02 - Introduction to NumPy

May 19, 2018

1 Introduction to NumPy

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

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

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.

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

If you followed the advice outlined in the Preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to http://www.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]: '1.14.0'
```

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 srikanth # you can import with your name
```

```
In [3]: 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.

1.1 Reminder about Built In Documentation

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

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

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

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

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

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

2 Understanding Data Types in Python

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

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

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

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

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

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

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

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

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

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

2.1 A Python Integer Is More Than Just an Integer

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

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

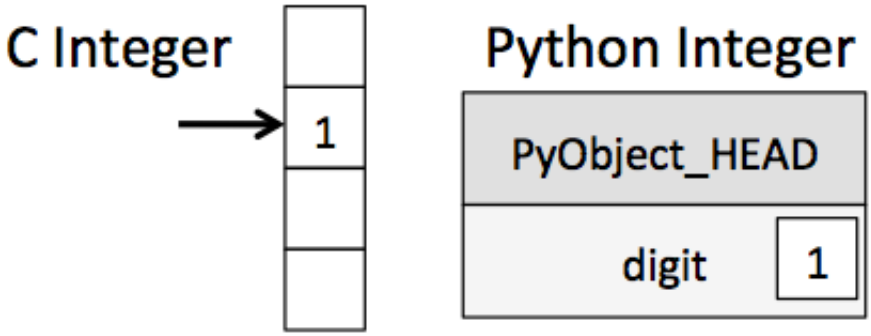
A single integer in Python 3.4 actually contains four pieces:

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

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in the following figure:

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.



Integer Memory Layout

2.2 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 multi-element container in Python is the list. We can create a list of integers as follows:

```
In [4]: L = list(range(10))
        L

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

In [5]: type(L[0])

Out[5]: int
```

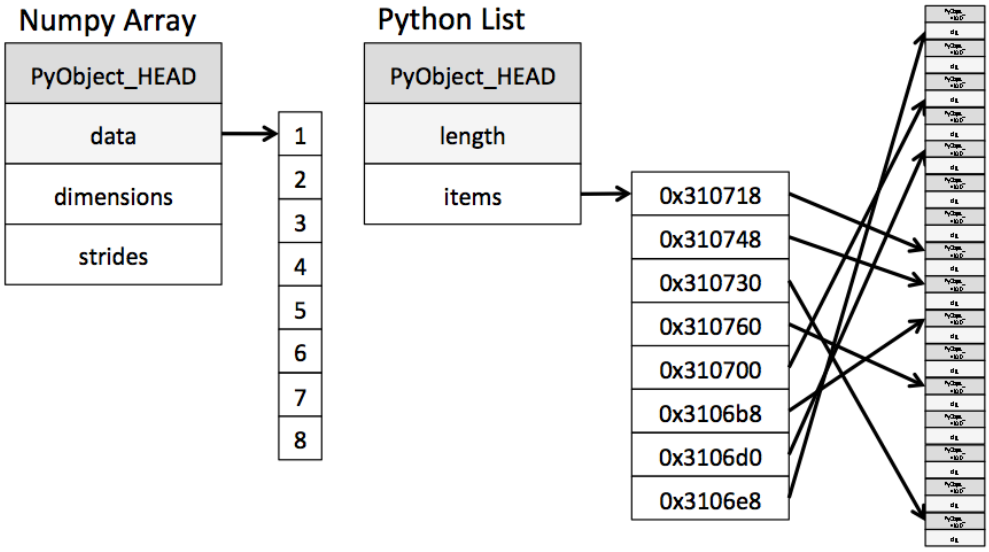
Or, similarly, a list of strings:

```
In [6]: L2 = [str(c) for c in L]
        L2

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

In [7]: type(L2[0])

Out[7]: str
```

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

```
In [8]: L3 = [True, "2", 3.0, 4]
        [type(item) for item in L3]

Out[8]: [bool, str, float, int]
```

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



Array Memory Layout

between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:

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.

2.3 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 [9]: import array
        L = list(range(10))
        A = array.array('i', L)
        A
```

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

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

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

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

```
In [10]: import numpy as np
```

2.4 Creating Arrays from Python Lists

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

```
In [11]: # integer array:
         np.array([1, 4, 2, 5, 3])

Out[11]: array([1, 4, 2, 5, 3])
```

