Rob Mastrodomenico

The Python Book

Table of Contents

List of Illustrations

5. Chapter 19

The Python Book

Rob Mastrodomenico

Global Sports Statistics

Swindon, United Kingdom

1
Introduction

Welcome to *The Python Book*, over the following pages you will be given an insight into the Python language. The genesis of this book has come from my experience of using and more importantly teaching Python over the last 10 years. With my background as a Data Scientist, I have used a number of different programming languages over the course of my career and Python being the one that has stuck with me. Why Python? For me I enjoy Python because its fast to develop with and covers many different application allowing me to use Python for pretty much everything. However for you the reader, Python is a great choice of language to learn as its easy to pick up and fast to get going with which means that for the novice programmers they can feel like they are making progress. This book is not just for complete novices, if you have some experience with Python, then this book is a great reference. The fact that you can pick up Python quickly means that many users skip the basics. This book looks to cover all the basics giving you the building blocks to do great things with the language. What this book is not intended to do is over complicating anything. Python is beautiful in its simplicity and this book looks to stick to that approach. Concepts will be explained in simple

terms and examples will be used to show how to practically use the introduced concepts.

Now having discussed what this book is intended to do, what is Python? Simply put Python is a programming language, its general purpose meaning that it can do lots of things. In this book, we will specialise in applying Python to data-driven applications, however Python can be used for many other applications including AI, machine learning, web development, to name just a few. The language itself is of high level and also interpreted meaning that code need not be compiled before running. One of the big attractions to the language is the simplicity of its syntax, which makes it great to learn and even better to write code. Aside from the clear, easy to understand syntax, the language makes use of indentation as an important tool to distinguish different elements of the code. Python is an object-orientated language and we will demonstrate this in more detail throughout this book. However, you can write Python code how you prefer be it object orientated, functional or interactively. The best way to demonstrate Python is by doing, so let's get started but to do so we need to get Python installed.

2

Getting Started

For the purposes of this book, we want you to install the Anaconda distribution of Python that is available at https://www.anaconda.com. Here, you have distributions for Windows, Mac, and Linux, which can be easily installed on your computer. Once you have the Anaconda installed, you will have access to the Anaconda navigator as shown in Figure 2.1.

Here, you get the following included by default:

- JupyterLab
- Notebook
- Qt Console
- Spyder

To follow the examples within this book you can use the Notebook or Qt Console. The Notebook is an interactive web based editor as shown in Figure 2.2.

Here, you can type your code, run the command, and then see the result, which is a nice way to work and is very popular. Here, we will

show how we can define a variable x and then just type x and run the command with the run button to show the result (Figure 2.3).

However for the purposes of the book we will use a console-based view that you can easily obtain through the Qt Console. An example is shown in Figure 2.4.

Like with the notebook, we show the same example using Qt Console in Figure 2.5.

Within this book we will denote anything that is an input with >>> and with any output having no arrows preceding it (Figure 2.6).

Another concept that the reader will need to be familiar with is the ability to navigate using the terminal (linux systems including mac) or command prompt (windows). These can be obtained through various approaches but simply using the search procedures with the word terminal or command prompt will bring up the relevant screen. To navigate through the file system you can use the command cd to change directory. This essentially is like us clicking on a folder to see what is in it. Unlike using a file viewing interface you cannot see what is in a given directory by default so to do so you need to use the command ls. This command lists the files and directories within the current locations. Let's demonstrate with an example of navigating to a directory and then running a python file.

Aside from the Anaconda navigator we have over 250 open-source data science and machine learning packages are automatically installed. You can also make use of the conda installer to install over 7500 packages easily into Python. A full list of packages that come with Anaconda is available for the relevant operating system from https://repo.anaconda.com/pkgs/. Details on the using the conda installer is available from https://docs.anaconda.com/anaconda/user-guide/tasks/install-packages/ however this is outside the scope of this book. The last concept we will raise but not cover in detail is that of virtual environments. This concept is where the user develops in an isolated Python environment and adds packages as needed. It is a very popular approach to development however as this book is aimed at beginners we use all packages included in the Anaconda installation.
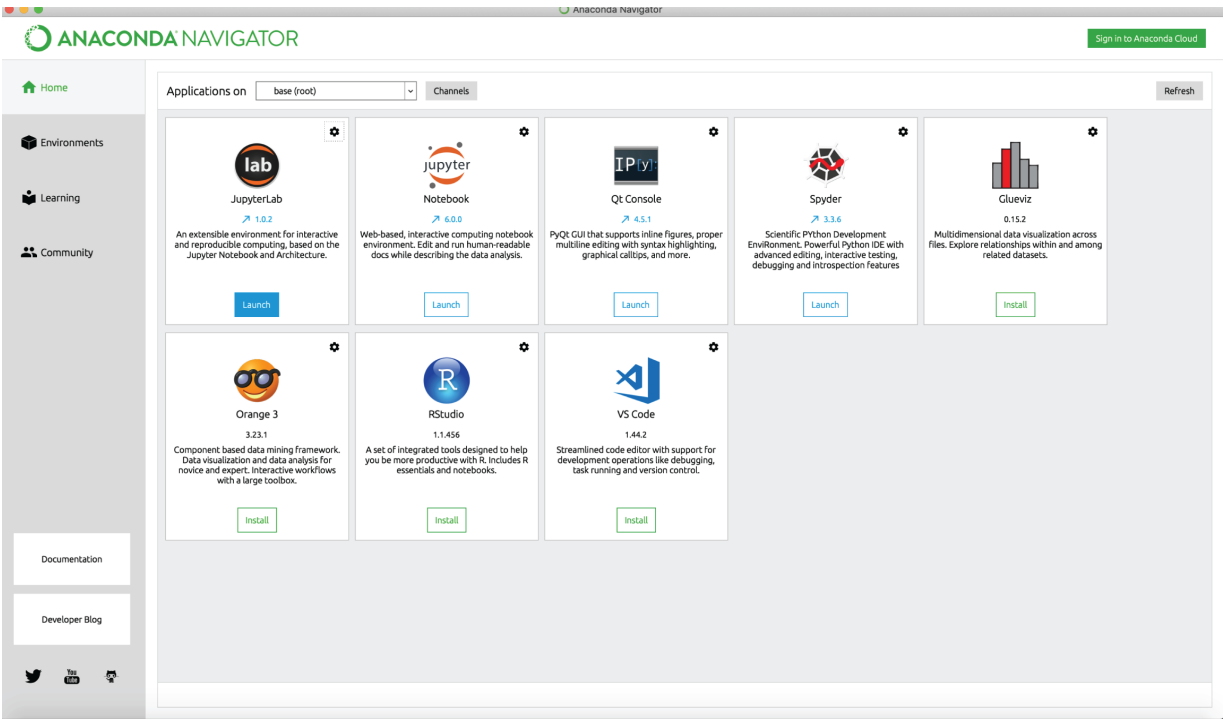
Figure 2.1 Anaconda navigator.
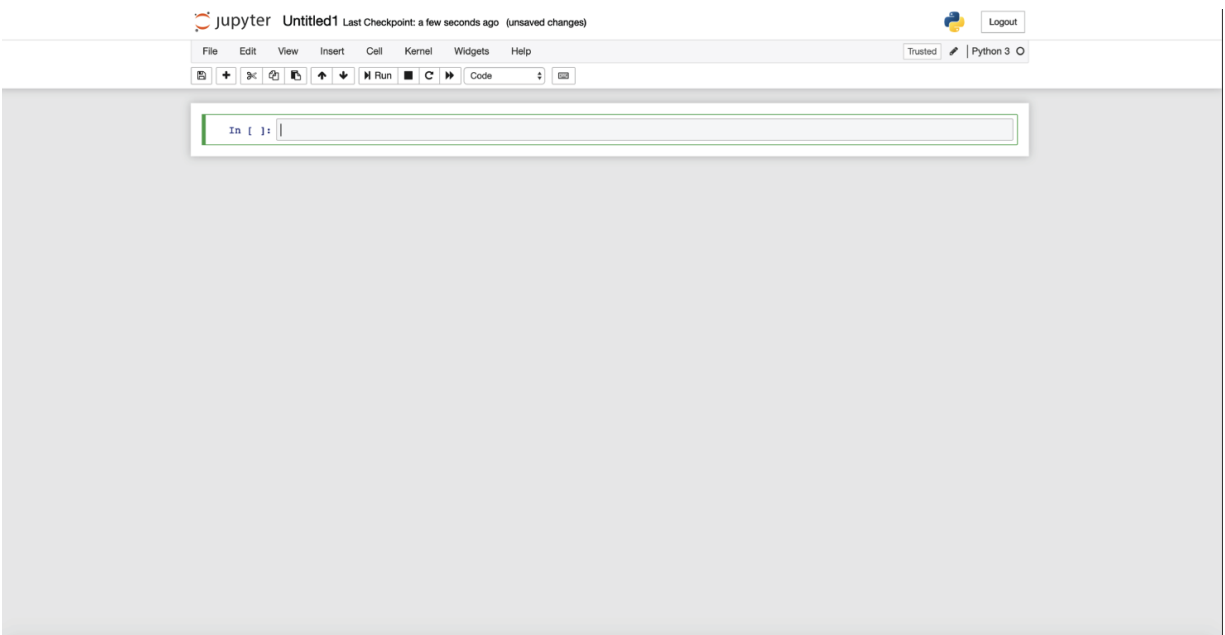
Figure 2.2 Jupyter Notebook.
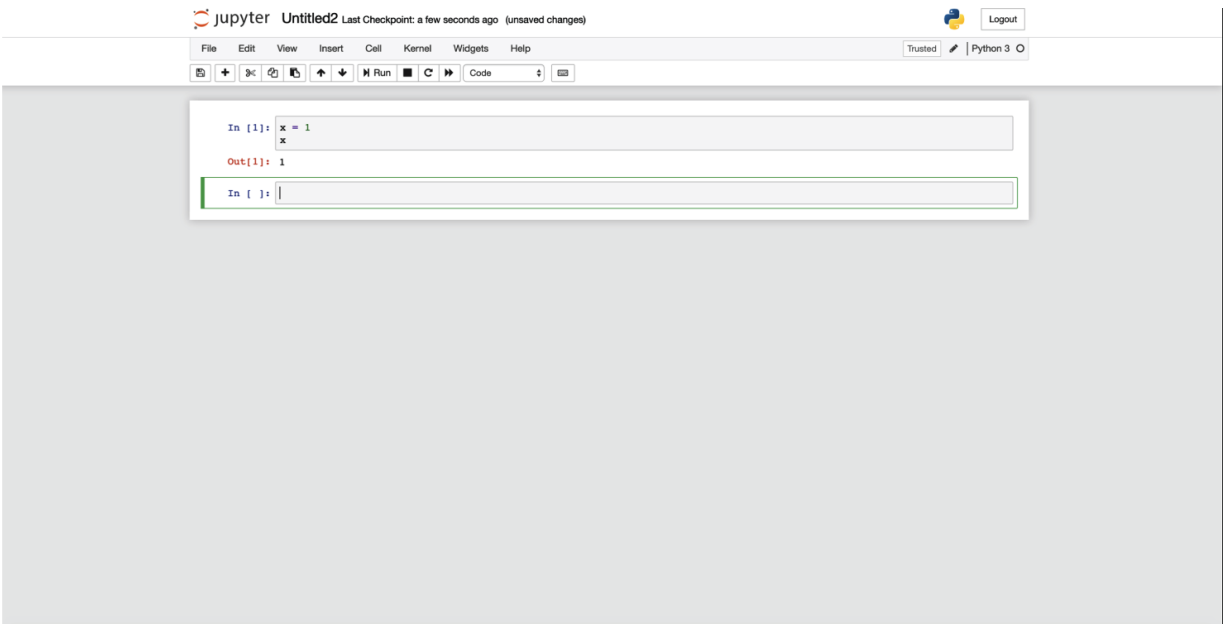
Figure 2.3 Jupyter Notebook example.
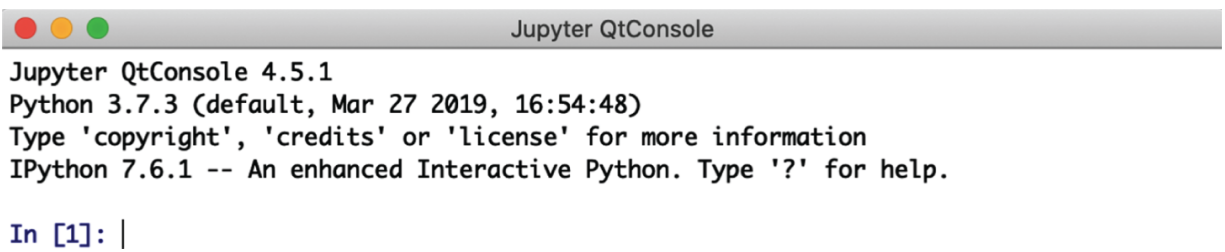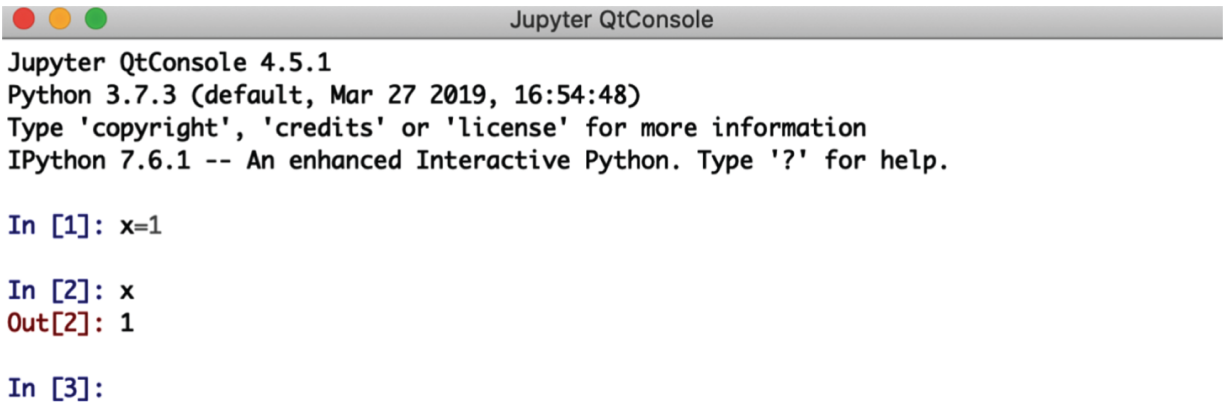
Figure 2.4 Qt Console.
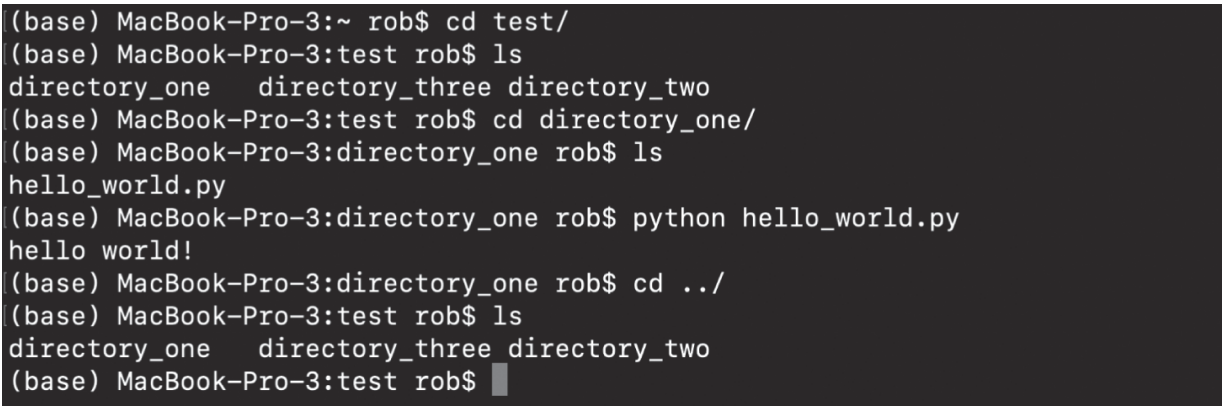
Figure 2.5 Qt Console example.

Figure 2.6 Command line example.

3
Packages and Builtin Functions

We have discussed packages without really describing what they are so let's look at packages and how it sits within the general setup of Python. As mentioned previously, Python is object orientated which means that everything is an object, you'll get to understand this in practice, however there are a few important builtin functions which aren't objects and they are worth mentioning here as they will be used within the book. These builtin types will be used throughout the book so keep an eye out for them. Below we show some useful ones, for a full list refer to https://docs.python.org/3/library/functions.html.

- dir(): This function takes in an object and returns the _dir_() of that object giving us the attributes of the object.

```
>>> name = 'Rob'
>>> dir(name)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- float(): Returns a floating point number from an integer of string

```
>>> x = '1'
>>> float(1)
1.0
```

- int(): Returns an integer from a float of string

```
>>> x = '1'
>>> int(1)
1
```

- len(): Returns the length of an object

```
>>> name = 'Rob'
>>> len(name)
3
>>> x = [1, 2, 3, 4]
>>> len(x)
4
```

- list(): Creates a list from the argument given

```
>>> name = 'rob'
>>> list(name)
['r', 'o', 'b']
```

- max(): Gives the maximum value from the argument provided

```
>>> x = [1, 2, 3, 4]
>>> max(x)
4
>>> name = ['r', 'o', 'b']
>>> max(name)
'r'
```

- min(): Gives the minimum value from the argument provided

```
>>> x = [1, 2, 3, 4]
>>> min(x)
1
>>> name = ['r', 'o', 'b']
>>> min(name)
'b'
```

- print(): Prints the object to the text stream

```
>>> x = [1, 2, 3, 4]
>>> print(x)
[1, 2, 3, 4]
```

- round(): Rounds the number to a specified precision

```
>>> y = 1.387668
>>> round(y, 2)
1.39
```

- str(): Converts the object to type string

```
>>> y = 1.387668
>>> str(y)
'1.387668'
```

- type(): Returns the type of an object

```
>>> y = 1.387668
>>> type(y)
<class 'float'>
```

- abs(): Returns the absolute value of a numeric value passed in

```
>>> z = -0.657
>>> abs(z)
0.657
```

- help(): Gives access to the Python help system

```
>>> help(list)

Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
```

Now if you are unfamiliar with the Python the concepts used above they will be introduced throughout this book.

Alongside these builtin functions Python also comes with a number of packages. These packages perform specific tasks and are imported into our code. Python has a number of packages that come as default however there are lots of third-party packages which we can also use. In using the Anaconda distribution we get all the default packages as well as the packages that are described previously. We will cover both default and third-party packages throughout this book. To

demonstrate this we will introduce how to import a package. The package we are going to introduce is datetime which is part of the standard Python library. What this means is it comes with the Python and is not developed by a third party. Now to import the datetime package you just need to type the following:

```
>>> import datetime
```

In doing this we now have access to everything within datetime and to see what datetime contains we can run the built in function dir which as we showed earlier gives us the attribute of the object.

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Now if we want to see what these attributes are we use the dot syntax to access attributes of the object. So to see what MINYEAR and MAXYEAR are we can do so as follows.

```
>>> import datetime
>>> datetime.MAXYEAR
9999
>>> datetime.MINYEAR
1
```

Now we can import specific parts of a package by using the from syntax as demonstrate below.

```
>>> from datetime import date
```

So what this says is from the package datetime import the specific date attribute. This is then the only aspect of datetime that we have access to. This is good practice to only import what you need from a package. Now every time we want to use date we have to call date, in this case its easy enough but you can also give the import an alias which can reduce your code down.

```
>>> from datetime import date as d
```

That is the basics of importing packages, throughout this book we will import from various packages as well as show how this same syntax can be used to import our own code. Alongside builtin functions these are key concepts that we will use extensively within this book.

4
Data Types

The next concept of Python that we will introduce is data types and in this chapter we will introduce a number of these and show how they behave when applied to some basic operators. We first start by introducing integers which are a number without a decimal point, written as follows:

```
>>> 1
1
>>> 2
2
```

A float is by definition a floating point number so we can write the previous as follows:

```
>>> 1.0
1.0
>>> 2.0
2.0
```

A string is simply something enclosed in either a double or single quote. So again we can rewrite what we have seen as follows:

```
>>> "2.0"
'2.0'
```

Given the fact that we know how to define these variables, how can we check what they are? Well, conveniently Python has a type method that will allow us to determine the type of a variable. So we will rewrite what we have done and assign each instance to a variable and then see what type Python thinks they are

```
>>> = 1
>>> type(x)
<class "int">
>>> y = 1.0
>>> type(y)
<class "float">
>>> z = "1.0"
>>> type(z)
<class "str">
```

So now we can define the variables, the question is what can we do with them? Initially we will consider the following operations:

- +

- −

- *

- /

These are commonly known as the mathematical operation: addition, subtraction, multiplication, and division.

So let's start with + now if we have two integers applying + is mathe-matical addition as we will show

```
>>> x = 10
>>> y = 16
>>> x + y
26
```

Similarly if we do the same with two floats we get a similar result

```
>>> x = 10.0
>>> y = 16.0
>>> x + y
26.0
```

But what happens if we apply addition to a float and an integer, let's see

```
>>> x = 10
>>> y = 16.0
>>> z = x + y
>>> z
26.0
>>> type(z)
<class "float">
```

What we see is that addition works on a float and an integer but it re-turns a float, so it's converting the integer into a float.

What if we use addition on a string? Well this is the interesting part, let's run the same example from before with x and y as string representations.

```
>>> x = "10"
>>> y = "16.0"
>>> z = x + y
>>> z
"1016.0"
```

What has happened here? Well we have stuck together x and y, this is known as concatenation and is a very powerful tool in dealing with strings.

We considered the + operation with integers and floats but what will happen if we do the + operation with a string and say an integer

```
>>> x = "10"
>>> y = 16
>>> z = x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not float
```

What we see here is an error message saying we cannot concatenate a str and int object. So Python in this instance wants to use the + operation as concatenation but due to the fact it doesn't have two strings it can't do that and hence throws an error. In Python you can-

not mix a string and integer or string and float so we won't consider operations between these types for the rest of this section.

Let us now look at the − operation. First considering two integers we get the following:

```
>>> x = 10
>>> y = 16
>>> z = x - y
>>> z
-6
```

As you may have expected the − operation with two integers acts as mathematical subtraction. If we apply it to two floats, or to a mix of floats and integers it acts as subtraction.

What about for strings, can we apply − to two strings?

```
>>> x = "10"
>>> y = "16"
>>> z = x - y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Here we get another error but this time it is because the $-$ operation doesn't support strings. What this means is that when you try to operate on two strings using this operation, it doesn't know what to do. The same is true for * and / operations on string. So, if we are dealing with strings the only operation from this section that we can use is + which is concatenation.

The next operation we will consider is * which is generally known as mathematical multiplication to most. So considering its use on two integers we get the following:

```
>>> x = 10
>>> y = 16
>>> x * y
160
```

As we can see its mathematical multiplication, the same is true when we run the same on two floats. Let us see what happens when we mix floats and integers.

```
>>> x = 10
>>> y = 16.0
>>> x * y
160.0
```

As we can see it returns multiplication in float format, so like with addition and subtraction it converts integers to floats.

Next, we need to see how / operation works on integers and floats, so first we consider the same types, so we will apply / on integers:

```
>>> x = 10
>>> y = 16
>>> x / y
0.625
```

There are other data types beyond these and the first we consider are complex numbers which can be defined as follows

```
>>> x = 3+5j
>>> x
(3+5j)
>>> y = 5j
>>> y
5j
>>> z = -5j
>>> z
(-0-5j)
```

We can obtain the real and imaginary parts of our complex numbers as follows

```
>>> x.real
3.0
>>> x.imag
5.0
>>> y.real
0.0
>>> y.imag
5.0
>>> z.real
-0.0
>>> z.imag
-5.0
```

We can also use the built-in function complex

```
>>> a = 3
>>> b = 5
>>> c = complex(a, b)
>>> c
(3+5j)
>>> c.real
3.0
>>> c.imag
5.0
```

In terms of operation we can use the standard operators shown earlier to complex numbers and the results are as follows

```
>>> x = 3+5j
>>> x
(3+5j)
>>> y = 5j
>>> y
5j
>>> z = -5j
>>> z
(-0-5j)
>>> x + y
(3+10j)
>>> x - y
(3+0j)
>>> x / y
(1-0.6j)
>>> x * y
(-25+15j)
```

We can also add, subtract, divide or multiply integers or floats to a complex numbers as we show

```
10.2
(13.2+5j)
>>> x - 10
(-7+5j)
>>> x - 10.2
(-7.199999999999999+5j)
>>> x * 10
(30+50j)
>>> x
(3+5j)
>>> x * 10.2
(30.599999999999998+51j)
>>> x * 10.2
(30.599999999999998+51j)
>>> x / 10
(0.3+0.5j)
>>> x / 10.2
(0.29411764705882354+0.4901960784313726j)
```

In adding or subtracting an integer or float with a complex we change only the real part which is to be expected, however if we multiply or divide we apply that value across both real and imaginary parts.

Next we look at boolean values in Python, these can be defined using True or False

```
>>> x = True
>>> x
True
>>> y = False
>>> y
False
```

Integers or floats can be converted into a boolean using the built-in function bool. This treats any value as 0 or 0.0 as False and any other value to be True.

```
>>> x = bool(1)
>>> x
True
>>> y = bool(0.0)
>>> y
False
>>> z = bool(-10)
>>> z
True
```

Surprisingly we can use the operators in this chapter on boolean variables. The key to note is that a value of True is evaluated as 1 and False as 0, so you can see examples of this below.

```
>>> x = True
>>> y = False
>>> x + y
1
>>> x - y
1
>>> x * y
0
>>> x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> x = True
>>> y = True
>>> x + y
2
>>> x - y
0
>>> x + y
2
>>> x * y
1
>>> x / y
1.0
>>> x = False
>>> y = False
>>> x + y
0
>>> x - y
0
>>> x * y
0
>>> x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In most cases the results are as expected considering that we are dealing with 1 or 0 in the operation. However anytime that we divide by zero we get a ZeroDivisonError so be careful with zero division.

We can also create byte, byte arrays and memory view objects with the following syntax.

```
>>> x = b"Hello World"
>>> x
b'Hello World'
>>> y = bytearray(6)
>>> y
bytearray(b'\x00\x00\x00\x00\x00\x00')
>>> z = memoryview(bytes(5))
>>> z
<memory at 0x7fdde6fea348>
```

We can concatenate byte strings together in the way we have shown with strings

```
>>> x = b"Hello World"
>>> x
b'Hello World'
>>> y = b" My name is Rob"
>>> y
b' My name is Rob'
>>> x + y
b'Hello World My name is Rob'
```

What we have covered in this chapter is some of the data types in Python and how to operate on them using standard mathematical methods. One thing to take from this is the mechanism that Python uses to operate on objects and that is simply look for a method that can take in the arguments that you pass into it.

5

Operators

The previous chapter introduced data types and some basic opera-tors but in this chapter we build on this by introducing a number of key operators that are important to understand. As shown in our intro-ductory examples in the previous chapter, we can define a variable as follows:

```
>>> x = 2 + 1
>>> x
3
```

Here, we are assigning the variable x with the result of $1 + 2$ so x is 3. Now, if we wanted to see if the value of x was equal to 3 we would use == which refers to equality.

```
>>> x = 2 + 1
>>> x == 3
True
```

We have shown how to test for equality but what about inequality. Well Python has that sorted as well, instead of using == we use != which is not equals. Using the example from before we get the following:

```
>>> X = 2 + 1
>>> X != 3
False
```

What we have here is the result of an equality statement like this being of type boolean (True or False).

We can also test to see if something is greater or less than another element.

```
>>> X = 2 + 1
>>> X > 4
False
>>> X < 4
True
>>> X >= 4
False
>>> X >= 3
True
>>> X <= 3
True
```

Here we have introduced the following tests which takes the value on the left against the value on the right and tests for

- `>` for greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

You can also test for equality using the statement is. Now it is not strictly the same as using == which we demonstrate earlier. Essentially the difference is that it returns True if the variables in question points to the same object whilst == returns True if the values are equal. It is a very subtle difference so you need to be careful with it. A simpler explanation is that == returns True if the variables being compared are equal, whereas is checks whether they are the same. The below examples shows the dangers in using is

```
>>> a = 1
>>> a is 1
True
>>> a == 1
True
>>> a = []
>>> b = []
>>> a is b
False
>>> a == b
True
```

In the first instance a is assigned to be 1 and we say is a 1 and it is so we get True returned. In the second instance we assign a and b to be empty list (we will cover what a list is later) and we can see we return False with the is statement and True with the equals. The reason behind this is that they are not the same lists so

```
>>> a is b
False
```

However, they are both lists so using the comparison statement ==
we return True as they are both empty lists. If we assigned a as a list
and b = a we would get the following:

```
>>> a = []
>>> b = a
>>> a is b
True
```

The reason being is that b is the same as a so they are the same
thing. As with == and != we have is not again the != is a test of equali-
ty between two variables whereas the Python statement is not, is a
test of identity. A good example of this is when you compare a vari-
able to the Python Null value denoted as None. Here, the preferred
way to write it is

```
>>> a = 21
>>> a is not None
True
```

You can override the variable by assigning something to it like we did
before

```
>>> x = 1
>>> x
1
>>> x = 10
>>> x
10
```

All pretty simple stuff. Now if we have three variables that we want to assign we can do it as follows

```
>>> x = 1
>>> y = 2
>>> z = 3
>>> x
1
>>> y
2
>>> z
3
```

This is fine to do however it takes up a lot of space so instead you can write your assignment as follows

```
>>> x, y, z = 1, 2, 3
>>> x
1
>>> y
2
>>> z
3
```

This just makes it easier to assign variables and makes your code shorter and hopefully more readable. Obviously the naming convention I've used for the variables isn't the best and it makes for better code to give your variables meaningful names as it will help those who have to go back and read your code.

We have looked at assigning variables however what if we want to do something to the variable like say add something to the value. Lets assume we have a variable profit and we want to add 100.0 to it, then we could do it as follows

```
>>> profit = 1000.0
>>> profit
1000.0
>>> profit = profit + 100
>>> profit
1100.0
```

What we are doing here is assigning profit the initial value then assigning it its value plus 100. There is nothing wrong with what we did here however the more Pythonic way would be to do this

```
>>> profit = 1000.0
>>> profit
1000.0
>>> profit += 100.0
>>> profit
1100.0
```

Similarly is we wanted to multiply the value by 20% we could do so as follows

```
>>> profit = 1000.0
>>> profit
1000.0
>>> profit *= 1.2
>>> profit
1200.0
```

As you would think we can do the same for division and subtraction

```
>>> profit
1000.0
>>> profit *= 1.2
>>> profit
1200.0
>>> profit = 1000.0
>>> profit
1000.0
>>> profit -= 1.2
>>> profit
998.8
>>> profit /= 2
>>> profit
499.4
```

There are some other non typical operators that we can use in Python, the first one being modulus which returns the remainder of integer division.

```
>>> x = 2
>>> y = 10
>>> x/y
0.2
>>> x % y
2
```

We can also perform exponentiation as follows

```
>>> x = 2
>>> y = 10
>>> y ** x
100
```

Python also gives us the operator for floor division which was how division was used in Python 2 however it is now performed explicitly using the floor operator

```
>>> x = 2
>>> y = 10
>>> x/y
0.2
>>> x // y
0
```

Following on from what we showed earlier these operations can be performed using the equals operator approach to assign back to the variable. So we can perform modulus, exponentiation and floor operations as follows.

```
>>> x = 2
>>> y = 10
>>> x %= y
>>> x
2
>>> x = 2
>>> y = 10
>>> y **= x
>>> y
100
>>> x = 2
>>> y = 10
>>> x //= y
>>> x
0
```

We can also chain together these types of operators as follows

```
>>> x = 2
>>> y = 10
>>> x < y
True
>>> z = 3
>>> a = 3
>>> b = 3
>>> a == b
True
>>> x < y and a == b
True
>>> x > y
False
>>> x > y and a == b
False
>>> a != b
False
>>> x > y and a != b
False
```

So we can combine our logic statements together to form more complex statements. This will become useful later in the book when we apply these with other Python functionality.

This chapter has shown how we perform various operations on Python data types and this will form the basis for lots of the logic that we apply throughout this book.

6
Dates

In the previous chapters we covered some of the main data types in Python but one thing that is quite important is dates. For anyone who has worked with dates they can be tricky things, there are different ways to format them and they can be hard to manipulate, however Python has you covered. If we want to create a datetime object we do so as follows (we earlier showed how to import the datetime package):

```
>>> from datetime import datetime as dt
>>> d = dt(2017, 5, 17, 12, 10, 11)
>>> d
datetime.datetime(2017, 5, 17, 12, 10, 11)
>>> str(d)
"2017-05-17 12:10:11"
```

What we have done here is create a datetime by passing into dt the year, month, day, hour, minute, second to give us a datetime object. If we want to see it in a more friendly way we can type str around it and we get the string representation. Given we can define a date we can operate on dates as follows:

```
>>> d1 = dt(2017, 5, 17, 12, 10, 11)
>>> d1
datetime.datetime(2017, 5, 17, 12, 10, 11)
>>> d2 = dt(2016, 4, 7, 1, 1, 1)
>>> d2
datetime.datetime(2016, 4, 7, 1, 1, 1)
>>> d1 - d2
datetime.timedelta(405, 40150)
>>> str(d1 - d2)
"405 days, 11:09:10"
```

Here, we have created two dates and then subtracted one from the other. What we get back is a timedelta object and converting it to a string we can see that it represents the days and time from the date subtraction. To understand timedelta we can import it just as we did with datetime.

