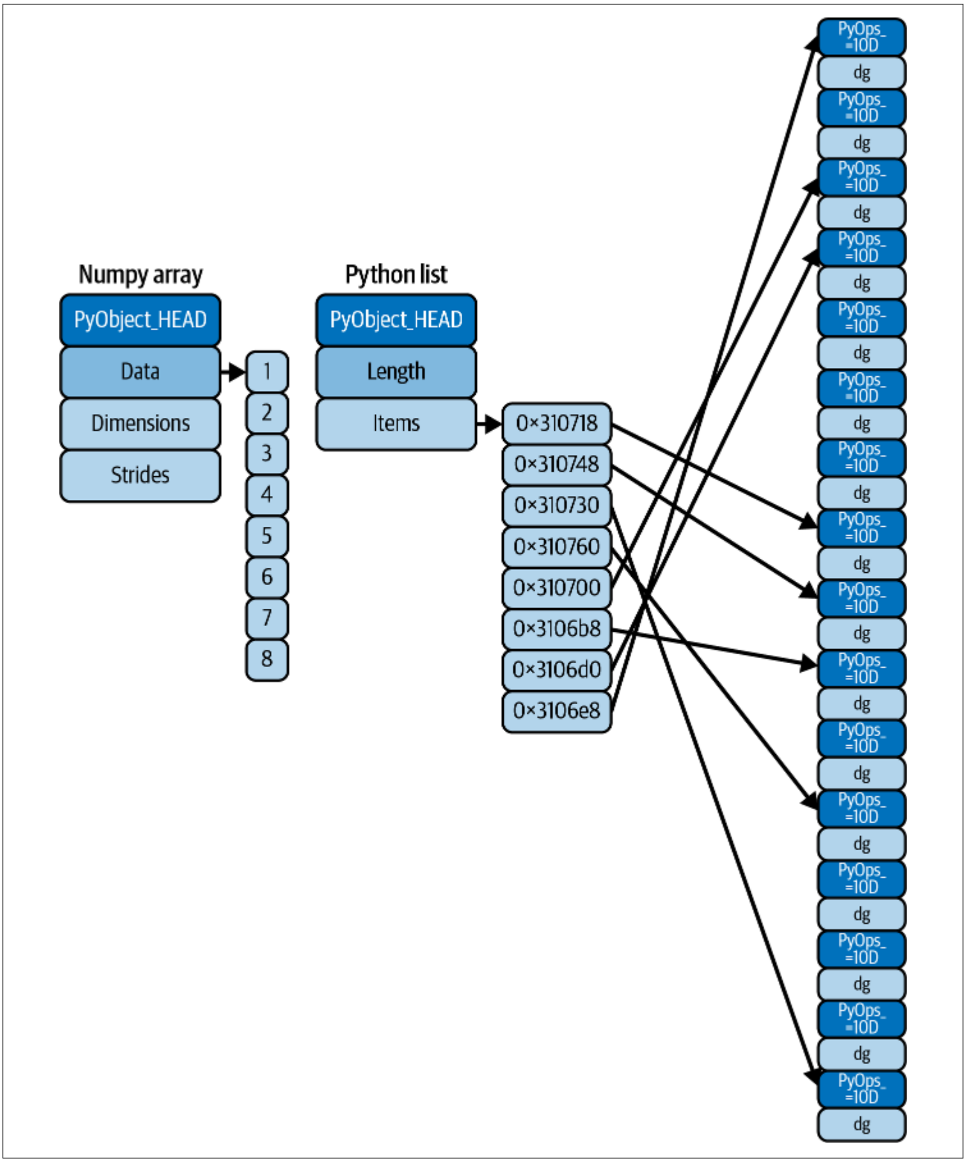

Figure 4-2. The difference between C and Python lists

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

Fixed-Type Arrays in Python

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

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later chapters; next, I'll show you a few different ways of creating a NumPy array.

Creating Arrays from Python Lists

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

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

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

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

Remember that unlike Python lists, NumPy arrays can only contain data of the same type. If the types do not match, NumPy will upcast them according to its type promotion rules; here, integers are upcast to floating point:

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

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

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

Finally, unlike Python lists, which are always one-dimensional sequences, NumPy arrays can be multidimensional. Here's one way of initializing a multidimensional array using a list of lists:

```
In [11]: # Nested lists result in multidimensional arrays
         np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

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

Creating Arrays from Scratch

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

```
In [12]: # Create a length-10 integer array filled with 0s
         np.zeros(10, dtype=int)
Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

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

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

In [15]: # Create an array filled with a linear sequence
         # starting at 0, ending at 20, stepping by 2
         # (this is similar to the built-in range function)
         np.arange(0, 20, 2)
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [16]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)
Out[16]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])

In [17]: # Create a 3x3 array of uniformly distributed
         # pseudorandom values between 0 and 1
         np.random.random((3, 3))
Out[17]: array([[0.09610171, 0.88193001, 0.70548015],
```

```
                [0.35885395, 0.91670468, 0.8721031 ],
                [0.73237865, 0.09708562, 0.52506779]])

In [18]: # Create a 3x3 array of normally distributed pseudorandom
         # values with mean 0 and standard deviation 1
         np.random.normal(0, 1, (3, 3))
Out[18]: array([[-0.46652655, -0.59158776, -1.05392451],
                [-1.72634268,  0.03194069, -0.51048869],
                [ 1.41240208,  1.77734462, -0.43820037]])

In [19]: # Create a 3x3 array of pseudorandom integers in the interval [0, 10)
         np.random.randint(0, 10, (3, 3))
Out[19]: array([[4, 3, 8],
                [6, 5, 0],
                [1, 1, 4]])

In [20]: # Create a 3x3 identity matrix
         np.eye(3)
Out[20]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])

In [21]: # Create an uninitialized array of three integers; the values will be
         # whatever happens to already exist at that memory location
         np.empty(3)
Out[21]: array([1., 1., 1.])
```

NumPy Standard Data Types

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

The standard NumPy data types are listed in Table 4-1. Note that when constructing an array, they can be specified using a string:

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

Or using the associated NumPy object:

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

More advanced type specification is possible, such as specifying big- or little-endian numbers; for more information, refer to the NumPy documentation. NumPy also supports compound data types, which will be covered in Chapter 12.

Table 4-1. Standard NumPy data types

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

CHAPTER 5

The Basics of NumPy Arrays

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

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

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

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

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

Reshaping of arrays
: Changing the shape of a given array

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

NumPy Array Attributes

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

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)  # seed for reproducibility

        x1 = rng.integers(10, size=6)  # one-dimensional array
        x2 = rng.integers(10, size=(3, 4))  # two-dimensional array
        x3 = rng.integers(10, size=(3, 4, 5))  # three-dimensional array
```

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

```
In [2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
        print("dtype:   ", x3.dtype)
Out[2]: x3 ndim:  3
        x3 shape: (3, 4, 5)
        x3 size:  60
        dtype:    int64
```

For more discussion of data types, see Chapter 4.

Array Indexing: Accessing Single Elements

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

```
In [3]: x1
Out[3]: array([9, 4, 0, 3, 8, 6])

In [4]: x1[0]
Out[4]: 9

In [5]: x1[4]
Out[5]: 8
```

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

```
In [6]: x1[-1]
Out[6]: 6

In [7]: x1[-2]
Out[7]: 8
```

In a multidimensional array, items can be accessed using a comma-separated (*row*, *column*) tuple:

```
In [8]: x2
Out[8]: array([[3, 1, 3, 7],
               [4, 0, 2, 3],
               [0, 0, 6, 9]])
```

```
In [9]: x2[0, 0]
Out[9]: 3
```

```
In [10]: x2[2, 0]
Out[10]: 0
```

```
In [11]: x2[2, -1]
Out[11]: 9
```

Values can also be modified using any of the preceding index notation:

```
In [12]: x2[0, 0] = 12
         x2
Out[12]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])
```

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

```
In [13]: x1[0] = 3.14159  # this will be truncated!
         x1
Out[13]: array([3, 4, 0, 3, 8, 6])
```

Array Slicing: Accessing Subarrays

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

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

If any of these are unspecified, they default to the values `start=0`, `stop=<size of dimension>`, `step=1`. Let's look at some examples of accessing subarrays in one dimension and in multiple dimensions.