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

```
In [12]: np.array([3.14, 4, 2, 3])

Out[12]: 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 [13]: np.array([1, 2, 3, 4], dtype='float32')

Out[13]: array([1., 2., 3., 4.], dtype=float32)
```

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

```
In [14]: # nested lists result in multi-dimensional arrays
         np.array([range(i, i + 3) for i in [2, 4, 6]])

Out[14]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

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

2.5 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 [15]: # Create a length-10 integer array filled with zeros
         np.zeros(10, dtype=int)

Out[15]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [16]: # Create a 3x5 floating-point array filled with ones
         np.ones((3, 5), dtype=float)

Out[16]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])
```

```
In [17]: # Create a 3x5 array filled with 3.14
         np.full((3, 5), 3.14)

Out[17]: 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 [18]: # 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[18]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [19]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)

Out[19]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])

In [20]: # Create a 3x3 array of uniformly distributed
         # random values between 0 and 1
         np.random.random((3, 3))

Out[20]: array([[0.69988982, 0.90751641, 0.78648773],
                [0.54750317, 0.83751288, 0.50806342],
                [0.57461256, 0.1559891 , 0.92646137]])

In [21]: # Create a 3x3 array of normally distributed random values
         # with mean 0 and standard deviation 1
         np.random.normal(0, 1, (3, 3))

Out[21]: array([[ 0.10714654, -0.16595739, -0.82601771],
                [-0.26254224,  0.15817754,  0.63660822],
                [-1.41356122, -1.66753375, -0.30863589]])

In [22]: # Create a 3x3 array of random integers in the interval [0, 10)
         np.random.randint(0, 10, (3, 3))

Out[22]: array([[8, 0, 5],
                [0, 2, 0],
                [1, 2, 5]])

In [23]: # Create a 3x3 identity matrix
         np.eye(3)

Out[23]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])

In [24]: # Create an uninitialized array of three integers
         # The values will be whatever happens to already exist at that memory location
         np.empty(3)

Out[24]: array([1., 1., 1.])
```

2.6 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 the following table. 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)
```

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)

Data type	Description
`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

3 The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 3) are built around the NumPy array. This section will present several examples 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 value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

3.1 NumPy Array Attributes

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

```
In [25]: import numpy as np
         np.random.seed(0)  # seed for reproducibility

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

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

```
In [26]: print("x3 ndim: ", x3.ndim)
         print("x3 shape:", x3.shape)
         print("x3 size: ", x3.size)
```

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

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

```
In [27]: print("dtype:", x3.dtype)
```

```
dtype: int32
```

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

```
In [28]: print("itemsize:", x3.itemsize, "bytes")
         print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 4 bytes
nbytes: 240 bytes
```

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

3.2 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 [29]: x1

Out[29]: array([5, 0, 3, 3, 7, 9])

In [30]: x1[0]

Out[30]: 5

In [31]: x1[4]

Out[31]: 7
```

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

```
In [32]: x1[-1]

Out[32]: 9

In [33]: x1[-2]

Out[33]: 7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
In [34]: x2

Out[34]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])

In [35]: x2[0, 0]

Out[35]: 3

In [36]: x2[2, 0]

Out[36]: 1

In [37]: x2[2, -1]

Out[37]: 7
```

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

```
In [38]: x2[0, 0] = 12
         x2
```

```
Out[38]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

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

```
In [39]: x1[0] = 3.14159  # this will be truncated!
         x1

Out[39]: array([3, 0, 3, 3, 7, 9])
```

3.3 Array Slicing: Accessing Subarrays

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

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

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

3.3.1 One-dimensional subarrays

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

Out[40]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [41]: x[:5]  # first five elements

Out[41]: array([0, 1, 2, 3, 4])

In [42]: x[5:]  # elements after index 5

Out[42]: array([5, 6, 7, 8, 9])

In [43]: x[4:7]  # middle sub-array

Out[43]: array([4, 5, 6])

In [44]: x[::2]  # every other element

Out[44]: array([0, 2, 4, 6, 8])

In [45]: x[1::2]  # every other element, starting at index 1

Out[45]: array([1, 3, 5, 7, 9])
```

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

```
In [46]: x[::-1]  # all elements, reversed

Out[46]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [47]: x[5::-2]  # reversed every other from index 5