```
>>> from datetime import timedelta as td
```

Now timedelta behaves a little different from datetime in that we do not pass in years and months but instead days, hours, minutes, and seconds. If you think about it setting a timedelta using years and months does not make much sense as they are not consistent units of time. We can create a timedelta object of 1 day, 2 hours 10 minutes as follows.

```
>>> td(days=1, hours=2, minutes=10)
datetime.timedelta(1, 7800)
>>> change = td(days=1, hours=2, minutes=10)
>>> d1 = dt(2017, 5, 17, 12, 10, 11)
>>> d1 - change
datetime.datetime(2017, 5, 16, 10, 0, 11)
>>> str(d1 - change)
"2017-05-16 10:00:11"
```

What we then did with the timedelta is subtract it from the datetime and it returns the datetime with the period in the timedelta subtracted from it. When this is shown as a str we see we have retained the datetime format, so this makes it much easier to do date subtraction, the same is true if we wanted to add to the datetime.

```
>>> str(d1)
"2017-05-17 12:10:11"
>>> d1 + change
datetime.datetime(2017, 5, 18, 14, 20, 11)
>>> str(d1 + change)
"2017-05-18 14:20:11"
```

If we just want to work with dates we can import date from datetime:

```
>>> from datetime import date as d
>>> d.today()
datetime.date(2017, 5, 17)
>>> str(d.today())
"2017-05-17"
```

So we imported in much the same way we did before, but let's say we want the current date we can use the method today to show the current date.

Taking the above we can apply what we have seen here to some examples. First let us consider a date in time and how we can calculate how long ago it was. The date that we will use is the 20 July 1969 which is the date that Neil Armstrong stepped on the moon. We can work out how far back from the current date it is:

```
>>> import datetime as dt
>>> now_date = dt.datetime.now()
>>> now_date
datetime.datetime(2020, 3, 25, 9, 55, 17, 272032)
>>> moon_date = dt.datetime(1969, 7, 20)
>>> moon_date
datetime.datetime(1969, 7, 20, 0, 0)
>>> date_since_moon = now_date - moon_date
>>> date_since_moon
datetime.timedelta(days=18511, seconds=35717, microseconds=272032)
>>> date_since_moon.days
18511
>>> date_since_moon.seconds
35717
```

What we have seen is we can subtract a datetime from a date and get back a resulting timedelta that takes into account the different types.

Next, we can look at how far in the future a date is so we look at the date 2030-01-01 relative to now (at the time of print).

```
>>> import datetime as dt
>>> now_date = dt.date.today()
>>> now_date
datetime.date(2020, 3, 25)
>>> future_date = dt.date(2030, 1, 1)
>>> future_date
datetime.date(2030, 1, 1)
>>> distance = future_date - now_date
>>> distance
datetime.timedelta(days=3569)
>>> distance.days
3569
```

What we do here is set the date as today using the today method and
then we can subtract our future date which we set as a date giving us
a timedelta object. As we have seen before we can access the spe-
cific number of days between 2020-03-05 and 2030-01-01 as
3569 days.

In this chapter, we have introduced how we manipulate dates within
Python but also shown how packages including how we import them
but also how we can access methods and attributes within them.

7
Lists

In this chapter we will cover lists, not the kind you write down your food shop but a really important part of Python. Lists are the first of the storage types we consider from core Python, the others being Tuples, Dictionaries, and Sets. Fundamentally lists allow us to store things, let's say we want to have a variable containing the numbers 1–10, we can store them in a list as follows:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

What we have here is a list of integers, we could do the same with a list of strings.

```
>>> numbers = ["1","2","3","4","5","6","7","8","9","10"]
>>> numbers
["1","2","3","4","5","6","7","8","9","10"]
```

The beauty with lists is that we could have a mixture of things in it, so it could look like this:

```
>>> stuff = ["longing","rusted","furnace","daybreak",17,"Benign",9]
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
```

We could even put variables in a list, so if we take the previous example we could have replaced some of the entries with the variable equivalent

```
>>> x = "longing"
>>> y = 17
>>> stuff = [x,"rusted","furnace","daybreak",y,"Benign",9]
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
```

We can even put lists in lists, like so

```
>>> first_names = ["Steve", "Peter", "Tony", "Natasha"]
>>> last_names = ["Rodgers", "Parker", "Stark", "Romanoff"]
>>> names = [first_names, last_names]
>>> names
[["Steve", "Peter", "Tony", "Natasha"], ["Rodgers", "Parker",
 "Stark", "Romanoff"]]
```

Basically lists are really powerful, however given you have put something in it you then need to be able access it. Lists in Python are 0 indexed which means to access the first element of the list you need to do the following:

```
>>> stuff = ["longing","rusted","furnace","daybreak",17,"Benign",9]
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff[0]
"longing"
```

Similarly to get the second and fifth elements you would do the following:

```
>>> stuff[1]
"rusted"
>>> stuff[4]
17
```

If you've not worked with something that is zero indexed then it can be a tad annoying to start off with but once you get the hang of it then it will just become second nature. We will cover in more detail how to access lists later in the chapter.

The first method that we will consider is pop, this removes the last item of the list.

```
>>> stuff.pop()
9
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign"]
```

We can also specify the index position that we wish to pop from the list as follows:

```
>>> stuff.pop(4)
17
>>> stuff
["longing", "rusted", "furnace", "daybreak", "Benign"]
```

Note that what you return from the pop is the value you are popping, you do not need to assign this back to the list you have changed your list permanently by doing this.

You may ask how useful that is but it leads nicely onto append which allows us to add an element to the end of the list.

```
>>> stuff.append(9)
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
```

There is another way to remove items from a list and that is by using the attribute remove. So let's say we wanted to remove 9 from the list we could use the following:

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff.remove(9)
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign"]
```

This could work on any element of the list, we would just need to specify the name of the item we wanted to remove. It is worth noting that remove does not remove all instances, only the initial instance of that value within the list.

In this case we will put the list back to its old ways by again using the append method.

```
>>> stuff.append(9)
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
```

Next, we will show how to use count, here we pass in something we want to get the count of in the list, to show this we will define a slightly more interesting list:

```
>>> count_list = [1,1,1,2,3,4,4,5,6,9,9,9]
>>> count_list.count(1)
3
>>> count_list.count(4)
2
```

We will now show how to use reverse which not surprisingly reverses the elements in a list:

```
>>> count_list.reverse()
>>> count_list
[9, 9, 9, 6, 5, 4, 4, 3, 2, 1, 1, 1]
```

The last method we will show you is an interesting one, the sort method. Now on a list of integers the way it works is as you would expect the following:

```
>>> count_list
[9, 9, 9, 6, 5, 4, 4, 3, 2, 1, 1, 1]
>>> count_list.sort()
>>> count_list
[1, 1, 1, 2, 3, 4, 4, 5, 6, 9, 9, 9]
```

But what if you did this with strings or a mix of data types, well we can see by using the stuff list defined earlier:

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "<" not supported between instances of "int" and "str"
```

What happened here is Python doesn't know how to support sorting integers and strings, so we can only use the sort method on a list of numeric values.

That will do us in terms of looking at the attributes of a list, we will go back to how we select items from a list. As you will remember, we introduced the concept of indexing and selecting items in the list according to the index. We will take that a step further by showing how negative indexes work. Simply put it works from the end of the list and works backwards, so the last element of the list is -1 and the first is the negative value of the length of the list. Now we can get the length of the list by using the function len as we will demonstrate:

```
>>> stuff = ["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> len(stuff)
7
>>> stuff[-1]
9
>>> stuff[-2]
"Benign"
>>> stuff[-len(stuff)]
"longing"
```

What have we done here? Initially we have shown the list to be what we defined earlier then we have applied len to it and obtained the result 7. What this says is that the number of elements in the list is 7. After that we used negative indexing to choose the last and second last item and showed how we could get the first element using negative indexing.

Now choosing a single item from a list is cool and all but what if we want to select subsets of the list. Well Python allows you to do this as well. Let us say we wanted to choose the second and third elements and have the results in a list. We could do so using the approach we showed so far as follows:

```
>>> x = stuff[1]
>>> y = stuff[2]
>>> new_stuff = [x, y]
>>> new_stuff
["rusted", "furnace"]
```

That is fine but it is pretty ugly, it would be much nicer to do it in a single line, and what do you know you can do exactly that.

```
>>> stuff[1:3]
["rusted", "furnace"]
```

What you are saying is take from the element in index 1 in the list (which is the second element as we are 0 indexed) and show up to

but not including element in index 3 in the list. In a similar way you could select everything except the first element in a similar way:

```
>>> stuff[1:]
["rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
```

Note we have shown the list after taking what we wanted and it shows the full list. This is because splicing the list (which is what we are doing) doesn't change the list it just returns the splice of our list. So if we want to use the splice we need to assign it to a variable as follows:

```
>>> new_stuff = stuff[1:]
>>> new_stuff
["rusted", "furnace", "daybreak", 17, "Benign", 9]
```

Splicing works using negative indexing as well. If we run the above code using -1 instead of 1 we get the following:

```
>>> new_stuff = stuff[-1:]
>>> new_stuff
[9]
```

What we are doing here is staying we want everything from the -1 index to the end of the list. Splicing works if we have the index after

the colon. So again rewriting the previous example with -1 after the colon gives us

```
>>> new_stuff = stuff[:-1]
>>> new_stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign"]
```

So its basically the opposite of what we did before as we take the everything from the start of the list up to the -1 index position.

Let's say we now want to select every second element of the list then we would have to run the following:

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff[1:8:2]
["rusted", "daybreak", "Benign"]
```

So here we are selecting everything between index 1 and 7 incrementing the index by 2 starting at the first position. Its pretty powerful stuff and gives us a lot of control over lists.

The next thing we consider is how to join lists together. Luckily we have covered concatenation earlier with strings and its very much the same for lists, lets demonstrate:

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> count_list
[1, 1, 1, 2, 3, 4, 4, 5, 6, 9, 9, 9]
>>> stuff + count_list
["longing", "rusted", "furnace", "daybreak", 17,
"Benign", 9, 1, 1, 1, 2, 3, 4, 4, 5, 6, 9, 9, 9]
```

By using the addition symbol we can concatenate two or more lists together in the same way we would a string.

If we want to check if an element is in a list we can easily do so by using the Python expression in argument.

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> 9 in stuff
True
```

If the value is in we get back a boolean value True or False.

Similarly, we can use not in to see if a value is not in a list so repeating the previous example we would get the following result:

```
>>> stuff
["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> 9 not in stuff
False
```

Let's now consider the copy method. To demonstrate this we will create a list and then assign it to a new list.

```
>>> stuff = ["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9]
>>> new_stuff = stuff
>>> new_stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9]
>>> stuff.append(21)
>>> stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9, 21]
>>> new_stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9, 21]
```

As we can see when we create the second list new_stuff anything we do to stuff is reflected in new_stuff. If we do not want to do this and want to take a copy of the list where they are independent then we use the copy method. Taking the last example we can repeat using the copy method.

```
>>> stuff = ["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9]
>>> new_stuff = stuff.copy()
>>> new_stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9]
>>> stuff.append(21)
>>> stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9, 21]
>>> new_stuff
['longing', 'rusted', 'furnace', 'daybreak', 17, 'Benign', 9]
```

The next method we will consider in this chapter is the clear method, simply put this method clears the list of all its content.

```
>>> stuff = ["longing", "rusted", "furnace", "daybreak", 17, "Benign", 9]
>>> stuff.clear()
>>> stuff
[]
```

The last method we will consider is not strictly a list method, its not even a list type, but it used to be. Here we will look at a range object. Now in Python 2 you could create a range using the following syntax

```
>>> x = range(7)
>>> x
[0, 1, 2, 3, 4, 5, 6]
>>> type(x)
<type 'list'>
```

Here range created a list of length 7 starting at 0. We can modify this by giving it a start and end point as follows

```
>>> x = range(1,7)
>>> x
[1, 2, 3, 4, 5, 6]
```

So the list starts at 1 and ends at 6, which mimics what we have seen when using the colon syntax to access a list earlier. We can take this a step further by adding the third argument

```
>>> x = range(1,7,2)
>>> x
[1, 3, 5]
```

This gives us a list of every other item starting at 1 ending at 6. We can think of the range method having three arguments start, stop and step where stop is the only necessary argument needed. This

method is very useful for creating on the fly lists, however in Python 3 this changed and range no longer created a list object but instead created a range object. Let's demonstrate this by using the previous example in Python 3.

```
>>> x = range(7)
>>> x
range(0, 7)
>>> type(x)
<class 'range'>
```

We can pass in a start and stop values as before.

```
>>> x = range(1,7)
>>> x
range(1, 7)
```

We can also add the step value.

```
>>> x = range(1,7,2)
>>> x
range(1, 7, 2)
```

We can access the elements of a range object in the same way we can a list by just passing the index value

```
>>> x = range(7)
>>> x[0]
0
>>> x[-1]
6
>>> x[3]
3
```

We can also splice up our range in the same manner as a list and access elements within it.

```
>>> x = range(7)
>>> x[1:]
range(1, 7)
>>> x[:-1]
range(0, 6)
>>> x[:-1][-1]
5
>>> x[3]
3
>>> x[:4:2]
range(0, 4, 2)
```

Unlike with a list we don't have the variety of methods associated with them but we can obtain the start, stop and step of the range object alongside the count and index.

```
>>> x = range(7)
>>> x = x[1:]
>>> x
range(1, 7)
>>> x.start
1
>>> x.stop
7
>>> x.step
1
>>> x.index(1)
0
>>> x.count(1)
1
>>> x = range(7)
>>> x = x[1:5:2]
>>> x
range(1, 5, 2)
>>> x.start
1
>>> x.stop
5
>>> x.step
2
>>> x.index(3)
1
>>> x.count(3)
1
```

Range objects can be very useful for creating on the fly objects containing integers objects and we will use these in some of the examples later on in this book. That covers lists and gives us a good refer-

ence on how to create, access and operate on them, we will use lists throughout the rest of the book.

8
Tuples

The good thing about covering lists is that it then makes tuples much easier to cover. Tuples are essentially lists. You access them in exactly the same way and many things we covered in lists are relevant to tuples, the big difference is tuples can't be modified. Why would you want something that cannot be modified? Well its actually more useful than you would think, it prevents you from accidentally changing something that you may rely on in your code.

Let's take the numbers list we defined at the start of the list section:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can rewrite this as a tuple as follows:

```
>>> numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> numbers
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Looks very similar doesn't it! Now let's try and access the first element of the tuple, again we do it in the same way as we would do in a

list

```
>>> numbers[0]
1
```

And to get the last number we would do

```
>>> numbers[-1]
10
```

We can splice the tuple just like we would a list, so if we wanted everything except the last two values then we do it in exactly the same way we would do in a list

```
>>> numbers[:-2]
(1, 2, 3, 4, 5, 6, 7, 8)
```

So what is the point in tuples? Well let's try and change the second element of the tuple as we would if it were a list:

```
>>> numbers[1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

We cannot assign a new value to a defined tuple, well what can we do to a tuple? Here, we consider two methods for the numbers tuple, namely count and index.

Count simply tells us how many of a value are present in the tuple. To demonstrate it, we will define a new tuple as follows:

```
>>> new_numbers = (1, 2, 2, 2, 5, 5, 7, 9, 9, 10)
>>> new_numbers.count(2)
3
```

So we see that new_numbers contains three instances of the integer value 2.

```
>>> new_numbers = (1, 2, 2, 2, 5, 5, 7, 9, 9, 10)
>>> new_numbers.index(2)
1
```

For the index value we ask it what index the integer value 2 has in the tuple and here you can see it returns 1 which is correct, however we also have 2 at index values 2 and 3 so care needs to be taken with this method.

In this chapter we have introduced the brief but important concept of a tuple. Having an object which you cannot modify is very useful and is used throughout various Python packages.

9
Dictionaries

When we say dictionary in Python, we aren't speaking of the Oxford variety. Dictionaries are used to contain data in a very specific way as they hold key value pairs. What do we mean by key value pairs? For example, we could have a key as first name and a value as Rob. We could also have a key as wins and the value as 21. The key to dictionaries (pun intended) is that to access values we need to have a key associated with it. Let us start with an example about personal details, we may have the following fields: first name, surname, gender, favourite food. This could be written in the form of a dictionary as follows:

```
>>> dict_detail = {}
>>> dict_detail["first name"] = "Rob"
>>> dict_detail["surname"] = "Mastrodomenico"
>>> dict_detail["gender"] = "Male"
>>> dict_detail["favourite food"] = "Pizza"
>>> dict_detail
{"first name": "Rob", "favourite food": "Pizza",
 "surname": "Mastrodomenico", "gender": "Male"}
>>>
```

Its important to note that the name of the dictionary is arbitrary, I used dict as I am not very creative, I could easily have written it as follows:

```
>>> ham_sandwich = {}
>>> ham_sandwich["first name"] = "Rob"
>>> ham_sandwich["surname"] = "Mastrodomenico"
>>> ham_sandwich["gender"] = "Male"
>>> ham_sandwich["favourite food"] = "Pizza"
>>> ham_sandwich
{"first name": "Rob", "favourite food": "Pizza",
 "surname": "Mastrodomenico", "gender": "Male"}
>>>
```

You could argue that the second one is funnier and I would agree, however the name of your dictionary or variable should reflect what you are trying to do as someone will no doubt have to read your code. The problem with us using dict is that dict is a builtin function with Python that means it actually does something. Now, there is nothing to prevent you from using this, however if you override the builtin function dict with your dictionary dict, then you cannot use that within the current environment.

```
>>> person_details = dict(first_name="Rob", surname="Mastrodomenico",
gender="Male", favourite_food="Pizza")
>>> person_details
{"gender": "Male", "first_name": "Rob", "surname": "Mastrodomenico",
"favourite_food": "Pizza"}
```

For you eagle eyed viewers you can see that I renamed favourite food to favourite_food. The reason being is that Python interprets the space between the two words as two separate entries and throws an

error. So in the remainder of this section we will refer to favourite food as favourite_food.

So recreating the person_details dictionary again we get the following:

```
>>> personal_details = {}
>>> personal_details["first name"] = "Rob"
>>> personal_details["surname"] = "Mastrodomenico"
>>> personal_details["gender"] = "Male"
>>> personal_details["favourite_food"] = "Pizza"
>>> personal_details
{"first name": "Rob", "favourite_food": "Pizza", "surname": "Mastrodomenico",
"gender": "Male"}
```

What we did here in the first line was create the dictionary using:

```
>>> personal_details = {}
```

In the subsequent lines we then assign key value pairs to the dictionary. We could have done this in another way though.

```
>>> personal_details = {"first name": "Rob", "surname": "Mastrodomenico",
"gender": "Male", "favourite_food": "Pizza"}
>>> personal_details
{"first name": "Rob", "favourite_food": "Pizza", "surname": "Mastrodomenico",
"gender": "Male"}
```

Similarly we could have used the dict method with the above dictionary setup to achieve the same outcome.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{"first name": "Rob", "gender": "Male", "surname": "Mastrodomenico",
"favourite_food": "Pizza"}
```

Earlier we talked about error handling and we will briefly cover it now. Let's say we try to access a key in dictionary that doesn't exist.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{"first name": "Rob", "surname": "Mastrodomenico", "favourite_food": "Pizza",
"gender": "Male"}
>>> personal_details["age"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "age"
```

Let's say we wanted to allow us to try and access a key that doesn't exist and instead of throwing us an error we want to deal with it in a more sophisticated way. To do this we can handle the exception (or error) that Python shows back. Now looking back at the error we can see very specifically that Python throws us a KeyError. Now if we can handle that then we can do something a bit better with it.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{"first name": "Rob", "surname": "Mastrodomenico", "favourite_food": "Pizza",
"gender": "Male"}
>>> try:
...     age = personal_details["age"]
... except KeyError:
...     age = None
...
>>> age
>>>
```

What we have done here is we used a try except statement. What this does is try the code we want to run and if doesn't work, then it deals with the error. In this case, it tries to access the key age and assigns it to the variable age and if an error is thrown and it is a KeyError, then it assigns the value to None, which is the Python null value. So, what we see now is that as opposed to throwing an error, it deals with the fact that the key can't be found in a much better way. Whilst try excepts method can be really useful, they can be very dangerous as you don't need to specify the error so we could rewrite it as follows:

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{"first name": "Rob", "surname": "Mastrodomenico", "favourite_food": "Pizza",
 "gender": "Male"}
>>> try:
...     age = personal_details["age"]
... except:
...     age = None
...
>>> age
>>>
```

Try excepts are useful however it would be better if we had a method that let us try and get a key and if it didn't exist not throw an error. Fortunately enough we have a get method that we can use to get the value from a dictionary by passing a key.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details.get('gender')
'Male'
>>> personal_details.get('age')
```

When we covered lists we looked at the pop method and we have something similar for dictionaries

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details.pop('gender')
'Male'
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'favourite_food': 'Pizza'}
>>> personal_details.popitem()
('favourite_food', 'Pizza')
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico'}
```

In the first example we used pop with the key that we wanted to remove and in doing so this removed the specific key and value from the dictionary. The second approach used the popitem method which just pops off the last key value pair in the dictionary.

Another way to remove items is to use the del method where we pass the dictionary name and key combination to remove that key value pair from the dictionary.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> del personal_details['gender']
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'favourite_food': 'Pizza'}
```

Earlier we mentioned how if we assign one list to another the changes are reflected. The same is true for dictionaries

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> his_details = personal_details
>>> his_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details['age'] = 24
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza', 'age': 24}
>>> his_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza', 'age': 24}
```

If we want to take a copy of a dictionary and independently make changes to it we can use the copy method in a similar way that we did with lists.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> his_details = personal_details.copy()
>>> his_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details['age'] = 24
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza', 'age': 24}
>>> his_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
```

We also have the ability to clear out all contents of dictionary using the clear method

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details.clear()
>>> personal_details
{}
```

Earlier in the chapter we looked at approaches to create dictionaries however if we want to create a set of keys with the same value we can do using the fromkeys method.

```
>>> x = ('key1', 'key2', 'key3')
>>> y = 0
>>> res = dict.fromkeys(x, y)
>>> res
{'key1': 0, 'key2': 0, 'key3': 0}
```

We have thus far accessed values from dictionaries using the keys, however we can access all keys and values from a dictionary using the following methods

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details.items()
dict_items([('first name', 'Rob'), ('surname', 'Mastrodomenico'), ('gender', 'Male'),
 ('favourite_food', 'Pizza')])
>>> personal_details.keys()
dict_keys(['first name', 'surname', 'gender', 'favourite_food'])
>>> personal_details.values()
dict_values(['Rob', 'Mastrodomenico', 'Male', 'Pizza'])
```

The objects that we return can be iterated over and this is covered later when we introduce loops however if you want to access them like we would a list we can cast them as such and access the relevant index positions.

```
>>> personal_details = dict([("first name", "Rob"), ("surname", "Mastrodomenico"),
("gender", "Male"), ("favourite_food", "Pizza")])
>>> personal_details
{'first name': 'Rob', 'surname': 'Mastrodomenico', 'gender': 'Male', 'favourite_food':
 'Pizza'}
>>> personal_details.items()
dict_items([('first name', 'Rob'), ('surname', 'Mastrodomenico'), ('gender', 'Male'),
 ('favourite_food', 'Pizza')])
>>> list(personal_details.items())[0]
('first name', 'Rob')
>>> personal_details.keys()
dict_keys(['first name', 'surname', 'gender', 'favourite_food'])
>>> list(personal_details.keys())[-1]
'favourite_food'
>>> personal_details.values()
dict_values(['Rob', 'Mastrodomenico', 'Male', 'Pizza'])
>>> list(personal_details.values())[:-1]
['Rob', 'Mastrodomenico', 'Male']
```

That gives summary of dictionary methods and what you can do with them, compared to lists and tuples these types of objects can be very powerful and flexible container. We will return to these throughout the book as they form a key concept in Python.

10

Sets

In this chapter, we cover the last object for data collection and that is set. Perhaps the best use for sets is that they cannot contain duplicate values so can be useful for storing unique values. They are also unordered and cannot be changed. Let's start by creating a set.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Peter', 'Tony', 'Wanda', 'Natasha'}
```

Here, we use the curly brackets as we did with a dictionary, however now we have no key value pairs and the content resembles that of a list of tuple. We are not constrained by having strings in it, let's add some integers to our set.

```
>>> names = {'Tony','Peter','Natasha','Wanda', 1, 2, 3}
>>> names
{1, 2, 3, 'Wanda', 'Peter', 'Tony', 'Natasha'}
```

Here, we can see that the ordering of the set doesn't resemble what we put into it.

We can also create a set using the set builtin function as covered earlier.

```
>>> names = set(('Tony','Peter','Natasha','Wanda', 1, 2, 3))
>>> names
{1, 2, 3, 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names = set(['Tony','Peter','Natasha','Wanda', 1, 2, 3])
>>> names
{1, 2, 3, 'Natasha', 'Peter', 'Tony', 'Wanda'}
```

We can create a set from a string but we need to be aware of how the curly brackets and set builtin work

```
>>> names = {'Wanda'}
>>> names
{'Wanda'}
>>> names = set('Wanda')
>>> names
{'n', 'a', 'W', 'd'}
```

What happens is that when you pass in a string using the curly brackets you retain the full string in but when passed in using set the string is split into the individual characters. Again note when the characters are split there is no ordering to them.

Next, we try and add a list to the set but we raise a type error as the list cannot be added.

```
>>> my_set = {'Tony','Wanda', 1, 2, ['hello','world']}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

The same is true for dictionaries and sets as we show below:

```
>>> my_set = {'Tony','Wanda', 1, 2, {'key':'value'}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
>>> my_set = {'Tony','Wanda', 1, 2, {1,2,3}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

However, if we include a tuple in the set we get the following:

```
>>> my_set = {'Tony','Wanda', 1, 2, (1,2,3)}
>>> my_set
{'Wanda', 1, 2, 'Tony', (1, 2, 3)}
```

The reason we can include the tuple over the dictionary, list and set is that the tuple cannot be changed so is supported in a set.

We can see if the value is in the set by using the following syntax:

```
>>> 'Tony' in names
True
>>> 'Steve' in names
False
```

An item could be added to the set by using the add method:

```
>>> names.add('Steve')
>>> names
{1, 2, 3, 'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names.add('Tony')
>>> names
{1, 2, 3, 'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
```

Note here when we add in Tony again to the set we don't get duplicate values of Tony in the set, but Steve gets added in as it was not present. That aspect of not having duplicate values within the set is useful if we want to have a unique representation of values where we could have duplicates. For example, you could imagine a long list with lots of repeated values and you just want the unique values within it as we show below:

```
>>> days = ['Monday', 'Monday', 'Tuesday', 'Wednesday',
            'Sunday', 'Sunday']
>>> days_set = set(days)
>>> days_set
{'Monday', 'Wednesday', 'Sunday', 'Tuesday'}
```

Now, the example is fairly trivial as we can see all content within the list but you can imagine examples within a big dataset where you may want a similar representation.

This is useful for a single set but we also have the ability to operate on multiple sets. The next example looks to obtain the unique value

between two sets.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names | more_names
{'Wanda', 'Peter', 'Tony', 'Carol', 'Steve', 'Natasha'}
```

Here, we use the | operator and get the values that are in the names set OR the more_names set so any shared values are included only once. The same can be achieved using the set method union.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names.union(more_names)
{'Wanda', 'Peter', 'Tony', 'Carol', 'Steve', 'Natasha'}
```

Now, while the results are the same we didn't need to pass a set in the union method, we could rewrite passing a list into the union and get the same results.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> names.union(more_names)
{'Wanda', 'Peter', 'Tony', 'Carol', 'Steve', 'Natasha'}
```

So, we can achieve the same result as the list is converted into a set.

If we used the | operator with the list, then we see an error.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> names | more_names
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

We can extend the union example by passing more items into the union method, for example, if we have two sets that we want to take the union with our existing set we would do as follows:

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Steve', 'Peter', 'Carol', 'Wanda'}
>>> even_more_names = {'Tony', 'Jonny', 'Sue', 'Wade'}
>>> even_more_names
{'Tony', 'Jonny', 'Sue', 'Wade'}
>>> names | more_names | even_more_names
{'Wanda', 'Sue', 'Wade', 'Carol', 'Peter', 'Tony', 'Jonny', 'Steve', 'Natasha'}
>>> names.union(more_names, even_more_names)
{'Wanda', 'Sue', 'Wade', 'Peter', 'Tony', 'Carol', 'Jonny', 'Steve', 'Natasha'}
```

Now, if the values were lists and not sets, then we can still use the union method as shown below:

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> even_more_names = ['Tony', 'Jonny', 'Sue', 'Wade']
>>> even_more_names
['Tony', 'Jonny', 'Sue', 'Wade']
>>> names.union(more_names, even_more_names)
{'Wanda', 'Sue', 'Wade', 'Peter', 'Tony', 'Carol', 'Jonny', 'Steve', 'Natasha'}
```

These examples so far have looked the union between two or more sets, what if we want to look at values that are in all sets, or not in any sets, luckily Python has us covered.

Now where we used union and the | operator if we want to find out what values are in all sets we use the intersection method or the & operator.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names & more_names
{'Wanda', 'Peter'}
>>> names.intersection(more_names)
{'Wanda', 'Peter'}
```

As we saw with the union method and │ operator, we can include more than two sets.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> even_more_names = {'Peter', 'Jonny', 'Sue', 'Wade'}
>>> even_more_names
['Peter, 'Jonny', 'Sue', 'Wade']
>>> names & more_names & even_more_names
{'Peter'}
>>> names.intersection(more_names, even_more_names)
{'Peter'}
```

And similarly as shown with the union method we can add non-sets into the intersection method.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> even_more_names = ['Peter', 'Jonny', 'Sue', 'Wade']
>>> even_more_names
['Peter, 'Jonny', 'Sue', 'Wade']
>>> names.intersection(more_names, even_more_names)
{'Peter'}
```

If we want to look at the differences between two or more sets, then we can use the difference method or the − operator. Again the rules that we have seen before are consistent here in that we can only do comparison using the − operator on sets whereas for the difference method we can handle non-sets passed in.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names - more_names
{'Natasha', 'Tony'}
>>> names.difference(more_names)
{'Natasha', 'Tony'}
>>> even_more_names = {'Peter', 'Jonny', 'Sue', 'Wade'}
>>> even_more_names
{'Peter, 'Jonny', 'Sue', 'Wade'}
>>> names - more_names - even_more_names
{'Natasha', 'Tony'}
>>> names.difference(more_names, even_more_names)
{'Natasha', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Steve', 'Peter', 'Carol', 'Wanda']
>>> even_more_names = ['Peter', 'Jonny', 'Sue', 'Wade']
>>> even_more_names
['Peter, 'Jonny', 'Sue', 'Wade']
>>> names.difference(more_names, even_more_names)
{'Natasha', 'Tony'}
```

The manner in which difference is applied for more than one comparison is to work left to right so we first look at the difference between names and more_names and then look at the difference between this result and even_more_names.

Another set comparison that we can perform is using the symmetric_difference method or the ^ operator. What this does is return back the elements that are in either set but not in both, so its like the or method but doesn't include any common values. We demonstrate as follows:

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names ^ more_names
{'Carol', 'Tony', 'Steve', 'Natasha'}
>>> names.symmetric_difference(more_names)
{'Carol', 'Tony', 'Steve', 'Natasha'}
>>> even_more_names = {'Peter', 'Jonny', 'Sue', 'Wade'}
>>> even_more_names
{'Peter, 'Jonny', 'Sue', 'Wade'}
>>> names ^ more_names ^ even_more_names
{'Sue', 'Wade', 'Carol', 'Peter', 'Jonny', 'Tony', 'Steve', 'Natasha'}
>>> names.symmetric_difference(more_names, even_more_names)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: symmetric_difference() takes exactly one argument (2 given)
```

Unlike with previous methods symmetric_difference doesn't allow more than one set, however it still allows us to pass in a non-set as shown below:

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = ['Steve', 'Peter', 'Carol', 'Wanda']
>>> more_names
['Wanda', 'Peter', 'Steve', 'Carol']
>>> names.symmetric_difference(more_names)
{'Carol', 'Tony', 'Steve', 'Natasha'}
```

We can also see if a set has any elements in common by using the isdisjoint method.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Bruce', 'Carol', 'Wanda'}
>>> more_names
{'Wanda', 'Peter', 'Steve', 'Carol'}
>>> names.isdisjoint(more_names)
False
>>> more_names = {'Steve', 'Bruce', 'Carol', 'Sue'}
>>> more_names
{'Sue', 'Bruce', 'Steve', 'Carol'}
>>> names.isdisjoint(more_names)
True
```

For any readers interested in set theory, there are a couple of other set methods:

- issubset
- issuperset

but we will leave it up to the reader to discover the wonders of them.