One-Dimensional Subarrays

Here are some examples of accessing elements in one-dimensional subarrays:

```
In [14]: x1
Out[14]: array([3, 4, 0, 3, 8, 6])
```

```
In [15]: x1[:3]  # first three elements
Out[15]: array([3, 4, 0])

In [16]: x1[3:]  # elements after index 3
Out[16]: array([3, 8, 6])

In [17]: x1[1:4]  # middle subarray
Out[17]: array([4, 0, 3])

In [18]: x1[::2]  # every second element
Out[18]: array([3, 0, 8])

In [19]: x1[1::2]  # every second element, starting at index 1
Out[19]: array([4, 3, 6])
```

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

```
In [20]: x1[::-1]  # all elements, reversed
Out[20]: array([6, 8, 3, 0, 4, 3])

In [21]: x1[4::-2]  # every second element from index 4, reversed
Out[21]: array([8, 0, 3])
```

Multidimensional Subarrays

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

```
In [22]: x2
Out[22]: array([[12,  1,  3,  7],
                [ 4,  0,  2,  3],
                [ 0,  0,  6,  9]])

In [23]: x2[:2, :3]  # first two rows & three columns
Out[23]: array([[12,  1,  3],
                [ 4,  0,  2]])

In [24]: x2[:3, ::2]  # three rows, every second column
Out[24]: array([[12,  3],
                [ 4,  2],
                [ 0,  6]])

In [25]: x2[::-1, ::-1]  # all rows & columns, reversed
Out[25]: array([[ 9,  6,  0,  0],
                [ 3,  2,  0,  4],
                [ 7,  3,  1, 12]])
```

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

```
In [26]: x2[:, 0]  # first column of x2
Out[26]: array([12,  4,  0])
```

```
In [27]: x2[0, :]  # first row of x2
Out[27]: array([12,  1,  3,  7])
```

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

```
In [28]: x2[0]  # equivalent to x2[0, :]
Out[28]: array([12,  1,  3,  7])
```

Subarrays as No-Copy Views

Unlike Python list slices, NumPy array slices are returned as *views* rather than *copies* of the array data. Consider our two-dimensional array from before:

```
In [29]: print(x2)
Out[29]: [[12  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

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

```
In [30]: x2_sub = x2[:2, :2]
         print(x2_sub)
Out[30]: [[12  1]
          [ 4  0]]
```

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

```
In [31]: x2_sub[0, 0] = 99
         print(x2_sub)
Out[31]: [[99  1]
          [ 4  0]]

In [32]: print(x2)
Out[32]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

Some users may find this surprising, but it can be advantageous: for example, when working with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating Copies of Arrays

Despite the features of array views, it's sometimes useful to instead explicitly copy the data within an array or a subarray. This is easiest to do with the `copy` method:

```
In [33]: x2_sub_copy = x2[:2, :2].copy()
         print(x2_sub_copy)
Out[33]: [[99  1]
          [ 4  0]]
```

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

```
In [34]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
Out[34]: [[42  1]
          [ 4  0]]
```

```
In [35]: print(x2)
Out[35]: [[99  1  3  7]
          [ 4  0  2  3]
          [ 0  0  6  9]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays, which can be done with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
In [36]: grid = np.arange(1, 10).reshape(3, 3)
         print(grid)
Out[36]: [[1 2 3]
          [4 5 6]
          [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array, and in most cases the `reshape` method will return a no-copy view of the initial array.

A common reshaping operation is converting a one-dimensional array into a two-dimensional row or column matrix:

```
In [37]: x = np.array([1, 2, 3])
         x.reshape((1, 3))  # row vector via reshape
Out[37]: array([[1, 2, 3]])
```

```
In [38]: x.reshape((3, 1))  # column vector via reshape
Out[38]: array([[1],
                [2],
                [3]])
```

A convenient shorthand for this is to use `np.newaxis` in the slicing syntax:

```
In [39]: x[np.newaxis, :]  # row vector via newaxis
Out[39]: array([[1, 2, 3]])
```

```
In [40]: x[:, np.newaxis]  # column vector via newaxis
Out[40]: array([[1],
                [2],
                [3]])
```

This is a pattern that we will utilize often throughout the remainder of the book.

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. NumPy also provides tools to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

Concatenation of Arrays

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

```
In [41]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])
Out[41]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In [42]: z = np.array([99, 99, 99])
         print(np.concatenate([x, y, z]))
Out[42]: [ 1  2  3  3  2  1 99 99 99]
```

And it can be used for two-dimensional arrays:

```
In [43]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [44]: # concatenate along the first axis
         np.concatenate([grid, grid])
Out[44]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [45]: # concatenate along the second axis (zero-indexed)
         np.concatenate([grid, grid], axis=1)
Out[45]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

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

```
In [46]: # vertically stack the arrays
         np.vstack([x, grid])
Out[46]: array([[1, 2, 3],
                [1, 2, 3],
                [4, 5, 6]])

In [47]: # horizontally stack the arrays
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])
```

```
Out[47]: array([[ 1,  2,  3, 99],
                [ 4,  5,  6, 99]])
```

Similarly, for higher-dimensional arrays, `np.dstack` will stack arrays along the third axis.

Splitting of Arrays

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

```
In [48]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)
Out[48]: [1 2 3] [99 99] [3 2 1]
```

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

```
In [49]: grid = np.arange(16).reshape((4, 4))
         grid
Out[49]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [50]: upper, lower = np.vsplit(grid, [2])
         print(upper)
         print(lower)
Out[50]: [[0 1 2 3]
          [4 5 6 7]]
         [[ 8  9 10 11]
          [12 13 14 15]]
```

```
In [51]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
Out[51]: [[ 0  1]
          [ 4  5]
          [ 8  9]
          [12 13]]
         [[ 2  3]
          [ 6  7]
          [10 11]
          [14 15]]
```

Similarly, for higher-dimensional arrays, `np.dsplit` will split arrays along the third axis.

CHAPTER 6
Computation on NumPy Arrays: Universal Functions

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

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

The Slowness of Loops

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

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

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)

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

        values = rng.integers(1, 10, size=5)
        compute_reciprocals(values)
Out[1]: array([0.11111111, 0.25      , 1.        , 0.33333333, 0.125     ])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow—perhaps surprisingly so! We'll benchmark this with IPython's `%timeit` magic (discussed in "Profiling and Timing Code" on page 26):

```
In [2]: big_array = rng.integers(1, 100, size=1000000)
        %timeit compute_reciprocals(big_array)
Out[2]: 2.61 s ± 192 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

Introducing Ufuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. For simple operations like the element-wise division here, vectorization is as simple as using Python arithmetic operators directly on the array object. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Compare the results of the following two operations:

```
In [3]: print(compute_reciprocals(values))
        print(1.0 / values)
Out[3]: [0.11111111 0.25       1.         0.33333333 0.125     ]
        [0.11111111 0.25       1.         0.33333333 0.125     ]
```

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

```
In [4]: %timeit (1.0 / big_array)
Out[4]: 2.54 ms ± 383 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

```
In [5]: np.arange(5) / np.arange(1, 6)
Out[5]: array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])
```

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

```
In [6]: x = np.arange(9).reshape((3, 3))
        2 ** x
Out[6]: array([[  1,   2,   4],
               [  8,  16,  32],
               [ 64, 128, 256]])
```

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

Exploring NumPy's Ufuncs

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

Array Arithmetic

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

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

```
        print("x * 2  =", x * 2)
        print("x / 2  =", x / 2)
        print("x // 2 =", x // 2)  # floor division
Out[7]: x      = [0 1 2 3]
        x + 5  = [5 6 7 8]
        x - 5  = [-5 -4 -3 -2]
        x * 2  = [0 2 4 6]
        x / 2  = [0.  0.5 1.  1.5]
        x // 2 = [0 0 1 1]
```

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

```
In [8]: print("-x     = ", -x)
        print("x ** 2 = ", x ** 2)
        print("x % 2  = ", x % 2)
Out[8]: -x     =  [ 0 -1 -2 -3]
        x ** 2 =  [0 1 4 9]
        x % 2  =  [0 1 0 1]
```

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

```
In [9]: -(0.5*x + 1) ** 2
Out[9]: array([-1.  , -2.25, -4.  , -6.25])
```

All of these arithmetic operations are simply convenient wrappers around specific ufuncs built into NumPy. For example, the + operator is a wrapper for the add ufunc:

```
In [10]: np.add(x, 2)
Out[10]: array([2, 3, 4, 5])
```

Table 6-1 lists the arithmetic operators implemented in NumPy.

Table 6-1. Arithmetic operators implemented in NumPy

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

Additionally, there are Boolean/bitwise operators; we will explore these in Chapter 9.

Absolute Value

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

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

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

```
In [12]: np.absolute(x)
Out[12]: array([2, 1, 0, 1, 2])

In [13]: np.abs(x)
Out[13]: array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which case it returns the magnitude:

```
In [14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
         np.abs(x)
Out[14]: array([5., 5., 2., 1.])
```

Trigonometric Functions

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

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

Now we can compute some trigonometric functions on these values:

```
In [16]: print("theta      = ", theta)
         print("sin(theta) = ", np.sin(theta))
         print("cos(theta) = ", np.cos(theta))
         print("tan(theta) = ", np.tan(theta))
Out[16]: theta      =  [0.         1.57079633 3.14159265]
         sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
         cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
         tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

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

```
In [17]: x = [-1, 0, 1]
         print("x         = ", x)
         print("arcsin(x) = ", np.arcsin(x))
         print("arccos(x) = ", np.arccos(x))
         print("arctan(x) = ", np.arctan(x))
Out[17]: x         =  [-1, 0, 1]
         arcsin(x) =  [-1.57079633  0.          1.57079633]
```

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

Exponents and Logarithms

Other common operations available in NumPy ufuncs are the exponentials:

```
In [18]: x = [1, 2, 3]
         print("x   =", x)
         print("e^x =", np.exp(x))
         print("2^x =", np.exp2(x))
         print("3^x =", np.power(3., x))
Out[18]: x   = [1, 2, 3]
         e^x = [ 2.71828183  7.3890561  20.08553692]
         2^x = [2. 4. 8.]
         3^x = [ 3.  9. 27.]
```