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

3.3.2 Multi-dimensional subarrays

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

```
In [48]: x2

Out[48]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])

In [49]: x2[:2, :3]  # two rows, three columns

Out[49]: array([[12,  5,  2],
                [ 7,  6,  8]])

In [50]: x2[:3, ::2]  # all rows, every other column

Out[50]: array([[12,  2],
                [ 7,  8],
                [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
In [51]: x2[::-1, ::-1]

Out[51]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Accessing array rows and columns One commonly needed routine is accessing of 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 [52]: print(x2[:, 0])  # first column of x2

[12  7  1]


In [53]: print(x2[0, :])  # first row of x2
```

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

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

```
In [54]: print(x2[0])  # equivalent to x2[0, :]
```

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

3.3.3 Subarrays as no-copy views

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

```
In [55]: print(x2)
```

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

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

```
In [56]: x2_sub = x2[:2, :2]
         print(x2_sub)
```

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

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

```
In [57]: x2_sub[0, 0] = 99
         print(x2_sub)
```

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

```
In [58]: print(x2)
```

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

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

3.3.4 Creating copies of arrays

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

```
In [59]: x2_sub_copy = x2[:2, :2].copy()
         print(x2_sub_copy)
```

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

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

```
In [60]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

```
In [61]: print(x2)
```

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

3.4 Reshaping of Arrays

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

```
In [62]: grid = np.arange(1, 10).reshape((3, 3))
         print(grid)
```

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

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

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

```
In [63]: x = np.array([1, 2, 3])

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

Out[63]: array([[1, 2, 3]])

In [64]: # row vector via newaxis
         x[np.newaxis, :]

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

In [65]: # column vector via reshape
         x.reshape((3, 1))

Out[65]: array([[1],
                [2],
                [3]])

In [66]: # column vector via newaxis
         x[:, np.newaxis]

Out[66]: array([[1],
                [2],
                [3]])
```

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

3.5 Array Concatenation and Splitting

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

3.5.1 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 we can see here:

```
In [67]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])

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

You can also concatenate more than two arrays at once:

```
In [68]: z = [99, 99, 99]
         print(np.concatenate([x, y, z]))
```

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

It can also be used for two-dimensional arrays:

```
In [69]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [70]: # concatenate along the first axis
         np.concatenate([grid, grid])

Out[70]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [71]: # concatenate along the second axis (zero-indexed)
         np.concatenate([grid, grid], axis=1)

Out[71]: 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 [72]: x = np.array([1, 2, 3])
         grid = np.array([[9, 8, 7],
                          [6, 5, 4]])

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

Out[72]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In [73]: # horizontally stack the arrays
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])

Out[73]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

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

3.5.2 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 [74]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)
```

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

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

```
In [75]: grid = np.arange(16).reshape((4, 4))
         grid
```

```
Out[75]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [76]: upper, lower = np.vsplit(grid, [2])
         print(upper)
         print(lower)
```

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

```
In [77]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
```

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

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

4 The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 3) are built around the NumPy array. This section will present several examples 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 value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

4.1 NumPy Array Attributes

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

```
In [78]: import numpy as np
         np.random.seed(0)  # seed for reproducibility

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

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

```
In [79]: print("x3 ndim: ", x3.ndim)
         print("x3 shape:", x3.shape)
         print("x3 size: ", x3.size)
```

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

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

```
In [80]: print("dtype:", x3.dtype)
```

```
dtype: int32
```

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

```
In [81]: print("itemsize:", x3.itemsize, "bytes")
         print("nbytes:", x3.nbytes, "bytes")

itemsize: 4 bytes
nbytes: 240 bytes
```

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

4.2 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 [82]: x1

Out[82]: array([5, 0, 3, 3, 7, 9])

In [83]: x1[0]

Out[83]: 5

In [84]: x1[4]

Out[84]: 7
```

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

```
In [85]: x1[-1]

Out[85]: 9

In [86]: x1[-2]

Out[86]: 7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
In [87]: x2

Out[87]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])
```

```
In [88]: x2[0, 0]

Out[88]: 3

In [89]: x2[2, 0]

Out[89]: 1

In [90]: x2[2, -1]

Out[90]: 7
```

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

```
In [91]: x2[0, 0] = 12
         x2

Out[91]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

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

```
In [92]: x1[0] = 3.14159  # this will be truncated!
         x1

Out[92]: array([3, 0, 3, 3, 7, 9])
```

4.3 Array Slicing: Accessing Subarrays

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

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

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

4.3.1 One-dimensional subarrays