Now aside from these methods which have allowed us to compare sets there are a number of other methods that can be applied to sets, some of which we have covered in the previous chapters so we will briefly demonstrate them.

```
>>> names = {'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Natasha'}
>>> names.pop()
'Wanda'
```

As before pop removes an item from the set.

We can also remove an item from the set explicitly using the remove method:

```
>>> names = {'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Natasha'}
>>> names.remove('Tony')
>>> names
{'Wanda', 'Peter', 'Steve', 'Natasha'}
```

Now the problem with the above is that if we try and remove something from the set that isn't in there, then we get a key error. Another way to remove something from the set that allows for a value not to be in the set is the discard method which is demonstrated below:

```
>>> names = {'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Natasha'}
>>> names.remove('Sue')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Sue'
>>> names.discard('Sue')
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Natasha'}
>>> names.discard('Peter')
>>> names
{'Wanda', 'Tony', 'Steve', 'Natasha'}
```

We can also clear the set by using the clear method which gives us an empty set.

```
>>> names = {'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names.clear()
>>> names
set()
```

Given we can remove items from a set similar to how we can add to it, the first method we cover that does this is the add method:

```
>>> names = {'Steve', 'Wanda', 'Peter', 'Tony', 'Natasha'}
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Natasha'}
>>> names.add('Bruce')
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Bruce', 'Natasha'}
>>> names.add('Peter')
>>> names
{'Wanda', 'Peter', 'Tony', 'Steve', 'Bruce', 'Natasha'}
```

Now the add method adds values one at a time but we can utilise some methods which are similar in nature to those that looked at comparison on the sets and allow us to modify a set, the first of these we will show is the update method:

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Steve', 'Peter', 'Carol', 'Wanda'}
>>> names | more_names
{'Carol', 'Peter', 'Steve', 'Natasha', 'Wanda', 'Tony'}
>>> names
{'Tony', 'Peter', 'Wanda', 'Natasha'}
>>> more_names
{'Carol', 'Peter', 'Steve', 'Wanda'}
>>> names.update(more_names)
>>> names
{'Carol', 'Peter', 'Steve', 'Natasha', 'Wanda', 'Tony'}
```

You can see the return value using the | operator applied to two sets is the same as the update value. The big difference here is that when you use the | operator you don't change either of the sets, however using the update method changes the set that you have used the method for so in this case the names set is now the result of names | more_names.

```
>>> names = {'Tony','Peter','Natasha','Wanda'}
>>> names
{'Wanda', 'Natasha', 'Peter', 'Tony'}
>>> more_names = {'Steve', 'Peter', 'Carol', 'Wanda'}
>>> more_names
{'Steve', 'Peter', 'Carol', 'Wanda'}
>>> names & more_names
{'Peter', 'Wanda'}
>>> names
{'Tony', 'Peter', 'Wanda', 'Natasha'}
>>> more_names
{'Carol', 'Peter', 'Steve', 'Wanda'}
>>> names.intersection_update(more_names)
>>> names
{'Peter', 'Wanda'}
```

Like with the update method the intersection_update applies the &
operation but assigns the result back to the set that its applied to. The
same is true for the symmetric_difference:update which gives the
same result as ∧ and difference:update which gives the difference be-
tween two sets.

The last concept that we look at in this chapter is the concept of the
frozen set. The frozen set is what the tuple is to a list in that it cannot
be altered:

```
>>> frozen_names = frozenset({'Tony','Peter','Natasha','Wanda'})
>>> frozen_names
frozenset({'Tony', 'Peter', 'Wanda', 'Natasha'})
>>> frozen_names = frozenset(['Tony','Peter','Natasha','Wanda'])
>>> frozen_names
frozenset({'Tony', 'Peter', 'Wanda', 'Natasha'})
>>> frozen_names = frozenset(('Tony','Peter','Natasha','Wanda'))
>>> frozen_names
frozenset({'Tony', 'Peter', 'Wanda', 'Natasha'})
>>> frozen_names = frozenset('Tony')
>>> frozen_names
frozenset({'T', 'y', 'n', 'o'})
>>> dir(frozen_names)
['__and__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__',
'__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__xor__', 'copy', 'difference', 'intersection', 'isdisjoint', 'issubset',
'issuperset', 'symmetric_difference', 'union']
```

We use frozenset around a set, list, tuple, or string to give us our set but looking at the methods available you can see that the ones which change the set are not available to a frozen set, however the methods that allow us to compare sets are still available. This concludes this chapter and gives us an introduction into sets and how they can be used within Python.

11
Loops, if, Else, and While

This chapter is now going to cover the following important aspects of programming namely loops, if, else, and while statements. These statements are the key components of programming and allow us to operate on our objects. We have covered aspects of logic earlier when we considered comparison and equality and that is key when we look at if statements which is the first aspect we will consider here. The key to an if statement is that we need a statement that returns a true or false value. So let's consider the following statement:

```
>>> x = 1
>>> x == 1
True
```

Now, the first line is an assignment and the second is comparison and its with the second line that we can introduce an if statement:

```
>>> x == 1
True
>>> if x == 1:
...        x = x + 1
...
>>> x
2
```

What we have done here is test the return value of the statement and if its true then we increment the variable x by 1. That is the essence of an if statement, we test to see if the return of a logic statement is true and if it is we do something within that statement. We can extend the example by introducing the concept of an else statement. Essentially given the if statement tests if a statement is true the else deals with if the statement is false.

```
>>> x = 2
>>> x == 1
False
>>> if x == 1:
...         x = x + 1
... else:
...         x = x - 1
...
>>> x
1
```

Now what we have done is assign x the variable 2 and then test if x is equal to 1 which returns False. We then set an if else statement where if the value is equal to 1 we add one to x and if not we subtract one from the value x. As we can see the value isn't equal to one so we enter the else aspect of the loop and decrement the value of x by 1.

Logic like this is really useful to allow us to make decisions in our code based on what a variable may or may not be. However, the if

else loop only allows us to make a decision based on a single true or false condition. If we require more control, then we can use the elif statement alongside an if else statement. Let's improve on our example from before and introduce the concept of elif.

```
>>> x = 2
>>> x == 1
False
>>> x == 2
True
>>> if x == 1:
...        x = x + 1
... elif x == 2:
...        x = x * x
... else:
...        x = x - 1
...
>>> x
4
```

Here, we have set the variable x equal to 2 and then show the logic statements that we plan to use. In the first if statement we test if x is equal to 1, it isn't so we then enter the elif statement and see if x is equal to 2, it is so we then run the command in the statement and x is shown to be 2. We can expand the statement further with more elif statements so we can check for a variety of things. It is important to note that if one of the conditions on the if or elif is met we then exit the statement and continue through our code. So as soon as a condition

is seen we exit the statement. Therefore, when thinking about using if statements you need to understand the context you are using them in.

If we have a fixed set of outcomes that we want to work with then an if elif statement like we showed before would work well. We next show an example where using an if elif statement doesn't quite work:

```
>>> home_score = 4
>>> away_score = 0
>>> if home_score > away_score:
...        result = "Home win"
... elif home_score < away_score:
...        result = "Away Win"
... elif home_score == away_score:
...        result = "Draw"
... elif home_score > (away_score + 1):
...        result = "Home win by more than 1 goal"
... elif (home_score + 1) < away_score:
...        result = "Away win by more than 1 goal"
... else:
...        result = "Unknown result"
...
>>> result
"Home win"
```

On the face of this you may think that this is a really good bit of code, however it isn't. The problem is we want to test if the win for either side is a win or a win by more than 1 goal. What happens here is the

condition for a home win is met before we test for the win by more than one (which would also return a true result). So using an if statement as we do here doesn't work, instead we need to use a nested if statement.

```
>>> home_score = 4
>>> away_score = 0
>>> if home_score > away_score:
...         if home_score > (away_score + 1):
...                 result = "Home win by more than 1 goal"
...         else:
...                 result = "Home win"
... elif home_score < away_score:
...         if (home_score + 1) < away_score:
...                 result = "Away win by more than 1 goal"
...         else:
...                 result = "Away win"
... elif home_score == away_score:
...         result = "Draw"
... else:
...         result = "Unknown result"
...
>>> result
"Home win by more than 1 goal"
```

We can see that we get the desired result. The point that this raises is that if statements can be very powerful but at the same time mistakes can be made if they are not thoroughly thought out. We next look to consider loops.

In earlier sections we have covered list (Chapter 7), tuples (Chapter 8), and dictionaries (Chapter 9) which are all important containers for values in Python. We showed how we can access and manipulate them. However, we concentrated on single instances of these objects whereas in reality this isn't the case. What you might expect is to have a collection of data which each may be contained in a list, tuple, or dictionary and you want to access it and perform some kind of operation on it. Let's create some data that allows us to do so.

```
>>> people = []
>>> person = ["Tony", "Stark",48]
>>> people.append(person)
>>> person = ["Steve","Rodgers",102]
>>> people.append(person)
>>> person = ["Stephen", "Strange",42]
>>> people.append(person)
>>> person = ["Natasha","Romanof",36]
>>> people.append(person)
>>> person = ["Peter","Parker",16]
>>> people.append(person)
>>> people
[["Tony", "Stark", 48], ["Steve", "Rodgers", 102],
   ["Stephen", "Strange", 42],
["Natasha", "Romanof", 36], ["Peter", "Parker", 16]]
```

What we have done here is setup a list of lists. Now if we want to access element 3 of the first list we can do so as follows:

```
>>> people
[["Tony", "Stark", 48], ["Steve", "Rodgers", 102],
    ["Stephen", "Strange", 42],
["Natasha", "Romanof", 36], ["Peter", "Parker", 16]]
>>> people[0][2]
45
```

So we access the first element of the outer list which returns us a list and then we access the value 3 of that list through the index 2. Now if we wanted to see everyones age we could write the following:

```
>>> people[0][2]
48
>>> people[1][2]
102
>>> people[2][2]
42
>>> people[3][2]
36
>>> people[4][2]
16
```

This is fairly tedious for five people, imagine if we had hundreds or thousands of people. To avoid writing the same code again and again we can use a loop to access the ages of everyone in the list. Now to make it more interesting let's work out the average age of the people in the list.

```
>>> total_age = 0.0
>>> for p in people:
...         age = p[2]
...         total_age += age
...
>>> total_age
244.0
>>> average_age = total_age / len(people)
>>> average_age
48.8
```

What we have done here is introduce the concept of a loop, along-side a few other things. Now, we initially setup a variable to contain the total age and set this to zero. The theory here is that we will incre-ment the value by everyones age within the loop. Next, we enter the for loop. The syntax says for p in person, and what this means is that we iterate over all the elements in people and assign whatever it is to the variable p. The name p is totally arbitrary, we could rewrite the loop as follows:

```
>>> total_age = 0.0
>>> for huge_massive_robot in people:
...         age = huge_massive_robot[2]
...         total_age += age
```

The name isn't technically import, although this is pretty bad naming convention. The point is that when you write your loops you don't al-ways have to use p. The loop then one by one access all the ele-ments of the list people, where each element is a list and the indented

code within the loop does something to the that element. What we do within the loop is assign the third value of the list to the variable age and then add that value to the total age variable which we initially set to be zero. This bit of code is shorthand for the following:

```
>>> age = people[0][2]
>>> total_age += age
>>> age = people[1][2]
>>> total_age += age
>>> age = people[2][2]
>>> total_age += age
>>> age = people[3][2]
>>> total_age += age
>>> age = people[4][2]
>>> total_age += age
>>> total_age
244.0
```

So we can see its a big reduction in amount of code written, which is a good thing. After that we divide the total age by the number of elements in the list which we get by using len. This then gives us the average for the list of lists, pretty cool.

That is the basics of for loops applied to lists, the key thing to remember is that you are iterating through the list, essentially going over every element of the list. Now as we discussed before the elements of the lists don't always have to be of the same type so care is needed when using loops as the logic applied to one element may not hold

for all elements of the loop. Everything we have shown here for lists also apply to tuples so you could write the same type of loop on a tuple, however you can't set it up in the way we did earlier as that won't work.

One neat thing we can do with lists and loops is something known as list comprehension. Now if the problem we had is that we wanted to create a list with the squared ages of our people then we could write a loop as follows:

```
>>> squared_ages = []
>>> for p in people:
...         squared_age = p[2] * p[2]
...         squared_ages.append(squared_age)
...
>>> squared_ages
[2304, 10404, 1764, 1089, 256]
```

That all looks fine, but we could write it in one line as follows:

```
>>> squared_ages = [p[2] * p[2] for p in people]
>>> squared_ages
[2304, 10404, 1764, 1089, 256]
```

Looks cool doesn't it! Which one is better, well they both do the same, some may say using list comprehension is more Pythonic but you could argue its harder to read than the standard for loop example. It is really down to personal preference and what you like doing more.

Next, we will look at how to loop over dictionaries which behave differently to lists and tuples. Now if we have a list of dictionaries we can loop across the list and access the dictionary similarly to how we accessed the list within a list in the previous example. However, if we just have a dictionary we can loop over it in the following way:

```
>>> person = {"first_name":"Steve", "last_name":"Rodgers", "age":102}
>>> person_list = []
>>> for p in person:
...     person_list.append(p)
...
>>> person_list
["first_name", "last_name", "age"]
```

The way the loop behaves in this case is very different to what we have seen before with lists. The p in person is key of the dictionary not the key and value of the dictionary, so in putting the p in a list we get the keys of the dictionary. If we want the values from the person dictionary we need to do it in the following way:

```
>>> person = {"first_name":"Steve", "last_name":"Rodgers", "age":102}
>>> person_list = []
>>> for p in person:
...     value = person[p]
...     person_list.append(value)
...
>>> person_list
["Steve", "Rodgers", 102]
```

The only difference here is that we access the value from the dictionary that uses the key obtained from the loop.

Next, we consider how loops work on strings, which is again different to what we have seen before. We can loop over a string in the same way that we would a list.

```
>>> name = "Rob Mastrodomenico"
>>> name_list = []
>>> for n in name:
...       name_list.append(n)
...
...
>>> name_list
["R", "o", "b", " ", "M", "a", "s", "t", "r", "o", "d", "o",
"m", "e", "n", "i", "c", "o"]
```

So, when we loop over a list we access each individual element of the string, so appending it to the list we see an entry for each letter in the name.

The last concept we will consider in this section is while loops, these are loops that continue whilst some logic is true. We will demonstrate as follows:

```
>>> score = 0
>>> while score < 4:
...       score
...       score += 1
...
0
1
2
3
```

What have we done here? Well initially we set a variable called score and set it equal to zero. Then we have said while score is less than four complete the code within the while loop. As for other logic considered here we need to use the colon after we have set the condition. What we see is that the output from this is that we show the values of score while score is less than four. Once it is completed we then leave the loop. Its much like a for loop, however while a for loop will work for loop will iterate over something a while loop doesn't and just continues until a condition is met which allows it to leave the loop. With a loop like this you need to be careful that the condition is met else it can stay in the loop forever!

Let's put all this together into a single example where we want to simulate a lottery draw. For those not familiar with lotteries we have a fixed number of balls that we randomly select to give a set of numbers. You win the lottery if you have the numbers selected. Now the example we will show will involve us generating six balls and a bonus ball. To generate the random balls we will use the following code:

```
>>> from random import randint
>>> ball = randint(min, max)
```

In this example we will use the min and max numbers to be 1 and 59, now using randint we can generate a random number between 1 and 59 inclusive.

```
>>> from random import randint
>>> min = 1
>>> max = 59
>>> ball = randint(min, max)
>>> ball
15
```

The key here is that we can generate a random integer but its not always unique, meaning we could get the same number out again so we need to have a way to ensure we don't get the same number out again. Now, there is not always one way to do something in Python so we will go through two different approaches to doing this, the first one is as follows:

```
>>> from random import randint
>>> min = 1
>>> max = 59
>>> result_list = []
>>> for i in range(7):
...         ball = randint(min, max)
...         while ball in result_list:
...                 ball = randint(min, max)
...         result_list.append(ball)
...
>>> result_list
[15, 19, 34, 12, 20, 31, 36]
```

Here, we have created a list to store the values that we want to append the results to. We then loop over a range object:

```
>>> range(7)
range(0, 7)
```

This essentially creates a range object that we could loop over and we have 7 as the value due to the fact we want to generate 7 numbers. We then generate a ball randomly and using a while loop if the ball isn't in the list we append it to the result list and this then gives us our 7 random numbers to simulate a lottery draw including the bonus ball.

We could however do this slightly differently and not need the range object as follows:

```
>>> from random import randint
>>> min = 1
>>> max = 59
>>> result_list = []
>>> while len(result_list)<7:
...         ball = randint(min, max)
...         if ball not in result_list:
...                 result_list.append(ball)
...
>>> result_list
[41, 48, 47, 55, 18, 15, 43]
```

So here we use the while loop to continue until the result_list has all the number of balls in so we can reduce the number of lines of code to achieve the same thing.

As you can see we have introduced the concepts of for, if, else, and while and shown how we can apply these to solve a problems and link them back to the concepts that we have covered so far.

12
Strings

Python is good at many things, I may be biased but I believe it's true, however one thing that its particularly good at is dealing with strings and string manipulation. This chapter will expand what we already know about strings showing how powerful Python is at dealing with them. So let's begin with looking at what a string is:

```
>>> name = "rob"
>>> name
"rob"
>>> name = "rob"
>>> name
"rob"
>>> name = """rob"""
>>> name
"rob"
>>> name = '''rob'''
>>> name
"rob"
```

In the above example it doesn't matter which way you create the string and it is all the same but there are a few examples where you need to create a string differently. This primarily revolves around the

use of single and double quotes. Now, if you put a single quote in a string separated by single quotes you get the following:

```
>>> single_quote_string = 'string with a ' single quote'
  File "<stdin>", line 1
    single_quote_string = 'string with a ' single quote'
                                            ^
SyntaxError: invalid syntax
```

To include a single quote in a string separated by single quotes you need to use an escape sequence. An escape sequence is basically the character with a backslash before it, so if we write the previous string with an escape sequence it looks like the following:

```
>>> single_quote_string = 'string with a \' single quote'
>>> single_quote_string
"string with a ' single quote"
```

You might say why use single quotes around a string with single quotes and you would be right, we could write the previous example as follows with double quotes:

```
>>> single_quote_string = 'string with a \' single quote'
>>> single_quote_string
"string with a ' single quote"
```

We can get away with not using escape sequences if we use triple quotes. So we can rewrite the previous using triple quotes as follows:

```
>>> single_quote_string = """string with a ' single quote"""
>>> single_quote_string
"string with a ' single quote"
```

The same would hold for a double quote, however if we have the double quote at the end of the string we encounter a problem.

```
>>> double_quote_string = """string with a double quote on the end""""
  File "<stdin>", line 1
    double_quote_string = """string with a double quote on the end""""
                                                                      ^
SyntaxError: EOL while scanning string literal
```

To get around this problem you would need to use an escape sequence on the double quote as shown below:

```
>>> double_quote_string = """string with a double quote on the end\""""
>>> double_quote_string
'string with a double quote on the end"'
```

Another benefit to triple quoted strings is that you can write strings over multiple lines.

```
>>> on_two_lines = """A quote on
...                             two lines"""
>>> on_two_lines
'A quote on \n\t\t\t\ttwo lines'
```

The string contains the line return denoted by backlash n and three tabs denoted by backslash t. If we were to try this with single quotation then we end up with the following:

```
>>> on_two_lines = "A quote on
  File "<stdin>", line 1
    on_two_lines = "A quote on
                              ^
SyntaxError: EOL while scanning string literal
```

We have touched briefly on escape sequences and although we have not covered all of them (you can search them out yourselves) an interesting issue comes up and that is what do we do if we use an escape sequence in our string. So, for example, typing backslash n gives us a carriage return in our string. But what if we want forward slash n in our string, we can use what is called a raw string:

```
>>> raw_string = r"This has a \n in it"
>>> raw_string
"This has a \\n in it"
```

This is the same as writing

```
>>> not_raw_string = "This has a \\n in it"
>>> not_raw_string
"This has a \\n in it"
```

In both examples when we show the content of the string it has an extra forward slash but when it is printed this disappears. And yes if you wanted an extra slash in there just use three slashes.

```
>>> not_raw_string = "This has a \\\n in it"
>>> not_raw_string
"This has a \\\n in it"
```

So given you have a string what can you do with it? You can access it in much the same way as you did when we covered lists earlier on. So if we want the 3rd element of a string we do so as follows:

```
>>> name = "Rob Mastrodomenico"
>>> name[2]
"b"
```

Remember from before the value in position 3 is indexed at 2 as strings like lists are 0 indexed. We can similarly take the values in positions 5–8.

```
>>> name = "Rob Mastrodomenico"
>>> name[4:8]
"Mast"
```

We can also work backwards and get the last three elements.

```
>>> name = "Rob Mastrodomenico"
>>> name[-3:]
"ico"
```

Or we can get everything but the last three elements.

```
>>> name = "Rob Mastrodomenico"
>>> name[:-3]
"Rob Mastrodomen"
```

We can apply much of the logic shown earlier to strings. So if we want to know if a string contains a character or combination of characters we can do so by seeing if the characters are in the string.

```
>>> name = "Rob Mastrodomenico"
>>> "ico" in name
True
>>> "z" in name
False
```

We can create custom strings using variables that we may have in our code. This is generally referred to as string formatting, so if we want to put a variable into our string we need only define the position in the string using curly brackets and then using the format method with the arguments passed in they get assigned to the appropriate positions.

```
>>> first_name = "Rob"
>>> last_name = "Mastrodomenico"
>>> name = "First name: {}, Last name: {}".format(first_name, last_name)
>>> name
"First name: Rob, Last name: Mastrodomenico"
```

In this example it doesn't seem to have any benefit as we could have easily define the whole string as we did before. However, consider the case where the names are changing. We could be looping over a

list of names or a file or names and at each iteration of the loop we want to show what the name is.

We can also give the positions in the curly brackets to where we want the variables assigned, so we could write the following:

```
>>> first_name = "Rob"
>>> last_name = "Mastrodomenico"
>>> name = "First name: {1}, Last name: {0}".format(first_name, last_name)
>>> name
"First name: Mastrodomenico, Last name: Rob"
```

That is wrong but you get the point. We can also define each value as a variable and pass that variable name in the curly brackets.

```
>>> first_name = "Rob"
>>> last_name = "Mastrodomenico"
>>> name = "First name: {f}, Last name: {1}".format(f=first_name, l=last_name)
>>> name
"First name: Rob, Last name: Mastrodomenico"
```

The first string method we will consider is how to convert a string to all uppercase letters and then all lowercase letters.

```
>>> name = "Rob Mastrodomenico"
>>> name
"Rob Mastrodomenico"
>>> name.lower()
"rob mastrodomenico"
>>> name.upper()
"ROB MASTRODOMENICO"
>>> name
"Rob Mastrodomenico"
```

We see using the lower and upper methods we create versions of the initial string in uppercase and lowercase, however we don't change the string itself. Why is this useful? You may want to check if certain characters in a string and may want to do it in a specific case so by changing the string to the correct case you can.

Next, we consider the split method which unsurprisingly can be used to split up strings.

```
>>> name = "Rob Mastrodomenico"
>>> name
"Rob Mastrodomenico"
>>> name.split(" ")
["Rob", "Mastrodomenico"]
>>> first_name, last_name = name.split(" ")
>>> first_name
"Rob"
>>> last_name
"Mastrodomenico"
```

Again while this example may seem trivial but it can be very useful as you can split on any character within a string so, for example, a comma separated string which you may find in a csv file can be split into the variables it contains:

```
>>> match_details = "Manchester United,Arsenal,2,0"
>>> match_details
"Manchester United,Arsenal,2,0"
>>> match_details.split(",")
["Manchester United", "Arsenal", "2", "0"]
>>> home_team, away_team = match_details.split(",")[0:2]
>>> home_team
"Manchester United"
>>> away_team
"Arsenal"
>>> home_goals, away_goals = match_details.split(",")[2:4]
>>> home_goals
"2"
>>> away_goals
"0"
```

So from that one line we can get out all the information it contains.

Another useful method to apply to a string is replace, again it does what it says on the tin.

```
>>> match_details = "Manchester United,Arsenal,2,0"
>>> match_details
"Manchester United,Arsenal,2,0"
>>> match_details.replace(",",":")
"Manchester United:Arsenal:2:0"
```

Here, we replace all the commas with colons, again it can come in very handy if say, for example, you want to change the separator in the string like we just did.

There is another string method that looks on the face of it like a list method, namely join.

```
>>> details = ['Manchester United', 'Arsenal', '2', '0']
>>> match_details = ','.join(details)
>>> match_details
'Manchester United,Arsenal,2,0'
```

Here, we apply the join method on the string containing just the comma and it then creates a string of the values in the list separated by the string that we applied the method on. This is a very useful method when it comes to creating strings from lists separated by a common value.

The last thing we shall consider is a Python builtin function that can be applied to strings and that is len. We earlier saw it applied to lists to get the length of them and it does the same with regards to strings, however now it tells us how many characters the string contains:

```
>>> match_details = "Manchester United,Arsenal,2,0"
>>> match_details
"Manchester United,Arsenal,2,0"
>>> len(match_details)
29
```

In this chapter, we have taken a deep dive into strings and what we can do with them and shown the level of flexibility and power Python

gives us when we have a variable of type string or if we want to create our own.

13
Regular Expressions

An extension to dealing with strings is the concept of regular expressions. The broad concept is that we can do advanced searches beyond the simple ones.

```
>>> text = "Hello in this is string"
>>> 'Hello' in text
True
```

To do this, we will use the package re from the standard Python library and as opposed to going through all the specifics let's just jump in with some examples.

The first example looks at finding all the characters a to m within the string Rob Mastrodomenico. Now to do this we pass a string containing a-m within list parenthesis and pass this into the findall method with the string of interest which has been called name. The result from this is a list containing all the characters in a-m which appear in the string.

```
>>> import re
>>> name = 'Rob Mastrodomenico'
>>> x = re.findall("[a-m]", name)
>>> x
['b', 'a', 'd', 'm', 'e', 'i', 'c']
```

Next, we see how we can find the integer values 0–9 within a sequence. We can do this in two ways; the first is by mimicking what we used in the previous example with the list convention but also by using d which gives us all values that are between 0 and 9. Both return a list of values that occur within the string. Now in the first example we can see that we get back each value but what if the value was repeated? In the next example, we see that repeated values are returned in the list as we had two instances of three within the string.

```
>>> import re
>>> txt = 'Find all numerical values like 1, 2, 3'
>>> x = re.findall("\d", txt)
>>> x
['1', '2', '3']
>>> x = re.findall("[0-9]", txt)
>>> x
['1', '2', '3']
>>> txt = 'Find all numerical values like 1, 2, 3, 3'
>>> x = re.findall("[0-9]", txt)
>>> x
['1', '2', '3', '3']
>>> x = re.findall("\d", txt)
>>> x
['1', '2', '3', '3']
```

This approach to finding if a value or values are present in the string and giving us all the values is useful but we can do more and look for specific patterns. In this next example, we use the standard text hello world and look for a specific pattern. We can pass the string "he..o" into the findall method and what this does is search for a sequence which starts with ho and has any two characters and is followed by an o, which fits nicely with the word hello. So in passing this in we get back the list containing the string hello. We can expand on this by changing the string to "hello helpo hesoo" in doing so we see that all these words are passed back from the findall. In using a different example like this, we can see how this could be applied across a bigger piece of text to see all the words that match this sequence.

```
>>> import re
>>> txt = "hello world"
>>> x = re.findall("he..o", txt)
>>> x
['hello']
>>> txt = "hello helpo hesoo"
>>> x = re.findall("he..o", txt)
>>> x
['hello', 'helpo', 'hesoo']
```

Next, we look at how to search specifically on the start of the string. To do so you use the ŝymbol prefixed to the string of interest, in this case we look for a string that starts with the string start. What the result of this gives is a list containing the word that is found so in the

first example we get back the list containing the string start and in the second example we get an empty list.

```
>>> import re
>>> txt ='starts at the end'
>>> x = re.findall("^start", txt)
>>> x
['start']
>>> txt ='ends at the start'
>>> x = re.findall("^start", txt)
>>> x
[]
```

We can achieve the same thing for looking at the last word in the string by using ending the searched string with the $ sign. In this example, we show what we get when searching for the last part of a given string and in a similar way to previous example we return a list containing that string if it does exist and an empty list if it doesn't.

```
>>> import re
>>> txt = 'the last word is end'
>>> x = re.findall("end$", txt)
>>> x
['end']
>>> txt = 'the last word is end sometimes'
>>> x = re.findall("end$", txt)
>>> x
[]
```

The last two examples look at finding something specific at the start or end of a given string, in the next example we look at all instances of a given string with another string. What we are looking to do here is find the occurrences of ai followed by 0 or more x values. So the first example shows that there are four instances of the string ai within the string when we search for aix. As in the previous examples if we don't have any instances then we get returned an empty string.

```
>>> import re
>>> txt = "The rain in Spain falls mainly in the plain!"
>>> x = re.findall("aix*", txt)
>>> x
['ai', 'ai', 'ai', 'ai']
>>> txt = 'This isnt like the other'
>>> x = re.findall("aix*", txt)
>>> x
[]
```

Expanding on the previous example you can find the number of instances of the string ai followed by one or more x by adding the + symbol. Applying that to the same string as before gives us the result of an empty string as we don't have aix within it.

```
>>> import re
>>> txt = "The rain in Spain falls mainly in the plain!"
>>> x = re.findall("aix+", txt)
>>> x
[]
```

If we are after a specified number of characters that we want to see we can use curly brackets containing the number of instances we are interested in. So in the next example we want to find moo within the string so we can do it as mo2 or moo, with each returning a string containing the character we have searched for.

```
>>> import re
>>> txt = 'The cow said moo'
>>> x = re.findall("mo{2}", txt)
>>> x
['moo']
>>> x = re.findall("moo", txt)
>>> x
['moo']
```

If we want to find one or another value we can do so by using the | symbol between the two strings that we are interested in searching for. In the example that we show we are looking for the strings avengers or heroes in our string. As we have the string Avengers with a capitalised A we only have an exact match on heroes. The second example uses Avengers with a capital A and therefore as that is exactly matched within the string we get back a list containing both strings. The last example shows what happens if we have multiple instances of one of the words that we are searching for giving us the number of instances in the order that we see them.

```
>>> import re
>>> txt = "The Avengers are earths mightiest heroes"
>>> x = re.findall("avengers|heroes", txt)
>>> x
['heroes']
>>> x = re.findall("Avengers|heroes", txt)
>>> x
['Avengers', 'heroes']
>>> txt = "The Avengers are earths mightiest heroes go Avengers"
>>> x = re.findall("Avengers|heroes", txt)
>>> x
['Avengers', 'heroes', 'Avengers']
```

We can also use special sequences like the one below which returns
the whitespace in a given string:

```
>>> txt = "Is there whitespace"
>>> x = re.findall("\s", txt)
>>> x
[' ', ' ']
```

There are other special sequences that we can use, they are listed as
follows:

- \A: This matches if the characters defined are at the beginning of
  the string "\AIt"
- \b: This matches if the characters defined are at the beginning or
  at the end of a word "\bain" r"ain\b"
- \B Returns a match where the specified characters are present,
  but NOT at the beginning (or at the end) of a word (the "r" in the

beginning is making sure that the string is being treated as a "raw string") r"\Bain" r"ain\B"

- \d Returns a match where the string contains digits (numbers from 0-9) "\d"
- \D Returns a match where the string DOES NOT contain digits "\D"
- \s Returns a match where the string contains a white space character "\s"
- \S Returns a match where the string DOES NOT contain a white space character "\S"
- \w Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character) "\w"
- \W Returns a match where the string DOES NOT contain any word characters "\W"
- \Z Returns a match if the specified characters are at the end of the string

Next, we use the split method.

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.split("\s", txt)
>>> x
['The', 'rain', 'in', 'Spain']
```

Expanding on this we can specify the number of times we want the split to be done by using the maxsplit argument. The below examples set the value to 1, 2, and 3. In each example, we see that the number of splits increases, so setting the value to 1 provides us with a list containing the results of a single split. As this increases we get more and more splits included.