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

```
In [19]: x = [1, 2, 4, 10]
         print("x        =", x)
         print("ln(x)    =", np.log(x))
         print("log2(x)  =", np.log2(x))
         print("log10(x) =", np.log10(x))
Out[19]: x        = [1, 2, 4, 10]
         ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
         log2(x)  = [0.         1.         2.         3.32192809]
         log10(x) = [0.         0.30103    0.60205999 1.        ]
```

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

```
In [20]: x = [0, 0.001, 0.01, 0.1]
         print("exp(x) - 1 =", np.expm1(x))
         print("log(1 + x) =", np.log1p(x))
Out[20]: exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
         log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

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

Specialized Ufuncs

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

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

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

```
In [22]: # Gamma functions (generalized factorials) and related functions
         x = [1, 5, 10]
         print("gamma(x)     =", special.gamma(x))
         print("ln|gamma(x)| =", special.gammaln(x))
         print("beta(x, 2)   =", special.beta(x, 2))
Out[22]: gamma(x)     = [1.0000e+00 2.4000e+01 3.6288e+05]
         ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
         beta(x, 2)   = [0.5        0.03333333 0.00909091]
```

```
In [23]: # Error function (integral of Gaussian),
         # its complement, and its inverse
         x = np.array([0, 0.3, 0.7, 1.0])
         print("erf(x)  =", special.erf(x))
         print("erfc(x) =", special.erfc(x))
         print("erfinv(x) =", special.erfinv(x))
Out[23]: erf(x)  = [0.         0.32862676 0.67780119 0.84270079]
         erfc(x) = [1.         0.67137324 0.32219881 0.15729921]
         erfinv(x) = [0.         0.27246271 0.73286908        inf]
```

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

Advanced Ufunc Features

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

Specifying Output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. For all ufuncs, this can be done using the `out` argument of the function:

```
In [24]: x = np.arange(5)
         y = np.empty(5)
         np.multiply(x, 10, out=y)
         print(y)
Out[24]: [ 0. 10. 20. 30. 40.]
```

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

```
In [25]: y = np.zeros(10)
         np.power(2, x, out=y[::2])
         print(y)
Out[25]: [ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

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

Aggregations

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

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

```
In [26]: x = np.arange(1, 6)
         np.add.reduce(x)
Out[26]: 15
```

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

```
In [27]: np.multiply.reduce(x)
Out[27]: 120
```

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

```
In [28]: np.add.accumulate(x)
Out[28]: array([ 1,  3,  6, 10, 15])
```

```
In [29]: np.multiply.accumulate(x)
Out[29]: array([  1,   2,   6,  24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in Chapter 7.

Outer Products

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

```
In [30]: x = np.arange(1, 6)
         np.multiply.outer(x, x)
Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods are useful as well, and we will explore them in Chapter 10.

We will also encounter the ability of ufuncs to operate between arrays of different shapes and sizes, a set of operations known as *broadcasting*. This subject is important enough that we will devote a whole chapter to it (see Chapter 8).

Ufuncs: Learning More

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

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab completion and help (`?`) functionality, as described in Chapter 1.

CHAPTER 7

Aggregations: min, max, and Everything in Between

A first step in exploring any dataset is often to compute various summary statistics. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregations are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and try out some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

```
In [1]: import numpy as np
        rng = np.random.default_rng()

In [2]: L = rng.random(100)
        sum(L)
Out[2]: 52.76825337322368
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

```
In [3]: np.sum(L)
Out[3]: 52.76825337322366
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

```
In [4]: big_array = rng.random(1000000)
        %timeit sum(big_array)
        %timeit np.sum(big_array)
Out[4]: 89.9 ms ± 233 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
        521 µs ± 8.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings (`sum(x, 1)` initializes the sum at `1`, while `np.sum(x, 1)` sums along axis `1`), and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
In [5]: min(big_array), max(big_array)
Out[5]: (2.0114398036064074e-07, 0.9999997912802653)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
In [6]: np.min(big_array), np.max(big_array)
Out[6]: (2.0114398036064074e-07, 0.9999997912802653)
```

```
In [7]: %timeit min(big_array)
        %timeit np.min(big_array)
Out[7]: 72 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
        564 µs ± 3.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
In [8]: print(big_array.min(), big_array.max(), big_array.sum())
Out[8]: 2.0114398036064074e-07 0.9999997912802653 499854.0273321711
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multidimensional Aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
In [9]: M = rng.integers(0, 10, (3, 4))
        print(M)
Out[9]: [[0 3 1 2]
```

```
 [1 9 7 0]
 [4 8 3 7]]
```

NumPy aggregations will apply across all elements of a multidimensional array:

```
In [10]: M.sum()
Out[10]: 45
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
In [11]: M.min(axis=0)
Out[11]: array([0, 3, 1, 0])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
In [12]: M.max(axis=1)
Out[12]: array([3, 9, 8])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the dimension of the array that will be *collapsed*, rather than the dimension that will be returned. So, specifying `axis=0` means that axis 0 will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

Other Aggregation Functions

NumPy provides several other aggregation functions with a similar API, and additionally most have a `NaN`-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value (see Chapter 16).

Table 7-1 provides a list of useful aggregation functions available in NumPy.

Table 7-1. Aggregation functions available in NumPy

Function name	NaN-safe version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value

Function name	NaN-safe version	Description
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

You will see these aggregates often throughout the rest of the book.

Example: What Is the Average Height of US Presidents?

Aggregates available in NumPy can act as summary statistics for a set of values. As a small example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a comma-separated list of labels and values:

```
In [13]: !head -4 data/president_heights.csv
Out[13]: order,name,height(cm)
         1,George Washington,189
         2,John Adams,170
         3,Thomas Jefferson,189
```

We'll use the Pandas package, which we'll explore more fully in Part III, to read the file and extract this information (note that the heights are measured in centimeters):

```
In [14]: import pandas as pd
         data = pd.read_csv('data/president_heights.csv')
         heights = np.array(data['height(cm)'])
         print(heights)
Out[14]: [189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
          174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
          177 185 188 188 182 185 191 182]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
In [15]: print("Mean height:       ", heights.mean())
         print("Standard deviation:", heights.std())
         print("Minimum height:    ", heights.min())
         print("Maximum height:    ", heights.max())
Out[15]: Mean height:        180.04545454545453
         Standard deviation: 6.983599441335736
         Minimum height:     163
         Maximum height:     193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
In [16]: print("25th percentile:   ", np.percentile(heights, 25))
         print("Median:            ", np.median(heights))
         print("75th percentile:   ", np.percentile(heights, 75))
```

```
Out[16]: 25th percentile:    174.75
         Median:             182.0
         75th percentile:    183.5
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in Part IV). For example, this code generates Figure 7-1:

```
In [17]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
```

```
In [18]: plt.hist(heights)
         plt.title('Height Distribution of US Presidents')
         plt.xlabel('height (cm)')
         plt.ylabel('number');
```

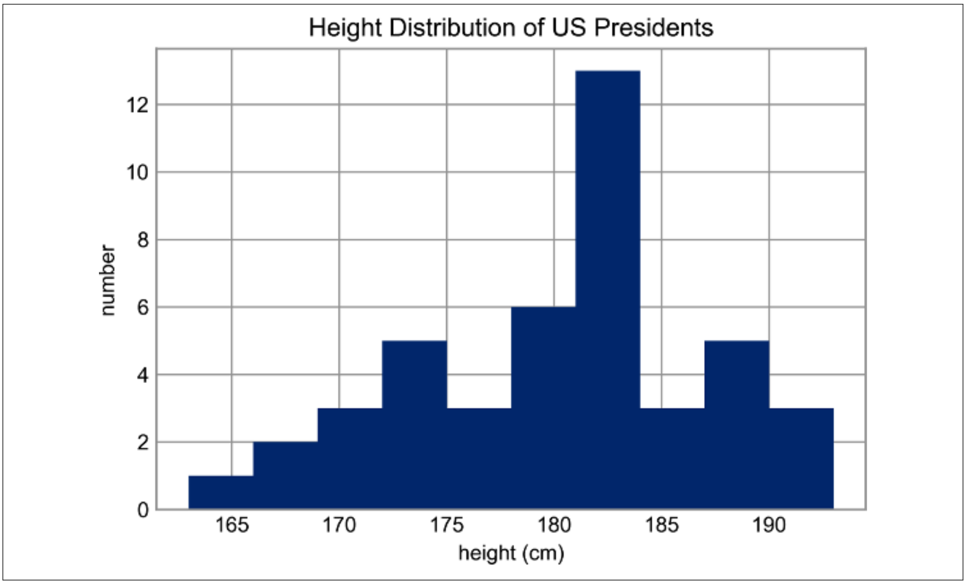

Figure 7-1. Histogram of presidential heights

CHAPTER 8

Computation on Arrays: Broadcasting

We saw in Chapter 6 how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. This chapter discusses *broadcasting*: a set of rules by which NumPy lets you apply binary operations (e.g., addition, subtraction, multiplication, etc.) between arrays of different sizes and shapes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
In [1]: import numpy as np

In [2]: a = np.array([0, 1, 2])
        b = np.array([5, 5, 5])
        a + b
Out[2]: array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes—for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
In [3]: a + 5
Out[3]: array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results.