```
In [93]: x = np.arange(10)
         x

Out[93]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [94]: x[:5]  # first five elements

Out[94]: array([0, 1, 2, 3, 4])
```

```
In [95]: x[5:]  # elements after index 5

Out[95]: array([5, 6, 7, 8, 9])

In [96]: x[4:7]  # middle sub-array

Out[96]: array([4, 5, 6])

In [97]: x[::2]  # every other element

Out[97]: array([0, 2, 4, 6, 8])

In [98]: x[1::2]  # every other element, starting at index 1

Out[98]: array([1, 3, 5, 7, 9])
```

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

```
In [99]: x[::-1]  # all elements, reversed

Out[99]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [100]: x[5::-2]  # reversed every other from index 5

Out[100]: array([5, 3, 1])
```

4.3.2 Multi-dimensional subarrays

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

```
In [101]: x2

Out[101]: array([[12,  5,  2,  4],
                 [ 7,  6,  8,  8],
                 [ 1,  6,  7,  7]])

In [102]: x2[:2, :3]  # two rows, three columns

Out[102]: array([[12,  5,  2],
                 [ 7,  6,  8]])

In [103]: x2[:3, ::2]  # all rows, every other column

Out[103]: array([[12,  2],
                 [ 7,  8],
                 [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
In [104]: x2[::-1, ::-1]

Out[104]: array([[ 7,  7,  6,  1],
                 [ 8,  8,  6,  7],
                 [ 4,  2,  5, 12]])
```

Accessing array rows and columns One commonly needed routine is accessing of 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 [105]: print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

```
In [106]: print(x2[0, :])  # first row of x2
```

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

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

```
In [107]: print(x2[0])  # equivalent to x2[0, :]
```

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

4.3.3 Subarrays as no-copy views

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

```
In [108]: print(x2)
```

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

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

```
In [109]: x2_sub = x2[:2, :2]
          print(x2_sub)
```

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

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

```
In [110]: x2_sub[0, 0] = 99
          print(x2_sub)
```

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

```
In [111]: print(x2)
```

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

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

4.3.4 Creating copies of arrays

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

```
In [112]: x2_sub_copy = x2[:2, :2].copy()
          print(x2_sub_copy)
```

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

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

```
In [113]: x2_sub_copy[0, 0] = 42
          print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

```
In [114]: print(x2)
```

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

4.4 Reshaping of Arrays

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

```
In [115]: grid = np.arange(1, 10).reshape((3, 3))
          print(grid)
```

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

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

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

```
In [116]: x = np.array([1, 2, 3])

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

Out[116]: array([[1, 2, 3]])

In [117]: # row vector via newaxis
          x[np.newaxis, :]

Out[117]: array([[1, 2, 3]])

In [118]: # column vector via reshape
          x.reshape((3, 1))

Out[118]: array([[1],
                 [2],
                 [3]])

In [119]: # column vector via newaxis
          x[:, np.newaxis]

Out[119]: array([[1],
                 [2],
                 [3]])
```

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

4.5 Array Concatenation and Splitting

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

4.5.1 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 we can see here:

```
In [120]: x = np.array([1, 2, 3])
          y = np.array([3, 2, 1])
          np.concatenate([x, y])
```

```
Out[120]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In [121]: z = [99, 99, 99]
          print(np.concatenate([x, y, z]))
```

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

It can also be used for two-dimensional arrays:

```
In [122]: grid = np.array([[1, 2, 3],
                           [4, 5, 6]])
```

```
In [123]: # concatenate along the first axis
          np.concatenate([grid, grid])
```

```
Out[123]: array([[1, 2, 3],
                 [4, 5, 6],
                 [1, 2, 3],
                 [4, 5, 6]])
```

```
In [124]: # concatenate along the second axis (zero-indexed)
          np.concatenate([grid, grid], axis=1)
```

```
Out[124]: 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 [125]: x = np.array([1, 2, 3])
          grid = np.array([[9, 8, 7],
                           [6, 5, 4]])

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

```
Out[125]: array([[1, 2, 3],
                 [9, 8, 7],
                 [6, 5, 4]])
```

```
In [126]: # horizontally stack the arrays
          y = np.array([[99],
                        [99]])
          np.hstack([grid, y])
```