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.split("\s", txt, maxsplit=1)
>>> x
>>> x = re.split("\s", txt, maxsplit=2)
>>> x
['The', 'rain', 'in Spain']
>>> x = re.split("\s", txt, maxsplit=3)
>>> x
['The', 'rain', 'in', 'Spain']
```

The next method we demonstrate is the submethod which behaves in a similar way to replace on a string. In the below example we replace the white space with the value 9:

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.sub("\s", "9", txt)
>>> x
'The9rain9in9Spain'
```

As with the previous example of split we have an extra argument that can be used here namely count, and again we apply it with values 1, 2, and 3. The result of this is the number of values that are replaced in the string with 1 giving only the first space being replaced by 9, 2 giving the first 2 spaces being replaced and so on.

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.sub("\s", "9", txt, 1)
>>> x
'The9rain in Spain'
>>> x = re.sub("\s", "9", txt, 2)
>>> x
'The9rain9in Spain'
>>> x = re.sub("\s", "9", txt, 3)
>>> x
'The9rain9in9Spain'
```

The last example that we look at is using the span method from a search result. Here, if we search for a set of characters in

```
>>> import re
>>> txt = "The rain in Spain"
>>> x = re.search("ai", txt)
>>> x.span()
(5, 7)
```

That is the last of the examples relating to regular expressions and that gives a good introduction into using the re packages and how

powerful this can be when we want to do complex searching of strings.

14
Dealing with Files

Thus far we have gone through what you can do with Python and the examples given have revolved around simple examples. In reality we want to work on data of some kind and as such we need to get it into Python. Now, we can obtain our data from a variety of sources such as databases, web API's but a common way to obtain data is through a good old fashioned file. So in this section, we will use a lot of what we have learnt so far to deal with reading data from and writing to files.

Python can read files pretty easily from the standard library. Its just a case of specifying where the file is and then creating a stream to that location. Let's demonstrate by having a file located in /Path/to/file/test.csv. This is the full path to the comma separated file test.csv.

```
>>> file_name = "/Path/to/file/test.csv"
>>> f = open(file_name,"r")
>>> f
<open file "/Path/to/file/test.csv", mode "r" at 0x1007e6270>
```

What we have done here is define a string file_name containing the name of the file and then used the open command with the argu-

ments of file_name and 'r' which is the mode to read the file in and in this case it refers to read. We have assigned the return of this to the variable f which is a stream to the file. Now to read in the data from the file we simply run:

```
>>> data = f.read()
```

What we get back is the data in a single index list, which isn't that useful. In text files what you find is that lines are separated by a line return which means that we could apply the split method which will split what it reads into new elements of the list every time it sees a line return. Its easier to demonstrate this by an example on a string:

```
>>> names = "steve\ntony\nbruce\n"
>>> names
"steve\ntony\nbruce\n"
>>> names.split("\n")
["steve", "tony", "bruce", ""]
```

So we can get our data into a readable format. The same theory works with reading data from a file where the lines are separated in the same way.

```
>>> data = f.read().split("\n")
```

With a comma separated file we have each item on a given line separated by a comma. So to get each item we need to split again based

on a comma. To do that we need to do the following:

```python
>>> names = "steve,rodgers\ntony,stark\nbruce,banner\n"
>>> names
"steve,rodgers\ntony,stark\nbruce,banner\n"
>>> names = names.split("\n")
>>> names
["steve,rodgers", "tony,stark", "bruce,banner", ""]
>>> for n in names:
...     row = n.split(",")
...     row
...
["steve", "rodgers"]
["tony", "stark"]
["bruce", "banner"]
[""]
```

Initially we split the string on the character \n to create a list containing three items. We then loop over the list and for each item in the list we split on the character comma to create a list containing first and last name. We can do this in a single line as opposed to looping the names list as follows:

```python
>>> names = "steve,rodgers\ntony,stark\nbruce,banner\n"
>>> names
"steve,rodgers\ntony,stark\nbruce,banner\n"
>>> names = names.split("\n")
>>> names
["steve,rodgers", "tony,stark", "bruce,banner", ""]
>>> names = [n.split(",") for n in names]
>>> names
[["steve", "rodgers"], ["tony", "stark"], ["bruce", "banner"], [""]]
```

So in a single line we can achieve what we did in the loop, here you can see we have basically moved the loop into a one liner. From both lines we can see that we get an empty list at the end of both implementations. What is happening here is that at the last line return when we split on the line return we get an empty string after it. So with any file where we separate on \n we need to make sure to account for the empty string, the way we can do this is refer back to the pop method we introduced earlier:

```
>>> names
[["steve", "rodgers"], ["tony", "stark"], ["bruce", "banner"], [""]]
>>> names.pop()
[""]
>>> names
[["steve", "rodgers"], ["tony", "stark"], ["bruce", "banner"]]
```

Now, we are able to read files. The next thing to cover is how to write to files. It works in much the same way as for reading from files in that we first need to define the file name and open a stream to write to file.

```
>>> file_name = "output.csv"
>>> f = open(file_name, "w")
>>> f
<_io.TextIOWrapper name="output.csv" mode="w" encoding="UTF-8">
```

This opens a stream under where your terminal window is open in write mode. To physically write something to a file you need to define something you want to be in the file.

```
>>> out_str = "something to go in the file\n"
>>> f.write(out_str)
28
>>> f.close()
```

What we have done is create a string that we want to be in our file and then using the streams method write we have written the string to file. One thing we missed from the first example when we read from file is that we forgot to close the file stream. Here, we see in the last line that we do this using the close method. Now, Python will generally tidy things like this when you quit Python or when your written program ends, however its good practice to include this in your code.

Next, we will consider how to append to a file. Now this is very similar to writing to a file however when we open a file in write mode we would override any existing file with the same name. With append we would keep the existing file and then add whatever we wanted to the end of it. The way we do this is very similar to what we have seen before, we just use the append option when opening the file, so to append to our output.csv we need to write the following:

```
>>> file_name = "output.csv"
>>> f = open(file_name, "a")
>>> f
<_io.TextIOWrapper name="output.csv" mode="a" encoding="UTF-8">
```

Let's expand on this example by applying reading and writing to a bigger example. What we are going to do is import a dataset from

sklean which is a package:

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
```

Now here we load up a dictionary object containing a dataset and relevant details that we want to work on. Here, we want to take the data and feature_name keys from this dictionary and write to a csv file.

```
>>> feature_names = boston['feature_names'][::2]
>>> list(feature_names)
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
```

To make things a little more difficult we will take every other column and not include the last two values.

```
>>> headers = list(feature_names)[::2][:-2]
>>> headers
['CRIM', 'NOX']
```

This will give us the values that we want to put into out file. So the next thing we will do is open up the file and write the headers to the file.

```
>>> file_name = "boston_output.csv"
>>> fo = open(file_name, "w")
>>> fo.write(','.join(headers) + '\n')
23
```

Here, we can see the output of 23 referring to the 23 characters that we wrote to the file. What we want to do next is write the relevant data referring to the headers to the file.

```python
>>> boston_data = boston['data']
>>> for bd in boston_data:
...     row_dict = dict(zip(feature_names, bd))
...     val_list = []
...     for h in headers:
...         val = row_dict[h]
...         val_list.append(str(val))
...     out_str = ','.join(val_list)
...     fo.write(out_str + '\n')
```

What we have done here is assign the data to boston_data and then loop over it. Each element of data can then be zipped with the feature_names to create a dictionary. The reason for doing this is to allow us to select the relevant values to write to file. To do this we loop over the headers and access the dictionary value using the key of the header. These values are then appended to a list and the join method is applied in much the same way we did for the headers to write each line to the file.

```python
>>> fo.close()
```

Lastly, we need to close the file, technically if we don't do this then Python will do it for us, however its good practice to do so.

We can then read the file in and loop over the contents using the fol-
lowing code:

```
>>> fo.close()
```

So, here we have created the file to output and have written the head-
ers to it, note the 23 denotes the number of characters written to the
file. Next, we will loop across the data and write it line by line to the
file making sure to select the columns that we want

```
>>> file_name = "boston_output.csv"
>>> f = open(file_name, "r")
>>> data = f.read().split('\n')
>>> data.pop()
>>> for d in data:
...         print(d)
```

That is really about it when it comes to reading, writing, and append-
ing to files. Its important to note that what we have shown only works
for single sheet data files.

14.1 Excel

A more common type of file that you might want to open in Python is
a spreadsheet like file containing sheets of data. This could be in the
form of an xls or xlsx file. Luckily Python has a library for us called

openpyxl which allows us to write the data to an excel file and read it back in as we will demonstrate.

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> feature_names = boston['feature_names'][::2]
>>> list(feature_names)
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
>>> headers = list(feature_names)[::2][:-2]
>>> headers
['CRIM', 'NOX']
```

Next, we want to read this data into a sheet of an excel sheet.

```
>>> from openpyxl import Workbook
>>> wb = Workbook()
>>> sheet1 = wb.create_sheet('boston_data', 0)
```

What we do here is import the relevant package and then create a Workbook. For this Workbook we then create a sheet to write to and call it boston_data and insert it into position 0 which is the first position of the spreadsheet.

```
>>> i = 1
>>> for h in headers:
...         sheet1.cell(1, i, h)
...         i += 1
```

Next, we write the headers to our sheet, note we want to insert the values into the first row so we set a counter i to 1 to start at the first column and then increment it to insert subsequent values into the relevant columns. Here, we use the cell method where we pass in row, column, and value, and here the row is fixed at 1.

```
>>> j = 2
>>> boston_data = boston['data'][0:5]
>>> for bd in boston_data:
...         k = 1
...         row_dict = dict(zip(feature_names, bd))
...         for h in headers:
...             val = row_dict[h]
...             sheet1.cell(j, k, val)
...             k += 1
...         j += 1
```

Next, we look to write the first five rows of the data to the file so to do so we use the same cell method, however now we need to increment rows and columns to deal with the fact we have multiple rows. So to do so counters are setup outside the loop for the row and inside the loop for the column. This is because we need to reset the columns for every row as we want to go back to column 1, hence the k value needs to change to 1 every time we finish writing a row.

```
>>> wb.save('boston.xlsx')
```

Lastly, to save the data we just use the save method on the workbook and pass in the name of the file we want to save.

To read the data back in is a relatively simple process.

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook('boston.xlsx')
>>> wb
<openpyxl.workbook.workbook.Workbook object at 0x7fe91d880b00>
```

We can see what worksheets we have through the worksheet method.

```
>>> wb.worksheets
[<Worksheet "boston_data">, <Worksheet "Sheet">]
>>> sheet = wb['boston_data']
>>> sheet
<Worksheet "boston_data">
```

We can then access the specific sheet using dictionary notation treating the sheet name as the key. To get the values we use row and column indexes:

```
>>> sheet[1][0].value
'CRIM'
>>> sheet[1][1].value
'NOX'
```

Note that our columns are zero indexed despite us writing to column 1 in the code to write to file but we can get the specific value by getting the value attribute.

14.2 JSON

JSON stands for JavaScript Object Notation and it has become very popular as a data type and is widely used. It's described as a lightweight data-interchange format. But what actually does that mean, well it's really a text format to store data that is easy, visually, for us to read and write, and also easy for the computers to parse and generate.

For a Python user, JSON will appear to be a mixture of lists and dictionaries in that you can have collections of key value pairs like in a dictionary but also have data stored in a manner like a list. Let's take the example that we have used previously and create a json representation of the data.

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> feature_names = boston['feature_names'][::2]
>>> list(feature_names)
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
>>> headers = list(feature_names)[::2][:-2]
>>> boston_data = boston['data'][0:5]
```

So, what we have done above is what has been done previously, however here we differ by selecting only the first five elements of the data which will allow us to show the data in json representation.

```
>>> json_list = []
>>> for bd in boston_data:
...     row_dict = dict(zip(feature_names, bd))
...     val_dict = {}
...     for h in headers:
...         val = row_dict[h]
...         val_dict[h] = val
...     json_list.append(val_dict)
>>> print(json_list)
[{'CRIM': 0.00632, 'NOX': 2.31}, {'CRIM': 0.02731, 'NOX': 7.07},
{'CRIM': 0.02729, 'NOX': 7.07}, {'CRIM': 0.03237, 'NOX': 2.18},
{'CRIM': 0.06905, 'NOX': 2.18}]
```

The next set of code gets the data into a format to export to json. As mentioned before we can achieve this via a combination of dictionaries and lists. So, initially we create a list to put every row of our data into. A row can be represented as a dictionary which in this case is simply a key value pair for two of the feature names which have been assigned to the headers. What we end up with is a list of dictionaries which we will look to export as json.

```
>>> import json
>>> file_name = "boston.json"
>>> with open(file_name, "w") as write_file:
...     json.dump(json_list, write_file, indent=4)
```

To create the json output we can use the json package and the dump method passing in the list and an open file as the arguments.

```
[
    {
            "CRIM": 0.00632,
            "NOX": 2.31
    },
    {
            "CRIM": 0.02731,
            "NOX": 7.07
    },
    {
            "CRIM": 0.02729,
            "NOX": 7.07
    },
    {
            "CRIM": 0.03237,
            "NOX": 2.18
    },
    {
            "CRIM": 0.06905,
            "NOX": 2.18
    }
]
```

The next part we need to cover is how to read the json file back into Python, luckily this is easily achieved using the json library.

```
>>> import json
>>> file_name = "boston_output.json"
>>> with open(file_name, "r") as read_file:
...     data = json.load(read_file)
...
>>> data
[{'CRIM': 0.00632, 'NOX': 2.31}, {'CRIM': 0.02731, 'NOX': 7.07},
{'CRIM': 0.02729, 'NOX': 7.07}, {'CRIM': 0.03237, 'NOX': 2.18},
{'CRIM': 0.06905, 'NOX': 2.18}]
>>> type(data)
<class 'list'>
```

As we did with writing to file we just use the load method with the open file mode read which assigns the values in the file to the data object which is of type list.

14.3 XML

XML stands for Extensible Markup Language and much like JSON it is a way to store data that is easy visually for us to read and write but at the same time easy for computers to parse and generate. Unlike JSON it doesn't have a natural link to Python data types and so needs a bit more of an introduction into its types and how it works. Let's explain using the example below.

```xml
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Rob, Mastrodomenico</author>
      <title>The Python book</title>
      <genre>Computer</genre>
      <price>Whatever</price>
      <publish_date>Whenever</publish_date>
      <description>Stuff to help you master Python</description>
   </book>
   <book id="bk102">
      <author>Rob, Mastrodomenico</author>
      <title>The Python book 2</title>
      <genre>Computer</genre>
      <price>More than the last one</price>
      <publish_date>Maybe never</publish_date>
      <description>Its like the first one but better</description>
   </book>
</catalog>
```

Now let's deconstruct the above example:

```xml
<?xml version="1.0"?>
```

The first line is the xml declaration and it could have simply been written as follows:

```xml
<?xml?>
```

If we had some specific encoding to use in the xml file we could rewrite it as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
```

Next, we have the following:

```
<catalog>
</catalog>
```

The catalog to catalog are the root elements of the XML and are the start and end of the content. The name used is arbitrary and in this case, just reflects the data we have. You will notice the use of a / on the closing content, this is common between the opening and closing elements.

Next, we add in a further level down as follows:

```
<book id="bk101">
</book>
```

Here, we have defined a book using the opening book and closing book and unlike at the root level we have attached data to this level with the addition of the id=bk101. This is the high level book data, to add more specific data about the book we can do so as follows:

```
<book id="bk101">
    <author>Rob, Mastrodomenico</author>
    <title>The Python book</title>
    <genre>Computer</genre>
    <price>Whatever</price>
    <publish_date>Whenever</publish_date>
    <description>Stuff to help you master Python</description>
  </book>
```

Under the book level, we have added variables for author, title, genre, price, publish_date, and description. As before you can see that the definition of each variable has an opening and closing using the terminology introduced earlier.

Lastly, to add another book you would do so as follows:

```
<book id="bk102">
    <author>Rob, Mastrodomenico</author>
    <title>The Python book 2</title>
    <genre>Computer</genre>
    <price>More than the last one</price>
    <publish_date>Maybe never</publish_date>
    <description>Its like the first one but better</description>
 </book>
```

We can create another book under our initial book in much the same way as we did before. The way we distinguish each book is by using its own id.

What we have shown here is how we can build interesting data structures using XML. The next question to address is how can we create and parse XML objects. To do this we use lxml which is a Python library that allows the user to take advantage of the C libraries libxml2 and libxslt. These are very fast XML processing libraries that are easily accessible through Python.

As we have done earlier in the chapter, we will use the same example and show how you can create XML from it.

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> feature_names = boston['feature_names'][::2]
>>> list(feature_names)
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
>>> headers = list(feature_names)[::2][:-2]
>>> boston_data = boston['data'][0:5]
```

The full code to write the data to xml is as follows:

```
>>> from lxml import etree
>>> root = etree.Element("root")
>>> for bd in boston_data:
...         row_dict = dict(zip(feature_names, bd))
...         row = etree.SubElement(root, "row")
...         for h in headers:
...                 child = etree.SubElement(row, h)
...                 val = row_dict[h]
...                 child.text = str(val)
>>> et = etree.ElementTree(root)
>>> et.write('boston.xml', pretty_print=True)
```

Breaking this down we first import lxml and then create the root of our xml document.

```
>>> from lxml import etree
>>> root = etree.Element("root")
```

Next, we have to loop over the data in a similar way that we have done before to put the data into our xml.

```
>>> for bd in boston_data:
...         row_dict = dict(zip(feature_names, bd))
...         row = etree.SubElement(root, "row")
...         for h in headers:
...             child = etree.SubElement(row, h)
...             val = row_dict[h]
...             child.text = str(val)
```

The mechanism of looping the data is no different to what we have seen and we create the same row_dict and loop the headers to get the values, however the difference is in how we setup the xml and where we write to. For each iteration across the boston_data we create another row called row under the root using the SubElement method assigning root as the parent. Then for every value we obtain from looping the headers we create another SubElement this time with parent row and having the name of the header. We assign the value for this by setting the text attribute to be that value. This then gives us the format of data.

```
>>> et = etree.ElementTree(root)
>>> et.write('boston.xml', pretty_print=True)
```

The last part is to write the data to file so we can make use of the write method by passing the root of the document through ElementTree. Note that we set the pretty_print to be True, which gives the following file:

```xml
<root>
  <row>
    <CRIM>0.00632</CRIM>
    <NOX>2.31</NOX>
  </row>
  <row>
    <CRIM>0.02731</CRIM>
    <NOX>7.07</NOX>
  </row>
  <row>
    <CRIM>0.02729</CRIM>
    <NOX>7.07</NOX>
  </row>
  <row>
    <CRIM>0.03237</CRIM>
    <NOX>2.18</NOX>
  </row>
  <row>
    <CRIM>0.06905</CRIM>
    <NOX>2.18</NOX>
  </row>
</root>
```

Now, we will show how you can read an XML file in using lxml in Python using the example below.

```
>>> from lxml import objectify
>>> xml = objectify.parse(open('boston.xml'))
>>> root = xml.getroot()
>>> children = root.getchildren()
>>> print(children)
[<Element row at 0x7fc64c57e548>, <Element row at 0x7fc64c57e748>,
<Element row at 0x7fc64c587248>, <Element row at 0x7fc64c587288>,
<Element row at 0x7fc64c5872c8>]
>>> for c in children:
...     print(c['CRIM'])
...     print(c['NOX'])
```

So, what we have done here is to import objectify from lxml, which will be used to read in the XML.

```
>>> xml = objectify.parse(open(file_name))
```

Here, we are reading in the XML file and parsing it using the parse method of objectify. This gives us an XML object which we can then use to try and parse out the information. Next, we look to get the root of the document using:

```
>>> root = xml.getroot()
```

Having obtained the root we look to get the children of this which represents the next level down which are the rows.

```
>>> children = root.getchildren()
```

Now, to access the values we can loop through the children as that object is simply a list. In doing so we can obtain and print the values

as follows:

```
>>> for c in children:
...     print(c['CRIM'])
...     print(c['NOX'])
```

These refer to the values in the dataset which we created.

This chapter has covered some important concepts relating to files and how to read from and write to them using Python. We have covered a number of different file types and given practical examples of how these work. We will show later in the book other approaches to reading and writing to file but these somewhat low level approaches are very important when we want to have a high level of control when it comes to manipulating the data and are a great tool to have in your arsenal.

15
Functions and Classes

In this chapter, we are going to introduce the concepts of functions and classes and how these concepts can be introduced into the way you code in Python. Thus far in this book everything has been shown as blocks of code. These blocks of code may be just snippets to demonstrate a specific concept or longer sections to demonstrate how to perform a task. When you come to practically write your code you can write it in the manner that you have seen within this book and that is perfectly acceptable. However, if you want to group code together or reuse it then functions and classes are a great thing to use.

Let's start with functions, these are very useful for code that you want to reuse or if you have repeated code, then a function can help you. Functions also allow you to run code with arguments which gives you the ability to have the behaviour of the function dictated by the variables you pass to it. Let's demonstrate how to setup a function by using the lottery example from before. Now to familiarise ourselves with the code we looked to generate the results of a lottery draw using elements of code that we had used up to that point of the book. The results look as follows:

```
>>> from random import randint
>>> min = 1
>>> max = 59
>>> result_list = []
>>> while len(result_list)<7:
...         ball = randint(min, max)
...         if ball not in result_list:
...             result_list.append(ball)
...
>>> result_list
[41, 48, 47, 55, 18, 15, 43]
```

Now, we can cast this a function called lottery as follows:

```
>>> from random import randint
>>> def lottery():
...         min = 1
...         max = 59
...         result_list = []
...         while len(result_list)<7:
...             ball = randint(min, max)
...             if ball not in result_list:
...                 result_list.append(ball)
...         return result_list
...
```

What we have done here is define a function called lottery. This is done by using the command def followed by the name that you want to give the function. We then use two round brackets to denote the arguments that we want to pass to the function. Here, we have nothing within the brackets which means that we pass no argument into

the function. Note that following the round brackets we use a colon in the same way we have for if, else, for, and while statements and in the same way as we do with these statements we indent the code one level from where the function was defined. From this point onwards the code used is exactly the same as we have seen in the lottery example. The main difference comes at the end in that we use the statement return with the return_list variable. What this does is return back what the variable return list after the code within the function has been run. In this instance we get back the list of lottery numbers that are generated. To run the above function we just do the following:

```
>>> results = lottery()
>>> results
[29, 42, 26, 37, 30, 8, 43]
```

If we consider what the function is doing we generate balls using the randint function. What if we didn't want numbers being 1–59 and instead want 1–49. We could just alter the code to have 49 instead of 59 but actually wouldn't it make sense for us to have arguments representing these max and min values. We can make that change relatively easily by rewriting the previous function as follows:

```
>>> from random import randint
>>> def lottery(min, max):
...        result_list = []
...        while len(result_list)<7:
...               ball = randint(min, max)
...               if ball not in result_list:
...                       result_list.append(ball)
...        return result_list
...
```

What we have done here is to have the min and max as variables we pass into the function. Despite the values being the minimum and maximum we can call them what we want to. We have just denoted them as min and max, however they could be x and y, and its just a question of referencing these in the appropriate place within the code. So here the values min and max are used only in the randint function to give us back the random ball. We can demonstrate how we would use this below.

```
>>> results = lottery(1, 49)
>>> results
[10, 48, 19, 46, 40, 42, 49]
```

In the previous example, we have passed in the minimum and maximum values we want to use in the function, however we may want them to have a default value such as 1 and 59 as in the original example. We can do that by just setting the values we pass into the function to have defaults, this is done as follows:

```
>>> from random import randint
>>> def lottery(min=1, max=59):
...         result_list = []
...         while len(result_list)<7:
...             ball = randint(min, max)
...             if ball not in result_list:
...                 result_list.append(ball)
...         return result_list
...
```

Here, we have given the min a default of 1 and max default as 59 by using the equals to set the values. This then gives us the flexibility to call the function as follows:

```
>>> lottery()
[12, 15, 31, 30, 57, 17, 22]
>>> lottery(1,49)
[11, 13, 30, 47, 31, 2, 19]
>>> lottery(max=49)
[34, 36, 26, 18, 13, 11, 46]
```

By passing nothing into the lottery function the min and max are set to the defaults 1 and 59. In the second example, we pass two values in the first getting assigned to the min and the second to the max. The last example of calling the lottery example sets the max value by passing it as an argument. What we can see is that by using arguments can give us a lot of flexibility when it comes to using functions.

If we look in more detail at the lottery example we can modify the example to be more flexible. The code we have used so far allows us to generate exactly 7 balls in our draw, however we may want to have more or less. To do this lets pass one more variable into the function definition namely draw length and we do so as follows:

```
>>> from random import randint
>>> def lottery(min=1, max=59, draw_length=7):
...         result_list = []
...         while len(result_list)<draw_length:
...             ball = randint(min, max)
...             if ball not in result_list:
...                 result_list.append(ball)
...         return result_list
...
```

We can apply our modified lottery example as follows:

```
>>> lottery()
[32, 23, 17, 6, 31, 4, 19]
>>> lottery(1,49,6)
[41, 46, 44, 47, 18, 24]
>>> lottery(max=49,draw_length=6)
[1, 15, 19, 26, 23, 29]
```

This works exactly the same way as seen before but every time we have called it we have passed in the exact arguments that are required what would happen if we passed in different values.

```
>>> lottery('1','49','6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in lottery
TypeError: '<' not supported between instances of 'int' and 'str'
```

Here, we see that the function returns an error as it cannot support the types that are passed in so it would make sense to allow the function to determine if it can deal with the arguments passed in. We need to specify the values be integers as both the min and max need to be integers as randint generates integers, similarly the draw length can only be integers as it relates to the length of a list. We can rewrite the function as follows:

```
>>> def lottery(min=1, max=59, draw_length=7):
...         if type(min)!=int:
...             print('min must be int')
...             return None
...         if type(max)!=int:
...             print('max must be int')
...             return None
...         if type(draw_length)!=int:
...             print('draw_length must be int')
...             return None
...         result_list = []
...         while len(result_list)<draw_length:
...             ball = randint(min, max)
...             if ball not in result_list:
...                 result_list.append(ball)
...         return result_list
...
```

What we have done here is use the type built in function to check if the value passed in for each variable is an integer. Note that this is done one by one so that an informative message can be sent back as to why the error occurred and what the problem was. To see this in action we only need to run the following:

```
>>> lottery('1','49','6')
min must be int
```

We see here that the function returned the message 'min must be an int, however we know that we would also have a problem with max and draw length which were passed in as strings but should also be an integer. We can expand upon the logic above by using a combination of if and else statements to determine which combination of variables are passed in with an invalid type.

```python
>>> def lottery(min=1, max=59, draw_length=7):
...     min_val = True
...     max_val = True
...     draw_length_val= True
...
...     if type(min) != int:
...         min_val = False
...     if type(max) != int:
...         max_val = False
...     if type(draw_length) != int:
...         draw_length_val = False
...
...     if min_val is False:
...         if max_val is False:
...             if draw_length_val is False:
...                 print('min, max, draw_length need to be integer')
...                 return
...             else:
...                 print('min and max need to be integer')
...                 return
...         else:
...             if draw_length_val is False:
...                 print('min and draw_length need to be integer')
...                 return
...             else:
...                 print('min needs to be integer')
...                 return
...     else:
...         if max_val is False:
...             if draw_length_val is False:
...                 print('max and draw_length need to be integer')
...                 return
...             else:
...                 print('max need to be integer')
...                 return
...         else:
...             if draw_length_val is False:
...                 print('draw_length needs to be integer')
...                 return
...             else:
...                 pass
...
...     result_list = []
...     while len(result_list)<draw_length:
...         ball = randint(min, max)
...         if ball not in result_list:
...             result_list.append(ball)
...     return result_list
...
```

What we have added here is variables that are set to True for each of the values that we pass in as arguments. We set these to be False if the value is not an integer and use a combination of if else statements to determine which combination are of the right type and print an informative message about what variables are not correct. Note that we don't return anything when values are not all correct and the return type is None.

```
>>> lottery(1,'2',3)
max need to be integer
>>> lottery('1',2,3)
min needs to be integer
>>> lottery('1','2',3)
min and max need to be integer
>>> lottery('1',2,'3')
min and draw_length need to be integer
>>> lottery('1','2','3')
min, max, draw_length need to be integer
>>> res = lottery(1,'2','3')
max and draw_length need to be integer
>>> res is None
True
```

Having introduced functions we will move onto classes within Python. Classes can be very powerful objects which allow us to bundle together lots of functions and variables. When we talk about functions and variables when related to a class we call them methods and attributes. We will demonstrate this by creating a simple class:

```
>>> class MyClass:
...         x = 10
...
>>> mc = MyClass()
>>> mc.x
10
```

What we have done here is create a class called my class by using the class definer. Within the class we have set a variable x to be equal to 10. We can then create an instance of MyClass and access the x variable using the dot syntax. Let us expand on this by creating a lottery example based on the function before.

```python
>>> class Lottery:
...     def __init__(self, min=1, max=59, draw_length=7):
...         self.min = min
...         self.max = max
...         self.draw_length = draw_length
...
...     def lottery(self):
...         min_val = True
...         max_val = True
...         draw_length_val= True
...
...         if type(self.min) != int:
...             min_val = False
...         if type(self.max) != int:
...             max_val = False
...         if type(self.draw_length) != int:
...             draw_length_val = False
...
...         if min_val is False:
...             if max_val is False:
...                 if draw_length_val is False:
...                     print('min, max, draw_length need to be integer')
...                     return
...                 else:
...                     print('min and max need to be integer')
...                     return
...             else:
...                 if draw_length_val is False:
...                     print('min and draw_length need to be integer')
...                     return
...                 else:
...                     print('min needs to be integer')
...                     return
...         else:
...             if max_val is False:
...                 if draw_length_val is False:
...                     print('max and draw_length need to be integer')
...                     return
...                 else:
...                     print('max need to be integer')
...                     return
...             else:
...                 if draw_length_val is False:
...                     print('draw_length needs to be integer')
...                     return
...                 else:
...                     pass
...
...         result_list = []
...         while len(result_list)<self.draw_length:
...             ball = randint(self.min, self.max)
...             if ball not in result_list:
...                 result_list.append(ball)
...         return result_list
```

In this code, we define the class in the way we did before and call it Lottery. Next, we create an init method using __init__ this initialiser is called when we define the class so we can pass in arguments from here that can be used within the class. Note that we can use the standard defaults but these are then assigned to the class by using the self-dot syntax which allows that value to be part of class and then allowed to be used anywhere within the class. We can create the class in the following example:

```
>>> l = Lottery()
>>> l.lottery()

>>> l = Lottery(1,49,6)
>>> l.lottery()

>>> l = Lottery(draw_length=8)
>>> l.lottery()
```

You may think this isn't very different to what we did with the function, however we can change up our code to take advantage of how classes work to simplify how we deal with variables not being of the right type.