We can similarly extend this idea to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
In [4]: M = np.ones((3, 3))
        M
Out[4]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])

In [5]: M + a
Out[5]: array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.]])
```

Here the one-dimensional array `a` is stretched, or broadcasted, across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
In [6]: a = np.arange(3)
        b = np.arange(3)[:, np.newaxis]

        print(a)
        print(b)
Out[6]: [0 1 2]
        [[0]
         [1]
         [2]]

In [7]: a + b
Out[7]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in Figure 8-1.

The light boxes represent the broadcasted values. This way of thinking about broadcasting may raise questions about its efficiency in terms of memory use, but worry not: NumPy broadcasting does not actually copy the broadcasted values in memory. Still, this can be a useful mental model as we think about broadcasting.

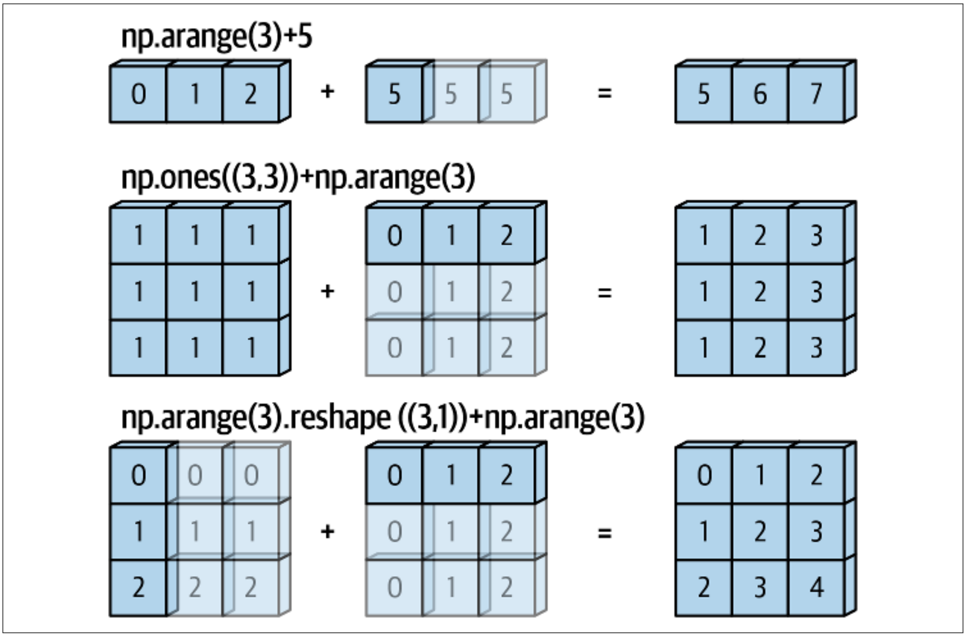

Figure 8-1. Visualization of NumPy broadcasting (adapted from a source published in the astroML documentation and used with permission)[1]

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

Rule 1
: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.

Rule 2
: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3
: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

1 Code to produce this plot can be found in the online appendix.

Broadcasting Example 1

Suppose we want to add a two-dimensional array to a one-dimensional array:

```
In [8]: M = np.ones((2, 3))
        a = np.arange(3)
```

Let's consider an operation on these two arrays, which have the following shapes:

- `M.shape` is `(2, 3)`
- `a.shape` is `(3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(2, 3)`

The shapes now match, and we see that the final shape will be `(2, 3)`:

```
In [9]: M + a
Out[9]: array([[1., 2., 3.],
               [1., 2., 3.]])
```

Broadcasting Example 2

Now let's take a look at an example where both arrays need to be broadcast:

```
In [10]: a = np.arange(3).reshape((3, 1))
         b = np.arange(3)
```

Again, we'll start by determining the shapes of the arrays:

- `a.shape` is `(3, 1)`
- `b.shape` is `(3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape` remains `(3, 1)`
- `b.shape` becomes `(1, 3)`

And rule 2 tells us that we must upgrade each of these 1s to match the corresponding size of the other array:

- a.shape becomes (3, 3)
- b.shape becomes (3, 3)

Because the results match, these shapes are compatible. We can see this here:

```
In [11]: a + b
Out[11]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

Broadcasting Example 3

Next, let's take a look at an example in which the two arrays are not compatible:

```
In [12]: M = np.ones((3, 2))
         a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix M is transposed. How does this affect the calculation? The shapes of the arrays are as follows:

- M.shape is (3, 2)
- a.shape is (3,)

Again, rule 1 tells us that we must pad the shape of a with ones:

- M.shape remains (3, 2)
- a.shape becomes (1, 3)

By rule 2, the first dimension of a is then stretched to match that of M:

- M.shape remains (3, 2)
- a.shape becomes (3, 3)

Now we hit rule 3—the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
In [13]: M + a
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making a and M compatible by, say, padding a's shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword introduced in Chapter 5 for this):

```
In [14]: a[:, np.newaxis].shape
Out[14]: (3, 1)
```

```
In [15]: M + a[:, np.newaxis]
Out[15]: array([[1., 1.],
                [2., 2.],
                [3., 3.]])
```

While we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

```
In [16]: np.logaddexp(M, a[:, np.newaxis])
Out[16]: array([[1.31326169, 1.31326169],
                [1.69314718, 1.69314718],
                [2.31326169, 2.31326169]])
```

For more information on the many available universal functions, refer to Chapter 6.

Broadcasting in Practice

Broadcasting operations form the core of many examples you'll see throughout this book. We'll now take a look at some instances of where they can be useful.

Centering an Array

In Chapter 6, we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly seen example in data science is subtracting the row-wise mean from an array of data. Imagine we have an array of 10 observations, each of which consists of 3 values. Using the standard convention (see Chapter 38), we'll store this in a 10×3 array:

```
In [17]: rng = np.random.default_rng(seed=1701)
         X = rng.random((10, 3))
```

We can compute the mean of each column using the `mean` aggregate across the first dimension:

```
In [18]: Xmean = X.mean(0)
         Xmean
Out[18]: array([0.38503638, 0.36991443, 0.63896043])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
In [19]: X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has a mean near zero:

```
In [20]: X_centered.mean(0)
Out[20]: array([ 4.99600361e-17, -4.44089210e-17,  0.00000000e+00])
```

To within machine precision, the mean is now zero.

Plotting a Two-Dimensional Function

One place that broadcasting often comes in handy is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
In [21]: # x and y have 50 steps from 0 to 5
         x = np.linspace(0, 5, 50)
         y = np.linspace(0, 5, 50)[:, np.newaxis]

         z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

We'll use Matplotlib to plot this two-dimensional array, shown in Figure 8-2 (these tools will be discussed in full in Chapter 28):

```
In [22]: %matplotlib inline
         import matplotlib.pyplot as plt
```

```
In [23]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5])
         plt.colorbar();
```

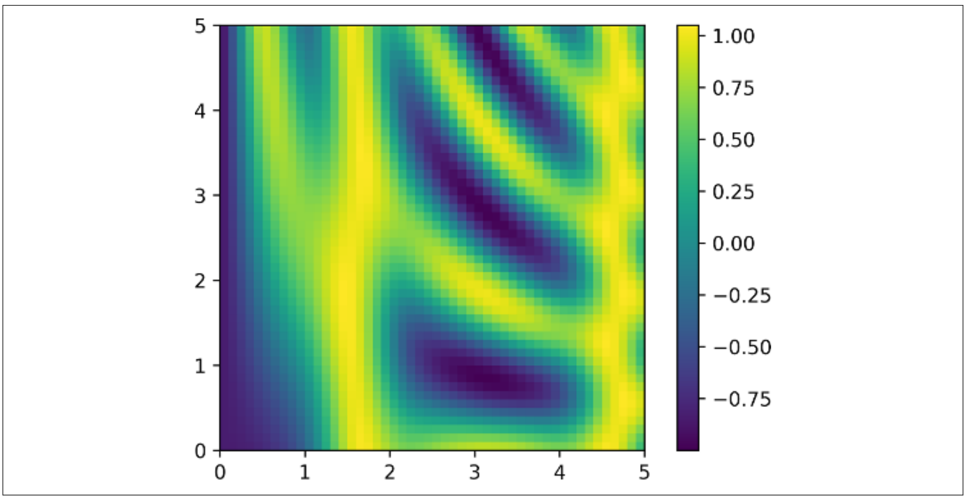

Figure 8-2. Visualization of a 2D array

The result is a compelling visualization of the two-dimensional function.

CHAPTER 9

Comparisons, Masks, and Boolean Logic

This chapter covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2015, using Pandas (see Part III):

```
In [1]: import numpy as np
        from vega_datasets import data

        # Use DataFrame operations to extract rainfall as a NumPy array
        rainfall_mm = np.array(
            data.seattle_weather().set_index('date')['precipitation']['2015'])
        len(rainfall_mm)
Out[1]: 365
```

The array contains 365 values, giving daily rainfall in millimeters from January 1 to December 31, 2015.