```
Out[126]: array([[ 9,  8,  7, 99],
                 [ 6,  5,  4, 99]])
```

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

4.5.2 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 [127]: x = [1, 2, 3, 99, 99, 3, 2, 1]
          x1, x2, x3 = np.split(x, [3, 5])
          print(x1, x2, x3)
```

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

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

```
In [128]: grid = np.arange(16).reshape((4, 4))
          grid
```

```
Out[128]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11],
                 [12, 13, 14, 15]])
```

```
In [129]: upper, lower = np.vsplit(grid, [2])
          print(upper)
          print(lower)
```

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

```
In [130]: left, right = np.hsplit(grid, [2])
          print(left)
          print(right)
```

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

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

5 Fancy Indexing

In the previous sections, we saw 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 section, we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing we've already seen, but 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.

5.1 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 [131]: import numpy as np
          rand = np.random.RandomState(42)

          x = rand.randint(100, size=10)
          print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

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

```
In [132]: [x[3], x[7], x[2]]
```

```
Out[132]: [71, 86, 14]
```

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

```
In [133]: ind = [3, 7, 4]
          x[ind]
```

```
Out[133]: array([71, 86, 60])
```

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

```
In [134]: ind = np.array([[3, 7],
                          [4, 5]])
          x[ind]
```

```
Out[134]: array([[71, 86],
                 [60, 20]])
```

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

```
In [135]: X = np.arange(12).reshape((3, 4))
          X
```

```
Out[135]: 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 [136]: row = np.array([0, 1, 2])
          col = np.array([2, 1, 3])
          X[row, col]

Out[136]: 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 Computation on Arrays: Broadcasting. So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
In [137]: X[row[:, np.newaxis], col]

Out[137]: 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 [138]: row[:, np.newaxis] * col

Out[138]: 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.

5.2 Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen:

```
In [139]: print(X)

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
In [140]: X[2, [2, 0, 1]]
```

```
Out[140]: array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
In [141]: X[1:, [2, 0, 1]]

Out[141]: array([[ 6,  4,  5],
                 [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
In [142]: mask = np.array([1, 0, 1, 0], dtype=bool)
          X[row[:, np.newaxis], mask]

Out[142]: array([[ 0,  2],
                 [ 4,  6],
                 [ 8, 10]])
```

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

5.3 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 by D matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

```
In [143]: mean = [0, 0]
          cov = [[1, 2],
                 [2, 5]]
          X = rand.multivariate_normal(mean, cov, 100)
          X.shape

Out[143]: (100, 2)
```

Using the plotting tools we will discuss in Introduction to Matplotlib, we can visualize these points as a scatter-plot:

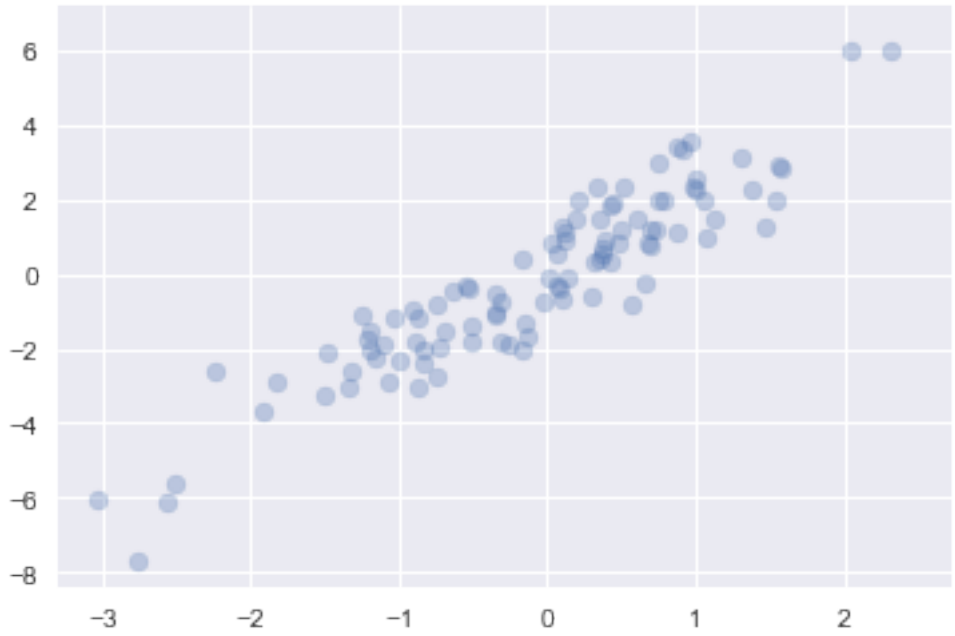
```
In [144]: %matplotlib inline
          import matplotlib.pyplot as plt
          import seaborn; seaborn.set()  # for plot styling