```python
>>> class Lottery:
...     def __init__(self, min=1, max=59, draw_length=7):
...         self.min = min
...         self.max = max
...         self.draw_length = draw_length
...         self.valid_data = True
...         if type(self.min) != int:
...             print('min value needs to be of type int')
...             self.valid_data = False
...         if type(self.max) != int:
...             print('max value needs to be of type int')
...             self.valid_data = False
...         if type(self.draw_length) != int:
...             print('draw_length value needs to be of type int')
...             self.valid_data = False
...
...     def lottery(self):
...         if self.valid_data:
...             min_val = True
...             max_val = True
...             draw_length_val= True
...             result_list = []
...             while len(result_list)<self.draw_length:
...                 ball = randint(self.min, self.max)
...                 if ball not in result_list:
...                     result_list.append(ball)
...             return result_list
```

What we have done here is move a lot of the complex logic around checking each types from the lottery function into the initialiser which means we can print what needs changing and also set the valid data attribute. Only if the valid data returns True can we run the lottery function. By using a class we have a lot more flexibility within it to make use of attributes and set logic that can affect what other methods do. Used correctly they can be a very powerful tool.

In the previous example, we have shown how to write functions and classes but as with the rest of the book we have done so in the interactive shell, however a more practical way to create a file with the content in. To do so is quite simple: you just simply write the exact same code into any blank file and save it with the suffix .py.

While in theory you can use any editor to write code in, you should use an integrated development environment (IDE). With the Anaconda installation of Python you get Spyder included, which is a great Python IDE. There are many more available with many free to use so once you get used to writing code in files you can choose which one is more suitable for you. For now, we will demonstrate how to develop within Spyder. Upon starting up you will be presented with a screen which will look something like the one shown in Figure 15.1.

The two windows we are most concerned with here are the editor and the console. In the editor window, we would type code as we have throughout the book but we would be able to save code to a physical file. You may ask why we would want to do this but it allows us to have saved code be it a set of Python commands, functions, classes, or any combination. In having saved the code, we can re-run anything that we have written in Python very simply and using an IDE like Spyder gives us the option to run code that we have. So here by clicking the run command we would execute all code within a file. You can also see we have a console available to us, this allows the testing of

commands and also allow you to access the variables within a script that you may have run. We will demonstrate this with a very simple example (Figure 15.2).
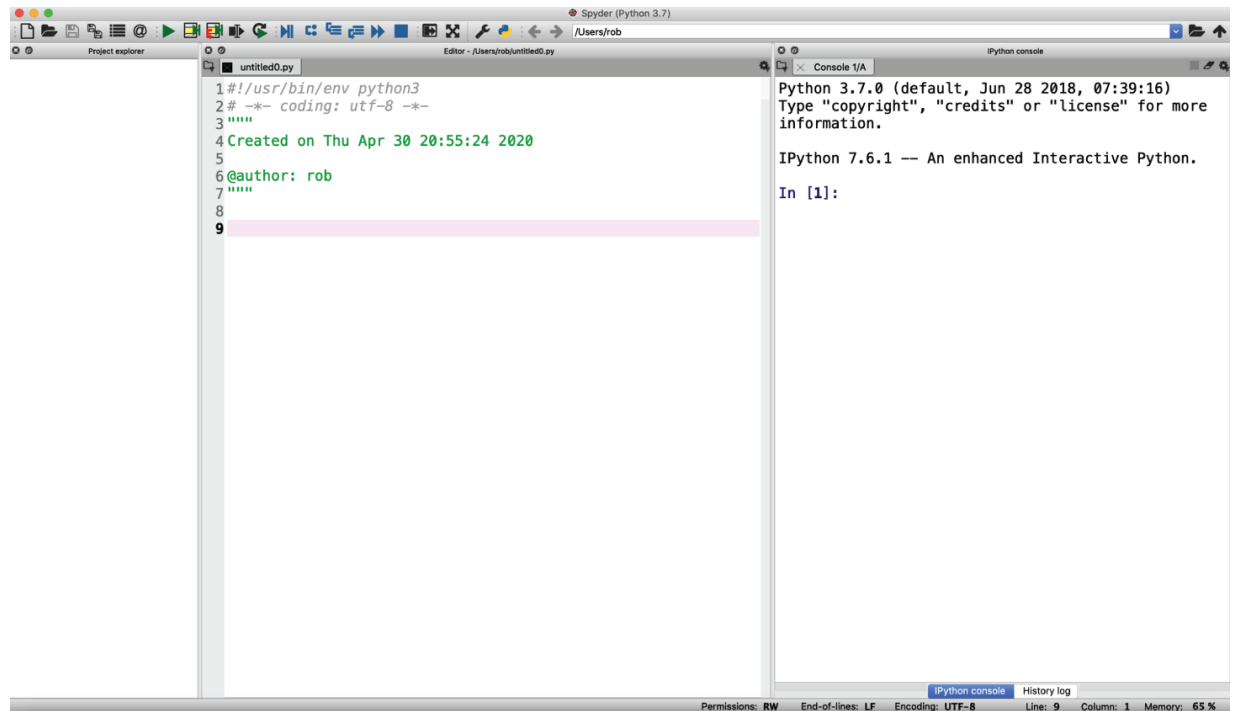
Figure 15.1 Spyder IDE.
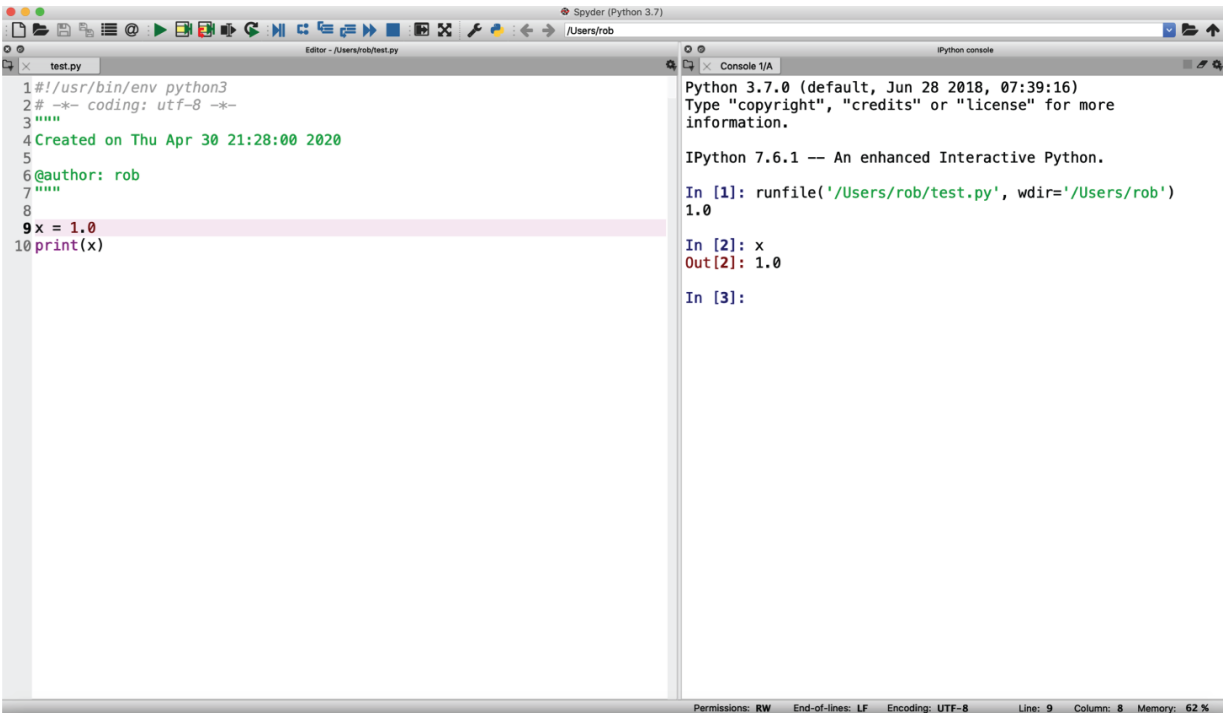
Figure 15.2 Run file in Spyder.

Here, we have written a very simple script called test.py where we define a variable x to be 1 and print it. By clicking the run file arrow in the toolbar we can run this script which is displayed on line 1 of the console as run file with the name of the file. Here, you can see the result of the run prints 1.0 to the screen however within the console we can access the x variable that we defined within the file.

Adding your code to files is great if you want to easily and quickly run a set of commands on demand and means you can create an archive and potentially version code that you have. What you can also do is share code with others and yourself. Let's demonstrate via an example, if we take the lottery function we defined at the start of the chapter and put this in a file called lottery.py we are then able to use this

by importing it into Python. We have covered how to import packages in earlier chapters and importing your own Python file is no different. When importing your own file it is important to understand how Python does an import. To do so we import the package sys and look at the sys.path list.

```
>>> import sys
>>> sys.path
['',
'/Users/rob/anaconda3/lib/python37.zip',
'/Users/rob/anaconda3/lib/python3.7',
'/Users/rob/anaconda3/lib/python3.7/lib-dynload',
'/Users/rob/anaconda3/lib/python3.7/site-packages',
'/Users/rob/anaconda3/lib/python3.7/site-packages/aeosa']
```

The sys.path list contains locations where Python can search for, so if we were looking to import our lottery file we can do so if it is present at one of the locations in sys.path. Note that the first entry is the current location you are in. We can also append a new location to this list if we require. Now, we can import the contents of the lottery.py file by just running the following code:

```
>>> from lottery import *
```

This then gives us access to everything within the file and we could then just run our lottery function by calling it. We could also import using other approaches used earlier to import a package. This concept of importing our own code allows the coder to be flexible in how code

is structured and also reduce repeatability by having key code written once and shared when needed. We could import our class into another file and run the code from there. If we create a file called import_lottery.py in the same directory as the lottery.py we can run the Lottery class as follows:

```python
from lottery import *
l = Lottery()
l.lottery()
```

It's as easy as that to share the code with other files. This makes it really easy to move sharable code into its own functions or classes and share it with other files very easily. Having both the console and an editor is important as sometime you want to work interactively to understand what you need to do but overall its much more efficient to have files with your code in.

16
Pandas

Previously we have looked at concepts and packages from the standard Python library, and now in this chapter we will look at a third-party package and one that is very relevant within the Python eco-system. Pandas is a package that is used for data analysis and data manipulation. It's used in a variety of packages and therefore understanding of it and its concepts is a crucial tool for a Python programmer to learn. In this chapter, we will introduce the package pandas from the basics up to some more advanced techniques. However, before we get started with pandas, we will briefly cover numpy arrays which alongside dictionaries and lists are concepts that should be understood to allow us to cover pandas.

16.1 Numpy Arrays

Numpy comes as part of the Anaconda distribution and is a key component in the scientific libraries within Python. It is very fast and underpins many other packages within Python. We concentrate on one specific aspect of it, numpy arrays. However, if you are interested in any of the machine learning libraries within Python, then numpy is certainly something worth exploring further.

We can import it as follows.

```
>>> import numpy as np
```

Why np? Its the standard convention used in the documentation, however you do not have to use that convention but we will. In this chapter, we won't cover everything to do with numpy but instead only introduce a few concepts and the first one we will do is introduce an array. An array in numpy is much like a list in Python. If we want to create an array of integers $0–10$ we can do so as follows:

```
>>> number_array = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> number_array
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

It looks like we passed a list into the method array and that is basically what we did as we can define the same array as follows:

```
>>> number_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> number_array = np.array(number_list)
>>> number_array
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

You may be thinking it looks like a list and we can use a list to create it, why is it different from a list. Very early on we looked at lists and operations on lists and we saw that using the common mathematical operators either didn't work or worked in an unexpected way. We will

now cover these again and compare them to what happens when using an array in numpy. We will begin by looking at addition:

```
>>> number_list + number_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> number_array + number_array
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

So, what we see is that with a list we have concatenation of two lists which is what we have seen before, however using an array we add together the two arrays and return a single array where the result is the element wise addition. Next, let's consider what happens when we use the mathematical subtraction symbol.

```
>>> number_list - number_list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: "list" and "list"
>>> number_array - number_array
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Here, we see that for two lists an error is thrown as it doesn't know how to use the operand on two lists. However, for two arrays it behaves how we might expect it to and subtracts element wise from the first array the value in the second. What happens if we look at the multiplication symbol * applied to two arrays and lists?

```
>>> number_list * number_list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type "list"
>>> number_array * number_array
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

Again, we see that this operand doesn't work on two lists but the arrays provide elementwise multiplication. Now for completion we will look at the division operand on both lists and arrays.

```
>>> number_list / number_list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: "list" and "list"
>>> number_array / number_array
__main__:1: RuntimeWarning: invalid value encountered in true_divide
array([ nan,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.])
```

Unsurprisingly, we see that this doesn't work on lists but on the arrays it performs elementwise division of the values in the first array by those in the second. Note that we get a warning for division by zero but ultimately it allows element by element division. This is great if we want to perform some mathematical operation on two lists which we cannot do and can be much faster. For the examples we have covered so far we can achieve the same thing using lists in Python in one line via list comprehension. So the three examples that didn't work here can be rewritten in as follows:

```
>>> number_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> subtraction_list = [nl - nl for nl in number_list]
>>> subtraction_list
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> multiplication_list = [nl * nl for nl in number_list]
>>> multiplication_list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> division_list = [nl / nl for nl in number_list]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

We can see that subtraction and multiplication can be done using list comprehension but division cannot be done as we get a problem with division by zero. Now, it should be noted that we have simply worked with operations on the same list which makes it easy to rewrite, however if we had two distinct lists we cannot use list comprehension and to rewrite using loops becomes more difficult. Let's demonstrate this by creating two random integer arrays. So in numpy, we can do this by using the random choice functionality as follows:

```
>>> import numpy as np
>>> np.random.choice(10,10)
array([7, 5, 3, 2, 0, 4, 6, 1, 6, 8])
```

Here, we generated an array of length 10 containing random numbers between 0 and 9. Now if we extend this example to 1 million random numbers and generate two arrays we can multiply them together as follows:

```
>>> x = np.random.choice(100, 1000000)
>>> x
array([11, 58, 13, ..., 11, 93, 27])
>>> y = np.random.choice(100, 1000000)
>>> y
array([49, 24, 11, ..., 34, 32, 15])
>>> result = x * y
>>> result
array([ 539, 1392,  143, ...,  374, 2976,  405])
```

What we have just done is complete 1 million multiplications instantly, if you try this using loops you would be waiting quite a bit longer than an instance! We have seen how powerful numpy arrays are but how do we access elements of them. Luckily we can access elements as we did for lists. We will give some examples below applied to the result array from the previous example:

```
>>> result[10]
2232
>>> result[10:20]
array([2232, 7140,  574, 1764,    0,   90,  632, 9207, 2520,  170])
>>> result[:-1]
array([ 539, 1392,  143, ...,  665,  374, 2976])
>>> result[-3:-1]
array([ 374, 2976])
```

You can see that we access elements in much the same we did for lists.

Now having introduced the concept of an array alongside everything else means we can start looking at Pandas starting with Series.

16.2 **Series**

We can import pandas as follows.

```
>>> import pandas as pd
```

Like before with numpy we use the alias pd which is the general convention used in the documentation for the package.

The first thing that we will cover here is the concept of a Series, we shall demonstrate this first by an example.

```
>>> import pandas as pd
>>> point_dict = {"Bulgaria": 45, "Romania": 43, "Israel": 30,
    "Denmark": 42}
>>> point_series = pd.Series(point_dict)
>>> point_series
Bulgaria    45
Romania     43
Israel      30
Denmark     42
dtype: int64
```

We created a dictionary containing the keys of country names and the median age of citizens (source worldomometers.info) in that country and passes then in to the Series method to create point series. We can access the elements of the series as follows:

```
>>> point_series[0]
45
>>> point_series[1:3]
Romania      43
Israel       30
dtype: int64
>>> point_series[-1]
42
>>> point_series[:-1]
Bulgaria     45
Romania      43
Israel       30
dtype: int64
>>> point_series[[1,3]]
Romania      43
Denmark      42
dtype: int64
```

You can see we can access the first element as if it was a list using the position of the value we want. We can also use the colon separated positional values as well as negative indices which we have covered earlier. There is a different way we can access elements of the series and that is by passing a list of the positions we want from the series. So if we want the first and third elements we have the values 0 and 2 in the list.

We have just accessed the values of the dict that we passed in to create the series but what about the keys and what use do they have in

the series? What we will now show is that the series can also be accessed like it was a dictionary:

```
>>> point_series.index
Index(['Bulgaria', 'Romania', 'Israel', 'Denmark'], dtype='object')
>>> point_series['Bulgaria']
45
```

We see the series has an index which is the key of the dictionary and we can access the values using the dictionary access approach we have seen earlier. Now, when we covered dictionaries earlier we saw that we could try and access a value from a key that isn't in the dictionary and it would throw an exception which is the same for the series.

```
>>> point_series["England"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/rob/anaconda/lib/python2.7/site-packages/pandas/core/series.py",
  line 601, in __getitem__
    result = self.index.get_value(self, key)
  File "/Users/rob/anaconda/lib/python2.7/site-packages/pandas/indexes/base.py",
  line 2183, in get_value
    raise e1
KeyError: "England"
>>> point_series.get("England")
```

Here, we have shown what happens but you can see we have used the method get to try and access the value for the index England. As opposed to throwing an exception it just returns None.

Given we can now access elements of a series we will now show how you can operate on it. Given the series is based on the concept of an

array in numpy you can do much of what you would in numpy to the series. So, now we will create series of random numbers and show how we can operate on them.

```
>>> import numpy as np
>>> np.random.rand(10)
array([ 0.97886013,  0.18080097,  0.84464838,  0.13603242,  0.23835642,
        0.69567161,  0.64771013,  0.67115685,  0.73923655,  0.22755168])
```

Here, we have used numpy's random methods to generate an array of 10 random numbers between 0 and 1. This can be assigned to a series relatively easily.

```
>>> import numpy as np
>>> import pandas as pd
>>> random_series = pd.Series(np.random.rand(10))
>>> random_series
0    0.173617
1    0.778444
2    0.700113
3    0.170871
4    0.846544
5    0.157117
6    0.420151
7    0.960854
8    0.759314
9    0.841746
dtype: float64
```

We can operate on this in much the same way as we do with a numpy array.

```
>>> import numpy as np
>>> import pandas as pd
>>> random_series_one = pd.Series(np.random.rand(10))
>>> random_series_two = pd.Series(np.random.rand(10))
>>> random_series_one + random_series_two
0    1.178019
1    0.940846
2    0.609092
3    0.888857
4    1.014994
5    0.531090
6    0.486204
7    0.793577
8    1.177649
9    0.661979
dtype: float64
>>> random_series_one - random_series_two
0    -0.410520
1    -0.757653
2    -0.412827
3    -0.154943
4    -0.465986
5    -0.372245
6    -0.245580
7     0.692818
8    -0.633033
9    -0.028016
dtype: float64
>>> random_series_one / random_series_two
0     0.483148
1     0.107856
2     0.192055
3     0.703117
4     0.370706
5     0.175844
6     0.328819
7    14.752071
8     0.300779
9     0.918792
dtype: float64
>>> random_series_one * random_series_two
0    0.304801
1    0.077789
2    0.050142
3    0.191515
4    0.203267
5    0.035873
6    0.044021
7    0.037442
8    0.246531
9    0.109358
dtype: float64
```

Now that all looks the same as we have seen for arrays earlier, however one key difference is that we can operate on splices of the series.

```
>>> random_series_one[1:] * random_series_two[:-1]
0              NaN
1        0.077789
2        0.050142
3        0.191515
4        0.203267
5        0.035873
6        0.044021
7        0.037442
8        0.246531
9              NaN
dtype: float64
```

What we see is that the multiplication is done on the elements of the series by index. So where we don't have an index for both series we get an NaN shown.

We earlier defined a series by using a dictionary but we can define a series using a list or array as follows:

```
>>> pd.Series([1,2,3,4,5,6])
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
>>> pd.Series(np.array([1,2,3,4,5,6]))
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

As you can see the index is defined automatically by pandas, however if we want a specific index we can define one as follows:

```
>>> pd.Series([1, 2, 3, 4, 5, 6], index=["a", "b", "c", "d", "e", "f"])
a    1
b    2
c    3
d    4
e    5
f    6
dtype: int64
```

So, here we pass an optional list to the index variable and this gets defined as the index for the series. It must be noted that the length of

the index list must match that of the list or array that we want to make a series.

16.3 **DataFrames**

Having looked at series, we will now turn our attention to data frames which are arguably the most popular aspect of pandas and are certainly what I use the most. They are essentially an object that carries data in column and row format, so for many they will mimic what is held in a spreadsheet or for others the content of a database table.

We will start off by looking at how we create a DataFrame and like with a series there are many ways we can do it.

```
>>> countries = ["United Kingdom","France","Germany","Spain",
                  "Italy"]
>>> median_age = [40,42,46,45,47]
>>> country_dict = {"name": countries, "median_age": median_age}
>>> country_df = pd.DataFrame(country_dict)
>>> country_df
             name  median_age
0  United Kingdom          40
1          France          42
2         Germany          46
3           Spain          45
4           Italy          47
```

What we have done above is begin by setting up two lists, one containing names and another containing values. These are then put into a dictionary with keys name and value. This dictionary is then passed

into the DataFrame method of pandas and what we get back is a DataFrame object with column names of name and values. We can see here that the index is automatically defined as 0–4 to correspond with the number of elements in each list.

```
>>> countries = pd.Series( ["United Kingdom","France",
                             "Germany","Spain","Italy"])
>>> median_age = pd.Series([40,42,46,45,47])
>>> country_dict = {"name": countries, "median_age": median_age}
>>> country_df = pd.DataFrame(country_dict)
>>> country_df
              name   median_age
0   United Kingdom           40
1           France           42
2          Germany           46
3            Spain           45
4            Italy           47
```

We can do the same using a dictionary of Series again assigning the Series to a dictionary and passing it into the DataFrame method. The same would happen if we used numpy arrays.

Next, we create a DataFrame using a list of tuples where the data is now country name, median age and density of the country.

```
>>> data = [("United Kingdom", 40, 281), ("France", 42, 119),
            ("Italy", 46, 206)]
>>> data_df = pd.DataFrame(data)
>>> data_df
                 0   1    2
0   United Kingdom  40  281
1           France  42  119
2            Italy  46  206
```

Here, we have created a list of tuples and we then pass those into the DataFrame method and it returns a three column by three row data frame. Unlike before we not only have auto assigned index values but we also have auto assigned column names which aren't the most useful, however we will later show how to assign both. The same applies here for a list of lists, list of series, or a list or arrays. It also works for a list of dictionaries, however the behaviour is slightly different.

```
>>> data = [{"country": "United Kingdom", "median_age":40, "density": 281},
            {"country": "France", "median_age": 42, "density": 119},
            {"country":"Italy", "median_age": 46, "density":206}]
>>> data_df = pd.DataFrame(data)
>>> data_df
          country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
```

When the list of dictionaries is passed in we get the same DataFrame, however now we have column names from the dictionary. On the face of it everything seems like it works the same as for lists of lists, however if we change some of the keys we get some different behaviour.

```
>>> data = [{"country": "United Kingdom", "median_age":40, "density": 281},
            {"country": "France", "median_age": 42, "density": 119},
            {"country":"Italy", "median": 46, "density":206}]
>>> data_df = pd.DataFrame(data)
>>> data_df
          country  density  median  median_age
0  United Kingdom      281     NaN        40.0
1          France      119     NaN        42.0
2           Italy      206    46.0         NaN
```

What we see here is that as every dictionary doesn't have all the same keys pandas fills in the missing values with NaN. Next, we will look at how to access elements of the data frame.

```
>>> data = [{"country": "United Kingdom", "median_age":40, "density": 281},
            {"country": "France", "median_age": 42, "density": 119},
            {"country":"Italy", "median_age": 46, "density":206}]
>>> data_df = pd.DataFrame(data)
>>> data_df
          country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
>>> data_df['country']
0    United Kingdom
1            France
2             Italy
Name: country, dtype: object
>>> data_df["country"][0]
'United Kingdom'
>>> data_df["country"][0:2]
0    United Kingdom
1            France
Name: country, dtype: object
```

We have done quite a lot here. The first thing is that we have defined the data frame based on the list of dictionaries as we showed previously. We then accessed all the elements of the column country by

passing the name of the column as the key of the data frame. We next showed how we could access the first element of that by adding the index of the value we wanted. This is an important distinction as we aren't asking for the first element, we are instead asking for the value of the column with index value 0. Lastly, we select the rows of the country with index 0 and 1 in the usual way we would for a list, but again we are asking for specific index rows.

```
>>> data_df["country"][-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/rob/anaconda/lib/python2.7/site-packages/pandas/core/series.py",
   line 601, in __getitem__
    result = self.index.get_value(self, key)
  File "/Users/rob/anaconda/lib/python2.7/site-packages/pandas/indexes/base.py"
, line 2169, in get_value
    tz=getattr(series.dtype, 'tz', None))
  File "pandas/index.pyx", line 105, in pandas.index.IndexEngine.get_value
  (pandas/index.c:3567)
  File "pandas/index.pyx", line 113, in pandas.index.IndexEngine.get_value
  (pandas/index.c:3250)
  File "pandas/index.pyx", line 161, in pandas.index.IndexEngine.get_loc
  (pandas/index.c:4289)
  File "pandas/src/hashtable_class_helper.pxi", line 404, in
  pandas.hashtable.Int64HashTable.get_item (pandas/hashtable.c:8555)
  File "pandas/src/hashtable_class_helper.pxi", line 410, in
  pandas.hashtable.Int64HashTable.get_item (pandas/hashtable.c:8499)
KeyError: -1
```

Here, we asked for the last value of the column country as we would with a list, however it threw an error because there is no index -1 in the index for the data frame. So we can't treat the data frame as we would a list and we need to have an understanding of the index. For any data frame we can find out the index and columns as follows:

```
>>> data_df.index
RangeIndex(start=0, stop=3, step=1)
>>> data_df.columns
Index(['country', 'density', 'median_age'], dtype='object')
```

Here, it shows the index starting at 0 and stopping at 3 with the step used each time, it also shows the columns as a list of each name. We can change the index of a data frame as follows:

```
>>> data_df.index = ["a","b","c"]
>>> data_df
             country  density  median_age
a     United Kingdom      281          40
b             France      119          42
c              Italy      206          46
```

Now if we want to access the first element of the country column we do so as follows:

```
>>> data_df["country"]["a"]
'United Kingdom'
```

Similarly if we want to change the column names of a data frame we do so as follows:

```
>>> data_df.columns = ["country_name","density","median_age"]
>>> data_df
      country_name  density  median_age
a   United Kingdom      281          40
b           France      119          42
c            Italy      206          46
```

Given we have changed the index to strings, the question is how do we access the nth row if we don't know what the index is. Luckily there is a method of data frames called iloc which allow us to access the nth row by just passing in the number of the row that we want. It works as follows:

```
>>> data_df.iloc[0]
country_name    United Kingdom
density                    281
median_age                  40
Name: a, dtype: object
>>> data_df.iloc[0:1]
      country_name  density  median_age
a   United Kingdom      281          40
>>> data_df.iloc[0:2]
      country_name  density  median_age
a   United Kingdom      281          40
b           France      119          42
>>> data_df.iloc[-1]
country_name    Italy
density           206
median_age         46
Name: c, dtype: object
```

We can see we can access rows from the data frame as if it was a list, which is cool.

Now that we have a grasp of DataFrames we will cover how to add to one. Let's say we want to add a column of all ones to our DataFrames

we can do so as follows:

```
>>> data_df["ones"] =1
>>> data_df
      country_name  density  median_age  ones
a  United Kingdom       281          40     1
b          France       119          42     1
c           Italy       206          46     1
```

We can then delete a column in a couple of ways:

```
>>> del data_df["ones"]
>>> data_df
      country_name  density  median_age
a   United Kingdom      281          40
b           France      119          42
c            Italy      206          46
>>> data_df["ones"] =1
>>> data_df
      country_name  density  median_age  ones
a   United Kingdom      281          40     1
b           France      119          42     1
c            Italy      206          46     1
>>> data_df.pop("ones")
a    1
b    1
c    1
Name: ones, dtype: int64
>>> data_df
      country_name  density  median_age
a   United Kingdom      281          40
b           France      119          42
c            Italy      206          46
```

Here, we first used the del method to delete the ones column, we then added it again and then used the pop method to remove the column. Note that when we use the del method we simply delete from the DataFrame but using the pop method we return the column we have popped as well as removing it from the DataFrame.

```
>>> data_df["ones"] = 1
>>> data_df["new_ones"] = data_df["ones"][1:2]
>>> data_df
     country_name  density  median_age  ones  new_ones
a   United Kingdom     281          40     1       NaN
b           France     119          42     1       1.0
c            Italy     206          46     1       NaN
>>> del data_df["ones"]
>>> del data_df["new_ones"]
```

What we see here is that when we use a partial column to form a new one pandas knows to fill in the gaps with the NaN value. There is another approach where we can insert a column and put it in a specific position:

```
>>> data_df.insert(1, "twos", 2)
>>> data_df
     country_name  twos  density  median_age
a   United Kingdom     2      281          40
b           France     2      119          42
c            Italy     2      206          46
>>> del data_df["twos"]
```

Here, we create a column containing the integer value 2 and puts it into position 1 (remember position 0 is the first position) under the title twos. This gives us full control of how we add to the data frame.

So, now we have a grasp of what a DataFrame is we can start doing some cool things to it. Let's say we want to take all data where the

value is less than 20.

```
>>> data_df['density']<200
a      False
b       True
c      False
Name: density, dtype: bool
>>> data_df[data_df['density']<200]
   country_name  density  median_age
b        France      119          42
```

What we have done here is test the values in data_df values column to see which ones are less than 20. This concept is a pretty key, we test every element in the column to see which ones are less than 20 and return a boolean column to show which ones meet the criteria. We can then pass this into the square brackets around a DataFrame and it returns the values where the condition is true. We can do this on multiple boolean statements where anything true across all the statements is returned, this is shown below.

```
>>> data_df[data_df['density']<250]
   country_name  density  median_age
b         France      119          42
c          Italy      206          46
>>> data_df["median_age"] > 42
a      False
b      False
c       True
Name: median_age, dtype: bool
>>> data_df[(data_df['density']<250)
            & (data_df["median_age"]>42)]
   country_name  density  median_age
c          Italy      206          46
```

It is important to note that the DataFrame isn't changed in this instance it stays the same. To use the DataFrame that is returned from such an operation you need to assign it to a variable to use later.

```
>>> (data_df['density']<250) & (data_df["median_age"]>42)
a      False
b      False
c       True
dtype: bool
>>> data_df['test'] = (data_df['density']<250)
                      & (data_df["median_age"]>42)
>>> data_df
     country_name  density  median_age    test
a  United Kingdom      281          40   False
b          France      119          42   False
c           Italy      206          46    True
>>> del data_df['test']
```

Here, we have used the same test as used in the previous example and assigned it to a column which is now part of the DataFrame. We could do the same thing if we wanted to create another column that uses the data in data frame.

```
>>> data_df["density"].sum()
606
>>> data_df['density_proportion'] = data_df['density']/data_df['density'].sum()
>>> data_df
    country_name  density  median_age   test  density_proportion
a   United Kingdom    281          40  False            0.463696
b           France    119          42  False            0.196370
c            Italy    206          46   True            0.339934
```

Here, we have divided all values in the value column with the sum of all the values in the column which we can see is 606 to give us a new column of data. We can also perform standard mathematical operations to a column. Below we use the numpy exponential function to exponentiate every element of the column:

```
>>> import numpy as np
>>> np.exp(data_df["density"])
a    1.088302e+122
b     4.797813e+51
c     2.915166e+89
Name: density, dtype: float64
```

We can also loop across a data frame as we have seen before with lists.

```
>>> for df in data_df:
...         df
...
'country_name'
'density'
'median_age'
'density_proportion'
```

This isn't exactly what we thought we would get as it only loops across the column names, we really want to get into the meat of the data frame to that we have to introduce the concept of transpose.

```
>>> data_df.T
                                    a           b           c
country_name          United Kingdom      France       Italy
density                          281         119         206
median_age                        40          42          46
density_proportion          0.463696     0.19637    0.339934
```

What we have done here is turn the DataFrame the other way so now the columns are the index. To loop over it we use the method iteritems method.

```
>>> for df in data_df.T.iteritems():
...     df
...
('a', country_name           United Kingdom
density                                  281
median_age                                40
density_proportion                  0.463696
Name: a, dtype: object)
('b', country_name                   France
density                             119
median_age                           42
density_proportion      0.19637
Name: b, dtype: object)
('c', country_name                    Italy
density                             206
median_age                           46
density_proportion      0.339934
Name: c, dtype: object)
```

What we see here is that when we use the iteritems method over the
data frame and at each instance of the loop it returns a two element
tuple. The first element is the index and the second the values in the
row stored in a series. The better way to access it would be to assign
each element to a variable allowing us to have better access to each
part.

```
>>> for ind, row in data_df.T.iteritems():
...        ind
...        row['country_name']
...
'a'
'United Kingdom'
'b'
'France'
'c'
'Italy'
```

We assign the first element of the tuple to the variable ind and the series of the row in the variable row. Then we access the country column of that row and show it here with the index. Also we can avoid using the transpose of the data frame by directly accessing the row via the iterrows method.

```
>>> for ind, row in data_df.iterrows():
...     ind
...     row
...
'a'
country_name        United Kingdom
density                        281
median_age                      40
density_proportion        0.463696
Name: a, dtype: object
'b'
country_name                France
density                        119
median_age                      42
density_proportion         0.19637
Name: b, dtype: object
'c'
country_name                 Italy
density                        206
median_age                      46
density_proportion        0.339934
Name: c, dtype: object
```

We have looked at how to add columns to a data frame but now we will look at how to add rows. The way we will consider is using the append method on data frames. We do so as follows:

```
>>> data = [{"country": "United Kingdom", "median_age":40, "density": 281},
            {"country": "France", "median_age": 42, "density": 119},
            {"country":"Italy", "median_age": 46, "density":206}]
>>> data_df = pd.DataFrame(data)
>>> data_df
        country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
>>> new_row = [{"country": "Iceland", "median_age": 37, "density": 3}]
>>> new_row_data_df = pd.DataFrame(new_row)
>>> new_row_data_df
   country  density  median_age
0  Iceland        3          37
>>> data_df.append(new_row_data_df)
        country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
0         Iceland        3          37
>>> data_df
        country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
```

So we setup the initial data as we have done earlier but here make a
fresh copy of the original data. We then setup a DataFrame of the
new row and pass that into the append method of the original data
frame. What we then see is the DataFrame containing the new row
however it has an index of zero which we already had in the original
DataFrame. We also see that when we call the DataFrame after this
operation it no longer has the new row. If we look at the index prob-
lem we can resolve this by using the argument ignore_index as
follows:

```
>>> data_df.append(new_row_data_df, ignore_index=True)
          country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
3         Iceland        3          37
>>> data_df
          country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
```