As a first quick visualization, let's look at the histogram of rainy days in Figure 9-1, which was generated using Matplotlib (we will explore this tool more fully in Part IV):

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
```

```
In [3]: plt.hist(rainfall_mm, 40);
```

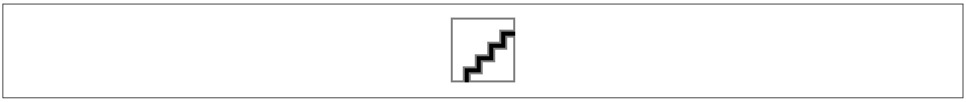

Figure 9-1. Histogram of 2015 rainfall in Seattle

This histogram gives us a general idea of what the data looks like: despite the city's rainy reputation, the vast majority of days in Seattle saw near zero measured rainfall in 2015. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What was the average precipitation on those rainy days? How many days were there with more than 10 mm of rainfall?

One approach to this would be to answer these questions by hand: we could loop through the data, incrementing a counter each time we see values in some desired range. But for reasons discussed throughout this chapter, such an approach is very inefficient from the standpoint of both time writing code and time computing the result. We saw in Chapter 6 that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as Ufuncs

Chapter 6 introduced ufuncs, and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and other operators on arrays leads to element-wise operations. NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

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

In [5]: x < 3  # less than
Out[5]: array([ True,  True, False, False, False])

In [6]: x > 3  # greater than
Out[6]: array([False, False, False,  True,  True])

In [7]: x <= 3  # less than or equal
Out[7]: array([ True,  True,  True, False, False])

In [8]: x >= 3  # greater than or equal
Out[8]: array([False, False,  True,  True,  True])
```

```
In [9]: x != 3  # not equal
Out[9]: array([ True,  True, False,  True,  True])
```

```
In [10]: x == 3  # equal
Out[10]: array([False, False,  True, False, False])
```

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

```
In [11]: (2 * x) == (x ** 2)
Out[11]: array([False,  True, False, False, False])
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufuncs is shown here:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`==`	`np.equal`	`!=`	`np.not_equal`
`<`	`np.less`	`<=`	`np.less_equal`
`>`	`np.greater`	`>=`	`np.greater_equal`

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
In [12]: rng = np.random.default_rng(seed=1701)
         x = rng.integers(10, size=(3, 4))
         x
Out[12]: array([[9, 4, 0, 3],
                [8, 6, 3, 1],
                [3, 7, 4, 0]])
```

```
In [13]: x < 6
Out[13]: array([[False,  True,  True,  True],
                [False, False,  True,  True],
                [ True, False,  True,  True]])
```

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created earlier:

```
In [14]: print(x)
Out[14]: [[9 4 0 3]
          [8 6 3 1]
          [3 7 4 0]]
```

Counting Entries

To count the number of True entries in a Boolean array, np.count_nonzero is useful:

```
In [15]: # how many values less than 6?
         np.count_nonzero(x < 6)
Out[15]: 8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use np.sum; in this case, False is interpreted as 0, and True is interpreted as 1:

```
In [16]: np.sum(x < 6)
Out[16]: 8
```

The benefit of np.sum is that, like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
In [17]: # how many values less than 6 in each row?
         np.sum(x < 6, axis=1)
Out[17]: array([3, 2, 3])
```

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are True, we can use (you guessed it) np.any or np.all:

```
In [18]: # are there any values greater than 8?
         np.any(x > 8)
Out[18]: True

In [19]: # are there any values less than zero?
         np.any(x < 0)
Out[19]: False

In [20]: # are all values less than 10?
         np.all(x < 10)
Out[20]: True

In [21]: # are all values equal to 6?
         np.all(x == 6)
Out[21]: False
```

np.all and np.any can be used along particular axes as well. For example:

```
In [22]: # are all values in each row less than 8?
         np.all(x < 8, axis=1)
Out[22]: array([False, False,  True])
```

Here all the elements in the third row are less than 8, while this is not the case for others.

Finally, a quick warning: as mentioned in Chapter 7, Python has built-in sum, any, and all functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using `np.sum`, `np.any`, and `np.all` for these examples!

Boolean Operators

We've already seen how we might count, say, all days with less than 20 mm of rain, or all days with more than 10 mm of rain. But what if we want to know how many days there were with more than 10 mm and less than 20 mm of rain? We can accomplish this with Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs that work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

```
In [23]: np.sum((rainfall_mm > 10) & (rainfall_mm < 20))
Out[23]: 16
```

This tells us that there were 16 days with rainfall of between 10 and 20 millimeters.

The parentheses here are important. Because of operator precedence rules, with the parentheses removed this expression would be evaluated as follows, which results in an error:

```
rainfall_mm > (10 & rainfall_mm) < 20
```

Let's demonstrate a more complicated expression. Using De Morgan's laws, we can compute the same result in a different manner:

```
In [24]: np.sum(~( (rainfall_mm <= 10) | (rainfall_mm >= 20) ))
Out[24]: 16
```

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`&`	`np.bitwise_and`		`np.bitwise_or`
`^`	`np.bitwise_xor`	`~`	`np.bitwise_not`

Using these tools, we can start to answer many of the questions we might have about our weather data. Here are some examples of results we can compute when combining Boolean operations with aggregations:

```
In [25]: print("Number days without rain:  ", np.sum(rainfall_mm == 0))
         print("Number days with rain:     ", np.sum(rainfall_mm != 0))
         print("Days with more than 10 mm: ", np.sum(rainfall_mm > 10))
         print("Rainy days with < 5 mm:    ", np.sum((rainfall_mm > 0) &
```

```
                                      (rainfall_mm < 5)))
Out[25]: Number days without rain:   221
         Number days with rain:      144
         Days with more than 10 mm:  34
         Rainy days with < 5 mm:     83
```

Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Let's return to our `x` array from before:

```
In [26]: x
Out[26]: array([[9, 4, 0, 3],
                [8, 6, 3, 1],
                [3, 7, 4, 0]])
```

Suppose we want an array of all values in the array that are less than, say, 5. We can obtain a Boolean array for this condition easily, as we've already seen:

```
In [27]: x < 5
Out[27]: array([[False,  True,  True,  True],
                [False, False,  True,  True],
                [ True, False,  True,  True]])
```

Now, to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
In [28]: x[x < 5]
Out[28]: array([4, 0, 3, 3, 1, 3, 4, 0])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on our Seattle rain data:

```
In [29]: # construct a mask of all rainy days
         rainy = (rainfall_mm > 0)

         # construct a mask of all summer days (June 21st is the 172nd day)
         days = np.arange(365)
         summer = (days > 172) & (days < 262)

         print("Median precip on rainy days in 2015 (mm):   ",
               np.median(rainfall_mm[rainy]))
         print("Median precip on summer days in 2015 (mm):  ",
               np.median(rainfall_mm[summer]))
         print("Maximum precip on summer days in 2015 (mm): ",
               np.max(rainfall_mm[summer]))
         print("Median precip on non-summer rainy days (mm):",
               np.median(rainfall_mm[rainy & ~summer]))
```

```
Out[29]: Median precip on rainy days in 2015 (mm):    3.8
         Median precip on summer days in 2015 (mm):   0.0
         Maximum precip on summer days in 2015 (mm):  32.5
         Median precip on non-summer rainy days (mm): 4.1
```

By combining Boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions about our dataset.

Using the Keywords and/or Versus the Operators &/|

One common point of confusion is the difference between the keywords `and` and `or` on the one hand, and the operators `&` and `|` on the other. When would you use one versus the other?

The difference is this: `and` and `or` operate on the object as a whole, while `&` and `|` operate on the elements within the object.

When you use `and` or `or`, it is equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as `True`. Thus:

```
In [30]: bool(42), bool(0)
Out[30]: (True, False)

In [31]: bool(42 and 0)
Out[31]: False

In [32]: bool(42 or 0)
Out[32]: True
```

When you use `&` and `|` on integers, the expression operates on the bitwise representation of the element, applying the *and* or the *or* to the individual bits making up the number:

```
In [33]: bin(42)
Out[33]: '0b101010'

In [34]: bin(59)
Out[34]: '0b111011'

In [35]: bin(42 & 59)
Out[35]: '0b101010'

In [36]: bin(42 | 59)
Out[36]: '0b111011'
```

Notice that the corresponding bits of the binary representation are compared in order to yield the result.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and `&` and `|` will operate similarly to in the preceding examples:

```
In [37]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
         B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
         A | B
Out[37]: array([ True,  True,  True, False,  True,  True])
```

But if you use or on these arrays it will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
In [38]: A or B
ValueError: The truth value of an array with more than one element is
          > ambiguous.
          a.any() or a.all()
```

Similarly, when evaluating a Boolean expression on a given array, you should use | or & rather than or or and:

```
In [39]: x = np.arange(10)
         (x > 4) & (x < 8)
Out[39]: array([False, False, False, False, False,  True,  True,  True, False,
                False])
```

Trying to evaluate the truth or falsehood of the entire array will give the same ValueError we saw previously:

```
In [40]: (x > 4) and (x < 8)
ValueError: The truth value of an array with more than one element is
          > ambiguous.
          a.any() or a.all()
```

So, remember this: and and or perform a single Boolean evaluation on an entire object, while & and | perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. For Boolean NumPy arrays, the latter is nearly always the desired operation.

CHAPTER 10
Fancy Indexing

The previous chapters discussed how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). In this chapter, we'll look at another style of array indexing, known as *fancy* or *vectorized* indexing, in which we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)

        x = rng.integers(100, size=10)
        print(x)
Out[1]: [90 40  9 30 80 67 39 15 33 79]
```

Suppose we want to access three different elements. We could do it like this:

```
In [2]: [x[3], x[7], x[2]]
Out[2]: [30, 15, 9]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
In [3]: ind = [3, 7, 4]
        x[ind]
Out[3]: array([30, 15, 80])
```

When using arrays of indices, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
In [4]: ind = np.array([[3, 7],
                        [4, 5]])
        x[ind]
Out[4]: array([[30, 15],
               [80, 67]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
In [5]: X = np.arange(12).reshape((3, 4))
        X
Out[5]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
In [6]: row = np.array([0, 1, 2])
        col = np.array([2, 1, 3])
        X[row, col]
Out[6]: array([ 2,  5, 11])
```

Notice that the first value in the result is `X[0, 2]`, the second is `X[1, 1]`, and the third is `X[2, 3]`. The pairing of indices in fancy indexing follows all the broadcasting rules that were mentioned in Chapter 8. So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
In [7]: X[row[:, np.newaxis], col]
Out[7]: array([[ 2,  1,  3],
               [ 6,  5,  7],
               [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
In [8]: row[:, np.newaxis] * col
Out[8]: array([[0, 0, 0],
               [2, 1, 3],
               [4, 2, 6]])
```

It is always important to remember with fancy indexing that the return value reflects the *broadcasted shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen. For example, given the array `X`:

```
In [9]: print(X)
Out[9]: [[ 0  1  2  3]
         [ 4  5  6  7]
         [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
In [10]: X[2, [2, 0, 1]]
Out[10]: array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
In [11]: X[1:, [2, 0, 1]]
Out[11]: array([[ 6,  4,  5],
                [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
In [12]: mask = np.array([True, False, True, False])
         X[row[:, np.newaxis], mask]
Out[12]: array([[ 0,  2],
                [ 4,  6],
                [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for efficiently accessing and modifying array values.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an $N \times D$ matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

```
In [13]: mean = [0, 0]
         cov = [[1, 2],
                [2, 5]]
         X = rng.multivariate_normal(mean, cov, 100)
         X.shape
Out[13]: (100, 2)
```

Using the plotting tools we will discuss in Part IV, we can visualize these points as a scatter plot (Figure 10-1).

```
In [14]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')

         plt.scatter(X[:, 0], X[:, 1]);
```

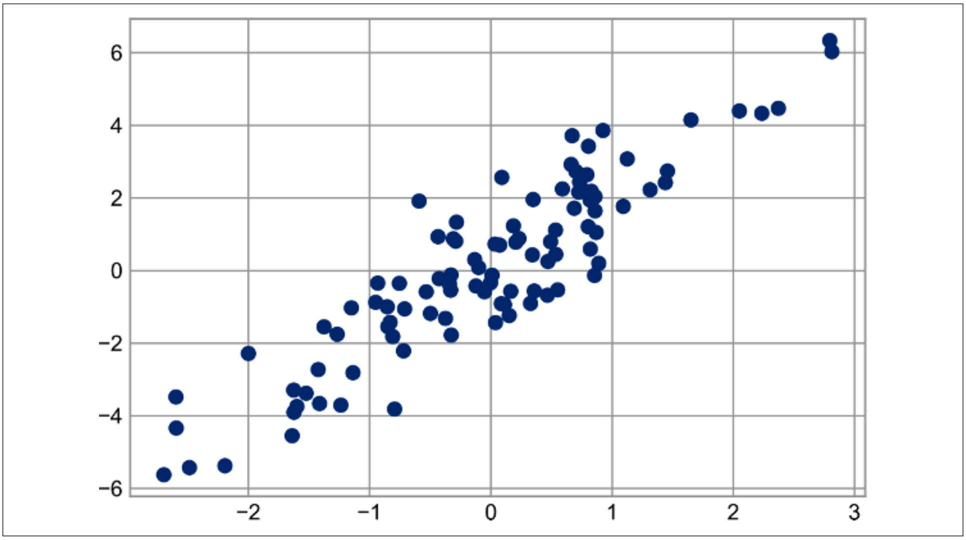

Figure 10-1. Normally distributed points

Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and using these indices to select a portion of the original array:

```
In [15]: indices = np.random.choice(X.shape[0], 20, replace=False)
         indices
Out[15]: array([82, 84, 10, 55, 14, 33,  4, 16, 34, 92, 99, 64,  8, 76, 68, 18, 59,
                80, 87, 90])
```

```
In [16]: selection = X[indices]  # fancy indexing here
         selection.shape
Out[16]: (20, 2)
```

Now to see which points were selected, let's overplot large circles at the locations of the selected points (see Figure 10-2).

```
In [17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
         plt.scatter(selection[:, 0], selection[:, 1],
                     facecolor='none', edgecolor='black', s=200);
```

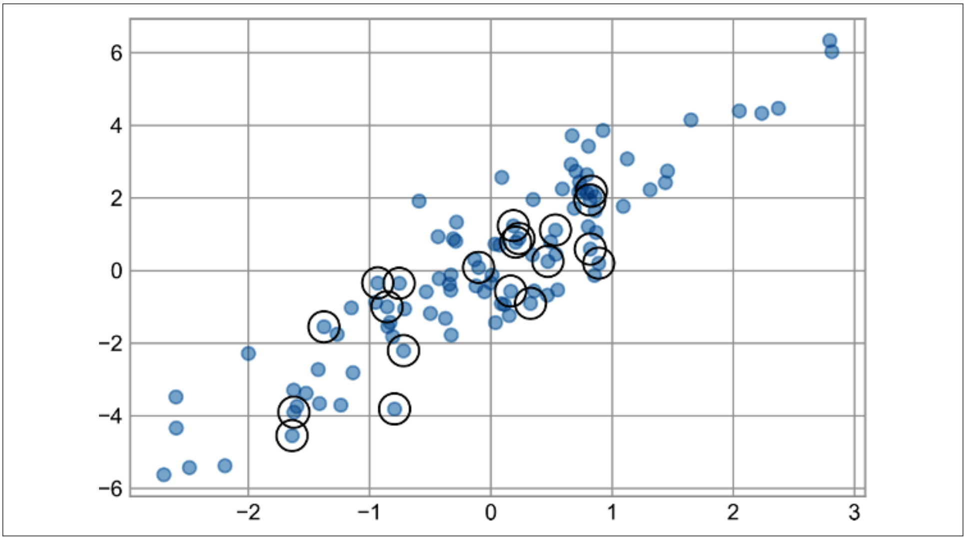

Figure 10-2. Random selection among points

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see Chapter 39), and in sampling approaches to answering statistical questions.

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value:

```
In [18]: x = np.arange(10)
         i = np.array([2, 1, 8, 4])
         x[i] = 99
         print(x)
Out[18]: [ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
In [19]: x[i] -= 10
         print(x)
Out[19]: [ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
In [20]: x = np.zeros(10)
         x[[0, 0]] = [4, 6]
         print(x)
Out[20]: [6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Where did the 4 go? This operation first assigns `x[0] = 4`, followed by `x[0] = 6`. The result, of course, is that `x[0]` contains the value 6.

Fair enough, but consider this operation:

```
In [21]: i = [2, 3, 3, 4, 4, 4]
         x[i] += 1
         x
Out[21]: array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])
```

You might expect that `x[3]` would contain the value 2 and `x[4]` would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because `x[i] += 1` is meant as a shorthand of `x[i] = x[i] + 1`. `x[i] + 1` is evaluated, and then the result is assigned to the indices in `x`. With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at` method of ufuncs and do the following:

```
In [22]: x = np.zeros(10)
         np.add.at(x, i, 1)
         print(x)
Out[22]: [0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
```

The `at` method does an in-place application of the given operator at the specified indices (here, `i`) with the specified value (here, 1). Another method that is similar in spirit is the `reduceat` method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning Data

You could use these ideas to efficiently do custom binned computations on data. For example, imagine we have 100 values and would like to quickly find where they fall within an array of bins. We could compute this using `ufunc.at` like this:

```
In [23]: rng = np.random.default_rng(seed=1701)
         x = rng.normal(size=100)

         # compute a histogram by hand
         bins = np.linspace(-5, 5, 20)
         counts = np.zeros_like(bins)

         # find the appropriate bin for each x
         i = np.searchsorted(bins, x)

         # add 1 to each of these bins
         np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin—in other words, a histogram (see Figure 10-3).