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

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

```
In [145]: indices = np.random.choice(X.shape[0], 20, replace=False)
          indices

Out[145]: array([25, 56, 31, 60, 68, 51,  7,  5, 86, 28, 54, 94,  9,  6, 73, 45, 74,
                 97,  4, 96])

In [146]: selection = X[indices]  # fancy indexing here
          selection.shape

Out[146]: (20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points:

```
In [147]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
          plt.scatter(selection[:, 0], selection[:, 1],
                      facecolor='none', s=200);
```



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 Hyperparameters and Model Validation), and in sampling approaches to answering statistical questions.

5.4 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 [148]: x = np.arange(10)
          i = np.array([2, 1, 8, 4])
          x[i] = 99
          print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```

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

```
In [149]: x[i] -= 10
          print(x)
```

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

[6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Where did the 4 go? The result of this operation is to first assign `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 [151]: i = [2, 3, 3, 4, 4, 4]
          x[i] += 1
          x

Out[151]: 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 (available since NumPy 1.8), and do the following:

```
In [152]: x = np.zeros(10)
          np.add.at(x, i, 1)
          print(x)

[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.

5.5 Example: Binning Data

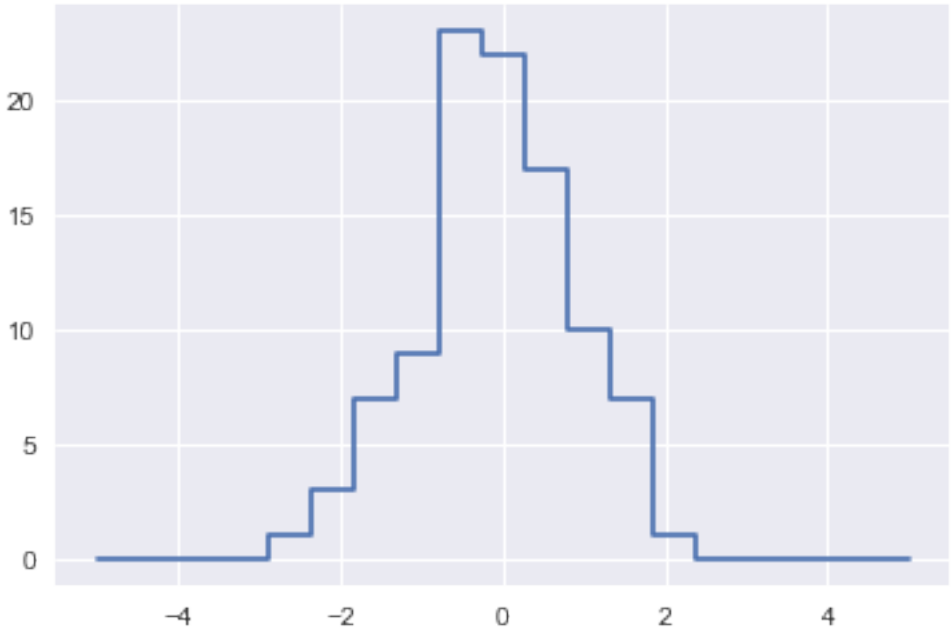
You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
In [153]: np.random.seed(42)
          x = np.random.randn(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:

```
In [154]: # plot the results
          plt.plot(bins, counts, linestyle='steps');
```



Of course, it would be silly 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 seen here. 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 [155]: print("NumPy routine:")
          %timeit counts, edges = np.histogram(x, bins)

          print("Custom routine:")
          %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
66.7 ţs ś 553 ns per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

```
Custom routine:
19 µs ± 484 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Our own one-line algorithm is several times faster than 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 [156]: x = np.random.randn(1000000)
          print("NumPy routine:")
          %timeit counts, edges = np.histogram(x, bins)

          print("Custom routine:")
          %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
80.4 ms ± 949 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Custom routine:
160 ms ± 2.05 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 Big-O Notation). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applications is 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.