So that is sorted but what about the fact that the new row hasn't become part of the data frame. Well to get that to work we need to assign the data frame to a new variable as the append method doesn't change the original DataFrame. We could re-assign the data frame to the same name data_df, however we would lose the memory of what we have done so we could assign it to a new variable.

```
>>> new_data_df = data_df.append(new_row_data_df, ignore_index=True)
>>> new_data_df
          country  density  median_age
0  United Kingdom      281          40
1          France      119          42
2           Italy      206          46
3         Iceland        3          37
```

16.4 Merge, Join, and Concatenation

Initially, we will consider the concept of concatenating DataFrames. The manner in which we can do this is to create a list of DataFrames

and pass them into the pd.concat method. These DataFrames will build on what we have looked at in the previous chapter by using the following country data:

- density
- median age
- population (millions)
- population change (%)

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                     index=['France', 'Italy', 'Germany', 'Spain'])
>>> df2 = pd.DataFrame({"density": [153, 464, 36, 25],
...                     "median_age": [38, 28, 38, 33],
...                     "population": [1439, 1380, 331, 212],
...                     "population_change": [0.39, 0.99, 0.59, 0.72]},
...                     index=['China', 'India', 'USA', 'Brazil'])
>>> df3 = pd.DataFrame({"density": [9, 66, 347, 103],
...                     "median_age": [40, 29, 48, 25],
...                     "population": [145, 128, 126, 102],
...                     "population_change": [0.04, 1.06, -0.30, 1.94]},
...                     index=['Russia', 'Mexico', 'Japan', 'Egypt'])
>>> frames = [df1, df2, df3]
>>> result = pd.concat(frames)
>>> result
         density  median_age  population  population_change
France       119          42          65               0.22
Italy        206          47          60              -0.15
Germany      240          46          83               0.32
Spain         94          45          46               0.04
China        153          38        1439               0.39
India        464          28        1380               0.99
USA           36          38         331               0.59
Brazil        25          33         212               0.72
Russia         9          40         145               0.04
Mexico        66          29         128               1.06
Japan        347          48         126              -0.30
Egypt        103          25         102               1.94
```

What we did was to create a list of DataFrames and then by passing
them into the pd.concat method we get the result shown which is
DataFrame with columns density, median_age, population, popula-
tion_change and rows indexed with country names. But what if we did
not have the index values as shown in the example:

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04],
...                     "country_name": ['France', 'Italy', 'Germany', 'Spain']})
>>> df2 = pd.DataFrame({"density": [153, 464, 36, 25],
...                     "median_age": [38, 28, 38, 33],
...                     "population": [1439, 1380, 331, 212],
...                     "population_change": [0.39, 0.99, 0.59, 0.72],
...                     "country_name": ['China', 'India', 'USA', 'Brazil']})
>>> df3 = pd.DataFrame({"density": [9, 66, 347, 103],
...                     "median_age": [40, 29, 48, 25],
...                     "population": [145, 128, 126, 102],
...                     "population_change": [0.04, 1.06, -0.30, 1.94],
...                     "country_name": ['Russia', 'Mexico', 'Japan', 'Egypt']})
>>> frames = [df1, df2, df3]
>>> result = pd.concat(frames)
>>> result
   density  median_age  population  population_change country_name
0      119          42          65               0.22       France
1      206          47          60              -0.15        Italy
2      240          46          83               0.32      Germany
3       94          45          46               0.04        Spain
0      153          38        1439               0.39        China
1      464          28        1380               0.99        India
2       36          38         331               0.59          USA
3       25          33         212               0.72       Brazil
0        9          40         145               0.04       Russia
1       66          29         128               1.06       Mexico
2      347          48         126              -0.30        Japan
3      103          25         102               1.94        Egypt
```

Here, we see that the index is retained for each DataFrame which when created all have the index 0, 1, 2, 3. To have an index 0–11 we need to use the ignore_index argument and set it to True.

```
>>> result = pd.concat(frames, ignore_index=True)
>>> result
    density   median_age   population   population_change   country_name
0       119           42           65                0.22         France
1       206           47           60               -0.15          Italy
2       240           46           83                0.32        Germany
3        94           45           46                0.04          Spain
4       153           38         1439                0.39          China
5       464           28         1380                0.99          India
6        36           38          331                0.59            USA
7        25           33          212                0.72         Brazil
8         9           40          145                0.04         Russia
9        66           29          128                1.06         Mexico
10      347           48          126               -0.30          Japan
11      103           25          102                1.94          Egypt
```

We can expand on this example by creating a list of DataFrames as we did previously and concat them together but now we use the argument keys and set it to a list containing region one, region two, and region three.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04],
...                     "country_name": ['France', 'Italy', 'Germany', 'Spain']})
>>> df2 = pd.DataFrame({"density": [153, 464, 36, 25],
...                     "median_age": [38, 28, 38, 33],
...                     "population": [1439, 1380, 331, 212],
...                     "population_change": [0.39, 0.99, 0.59, 0.72],
...                     "country_name": ['China', 'India', 'USA', 'Brazil']})
>>> df3 = pd.DataFrame({"density": [9, 66, 347, 103],
...                     "median_age": [40, 29, 48, 25],
...                     "population": [145, 128, 126, 102],
...                     "population_change": [0.04, 1.06, -0.30, 1.94],
...                     "country_name": ['Russia', 'Mexico', 'Japan', 'Egypt']})
>>> frames = [df1, df2, df3]
>>> result = pd.concat(frames, keys=["region_one", "region_two", "region_three"])
>>> result
                   density  median_age  population  population_change country_name
region_one    0        119          42          65               0.22       France
              1        206          47          60              -0.15        Italy
              2        240          46          83               0.32      Germany
              3         94          45          46               0.04        Spain
region_two    0        153          38        1439               0.39        China
              1        464          28        1380               0.99        India
              2         36          38         331               0.59          USA
              3         25          33         212               0.72       Brazil
region_three  0          9          40         145               0.04       Russia
              1         66          29         128               1.06       Mexico
              2        347          48         126              -0.30        Japan
              3        103          25         102               1.94        Egypt
>>> result.loc['region_two']
   density  median_age  population  population_change country_name
0      153          38        1439               0.39        China
1      464          28        1380               0.99        India
2       36          38         331               0.59          USA
3       25          33         212               0.72       Brazil
```

In running the code what we see is that passing the keys in means we have what appears to be another level of the DataFrame away from our index in the previous example which allows us to select the one of the DataFrames used in the concat. If we look at the index of the result we get the following:

```
>>> result.index
MultiIndex(levels=[['region_one', 'region_two', 'region_three'],
                  [0, 1, 2, 3]],
          codes=[[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2],
                 [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]])
```

This is commonly referred to a multilevel index as the name would suggest and what it does is tell us what the index value each element has. So the levels are ["region_one", "region_two", "region_three"] and [0, 1, 2, 3], which are denoted in levels. The index for each row is then determined using the label which has two lists of eight elements with the first one having values 0, 1, 2 which corresponds to region one, region two and region three whilst the second has values 0, 1, 2, 3 which refer to the levels 0, 1, 2, 3. We could name these levels by using the optional name argument.

```
>>> result = pd.concat(frames, keys=["region_one", "region_two",
"region_three"], names = ["region","item"])
>>> result
                density  median_age  population  population_change country_name
region        item
region_one    0      119         42          65               0.22       France
              1      206         47          60              -0.15        Italy
              2      240         46          83               0.32      Germany
              3       94         45          46               0.04        Spain
region_two    0      153         38        1439               0.39        China
              1      464         28        1380               0.99        India
              2       36         38         331               0.59          USA
              3       25         33         212               0.72       Brazil
region_three  0        9         40         145               0.04       Russia
              1       66         29         128               1.06       Mexico
              2      347         48         126              -0.30        Japan
              3      103         25         102               1.94        Egypt
```

In the previous example we used concat to concatenate the Data-Frames together however there are other ways to use it which we will demonstrate now by concatenating urban population percentage from France, Italy, Argentina, and Thailand to our initial DataFrame.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                    index=['France', 'Italy', 'Germany', 'Spain'])
>>> df4 = pd.DataFrame({"urban_population": [82, 69, 93, 51]},
...                    index=['France', 'Italy', 'Argentina', 'Thailand'])
>>> result = pd.concat([df1, df4], axis=1, sort=False)
>>> result
           density  median_age  population  population_change  urban_population
France       119.0        42.0        65.0               0.22              82.0
Italy        206.0        47.0        60.0              -0.15              69.0
Germany      240.0        46.0        83.0               0.32               NaN
Spain         94.0        45.0        46.0               0.04               NaN
Argentina      NaN         NaN         NaN                NaN              93.0
Thailand       NaN         NaN         NaN                NaN              51.0
```

Here, we have used concat with a list of DataFrames as we have done before but now we pass in the argument axis $= 1$. Now the axis argument says we concatenate on the columns, here 0 is index and 1 is columns. So, we see commonality in the index with France and Italy so we can add the extra column on and fill the values that are not common with NaN. Here, we have set the sort to be False which means we keep the order as if the two were joined one below the other. If we set the value to be True we get the following:

```
>>> result = pd.concat([df1, df4], axis=1, sort=True)
>>> result
           density  median_age  population  population_change  urban_population
Argentina    NaN        NaN         NaN             NaN              93.0
France     119.0       42.0        65.0            0.22              82.0
Germany    240.0       46.0        83.0            0.32               NaN
Italy      206.0       47.0        60.0           -0.15              69.0
Spain       94.0       45.0        46.0            0.04               NaN
Thailand     NaN        NaN         NaN             NaN              51.0
```

We can see that with sort set to True we get the values sorted by index order. Below we can also see what happens if we run the same query with the axis set to 0.

```
>>> result = pd.concat([df1, df4], axis=0, sort=False)
>>> result
           density  median_age  population  population_change  urban_population
France     119.0       42.0        65.0            0.22               NaN
Italy      206.0       47.0        60.0           -0.15               NaN
Germany    240.0       46.0        83.0            0.32               NaN
Spain       94.0       45.0        46.0            0.04               NaN
France       NaN        NaN         NaN             NaN              82.0
Italy        NaN        NaN         NaN             NaN              69.0
Argentina    NaN        NaN         NaN             NaN              93.0
Thailand     NaN        NaN         NaN             NaN              51.0
```

What we do is just concatenate the DataFrames one below each other with duplication of the index for France and Italy.

Concat also has an extra argument join that we will now explore and set the value to join.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                      index=['France', 'Italy', 'Germany', 'Spain'])
>>> df4 = pd.DataFrame({"urban_population": [82, 69, 93, 51]},
...                      index=['France', 'Italy', 'Argentina', 'Thailand'])
>>> result = pd.concat([df1, df4], axis=1, join="inner")
>>> result
        density  median_age  population  population_change  urban_population
France      119          42          65               0.22                82
Italy       206          47          60              -0.15                69
```

As you can see we only have two rows returned which if you look back at the example before are the only two rows where the two DataFrames have values in columns. The inner join is similar to that of a database join which we will cover later in the course however here we don't specify a key to use it on.

Next, we add argument join_axes and set it to df1.index.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                      index=['France', 'Italy', 'Germany', 'Spain'])
>>> df4 = pd.DataFrame({"urban_population": [82, 69, 93, 51]},
...                      index=['France', 'Italy', 'Argentina', 'Thailand'])
>>> result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
>>> result
        density  median_age  population  population_change  urban_population
France      119          42          65               0.22              82.0
Italy       206          47          60              -0.15              69.0
Germany     240          46          83               0.32               NaN
Spain        94          45          46               0.04               NaN
```

What we see is that all we get back only the values for in the index in df1 and show all the columns from the axis 1 argument. By default the join_axes are set to False.

Next, we will ignore the index by using the following arguments:

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                    index=['France', 'Italy', 'Germany', 'Spain'])
>>> df4 = pd.DataFrame({"urban_population": [82, 69, 93, 51]},
...                    index=['France', 'Italy', 'Argentina', 'Thailand'])
>>> result = pd.concat([df1, df4], ignore_index=True, sort=True)
>>> result
   density  median_age  population  population_change  urban_population
0    119.0        42.0        65.0               0.22               NaN
1    206.0        47.0        60.0              -0.15               NaN
2    240.0        46.0        83.0               0.32               NaN
3     94.0        45.0        46.0               0.04               NaN
4      NaN         NaN         NaN                NaN              82.0
5      NaN         NaN         NaN                NaN              69.0
6      NaN         NaN         NaN                NaN              93.0
7      NaN         NaN         NaN                NaN              51.0
```

Here, we see the result has lost index values from df1 and df2 and retained all the information filling the missing values with NaN.

We can achieve the same thing using the append method directly on a DataFrame.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                     index=['France', 'Italy', 'Germany', 'Spain'])
>>> df4 = pd.DataFrame({"urban_population": [82, 69, 93, 51]},
...                     index=['France', 'Italy', 'Argentina', 'Thailand'])
>>> result = df1.append(df4, ignore_index=True, sort=True)
>>> result
   density  median_age  population  population_change  urban_population
0    119.0        42.0        65.0               0.22               NaN
1    206.0        47.0        60.0              -0.15               NaN
2    240.0        46.0        83.0               0.32               NaN
3     94.0        45.0        46.0               0.04               NaN
4      NaN         NaN         NaN                NaN              82.0
5      NaN         NaN         NaN                NaN              69.0
6      NaN         NaN         NaN                NaN              93.0
7      NaN         NaN         NaN                NaN              51.0
```

The concat method is not only valid for DataFrames but can also work on Series.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                     index=['France', 'Italy', 'Germany', 'Spain'])
>>> s1 = pd.Series([82, 69, 93, 51],
...                index=['France', 'Italy', 'Germany', 'Spain'],
...                name="urban_population")
>>> s1
France     82
Italy      69
Germany    93
Spain      51
Name: urban_population, dtype: int64
>>> result = pd.concat([df1, s1], axis=1)
>>> result
         density  median_age  population  population_change  urban_population
France       119          42          65               0.22                82
Italy        206          47          60              -0.15                69
Germany      240          46          83               0.32                93
Spain         94          45          46               0.04                51
```

What is worth noting is that we give the series a name and then that is set to be the name of the column when the two are concatenated together. We could also pass in multiple series in the list and we will add a second series with world share percentage.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                    index=['France', 'Italy', 'Germany', 'Spain'])
>>> s1 = pd.Series([82, 69, 93, 51],
...                index=['France', 'Italy', 'Germany', 'Spain'],
...                name="urban_population")
>>> s2 = pd.Series([0.84, 0.78, 1.07, 0.60],
...                index=['France', 'Italy', 'Germany', 'Spain'],
...                name="world_share")
>>> s2
France     0.84
Italy      0.78
Germany    1.07
Spain      0.60
Name: world_share, dtype: float64
>>> result = pd.concat([df1, s1, s2], axis=1)
>>> result
         density  median_age  ...  urban_population  world_share
France       119          42  ...                82         0.84
Italy        206          47  ...                69         0.78
Germany      240          46  ...                93         1.07
Spain         94          45  ...                51         0.60

[4 rows x 6 columns]
```

Next, we pass in series as a list to create a DataFrame and by specifying keys we can rename the columns.

```
>>> s1 = pd.Series([82, 69, 93, 51],
...                         index=['France', 'Italy', 'Germany', 'Spain'],
...                         name="urban_population")
>>> s2 = pd.Series([0.84, 0.78, 1.07, 0.60],
...                         index=['France', 'Italy', 'Germany', 'Spain'],
...                         name="world_share")
>>> pd.concat([s1, s2], axis=1, keys=["urban_population","world_share"])
         urban_population  world_share
France                 82         0.84
Italy                  69         0.78
Germany                93         1.07
Spain                  51         0.60
```

Next, we take our three DataFrames from before and assign them to a dictionary each with a key. The dictionary is then passed into concat.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                     index=['France', 'Italy', 'Germany', 'Spain'])
>>> df2 = pd.DataFrame({"density": [153, 464, 36, 25],
...                     "median_age": [38, 28, 38, 33],
...                     "population": [1439, 1380, 331, 212],
...                     "population_change": [0.39, 0.99, 0.59, 0.72]},
...                     index=['China', 'India', 'USA', 'Brazil'])
>>> df3 = pd.DataFrame({"density": [9, 66, 347, 103],
...                     "median_age": [40, 29, 48, 25],
...                     "population": [145, 128, 126, 102],
...                     "population_change": [0.04, 1.06, -0.30, 1.94]},
...                     index=['Russia', 'Mexico', 'Japan', 'Egypt'])
>>> pieces = {"region_one": df1, "region_two": df2, "region_three": df3}
>>> result = pd.concat(pieces)
>>> result
```

		density	median_age	population	population_change
region_one	France	119	42	65	0.22
	Italy	206	47	60	-0.15
	Germany	240	46	83	0.32
	Spain	94	45	46	0.04
region_three	Russia	9	40	145	0.04
	Mexico	66	29	128	1.06
	Japan	347	48	126	-0.30
	Egypt	103	25	102	1.94
region_two	China	153	38	1439	0.39
	India	464	28	1380	0.99
	USA	36	38	331	0.59
	Brazil	25	33	212	0.72

In using a dictionary we automatically create a DataFrame with a multilevel index where the first level is the key of the dictionary and the second level the index of the DataFrame. We next do exactly the same but here pass in an optional keys list.

```
>>> df1 = pd.DataFrame({"density": [119, 206, 240, 94],
...                     "median_age": [42, 47, 46, 45],
...                     "population": [65, 60, 83, 46],
...                     "population_change": [0.22, -0.15, 0.32, 0.04]},
...                    index=['France', 'Italy', 'Germany', 'Spain'])
>>> df2 = pd.DataFrame({"density": [153, 464, 36, 25],
...                     "median_age": [38, 28, 38, 33],
...                     "population": [1439, 1380, 331, 212],
...                     "population_change": [0.39, 0.99, 0.59, 0.72]},
...                    index=['China', 'India', 'USA', 'Brazil'])
>>> df3 = pd.DataFrame({"density": [9, 66, 347, 103],
...                     "median_age": [40, 29, 48, 25],
...                     "population": [145, 128, 126, 102],
...                     "population_change": [0.04, 1.06, -0.30, 1.94]},
...                    index=['Russia', 'Mexico', 'Japan', 'Egypt'])
>>> pieces = {"region_one": df1, "region_two": df2, "region_three": df3}
>>> result = pd.concat(pieces, keys=["region_two", "region_three"])
>>> result
                       density  median_age  population  population_change
region_two    China        153          38        1439               0.39
              India        464          28        1380               0.99
              USA           36          38         331               0.59
              Brazil        25          33         212               0.72
region_three  Russia         9          40         145               0.04
              Mexico        66          29         128               1.06
              Japan        347          48         126              -0.30
              Egypt        103          25         102               1.94
```

Having looked at the concat and append methods we now consider
how pandas deals with database styles merging this is all done via
the merge method. We will explain the specifics around each join
type by example, however it is worth explaining the basics of data-
base joins. So when we speak of database style joins we mean the
mechanism to join together tables via common values. The way in
which the tables will look will depend on the type of join with us show-
ing examples for inner, outer, right, and left joins.

We will develop the example of country data to combine DataFrames that contain data relating to common countries and now add in the data for the countries relating to the percentage world share.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, on="country")
>>> result
   density  median_age  population  population_change  country  world_share
0      119          42          65               0.22   France         0.84
1      206          47          60              -0.15    Italy         0.78
2      240          46          83               0.32  Germany         1.07
3       94          45          46               0.04    Spain         0.60
```

In this example, we join two DataFrames on a common key which in this case is the country name. The result is a DataFrame with only one country column where both left and right are merged.

In the next example, we look at the merge method with a left and right DataFrame but this time will have two keys to join on which will be passed in as a list to the on argument. This allows us to join on multiple values being the same.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, on=["country", "population"])
>>> result
   density  median_age  population  population_change country  world_share
0      119          42          65               0.22  France         0.84
1      206          47          60              -0.15   Italy         0.78
2       94          45          46               0.04   Spain         0.60
```

Here, we have joined on country and population and the resulting DataFrame is where both DataFrames share the same country and population. So we lose one row of data from each DataFrame where we do not share the population and country on both.

Next, we run the same code with an added argument which is how equal to left.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="left", on=["country", "population"])
>>> result
   density  median_age  population  population_change  country  world_share
0      119          42          65               0.22   France         0.84
1      206          47          60              -0.15    Italy         0.78
2      240          46          83               0.32  Germany          NaN
3       94          45          46               0.04    Spain         0.60
```

The result of this is what is known as a left join. So we retain all the information of the left DataFrame and only the elements from the right DataFrame with the same keys as the left one. In this case we retain all information from the left DataFrame.

Next we consider a right join using the same example as before.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="right", on=["country", "population"])
>>> result
   density  median_age  population  population_change  country  world_share
0    119.0        42.0          65               0.22   France         0.84
1    206.0        47.0          60              -0.15    Italy         0.78
2     94.0        45.0          46               0.04    Spain         0.60
3      NaN         NaN          85                NaN  Germany         1.07
```

Essentially this does the same as the left join, however its now the left DataFrame that is joined onto the right one which is the reverse of what we saw with the left join.

The next join to consider is the outer join and again for completeness we use the previous example to show how it works.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="outer", on=["country", "population"])
>>> result
   density  median_age  population  population_change  country  world_share
0    119.0        42.0          65               0.22   France         0.84
1    206.0        47.0          60              -0.15    Italy         0.78
2    240.0        46.0          83               0.32  Germany          NaN
3     94.0        45.0          46               0.04    Spain         0.60
4      NaN         NaN          85                NaN  Germany         1.07
```

With the outer join its a combination of both the left and right joins so we have more rows than are in each DataFrame as the join of left and right give different results so we need all of these in the outer join result.

The last how option we consider is the inner join.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
...                       "population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="inner", on=["country", "population"])
>>> result
   density  median_age  population  population_change country  world_share
0      119          42          65               0.22  France         0.84
1      206          47          60              -0.15   Italy         0.78
2       94          45          46               0.04   Spain         0.60
```

This join gives only the result where we have commonality on both the left and right DataFrame. This is also the default when we pass no how argument:

```
>>> result = pd.merge(left, right, on=["country", "population"])
>>> result
   density  median_age  population  population_change country  world_share
0      119          42          65               0.22  France         0.84
1      206          47          60              -0.15   Italy         0.78
2       94          45          46               0.04   Spain         0.60
```

Next, we join two DataFrames with columns population and country but we join only on country using an outer join.

```
>>> left = pd.DataFrame({"population": [65, 60, 83, 46],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"population": [65, 60, 85, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, on="country", how="outer")
>>> result
   population_x  country  population_y
0            65   France            65
1            60    Italy            60
2            83  Germany            85
3            46    Spain            46
```

What we see here is that if the columns are the same and not used in the join the names get changed. Here, we now have population_x and population_y which could be problematic if you are assuming to operate on the column population. This makes sense as we need a way to distinguish the two and pandas takes care of it for us.

Next, we do a merge using the indicator option set to True. Here, we have two DataFrames with only a single column to merge on which is country and we want to do an outer join:

```
>>> left = pd.DataFrame({"population": [65, 60, 83, 46],
...                       "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"population": [65, 60, 85, 46],
...                        "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, on="country", how="outer", indicator=True)
>>> result
   population_x  country  population_y  _merge
0           65   France            65    both
1           60    Italy            60    both
2           83  Germany            85    both
3           46    Spain            46    both
```

What the result shows is how the join is done index by index position so this could be left, right, or both. Here, we see that the join from one to the other is done on both.

The merge method is a pandas method to take account of two Data-Frames, however we can use a DataFrames join method to join one onto another.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                      index= ['France', 'Italy', 'Germany', 'Spain'])
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                       index = ['France', 'Italy', 'Germany',
...                       'United Kingdom'])
>>> result = left.join(right)
>>> result
         density  median_age  population  population_change  world_share
France       119          42          65               0.22         0.84
Italy        206          47          60              -0.15         0.78
Germany      240          46          83               0.32         1.07
Spain         94          45          46               0.04          NaN
```

What we see is that the left DataFrame is retained and we join the
right one where the keys in right match the keys in left. Like with the
merge we have the option how to join the DataFrames so we can
specify that like we have seen earlier. Using the same example previ-
ously we can show this.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                      index= ['France', 'Italy', 'Germany', 'Spain'])
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                       index = ['France', 'Italy', 'Germany',
...                       'United Kingdom'])
>>> result = left.join(right, how="outer")
>>> result
                density  median_age  population  population_change  world_share
France            119.0        42.0        65.0               0.22         0.84
Germany           240.0        46.0        83.0               0.32         1.07
Italy             206.0        47.0        60.0              -0.15         0.78
Spain              94.0        45.0        46.0               0.04          NaN
United Kingdom      NaN         NaN         NaN                NaN         0.60
```

In using the outer join we retain all the information from both Data-Frames as we have seen when using a merge and where there are not values in one of the DataFrames they are filled in using NaN. Using the same example with an inner join show the following results:

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                      index= ['France', 'Italy', 'Germany', 'Spain'])
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                      index = ['France', 'Italy', 'Germany',
...                      'United Kingdom'])
>>> result = left.join(right, how="inner")
>>> result
         density  median_age  population  population_change  world_share
France       119          42          65               0.22         0.84
Italy        206          47          60              -0.15         0.78
Germany      240          46          83               0.32         1.07
```

As expected the inner join just retains where the DataFrames have common data which here is for index France, Italy, and Germany.

We can achieve the same result without using a how if we pass in some different arguments to the merge method. These arguments are left_index and right_index here in setting them to True. We are getting the same behaviour as for the join method with how set to inner.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04]},
...                     index= ['France', 'Italy', 'Germany', 'Spain'])
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                      index = ['France', 'Italy', 'Germany',
...                      'United Kingdom'])
>>> result = pd.merge(left, right, left_index=True, right_index=True,)
>>> result
         density  median_age  population  population_change  world_share
France       119          42          65               0.22         0.84
Italy        206          47          60              -0.15         0.78
Germany      240          46          83               0.32         1.07
```

Next, we use the argument 'on' with the join method when applied to the left DataFrame.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                      index = ['France', 'Italy', 'Germany',
...                      'United Kingdom'])
>>> left
   density  median_age  population  population_change  country
0      119          42          65               0.22   France
1      206          47          60              -0.15    Italy
2      240          46          83               0.32  Germany
3       94          45          46               0.04    Spain
>>> right
                world_share
France                 0.84
Italy                  0.78
Germany                1.07
United Kingdom         0.60
>>> result = left.join(right, on="country")
>>> result
   density  median_age  population  population_change  country  world_share
0      119          42          65               0.22   France         0.84
1      206          47          60              -0.15    Italy         0.78
2      240          46          83               0.32  Germany         1.07
3       94          45          46               0.04    Spain          NaN
```

In specifying the on column which is country we join the index of right on this column and we see that we now have a DataFrame indexed by the first DataFrame. This type of approach is what you may see when using databases and you want to join on the id of the column on the respective value in another table. This example can be extended to multiple values in the on argument however to do this you would require multilevel indexes which will be covered later in the book. We can remove any NaN values by adding the how argument and setting it to inner doing an inner join as shown below.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
...                      "median_age": [42, 47, 46, 45],
...                      "population": [65, 60, 83, 46],
...                      "population_change": [0.22, -0.15, 0.32, 0.04],
...                      "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60]},
...                       index = ['France', 'Italy', 'Germany',
...                                'United Kingdom'])
>>> result = left.join(right, on="country", how="inner")
>>> result
   density  median_age  population  population_change  country  world_share
0      119          42          65               0.22   France         0.84
1      206          47          60              -0.15    Italy         0.78
2      240          46          83               0.32  Germany         1.07
```

The next thing we will consider is the important concept of missing data. We all hope to work with perfect datasets but the reality is we generally won't and having the ability to work with missing or bad data is an important one. Luckily pandas offers some great tools for dealing with this and we begin by showing how to identify where we have NaN in our dataset.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
                         "median_age": [42, 47, 46, 45],
                         "population": [65, 60, 83, 46],
                         "population_change": [0.22, -0.15, 0.32, 0.04],
                         "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
                          "population": [65, 60, 85, 46],
                          "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="outer", on=["country", "population"])
>>> result
   density  median_age  population  population_change  country  world_share
0    119.0        42.0          65               0.22   France         0.84
1    206.0        47.0          60              -0.15    Italy         0.78
2    240.0        46.0          83               0.32  Germany          NaN
3     94.0        45.0          46               0.04    Spain         0.60
4      NaN         NaN          85                NaN  Germany         1.07
>>> pd.isna(result['density'])
0    False
1    False
2    False
3    False
4     True
Name: density, dtype: bool
>>> result['median_age'].notna()
0     True
1     True
2     True
3     True
4    False
Name: median_age, dtype: bool
>>> result.isna()
   density  median_age  population  population_change  country  world_share
0    False       False       False              False    False        False
1    False       False       False              False    False        False
2    False       False       False              False    False         True
3    False       False       False              False    False        False
4     True        True       False               True    False        False
```

Here, we have taken the DataFrames we have seen before and created a result DataFrame using the merge method with how set to outer. What this has done is given us a DataFrame with NaN values and we can now demonstrate how you can find where these values are

within your DataFrame. We first consider the pandas isna method on a column of the DataFrame which tests each element to see what is and what isn't NaN. To achieve the same thing we can use the notna() method on a column or all of our DataFrame, or we could use isna() method which does the opposite of notna(). This makes it very easy to determine what is and what isn't NaN in our DataFrame.

```
>>> left = pd.DataFrame({"density": [119, 206, 240, 94],
                         "median_age": [42, 47, 46, 45],
                         "population": [65, 60, 83, 46],
                         "population_change": [0.22, -0.15, 0.32, 0.04],
                         "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> right = pd.DataFrame({"world_share": [0.84, 0.78, 1.07, 0.60],
                          "population": [65, 60, 85, 46],
                          "country": ['France', 'Italy', 'Germany', 'Spain']})
>>> result = pd.merge(left, right, how="outer", on=["country", "population"])
>>> result
   density  median_age  population  population_change  country  world_share
0    119.0        42.0          65               0.22   France         0.84
1    206.0        47.0          60              -0.15    Italy         0.78
2    240.0        46.0          83               0.32  Germany          NaN
3     94.0        45.0          46               0.04    Spain         0.60
4      NaN         NaN          85                NaN  Germany         1.07
>>> result['density'].dropna()
0    119.0
1    206.0
2    240.0
3     94.0
Name: density, dtype: float64
>>> result.dropna()
   density  median_age  population  population_change country  world_share
0    119.0        42.0          65               0.22  France         0.84
1    206.0        47.0          60              -0.15   Italy         0.78
3     94.0        45.0          46               0.04   Spain         0.60
>>> result[result['density'].notna()]
   density  median_age  population  population_change  country  world_share
0    119.0        42.0          65               0.22   France         0.84
1    206.0        47.0          60              -0.15    Italy         0.78
2    240.0        46.0          83               0.32  Germany          NaN
3     94.0        45.0          46               0.04    Spain         0.60
```

Taking the example one step further we can drop values from a column of the whole DataFrame by using the dropna method. For the column we only drop the one value that is NaN, however across the whole DataFrame we remove any row that has NaN in it. This may not be ideal and instead we may want to remove the row where one column has NaN and we can do that by passing and columns notna to the whole DataFrame.

16.5 **DataFrame Methods**

Now in the next example we will show some of the methods we can apply to a DataFrame. Earlier we demonstrated the sum method, however pandas has lots more to offer and we will look at some of the more common mathematical ones. Here, we import the package seaborn and load the iris dataset that comes with it giving us the data in a DataFrame.

```
>>> import seaborn as sns
>>> iris = sns.load_dataset('iris')
>>> iris.head()
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
>>> iris.tail()
     sepal_length  sepal_width  petal_length  petal_width    species
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica
>>> iris.head(10)
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
5           5.4          3.9           1.7          0.4  setosa
6           4.6          3.4           1.4          0.3  setosa
7           5.0          3.4           1.5          0.2  setosa
8           4.4          2.9           1.4          0.2  setosa
9           4.9          3.1           1.5          0.1  setosa
>>> iris.columns
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
       'species'],
      dtype='object')
```

Following importing the package and the iris data we can access the top of the DataFrame by using head which by default gives us the top five rows, we can use the tail method to get the bottom five rows. We can get a defined number of rows by just passing the number into the

head or tail method and if we want just the columns back we can use the columns method.