```
In [24]: # plot the results
         plt.plot(bins, counts, drawstyle='steps');
```

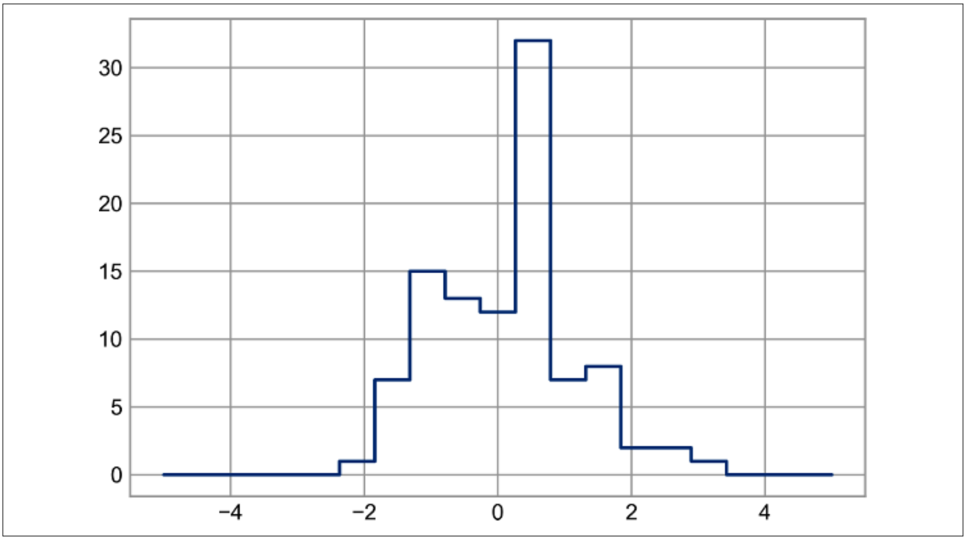

Figure 10-3. A histogram computed by hand

Of course, it would be inconvenient to have to do this each time you want to plot a histogram. This is why Matplotlib provides the `plt.hist` routine, which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

This function will create a nearly identical plot to the one just shown. To compute the binning, Matplotlib uses the `np.histogram` function, which does a very similar computation to what we did before. Let's compare the two here:

```
In [25]: print(f"NumPy histogram ({len(x)} points):")
         %timeit counts, edges = np.histogram(x, bins)

         print(f"Custom histogram ({len(x)} points):")
         %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
Out[25]: NumPy histogram (100 points):
         33.8 µs ± 311 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
         Custom histogram (100 points):
         17.6 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Our own one-line algorithm is twice as fast as the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing `np.histogram??`), you'll see that it's quite a bit more involved than the simple search-and-count that we've done; this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
In [26]: x = rng.normal(size=1000000)
         print(f"NumPy histogram ({len(x)} points):")
         %timeit counts, edges = np.histogram(x, bins)

         print(f"Custom histogram ({len(x)} points):")
         %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
Out[26]: NumPy histogram (1000000 points):
         84.4 ms ± 2.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
         Custom histogram (1000000 points):
         128 ms ± 2.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see Chapter 11). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, the sky is the limit: you're no longer constrained to built-in routines, but can create your own approaches to exploring the data. Key to efficiently using Python in data-intensive applications is not only knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to make use of lower-level functionality when you need more pointed behavior.

CHAPTER 11

Sorting Arrays

Up to this point we have been concerned mainly with tools to access and operate on array data with NumPy. This chapter covers algorithms related to sorting values in NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

Python has a couple of built-in functions and methods for sorting lists and other iterable objects. The `sorted` function accepts a list and returns a sorted version of it:

```
In [1]: L = [3, 1, 4, 1, 5, 9, 2, 6]
        sorted(L)  # returns a sorted copy
Out[1]: [1, 1, 2, 3, 4, 5, 6, 9]
```

By contrast, the `sort` method of lists will sort the list in-place:

```
In [2]: L.sort()  # acts in-place and returns None
        print(L)
Out[2]: [1, 1, 2, 3, 4, 5, 6, 9]
```

Python's sorting methods are quite flexible, and can handle any iterable object. For example, here we sort a string:

```
In [3]: sorted('python')
Out[3]: ['h', 'n', 'o', 'p', 't', 'y']
```

These built-in sorting methods are convenient, but as previously discussed, the dynamism of Python values means that they are less performant than routines designed specifically for uniform arrays of numbers. This is where NumPy's sorting routines come in.

Fast Sorting in NumPy: np.sort and np.argsort

The `np.sort` function is analogous to Python's built-in `sorted` function, and will efficiently return a sorted copy of an array:

```
In [4]: import numpy as np

        x = np.array([2, 1, 4, 3, 5])
        np.sort(x)
Out[4]: array([1, 2, 3, 4, 5])
```

Similarly to the `sort` method of Python lists, you can also sort an array in-place using the array `sort` method:

```
In [5]: x.sort()
        print(x)
Out[5]: [1 2 3 4 5]
```

A related function is `argsort`, which instead returns the *indices* of the sorted elements:

```
In [6]: x = np.array([2, 1, 4, 3, 5])
        i = np.argsort(x)
        print(i)
Out[6]: [1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
In [7]: x[i]
Out[7]: array([1, 2, 3, 4, 5])
```

You'll see an application of `argsort` later in this chapter.

Sorting Along Rows or Columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the `axis` argument. For example:

```
In [8]: rng = np.random.default_rng(seed=42)
        X = rng.integers(0, 10, (4, 6))
        print(X)
Out[8]: [[0 7 6 4 4 8]
         [0 6 2 0 5 9]
         [7 7 7 7 5 1]
         [8 4 5 3 1 9]]

In [9]: # sort each column of X
        np.sort(X, axis=0)
Out[9]: array([[0, 4, 2, 0, 1, 1],
               [0, 6, 5, 3, 4, 8],
```

```
                [7, 7, 6, 4, 5, 9],
                [8, 7, 7, 7, 5, 9]])
```

```
In [10]: # sort each row of X
         np.sort(X, axis=1)
Out[10]: array([[0, 4, 4, 6, 7, 8],
                [0, 0, 2, 5, 6, 9],
                [1, 5, 7, 7, 7, 7],
                [1, 3, 4, 5, 8, 9]])
```

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the *k* smallest values in the array. NumPy enables this with the `np.partition` function. `np.partition` takes an array and a number *k*; the result is a new array with the smallest *k* values to the left of the partition and the remaining values to the right:

```
In [11]: x = np.array([7, 2, 3, 1, 6, 5, 4])
         np.partition(x, 3)
Out[11]: array([2, 1, 3, 4, 6, 5, 7])
```

Notice that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
In [12]: np.partition(X, 2, axis=1)
Out[12]: array([[0, 4, 4, 7, 6, 8],
                [0, 0, 2, 6, 5, 9],
                [1, 5, 7, 7, 7, 7],
                [1, 3, 4, 5, 8, 9]])
```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is an `np.argsort` function that computes indices of the sort, there is an `np.argpartition` function that computes indices of the partition. We'll see both of these in action in the following section.

Example: k-Nearest Neighbors

Let's quickly see how we might use the `argsort` function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of 10 points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

```
In [13]: X = rng.random((10, 2))
```

To get an idea of how these points look, let's generate a quick scatter plot (see Figure 11-1).

```
In [14]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use('seaborn-whitegrid')
         plt.scatter(X[:, 0], X[:, 1], s=100);
```

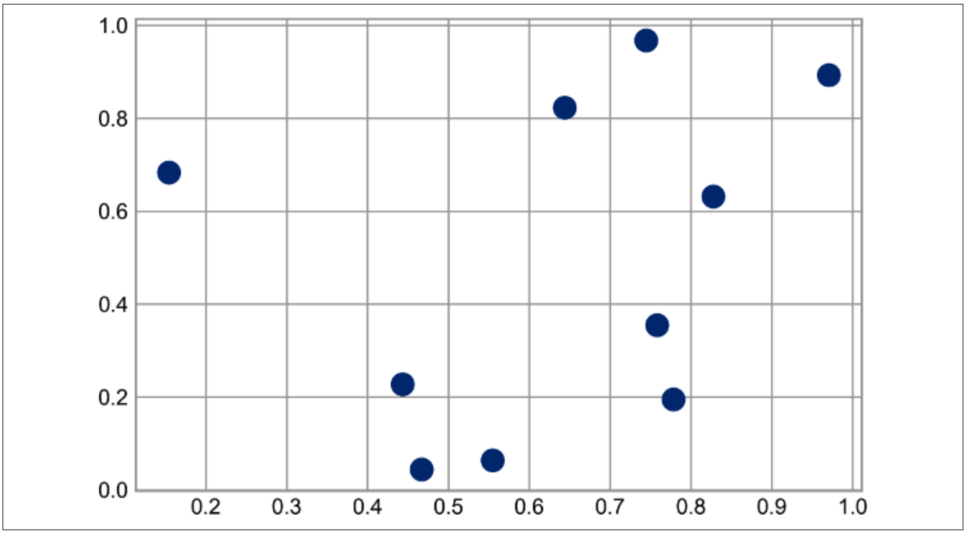

Figure 11-1. Visualization of points in the k*-neighbors example*

Now we'll compute the distance between each pair of points. Recall that the squared distance between two points is the sum of the squared differences in each dimension; using the efficient broadcasting (Chapter 8) and aggregation (Chapter 7) routines provided by NumPy we can compute the matrix of square distances in a single line of code:

```
In [15]: dist_sq = np.sum((X[:, np.newaxis] - X[np.newaxis, :]) ** 2, axis=-1)
```

This operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to break it down into its component steps:

```
In [16]: # for each pair of points, compute differences in their coordinates
         differences = X[:, np.newaxis] - X[np.newaxis, :]
         differences.shape
Out[16]: (10, 10, 2)
```

```
In [17]: # square the coordinate differences
         sq_differences = differences ** 2
         sq_differences.shape
Out[17]: (10, 10, 2)
```

```
In [18]: # sum the coordinate differences to get the squared distance
         dist_sq = sq_differences.sum(-1)
         dist_sq.shape
Out[18]: (10, 10)
```

As a quick check of our logic, we should see that the diagonal of this matrix (i.e., the set of distances between each point and itself) is all zeros:

```
In [19]: dist_sq.diagonal()
Out[19]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

With the pairwise square distances converted, we can now use `np.argsort` to sort along each row. The leftmost columns will then give the indices of the nearest neighbors:

```
In [20]: nearest = np.argsort(dist_sq, axis=1)
         print(nearest)
Out[20]: [[0 9 3 5 4 8 1 6 2 7]
          [1 7 2 6 4 8 3 0 9 5]
          [2 7 1 6 4 3 8 0 9 5]
          [3 0 4 5 9 6 1 2 8 7]
          [4 6 3 1 2 7 0 5 9 8]
          [5 9 3 0 4 6 8 1 2 7]
          [6 4 2 1 7 3 0 5 9 8]
          [7 2 1 6 4 3 8 0 9 5]
          [8 0 1 9 3 4 7 2 6 5]
          [9 0 5 3 4 8 6 1 2 7]]
```

Notice that the first column gives the numbers 0 through 9 in order: this is due to the fact that each point's closest neighbor is itself, as we would expect.

By using a full sort here, we've actually done more work than we need to in this case. If we're simply interested in the nearest k neighbors, all we need to do is partition each row so that the smallest $k + 1$ squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```
In [21]: K = 2
         nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along with lines representing the connections from each point to its two nearest neighbors (see Figure 11-2).

```
In [22]: plt.scatter(X[:, 0], X[:, 1], s=100)

         # draw lines from each point to its two nearest neighbors
         K = 2
```

```python
for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```

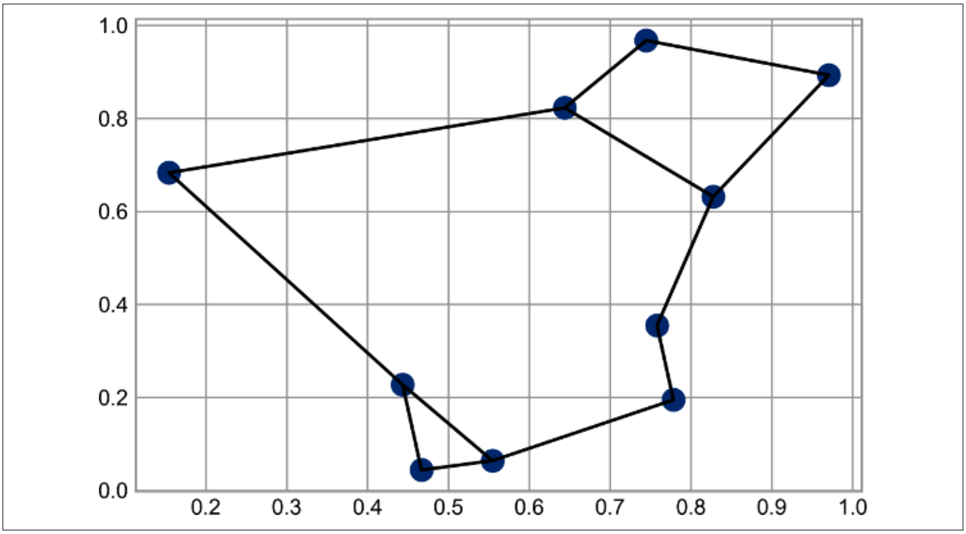

Figure 11-2. Visualization of the nearest neighbors of each point

Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Although the broadcasting and row-wise sorting of this approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used. The beauty of this approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, I'll note that when doing very large nearest neighbor searches, there are tree-based and/or approximate algorithms that can scale as $\mathcal{O}[N \log N]$ or better rather than the $\mathcal{O}\left[N^2\right]$ of the brute-force algorithm. One example of this is the KD-Tree, implemented in Scikit-Learn.

CHAPTER 12

Structured Data: NumPy's Structured Arrays

While often our data can be well represented by a homogeneous array of values, sometimes this is not the case. This chapter demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas `DataFrame`s, which we'll explore in Part III.

```
In [1]: import numpy as np
```

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
In [2]: name = ['Alice', 'Bob', 'Cathy', 'Doug']
        age = [25, 45, 37, 19]
        weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; NumPy's structured arrays allow us to do this more naturally by using a single structure to store all of this data.

Recall that previously we created a simple array using an expression like this:

```
In [3]: x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
In [4]: # Use a compound data type for structured arrays
        data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                  'formats':('U10', 'i4', 'f8')})
```

```
        print(data.dtype)
Out[4]: [('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here `'U10'` translates to "Unicode string of maximum length 10," `'i4'` translates to "4-byte (i.e., 32-bit) integer," and `'f8'` translates to "8-byte (i.e., 64-bit) float." We'll discuss other options for these type codes in the following section.

Now that we've created an empty container array, we can fill the array with our lists of values:

```
In [5]: data['name'] = name
        data['age'] = age
        data['weight'] = weight
        print(data)
Out[5]: [('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
         ('Doug', 19, 61.5)]
```

As we had hoped, the data is now conveniently arranged in one structured array.

The handy thing with structured arrays is that we can now refer to values either by index or by name:

```
In [6]: # Get all names
        data['name']
Out[6]: array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

```
In [7]: # Get first row of data
        data[0]
Out[7]: ('Alice', 25, 55.)
```

```
In [8]: # Get the name from the last row
        data[-1]['name']
Out[8]: 'Doug'
```

Using Boolean masking, we can even do some more sophisticated operations, such as filtering on age:

```
In [9]: # Get names where age is under 30
        data[data['age'] < 30]['name']
Out[9]: array(['Alice', 'Doug'], dtype='<U10')
```

If you'd like to do any operations that are any more complicated than these, you should probably consider the Pandas package, covered in Part IV. As you'll see, Pandas provides a `DataFrame` object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what you've seen here, as well as much, much more.

Exploring Structured Array Creation

Structured array data types can be specified in a number of ways. Earlier, we saw the dictionary method:

```
In [10]: np.dtype({'names':('name', 'age', 'weight'),
                   'formats':('U10', 'i4', 'f8')})
Out[10]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified using Python types or NumPy `dtype`s instead:

```
In [11]: np.dtype({'names':('name', 'age', 'weight'),
                   'formats':((np.str_, 10), int, np.float32)})
Out[11]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

```
In [12]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
Out[12]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
In [13]: np.dtype('S10,i4,f8')
Out[13]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may not be immediately intuitive, but they are built on simple principles. The first (optional) character < or >, means "little endian" or "big endian," respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see Table 12-1). The last character or characters represent the size of the object in bytes.

Table 12-1. NumPy data types

Character	Description	Example
`'b'`	Byte	`np.dtype('b')`
`'i'`	Signed integer	`np.dtype('i4') == np.int32`
`'u'`	Unsigned integer	`np.dtype('u1') == np.uint8`
`'f'`	Floating point	`np.dtype('f8') == np.int64`
`'c'`	Complex floating point	`np.dtype('c16') == np.complex128`
`'S'`, `'a'`	String	`np.dtype('S5')`
`'U'`	Unicode string	`np.dtype('U') == np.str_`
`'V'`	Raw data (void)	`np.dtype('V') == np.void`

More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3 × 3 floating-point matrix:

```
In [14]: tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
         X = np.zeros(1, dtype=tp)
         print(X[0])
         print(X['mat'][0])
Out[14]: (0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
         [[0. 0. 0.]
          [0. 0. 0.]
          [0. 0. 0.]]
```

Now each element in the `X` array consists of an `id` and a 3 × 3 matrix. Why would you use this rather than a simple multidimensional array, or perhaps a Python dictionary? One reason is that this NumPy `dtype` directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, structured arrays can provide a powerful interface.

Record Arrays: Structured Arrays with a Twist

NumPy also provides record arrays (instances of the `np.recarray` class), which are almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages in our sample dataset by writing:

```
In [15]: data['age']
Out[15]: array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
In [16]: data_rec = data.view(np.recarray)
         data_rec.age
Out[16]: array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax:

```
In [17]: %timeit data['age']
         %timeit data_rec['age']
         %timeit data_rec.age
Out[17]: 121 ns ± 1.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
         2.41 μs ± 15.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
         3.98 μs ± 20.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Whether the more convenient notation is worth the (slight) overhead will depend on your own application.

On to Pandas

This chapter on structured and record arrays is purposely located at the end of this part of the book, because it leads so well into the next package we will cover: Pandas. Structured arrays can come in handy in certain situations, like when you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. But for day-to-day use of structured data, the Pandas package is a much better choice; we'll explore it in depth in the chapters that follow.