Having imported and accessed the data we now demonstrate some methods which we can apply.

```
>>> iris.count()
sepal_length      150
sepal_width       150
petal_length      150
petal_width       150
species           150
dtype: int64
>>> iris.count().sepal_length
150
>>> iris['sepal_length'].count()
150
>>> len(iris)
150
```

We can apply count to both the DataFrame and the column. When applied to the DataFrame we return the length of each column. We can also get the specific column length by either using the column name on the end of the count method or by accessing the column and then applying the count method. If you want the number of rows in the DataFrame as a whole you can use the len method on DataFrame.

```
>>> iris.corr()
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      1.000000    -0.117570      0.871754     0.817941
sepal_width      -0.117570     1.000000     -0.428440    -0.366126
petal_length      0.871754    -0.428440      1.000000     0.962865
petal_width       0.817941    -0.366126      0.962865     1.000000
>>> iris.corr()['petal_length']
sepal_length     0.871754
sepal_width     -0.428440
petal_length     1.000000
petal_width      0.962865
Name: petal_length, dtype: float64
>>> iris.corr()['petal_length']['sepal_length']
0.8717537758865828
>>> iris.cov()
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      0.685694    -0.042434      1.274315     0.516271
sepal_width      -0.042434     0.189979     -0.329656    -0.121639
petal_length      1.274315    -0.329656      3.116278     1.295609
petal_width       0.516271    -0.121639      1.295609     0.581006
>>> iris.cov()['sepal_length']
sepal_length     0.685694
sepal_width     -0.042434
petal_length     1.274315
petal_width      0.516271
Name: sepal_length, dtype: float64
>>> iris.cov()['sepal_length']['sepal_width']
-0.04243400447427296
```

The corr method applied to the DataFrame gives us the correlation between each variable and we can limit that to one columns correlation with all others by passing the column name or get the correlation between two columns by passing both column names. You can also see the same applies with the cov method which calculates the covariance between variables.

```
>>> iris.cumsum().head()
  sepal_length sepal_width  ... petal_width                            species
0          5.1         3.5  ...         0.2                             setosa
1           10         6.5  ...         0.4                       setosasetosa
2         14.7         9.7  ...         0.6                 setosasetosasetosa
3         19.3        12.8  ...         0.8           setosasetosasetosasetosa
4         24.3        16.4  ...           1     setosasetosasetosasetosasetosa
>>> iris.columns
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
...         'species'], dtype='object')
>>> iris.cumsum()[['sepal_length', 'sepal_width', 'petal_length',
...         'petal_width']].tail()
    sepal_length sepal_width petal_length petal_width
145        851.6       446.7          543       171.9
146        857.9       449.2          548       173.8
147        864.4       452.2        553.2       175.8
148        870.6       455.6        558.6       178.1
149        876.5       458.6        563.7       179.9
```

Next, we consider the cumsum method. This provides the cumulative sum as the columns ascend. Now for those columns of numeric type the value ascends as expected with the current value added to the previous value and so on to create an increasing value. The difference comes when we consider a character-based column. The cumulative value here is just the concatenation of the values together with the results looking very strange. To make things easier to read we can restrict what we show for the return of the method by specifying a list of the columns to show and as you can see we can even chain the tail command on the end.

```
>>> iris.describe()
       sepal_length   sepal_width   petal_length   petal_width
count    150.000000    150.000000     150.000000    150.000000
mean       5.843333      3.057333       3.758000      1.199333
std        0.828066      0.435866       1.765298      0.762238
min        4.300000      2.000000       1.000000      0.100000
25\%       5.100000      2.800000       1.600000      0.300000
50\%       5.800000      3.000000       4.350000      1.300000
75\%       6.400000      3.300000       5.100000      1.800000
max        7.900000      4.400000       6.900000      2.500000
>>> iris.sepal_length.describe()
count    150.000000
mean       5.843333
std        0.828066
min        4.300000
25\%       5.100000
50\%       5.800000
75\%       6.400000
max        7.900000
Name: sepal_length, dtype: float64
```

Above we use the describe method which gives us a number of values namely the count, mean, standard deviation, minimum, maximum and the 25, 50, and 75 percentiles. This method only works on columns with the type to calculate the values so we not the column species is not included. We can also use this on individual columns, the manner in which we have done this in the example is not to use the square bracket method to accessing a column but instead the dot approach where we can use dot and the column name to access the value and then chain the describe method on the end.

```
>>> iris.max()
sepal_length               7.9
sepal_width                4.4
petal_length               6.9
petal_width                2.5
species              virginica
dtype: object
>>> iris.sepal_length.max()
7.9
```

Next, we consider the max value. Here, when applied to the entire DataFrame we get the max of every column where a maximum value can be obtained. We also show that we can apply the method on a column in the same manner as we showed in the previous example.

```
>>> iris.sepal_width.mean()
3.057333333333334
>>> iris.mean(0)
sepal_length    5.843333
sepal_width     3.057333
petal_length    3.758000
petal_width     1.199333
dtype: float64
>>> iris.mean(1).head()
0    2.550
1    2.375
2    2.350
3    2.350
4    2.550
dtype: float64
>>> iris.mean(1).tail()
145    4.300
146    3.925
147    4.175
148    4.325
149    3.950
dtype: float64
```

The next method we look at is the mean which is a common calcula-
tion that you may want to make and as before we can apply it on an
individual column and we have done so here using the dot syntax.
We then apply the mean method but now pass in a 0 or 1 referring to
whether we want to apply across columns or rows. There are a num-
ber of different methods that you can apply to a DataFrame and a list
of some of the more useful ones is given below.

- median: returns the arithmetic median

- min: returns the minimum value

- max: returns the maximum value

- mode: returns the most frequent number

- std: returns the standard deviation

- sum: returns the arithmetic sum

- var: returns the variance

These are demonstrated as follows:

```
>>> import seaborn as sns
>>> iris = sns.load_dataset('iris')
>>> iris.sepal_length.median()
5.8
>>> iris.sepal_length.min()
4.3
>>> iris.sepal_length.mode()
0    5.0
dtype: float64
>>> iris.sepal_length.max()
7.9
>>> iris.sepal_length.std()
0.8280661279778629
>>> iris.sepal_length.sum()
876.5
>>> iris.sepal_length.var()
0.6856935123042505
```

16.6 Missing Data

We next consider methods we can apply across the DataFrame and how missing data is dealt with. Here, we set the DataFrame up in the way we have done so far in the section and introduce some NaN entries into the DataFrame.

```
>>> data = pd.DataFrame({"A": [1, 2.1, np.nan, 4.7, 5.6, 6.8],
...                      "B": [.25, np.nan, np.nan, 4, 12.2, 14.4]})
>>> data
     A      B
0  1.0   0.25
1  2.1    NaN
2  NaN    NaN
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
>>> data.dropna(axis=0)
>>> data.dropna(axis=1)
>>> data.where(pd.notna(data), data.mean(), axis="columns")
>>> data.fillna(data.mean()["B":"C"])
>>> data.fillna(data.mean())
>>> data_2.fillna(method="pad")
>>> data_2.fillna(method="bfill")
>>> data.interpolate()
     A      B
0  1.0   0.25
1  2.1   1.50
2  3.4   2.75
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
>>> data.interpolate(method="barycentric")
      A       B
0  1.00   0.250
1  2.10  -7.660
2  3.53  -4.515
3  4.70   4.000
4  5.60  12.200
5  6.80  14.400
>>> data.interpolate(method="pchip")
        A          B
0  1.00000   0.250000
1  2.10000   0.672808
2  3.43454   1.928950
3  4.70000   4.000000
4  5.60000  12.200000
5  6.80000  14.400000
>>> data.interpolate(method="akima")
         A          B
0  1.000000   0.250000
1  2.100000  -0.873316
2  3.406667   0.320034
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
>>> data.interpolate(method="spline", order=2)
         A          B
0  1.000000   0.250000
1  2.100000  -0.428598
2  3.404545   1.206900
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
>>> data.interpolate(method="polynomial", order=2)
         A          B
0  1.000000   0.250000
1  2.100000  -2.703846
2  3.451351  -1.453846
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

So interpolate has a number of methods that you can use to interpolate between the NaN's. The default, which is executed with no argument is linear and what it does is ignore the index and treats the values as equally spaced and looks to linearly fill between the values. The remaining methods are all taken from scipy interpolate with a brief description given below.

- barycentric: Constructs a polynomial that passes through a given set of points.
- pchip: PCHIP one-dimensional monotonic cubic interpolation
- akima: Fit piecewise cubic polynomials, given vectors x and y
- spline: Spline data interpolator where we can pass the order of the spline
- polynomial: Polynomial data interpolator where we can pass the order of the polynomial

For more information please refer to the scipy documentation.

Next, we will consider interpolate on a series and show some of the optional arguments that we can pass.

```
>>> ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan,
...                   13, np.nan])
>>> ser
0     NaN
1     NaN
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
dtype: float64
>>> ser.interpolate()
0     NaN
1     NaN
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
dtype: float64
>>> ser.interpolate(limit=1)
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5     NaN
6    13.0
7    13.0
dtype: float64
>>> ser.interpolate(limit=1, limit_direction="backward")
0     NaN
1     5.0
2     5.0
3     NaN
4     NaN
5    11.0
6    13.0
7     NaN
dtype: float64
>>> ser.interpolate(limit=1, limit_direction="both")
0     NaN
1     5.0
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7    13.0
dtype: float64
>>> ser.interpolate(limit_direction="both")
0     5.0
1     5.0
2     5.0
3     7.0
4     9.0
5    11.0
6    13.0
7    13.0
dtype: float64
>>> ser.interpolate(limit_direction="both", limit_area="inside",
...                 limit=1)
0     NaN
1     NaN
2     5.0
3     7.0
4     NaN
5    11.0
6    13.0
7     NaN
dtype: float64
>>> ser.interpolate(limit_direction="backward",
...                 limit_area="outside")
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7     NaN
dtype: float64
>>> ser.interpolate(limit_direction="both",
...                 limit_area="outside")
0     5.0
1     5.0
2     5.0
3     NaN
4     NaN
5     NaN
6    13.0
7    13.0
dtype: float64
```

Initially we interpolate using the default method which is linear, and for the rest of the example we use the default method and vary the optional arguments. Next, we pass the limit option and set it to 1 which says we can only interpolate one past any value so we still have NaN data in the Series. We next keep limit set to 1 and add another argument limit direction and set it to backward. What this does is only interpolate one value next to an existing value but unlike before does it going backwards. We extend this in the next example by setting the limit direction to be both which interpolates both forwards and backwards for one value. We next remove the limit one and keep limit direction to be both and see that all values are interpolated. We next introduce the limit area option which has two options (aside from the default None) these are inside and outside. When set to inside NaN's are only filled when they are surrounded by valid values and when set to outside it only fills outside valid values. Here, we show examples using each of these alongside limit direction and limit.

Next, we introduce the replace method.

```
>>> import seaborn as sns
>>> iris = sns.load_dataset('iris')
>>> iris.sepal_length.unique()
array([5.1, 4.9, 4.7, 4.6, 5. , 5.4, 4.4, 4.8, 4.3, 5.8, 5.7, 5.2, 5.5,
       4.5, 5.3, 7. , 6.4, 6.9, 6.5, 6.3, 6.6, 5.9, 6. , 6.1, 5.6, 6.7,
       6.2, 6.8, 7.1, 7.6, 7.3, 7.2, 7.7, 7.4, 7.9])
>>> iris.sepal_width.unique()
array([3.5, 3. , 3.2, 3.1, 3.6, 3.9, 3.4, 2.9, 3.7, 4. , 4.4, 3.8, 3.3,
       4.1, 4.2, 2.3, 2.8, 2.4, 2.7, 2. , 2.2, 2.5, 2.6])
>>> iris.petal_length.unique()
array([1.4, 1.3, 1.5, 1.7, 1.6, 1.1, 1.2, 1. , 1.9, 4.7, 4.5, 4.9, 4. ,
       4.6, 3.3, 3.9, 3.5, 4.2, 3.6, 4.4, 4.1, 4.8, 4.3, 5. , 3.8, 3.7,
       5.1, 3. , 6. , 5.9, 5.6, 5.8, 6.6, 6.3, 6.1, 5.3, 5.5, 6.7, 6.9,
       5.7, 6.4, 5.4, 5.2])
>>> iris.petal_width.unique()
array([0.2, 0.4, 0.3, 0.1, 0.5, 0.6, 1.4, 1.5, 1.3, 1.6, 1. , 1.1, 1.8,
       1.2, 1.7, 2.5, 1.9, 2.1, 2.2, 2. , 2.4, 2.3])
>>> iris.replace(2.3, 2).head()
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
>>> iris.species.unique()
array(['setosa', 'versicolor', 'virginica'], dtype=object)
>>> iris.replace(['setosa', 'versicolor', 'virginica'],
                 ['set', 'ver', 'vir']).head()
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2     set
1           4.9          3.0           1.4          0.2     set
2           4.7          3.2           1.3          0.2     set
3           4.6          3.1           1.5          0.2     set
4           5.0          3.6           1.4          0.2     set
>>> iris.replace(['setosa', 'versicolor', 'virginica'],
                 ['set', 'ver', 'vir'])
...      ['species'].unique()
array(['set', 'ver', 'vir'], dtype=object)
```

16.7 Grouping

Next, we introduce the concept of grouping the data via the groupby method. Grouping data is a very powerful tool as we are able to create and operate on groups of data all at once.

```
>>> import seaborn as sns
>>> iris = sns.load_dataset('iris')
>>> groupby = iris.groupby('species')
>>> groupby.sum()
            sepal_length  sepal_width  petal_length  petal_width
species
setosa             250.3        171.4          73.1         12.3
versicolor         296.8        138.5         213.0         66.3
virginica          329.4        148.7         277.6        101.3
>>> groupby.mean()
            sepal_length  sepal_width  petal_length  petal_width
species
setosa             5.006        3.428         1.462        0.246
versicolor         5.936        2.770         4.260        1.326
virginica          6.588        2.974         5.552        2.026
```

Above we see the groupby applied to the iris dataset where we look to group the data based on the column species. This then allows us to apply methods to the groupby object and we show the results of the sum and mean method applied to this. What this is doing is applying this method to all the distinct types in species by all the columns in the dataset.

We next demonstrate how to loop over a group. Here, we set the DataFrame up as seen previously but now we loop over the group and in looping over it print the name of the group and what is in that

group. This gives us a good visualisation of what a groupby does to the data.

```
>>> groupby = iris.groupby('species')
>>> for name, group in groupby:
...     print(name)
...     print(group.head())
...
setosa
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
versicolor
    sepal_length  sepal_width  petal_length  petal_width     species
50           7.0          3.2           4.7          1.4  versicolor
51           6.4          3.2           4.5          1.5  versicolor
52           6.9          3.1           4.9          1.5  versicolor
53           5.5          2.3           4.0          1.3  versicolor
54           6.5          2.8           4.6          1.5  versicolor
virginica
     sepal_length  sepal_width  petal_length  petal_width    species
100           6.3          3.3           6.0          2.5  virginica
101           5.8          2.7           5.1          1.9  virginica
102           7.1          3.0           5.9          2.1  virginica
103           6.3          2.9           5.6          1.8  virginica
104           6.5          3.0           5.8          2.2  virginica
```

Next, we introduce the aggregate method applied to a groupby. We set the data up in the same way as seen earlier and then apply the aggregate method of the groupby object and inside it pass what we want to use for this aggregation. In the example we show we have used the np.sum method which will be applied to the group.

```
>>> grouped = iris.groupby('species')
>>> grouped.aggregate(np.sum)
      B
A
1     6
2     9
3    13
```

We can extend the previous example by introducing the as_index argument. Here, we use the same DataFrame from the previous examples and groupby species with as_index set to False. What this does is create a group on species but retain species in the output as its column with the value we want to group by. In this case, we apply the sum to the group and so all other columns are summed within the group.

```
>>> iris.groupby('species', as_index=False).sum()
      species  sepal_length  sepal_width  petal_length  petal_width
0      setosa         250.3        171.4          73.1         12.3
1  versicolor         296.8        138.5         213.0         66.3
2   virginica         329.4        148.7         277.6        101.3
```

There are also methods that we can apply to a groupby object which can be useful.

```
>>> grouped = iris.groupby('species')
>>> grouped.size()
species
setosa          50
versicolor      50
virginica       50
dtype: int64
>>> grouped['sepal_length'].describe()
           count    mean       std   min     25%    50%    75%    max
species
setosa      50.0   5.006   0.352490   4.3   4.800   5.0    5.2    5.8
versicolor  50.0   5.936   0.516171   4.9   5.600   5.9    6.3    7.0
virginica   50.0   6.588   0.635880   4.9   6.225   6.5    6.9    7.9
```

We can also apply different methods to the group and in this example we show multiple ways to apply the numpy methods sum, mean, and std to our grouped data. So we create the same DataFrame and group as in the last examples. What we can then do is use the agg method with the arguments being a list of methods to be applied and what we see is that each method is applied on the group of data. Lastly, here we can even apply a lambda function to the groupby.

```
>>> grouped = iris.groupby('species')
>>> grouped['sepal_length'].agg([np.sum, np.mean, np.std])
                sum     mean        std
species
setosa        250.3    5.006    0.352490
versicolor    296.8    5.936    0.516171
virginica     329.4    6.588    0.635880
>>> grouped.agg({lambda x: np.std(x, ddof=1)})
            sepal_length sepal_width petal_length petal_width
              <lambda>      <lambda>      <lambda>     <lambda>
species
setosa         0.352490      0.379064      0.173664     0.105386
versicolor     0.516171      0.313798      0.469911     0.197753
virginica      0.635880      0.322497      0.551895     0.274650
```

What we next show is that you can get the largest and smallest values with a group by using the nlargest and nsmallest methods. Here, the integer value you pass in gives you the number of values returned. What you see is that we get the largest and smallest per group.

```
>>> grouped = iris.groupby('species')
>>> grouped['sepal_length'].nlargest(3)
species
setosa       14     5.8
             15     5.7
             18     5.7
versicolor   50     7.0
             52     6.9
             76     6.8
virginica    131    7.9
             117    7.7
             118    7.7
Name: sepal_length, dtype: float64
>>> grouped['petal_length'].nsmallest(4)
species
setosa       22     1.0
             13     1.1
             14     1.2
             35     1.2
versicolor   98     3.0
             57     3.3
             93     3.3
             60     3.5
virginica    106    4.5
             126    4.8
             138    4.8
             121    4.9
Name: petal_length, dtype: float64
```

Our next example introduces the apply method which can be very useful. Here, we set the data up in the manner seen before and groupby column species. We can then use the apply method on the

group to apply whatever we pass through it to the groupby. It should be noted we can also use the apply method on DataFrames and Series. Here we see we have applied a custom function to the groupby.

```
>>> grouped = iris.groupby('species')
>>> def f(group):
...     return pd.DataFrame({"original" : group,
                             "demeaned" : group - group.mean()})
>>> grouped['petal_length'].apply(f).head()
   original  demeaned
0       1.4    -0.062
1       1.4    -0.062
2       1.3    -0.162
3       1.5     0.038
4       1.4    -0.062
```

In the next example, we introduce a nice pandas method called qcut. This cuts the data into equal sized buckets based on the arguments passed in. Here, we apply the qcut on the data which is the column sepal length of the iris dataset by the list of values $0, 0.25, 0.5, 0.75$, and 1. We assign the cut to the variable factor and when passed into the groupby the mean method gives the average on each bucket showing what the min and max values in the buckets are.

```
>>> factor = pd.qcut(iris['sepal_length'], [0, 0.25, 0.5, 0.75, 1.0])
>>> factor.head()
0    (4.2989999999999995, 5.1]
1    (4.2989999999999995, 5.1]
2    (4.2989999999999995, 5.1]
3    (4.2989999999999995, 5.1]
4    (4.2989999999999995, 5.1]
Name: sepal_length, dtype: category
Categories (4, interval[float64]): [(4.2989999999999995, 5.1]
 < (5.1, 5.8] < (5.8, 6.4] < (6.4, 7.9]]
>>> iris.groupby(factor).mean()
                             sepal_length  sepal_width  petal_length  petal_width
sepal_length
(4.2989999999999995, 5.1]        4.856098     3.175610      1.707317     0.353659
(5.1, 5.8]                       5.558974     3.089744      3.256410     0.989744
(5.8, 6.4]                       6.188571     2.868571      4.908571     1.682857
(6.4, 7.9]                       6.971429     3.071429      5.568571     1.940000
```

So far we have considered grouping on single columns, however we could also group on multiple columns. However, the iris dataset isn't best setup to allow us to do this so we instead load the tips dataset. The tips dataset contains the following columns:

- total_bill
- tip
- sex
- smoker
- day
- time
- size

Given some of the columns only have limited responses it makes it ideal to do a group by multiple columns so next we group by sex and

smoker.

```
>>> tips = sns.load_dataset('tips')
>>> tips.head()
   total_bill    tip     sex smoker  day    time  size
0       16.99   1.01  Female     No  Sun  Dinner     2
1       10.34   1.66    Male     No  Sun  Dinner     3
2       21.01   3.50    Male     No  Sun  Dinner     3
3       23.68   3.31    Male     No  Sun  Dinner     2
4       24.59   3.61  Female     No  Sun  Dinner     4
>>> grouped = tips.groupby(['sex', 'smoker'])
>>> grouped.sum()
                total_bill      tip  size
sex     smoker
Male    Yes        1337.07   183.07   150
        No         1919.75   302.00   263
Female  Yes         593.27    96.74    74
        No          977.68   149.77   140
>>> grouped = tips.groupby(['sex', 'smoker', 'time'])
>>> grouped.mean()
                     total_bill        tip      size
sex     smoker time
Male    Yes    Lunch    17.374615  2.790769  2.153846
               Dinner   23.642553  3.123191  2.595745
        No     Lunch    18.486500  2.941500  2.500000
               Dinner   20.130130  3.158052  2.766234
Female  Yes    Lunch    17.431000  2.891000  2.300000
               Dinner   18.215652  2.949130  2.217391
        No     Lunch    15.902400  2.459600  2.520000
               Dinner   20.004138  3.044138  2.655172
```

Here, we see that when we group by two or three variables we in-
crease the number of values that are returned by creating more com-

binations within the groups.

A similar approach to groupby is pivot table which is a common amongst spreadsheet users. The concept is to take a combination of variables and group the data by it, which can seem similar to group-by. The difference is you can extend upon this to create some more complicated groupings of your dataset, we will demonstrate these by example.

```
>>> tips = sns.load_dataset('tips')
>>> tips.head()
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
>>> pd.pivot_table(tips,index=["sex"])
            size       tip  total_bill
sex
Male    2.630573  3.089618   20.744076
Female  2.459770  2.833448   18.056897
>>> pd.pivot_table(tips,index=["sex","smoker","day"])
                      size       tip  total_bill
sex    smoker day
Male   Yes    Thur  2.300000  3.058000   19.171000
              Fri   2.125000  2.741250   20.452500
              Sat   2.629630  2.879259   21.837778
              Sun   2.600000  3.521333   26.141333
       No     Thur  2.500000  2.941500   18.486500
              Fri   2.000000  2.500000   17.475000
              Sat   2.656250  3.256563   19.929063
              Sun   2.883721  3.115349   20.403256
Female Yes    Thur  2.428571  2.990000   19.218571
              Fri   2.000000  2.682857   12.654286
              Sat   2.200000  2.868667   20.266667
              Sun   2.500000  3.500000   16.540000
       No     Thur  2.480000  2.459600   16.014400
              Fri   2.500000  3.125000   19.365000
              Sat   2.307692  2.724615   19.003846
              Sun   3.071429  3.329286   20.824286
>>> pd.pivot_table(tips,index=["sex","smoker","day"],
                   values=['tip'])
                      tip
sex    smoker day
Male   Yes    Thur  3.058000
              Fri   2.741250
              Sat   2.879259
              Sun   3.521333
       No     Thur  2.941500
              Fri   2.500000
              Sat   3.256563
              Sun   3.115349
Female Yes    Thur  2.990000
              Fri   2.682857
              Sat   2.868667
              Sun   3.500000
       No     Thur  2.459600
              Fri   3.125000
              Sat   2.724615
              Sun   3.329286
```

In the above code we use the tips as the dataset in each example and set a variety of index values starting at just sex and extending to sex and smoker and then with the combination of sex, smoker and day. In each example when we pivot the data by default we end up with the average of the index across all of the variables where we can take an average, so only numerical variables. We don't necessarily need to show all available variables as we have seen by passing the values argument as a list of columns we want to include.

```
>>> import numpy as np
>>> pd.pivot_table(tips,index=["sex","smoker","day"],
....     values=['tip'],aggfunc=[np.mean,len])
                      mean     len
                      tip      tip
sex     smoker day
Male    Yes    Thur   3.058000  10.0
               Fri    2.741250   8.0
               Sat    2.879259  27.0
               Sun    3.521333  15.0
        No     Thur   2.941500  20.0
               Fri    2.500000   2.0
               Sat    3.256563  32.0
               Sun    3.115349  43.0
Female  Yes    Thur   2.990000   7.0
               Fri    2.682857   7.0
               Sat    2.868667  15.0
               Sun    3.500000   4.0
        No     Thur   2.459600  25.0
               Fri    3.125000   2.0
               Sat    2.724615  13.0
               Sun    3.329286  14.0
```

As a default when we use the pivot table command we get the average of the variables, however we can control what we get back by passing the aggfunc argument which takes a list of functions we want to apply to the data. Note here that we pass the numpy mean function as well as the len from the standard Python library.

```
>>> pd.pivot_table(tips,index=["sex","smoker"],values=["tip"],
...     columns=["day"],aggfunc=[np.mean])
                    mean
                    tip
day                 Thur        Fri         Sat         Sun
sex     smoker
Male    Yes         3.0580      2.741250    2.879259    3.521333
        No          2.9415      2.500000    3.256563    3.115349
Female  Yes         2.9900      2.682857    2.868667    3.500000
        No          2.4596      3.125000    2.724615    3.329286
>>> pd.pivot_table(tips,index=["sex","smoker"],values=["tip"],
...     columns=["day"],aggfunc=[np.mean],margins=True)
                    mean
                    tip
day                 Thur        Fri         Sat         Sun         All
sex     smoker
Male    Yes         3.058000    2.741250    2.879259    3.521333    3.051167
        No          2.941500    2.500000    3.256563    3.115349    3.113402
Female  Yes         2.990000    2.682857    2.868667    3.500000    2.931515
        No          2.459600    3.125000    2.724615    3.329286    2.773519
All                 2.771452    2.734737    2.993103    3.255132    2.998279
```

We can expand on this example by adding in the margins variable which then gives us the totals associated with the rows and columns.

16.8 Reading in Files with Pandas

The examples in the chapter have used the datasets from Seaborn and while this is useful, pandas has a lot of methods to allow you to read in external files. If we relate this back to earlier in the book where we read and manipulated data within Python we can see that these methods are a lot easier to use. They also allow us to write back to file. To show how this works we will take one of the existing datasets that we have been using and write to a csv and read that back in:

```
>>> tips.head()
   total_bill   tip      sex smoker   day     time   size
0       16.99  1.01   Female     No   Sun   Dinner      2
1       10.34  1.66     Male     No   Sun   Dinner      3
2       21.01  3.50     Male     No   Sun   Dinner      3
3       23.68  3.31     Male     No   Sun   Dinner      2
4       24.59  3.61   Female     No   Sun   Dinner      4
>>> tips.to_csv('myfile.csv', index=False)
>>> data = pd.read_csv('myfile.csv')
>>> data.head()
   total_bill   tip      sex smoker   day     time   size
0       16.99  1.01   Female     No   Sun   Dinner      2
1       10.34  1.66     Male     No   Sun   Dinner      3
2       21.01  3.50     Male     No   Sun   Dinner      3
3       23.68  3.31     Male     No   Sun   Dinner      2
4       24.59  3.61   Female     No   Sun   Dinner      4
```

What we have done is use the to_csv method that the tips DataFrame has and write the data into a file called 'myfile.csv', note this will live in the directory where this is being run from, as well as the file name the

index argument is set to False which prevents the DataFrame index being written to the file with the other columns. Now, to read this back in we use the read_csv method from pandas and this takes the csv file and creates a DataFrame with the contents of this file. These methods are extremely useful as we do not have to worry about the process of writing to file or reading from file. Alongside read_csv we have other read methods for different file types and here are some of the more useful ones, for a complete list consult the pandas documentation:

- read_excel: reads in xls, xlsx, xlsm, xlsb, odf, ods and odt file types
- read_json: reads a valid json string

If we take the examples in previous chapters we created the following json and excel files called boston.json and boston.xlsx. We can read these into DataFrames using the following code:

```
>>> file_name = '/path/to/boston.json'
>>> data = pd.read_json(file_name)
>>> data.head()
      CRIM    NOX
0  0.00632   2.31
1  0.02731   7.07
2  0.02729   7.07
3  0.03237   2.18
4  0.06905   2.18
>>> file_name = '/path/to/boston.xlsx'
>>> data = pd.read_excel(file_name)
>>> data.head()
      CRIM    NOX
0  0.00632   2.31
1  0.02731   7.07
2  0.02729   7.07
3  0.03237   2.18
4  0.06905   2.18
```

As you can see these methods provide very simple ways to load data from these common formats into DataFrames. There is also a read-_table method which we can use for general delimited files. The read methods also support operations like querying databases or even reading html but that is beyond the scope of this book but well worth a look.

The to methods of the DataFrames are pretty similar with support for many different formats and a selection given as follows:

- to_dict

- to_json
- to_html
- to_latex
- to_string

These are all demonstrated as follows:

```
>>> import seaborn as sns
>>> tips = sns.load_dataset('tips')
>>> tips.head().to_json()
'{"total_bill":{"0":16.99,"1":10.34,"2":21.01,"3":23.68,"4":24.59},
"tip":{"0":1.01,"1":1.66,"2":3.5,"3":3.31,"4":3.61},
"sex":{"0":"Female","1":"Male","2":"Male","3":"Male","4":"Female"},
"smoker":{"0":"No","1":"No","2":"No","3":"No","4":"No"},
"day":{"0":"Sun","1":"Sun","2":"Sun","3":"Sun","4":"Sun"},
"time":{"0":"Dinner","1":"Dinner","2":"Dinner","3":"Dinner","4":"Dinner"},
"size":{"0":2,"1":3,"2":3,"3":2,"4":4}}'
>>> tips.head().to_dict()
{'total_bill': {0: 16.99, 1: 10.34, 2: 21.01, 3: 23.68, 4: 24.59},
'tip': {0: 1.01, 1: 1.66, 2: 3.5, 3: 3.31, 4: 3.61},
'sex': {0: 'Female', 1: 'Male', 2: 'Male', 3: 'Male', 4: 'Female'},
'smoker': {0: 'No', 1: 'No', 2: 'No', 3: 'No', 4: 'No'},
'day': {0: 'Sun', 1: 'Sun', 2: 'Sun', 3: 'Sun', 4: 'Sun'},
'time': {0: 'Dinner', 1: 'Dinner', 2: 'Dinner', 3: 'Dinner', 4: 'Dinner'},
'size': {0: 2, 1: 3, 2: 3, 3: 2, 4: 4}}
>>> tips.head().to_html()
'<table border="1" class="dataframe">\n  <thead>\n
<tr style="text-align: right;">\n      <th></th>\n
<th>total_bill</th>\n      <th>tip</th>\n      <th>sex</th>\n
<th>smoker</th>\n      <th>day</th>\n      <th>time</th>\n
<th>size</th>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n
<th>0</th>\n      <td>16.99</td>\n      <td>1.01</td>\n
<td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n
<th>1</th>\n      <td>10.34</td>\n      <td>1.66</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n
<th>2</th>\n      <td>21.01</td>\n      <td>3.50</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n
<th>3</th>\n      <td>23.68</td>\n      <td>3.31</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n
<th>4</th>\n      <td>24.59</td>\n      <td>3.61</td>\n
<td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n  </tbody>\n</table>'
>>> tips.head().to_latex()
'\\begin{tabular}{lrrlllr}\n\\toprule\n{} &  total\\_bill &   tip &
sex & smoker &  day &    time &  size \\\\\n\\midrule\n0 &       16.99 &
1.01 &  Female &     No &  Sun &  Dinner &     2 \\\\\n1 &       10.34 &
1.66 &    Male &     No &  Sun &  Dinner &     3 \\\\\n2 &       21.01 &
3.50 &    Male &     No &  Sun &  Dinner &     3 \\\\\n3 &       23.68 &
3.31 &    Male &     No &  Sun &  Dinner &     2 \\\\\n4 &       24.59 &
3.61 &  Female &     No &  Sun &  Dinner &     4 \\\\\n\\bottomrule\n
\\end{tabular}\n'
>>> tips.head().to_string()
'   total_bill   tip     sex smoker  day    time  size\n0        16.99
1.01  Female     No  Sun  Dinner     2\n1       10.34  1.66    Male
No  Sun  Dinner     3\n2       21.01  3.50    Male     No  Sun  Dinner
3\n3       23.68  3.31    Male     No  Sun  Dinner     2\n4
24.59  3.61  Female     No  Sun  Dinner     4'
```

We can also use some of these methods to write the data directly to file in the format with some examples below:

```
>>> import seaborn as sns
>>> tips = sns.load_dataset('tips')
>>> tips.head().to_json('tips.json')
>>> tips.to_html('tips.html')
>>> tips.to_latex('tips.latex')
>>> tips.to_latex('tips.tex')
```

These methods are really useful and for correctly formatted data are a very convenient way to read data into pandas and also export it from pandas.

What we have seen in this chapter is the advanced methods of pandas and how we can do complex data analysis. We have shown how pandas allows us to manipulate data as if it were in a database allows us to join, merge, group, and pivot the data in a variety of ways. We have also covered some of the built in methods that pandas has and shown how we can deal with missing data. The examples that we have covered have been rather simple in nature but pandas is powerful enough to deal with large datasets and that makes it an extremely powerful Python package. It is also worth noting that pandas plays well with many other Python packages meaning a mastery of it is essential for a Python programmer.

17
Plotting

Plotting is a key component when looking to work with data, and Python has a number of plotting libraries. In this chapter, we cover plotting in great detail starting with creating basic plots directly from Pandas DataFrames to getting more control using matplotlib all the way through to using Seaborn. The chapter is fully example driven with code snippets alongside the graphics they produce. The aim of including both is to give you the ability to see what the code produces but also act as a reference for when you want to produce graphs. The examples are based on datasets that come from with Python with many being taken from the documentation with the idea of giving them greater explanation to help you understand what is happening. The packages that we will look to use in this chapter are imported below.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib
>>> import seaborn as sns
```

17.1 Pandas

Initially, we will look at plotting methods on Series and DataFrame objects in pandas. One of the great things about pandas is that we have inbuilt plotting methods that we can call and produce plots from. This allows very fast visual presentation of datasets that we want to analyse.

```
>>> iris = sns.load_dataset('iris')
>>> iris.sepal_length.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x113e62290>
>>> plt.show()

>>> iris.sepal_length.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x113e62290>
>>> plt.savefig('/path/to/file/file_name.pdf')
```
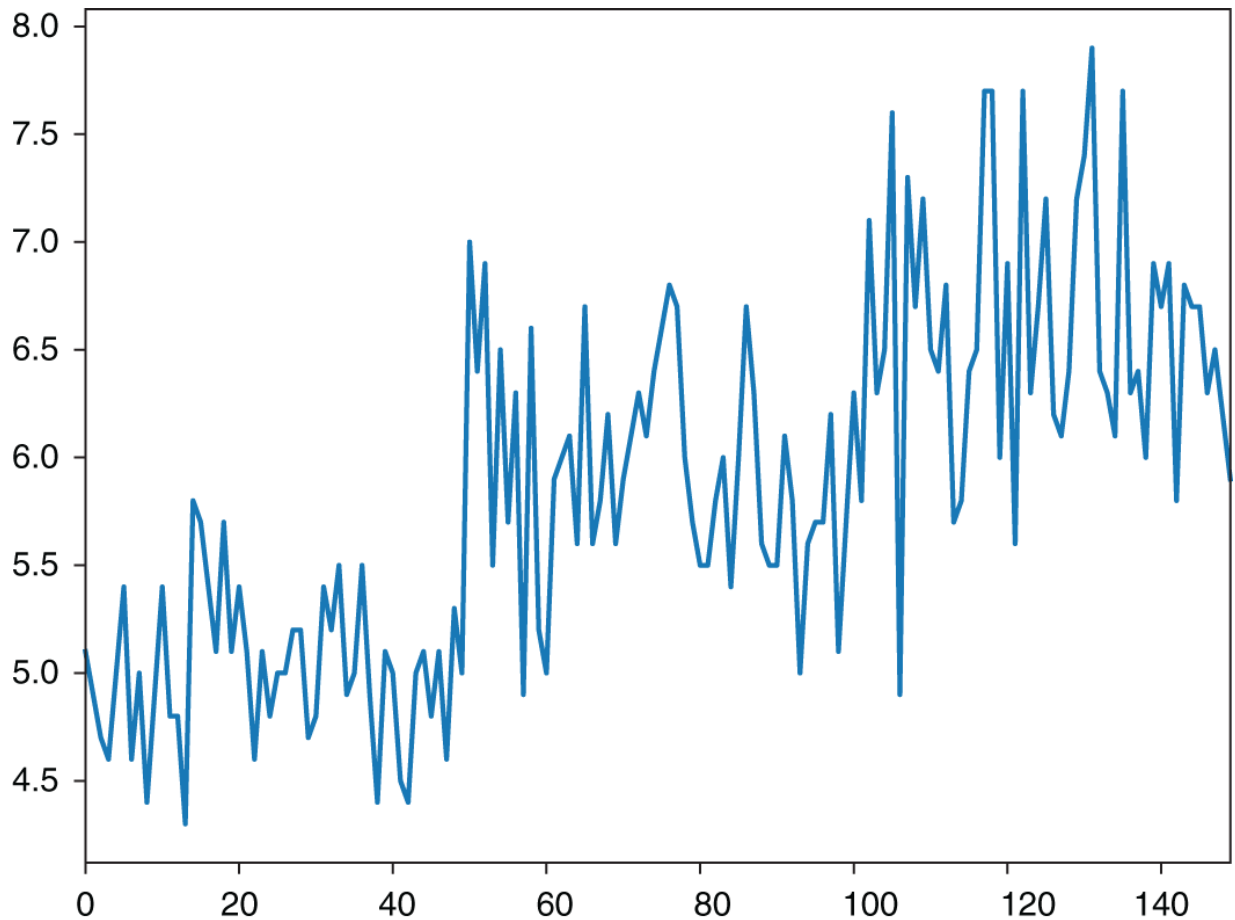
Figure 17.1 Line plot of sepal length.

The previous code snippets and associated plot show how we can produce a simple line plot of a Series. The two examples use plt.show() and plt.savefig() to show how to either show the graph or save the graph, with the graph shown in Figure 17.1 In the event of saving a graph, we need to supply the file path and name to where we want the plot to save.

```
>>> plt.clf()
>>> iris.sepal_length.hist()
<matplotlib.axes._subplots.AxesSubplot object at 0x1147cd9d0>
```

In the Figure 17.2, we use the method hist method of the series in the same way we used the plot method in the previous example. It should be noted that plt.clf() was used as the first line of code with the reason behind this to clear the figure meaning we do not see the previous plot when we plot the new image.

```
>>> plt.clf()
>>> iris.sepal_length.plot.box()
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f1a0b00>

>>> plt.clf()
>>> iris.sepal_length.plot.density()
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f6b7a58>
>>> plt.show()
```

The next two snippets show how we can create a box plot and a density as shown in Figures 17.3 and 17.4. In these examples, we use plot method of the series to give us access to the plot that we want to create. This is slightly different from what we have used before, and looking at the plot method, we can see the types of plots that we can create.
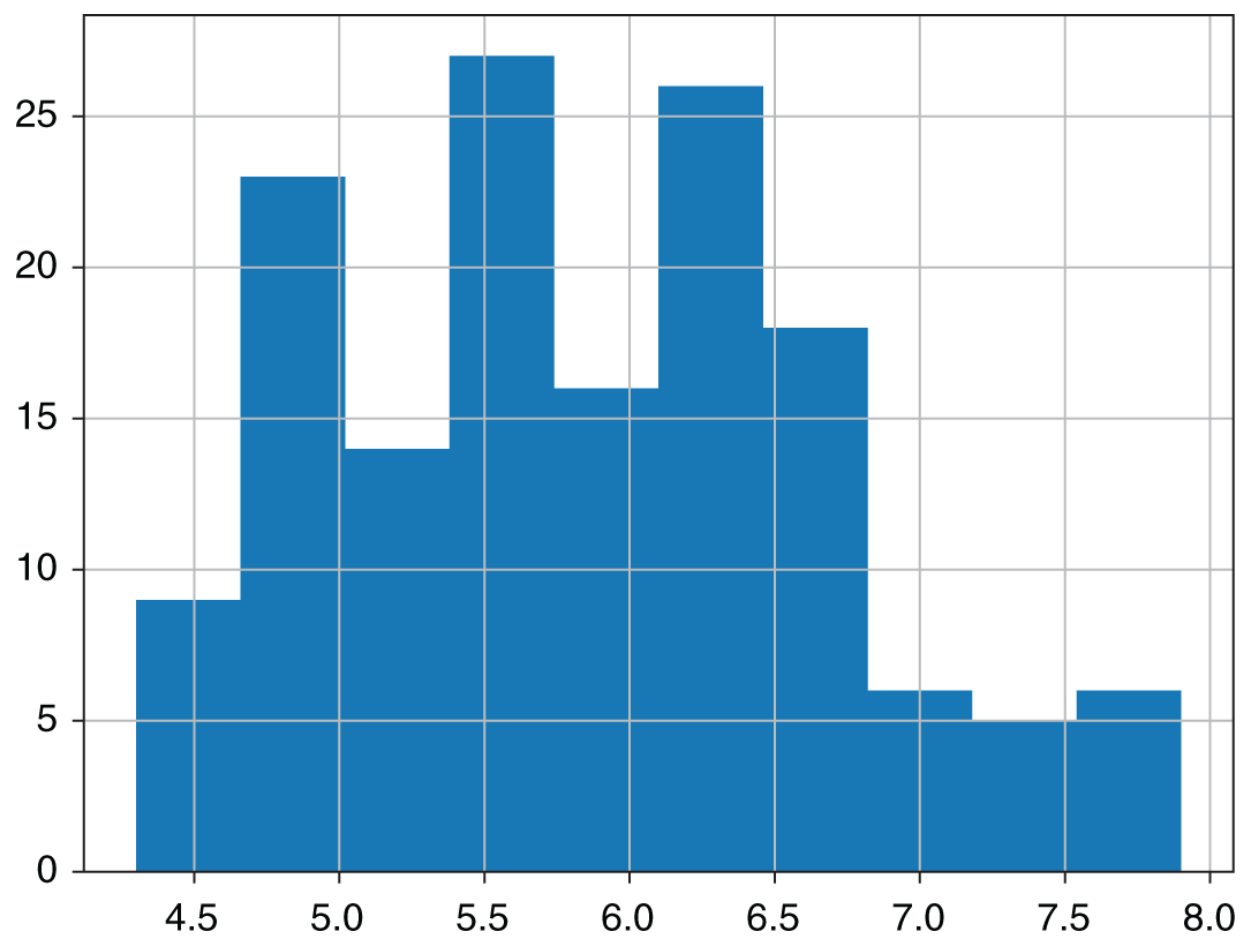
Figure 17.2 Histogram of sepal length.
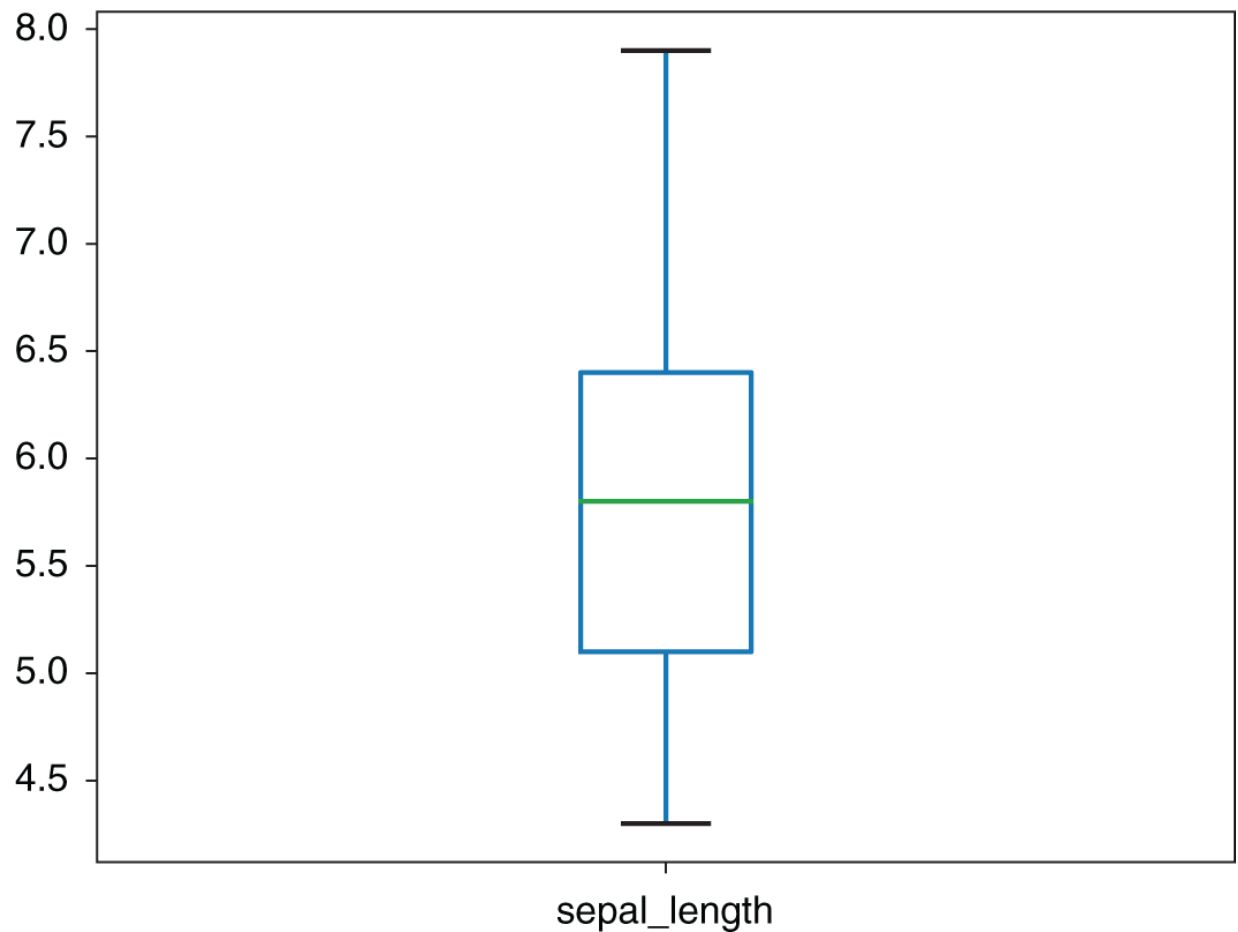
Figure 17.3 Boxplot of sepal length.

```
>>> dir(iris.sepal_length.plot)
['__bytes__', '__call__', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__unicode__', '__weakref__',
 '_accessors', '_constructor', '_deprecations', '_dir_additions', '_dir_deletions',
 '_reset_cache', 'area', 'bar', 'barh', 'box', 'density', 'hist', 'kde', 'line', 'pie']
```

Here we see that alongside box and density, we also have the following types of plots.
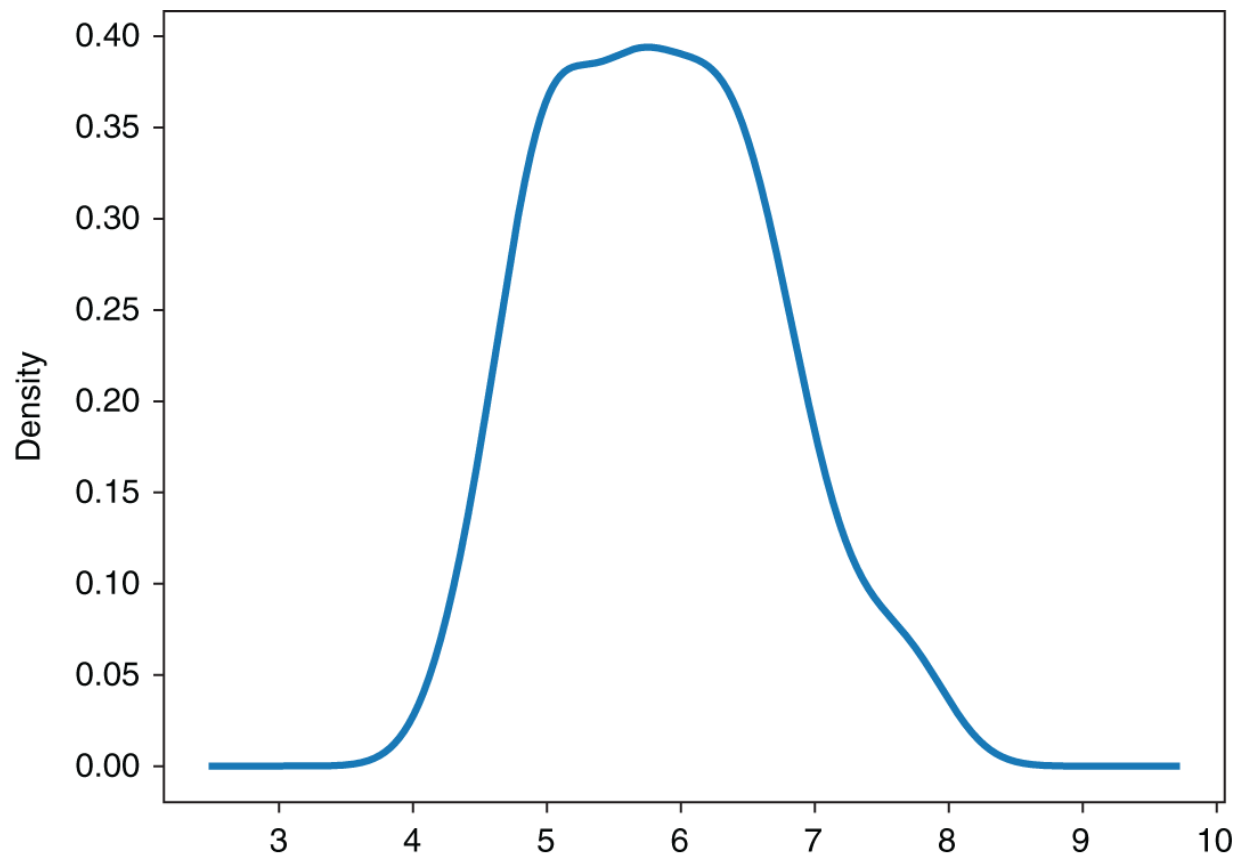
Figure 17.4 Density plot of sepal length.
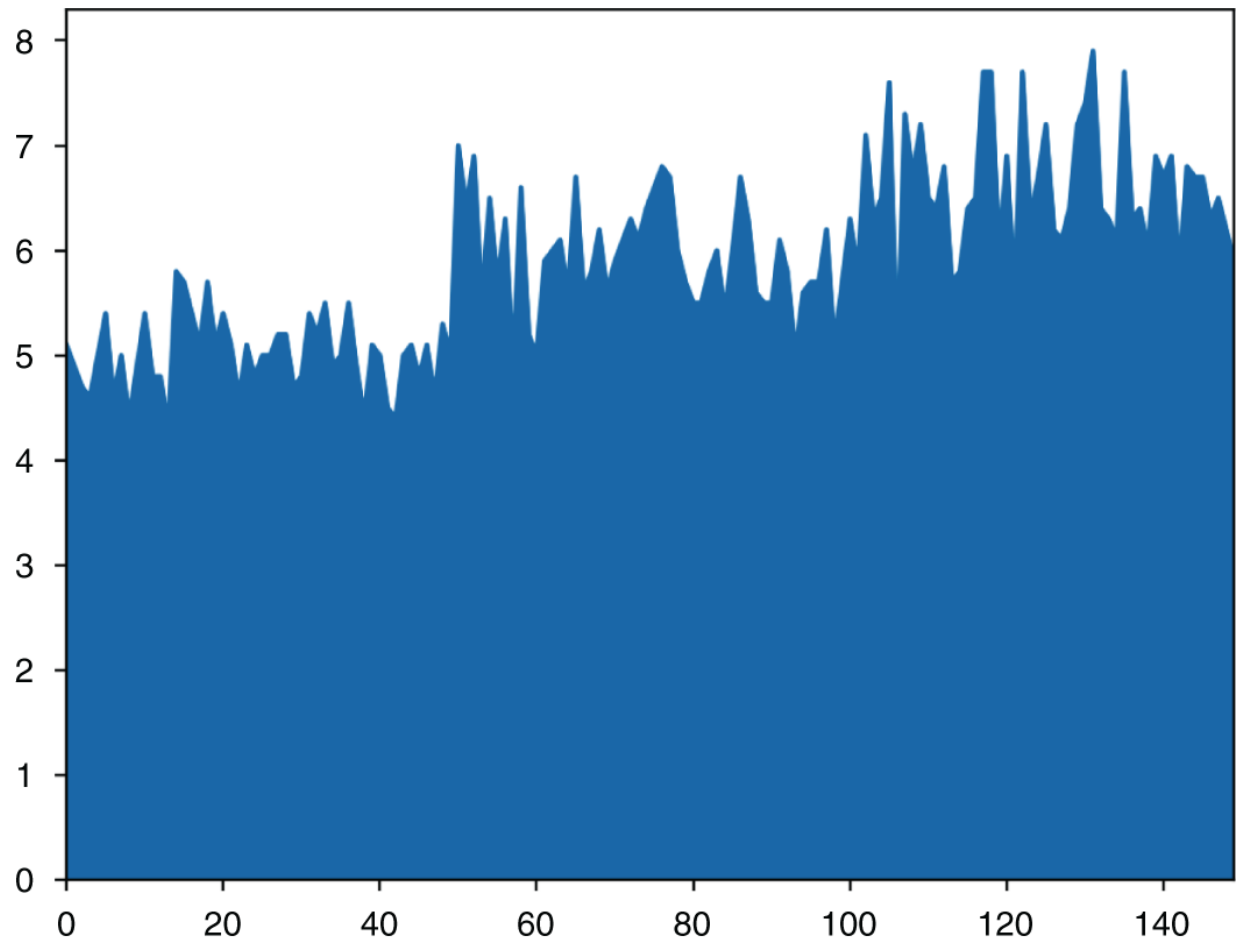
Figure 17.5 Area plot of sepal length.

- area as shown in Figure 17.5
- hist as shown in Figure 17.6
- kde as shown in Figure 17.7
- line as shown in Figure 17.8

It is worth noting that some of these types we have already covered as we can directly plot a line using plot and a histogram using hist. But we will show examples of each of these.

Thus far we have looked at plots on series which will apply to columns of data frames however we can also do plots on DataFrames as a whole. As opposed to working on a single column of the iris DataFrame, we now work on the DataFrame as a whole.
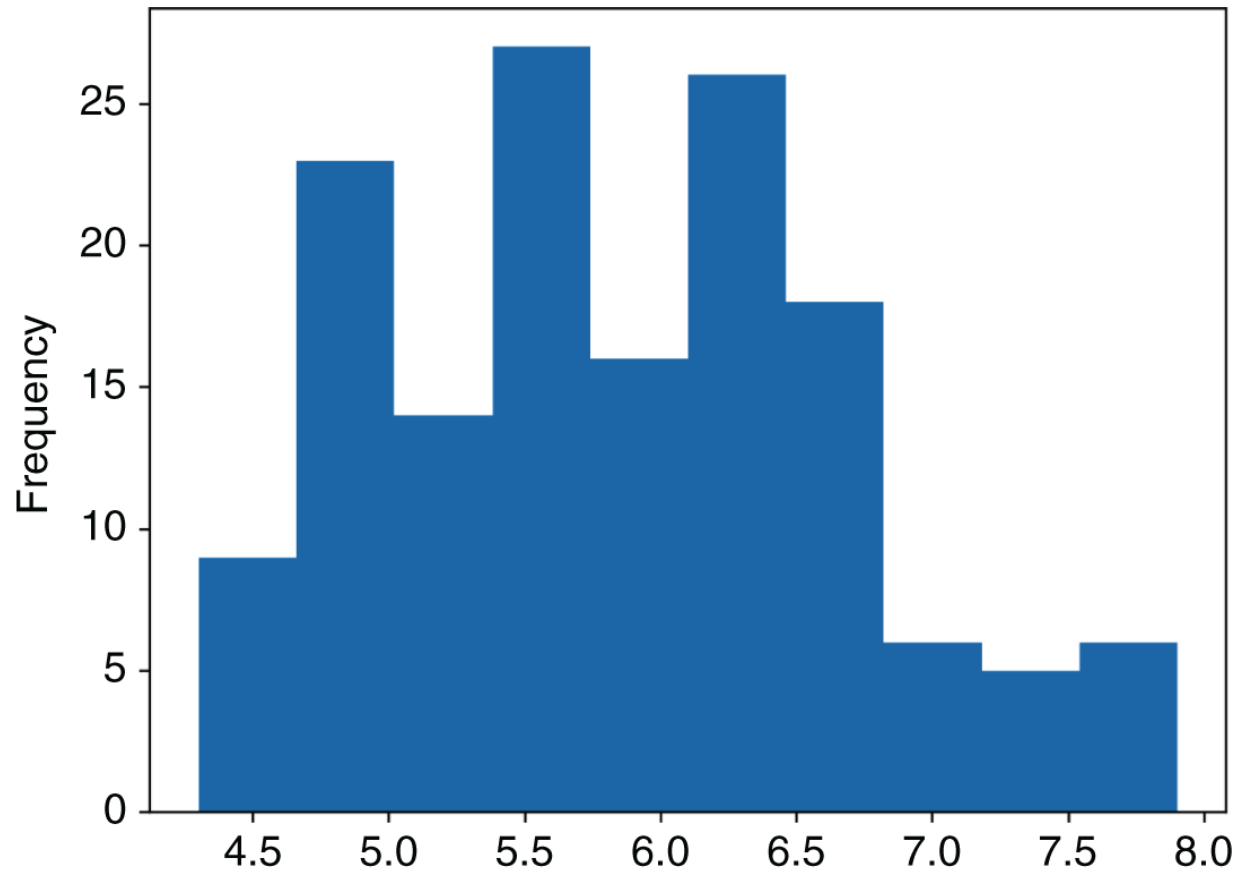
Figure 17.6 Histogram of sepal length.
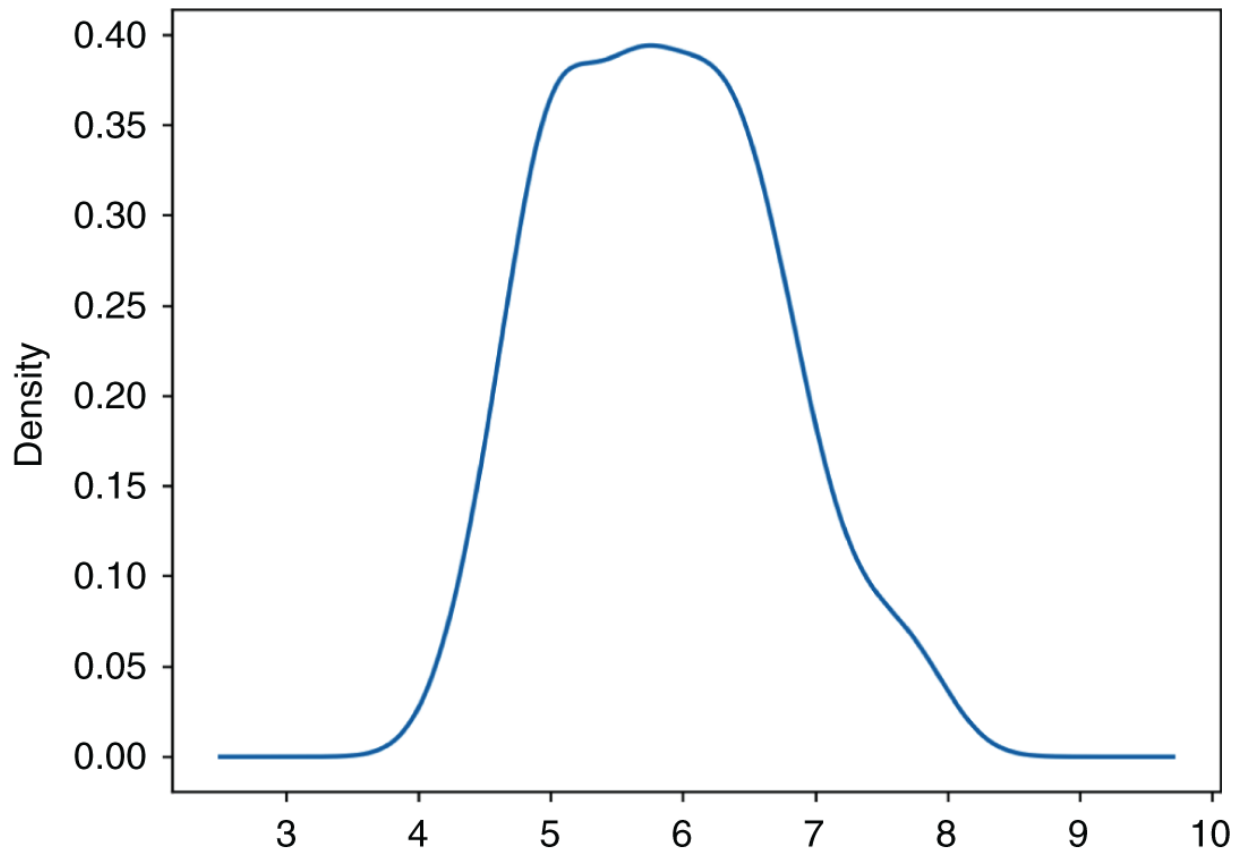
Figure 17.7 KDE of sepal length.

```
>>> plt.clf()
>>> iris.plot.box()
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1b1d1a58>
```

The first plot that we will consider is the box plot as shown in Figure 17.9. When applied to the DataFrame as a whole, we can see that we have box plots for each of the variables where a box plot can be created. Pandas also labels up the variables from the DataFrame names so out of the box you get a robust plot of all the data.
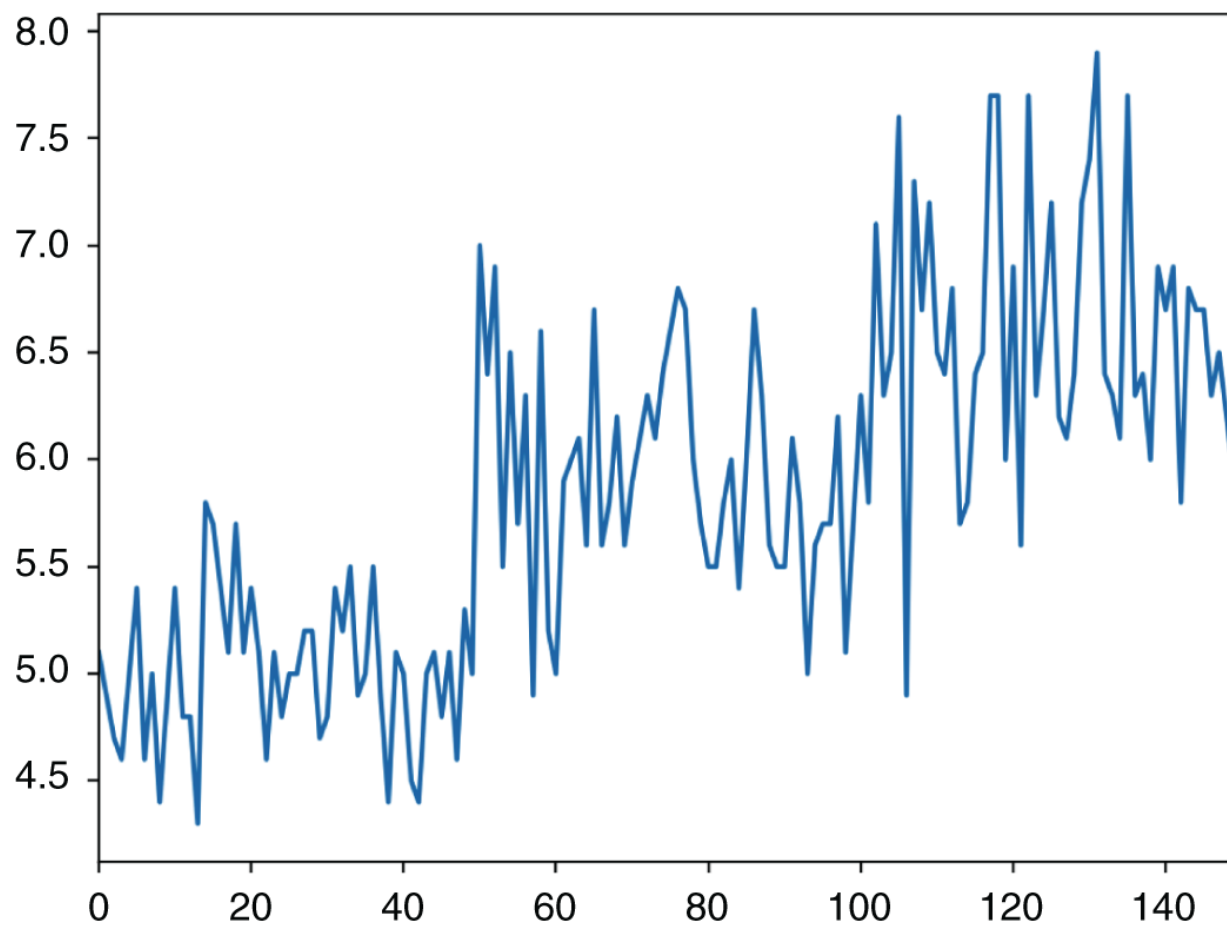
Figure 17.8 Line plot of sepal length.
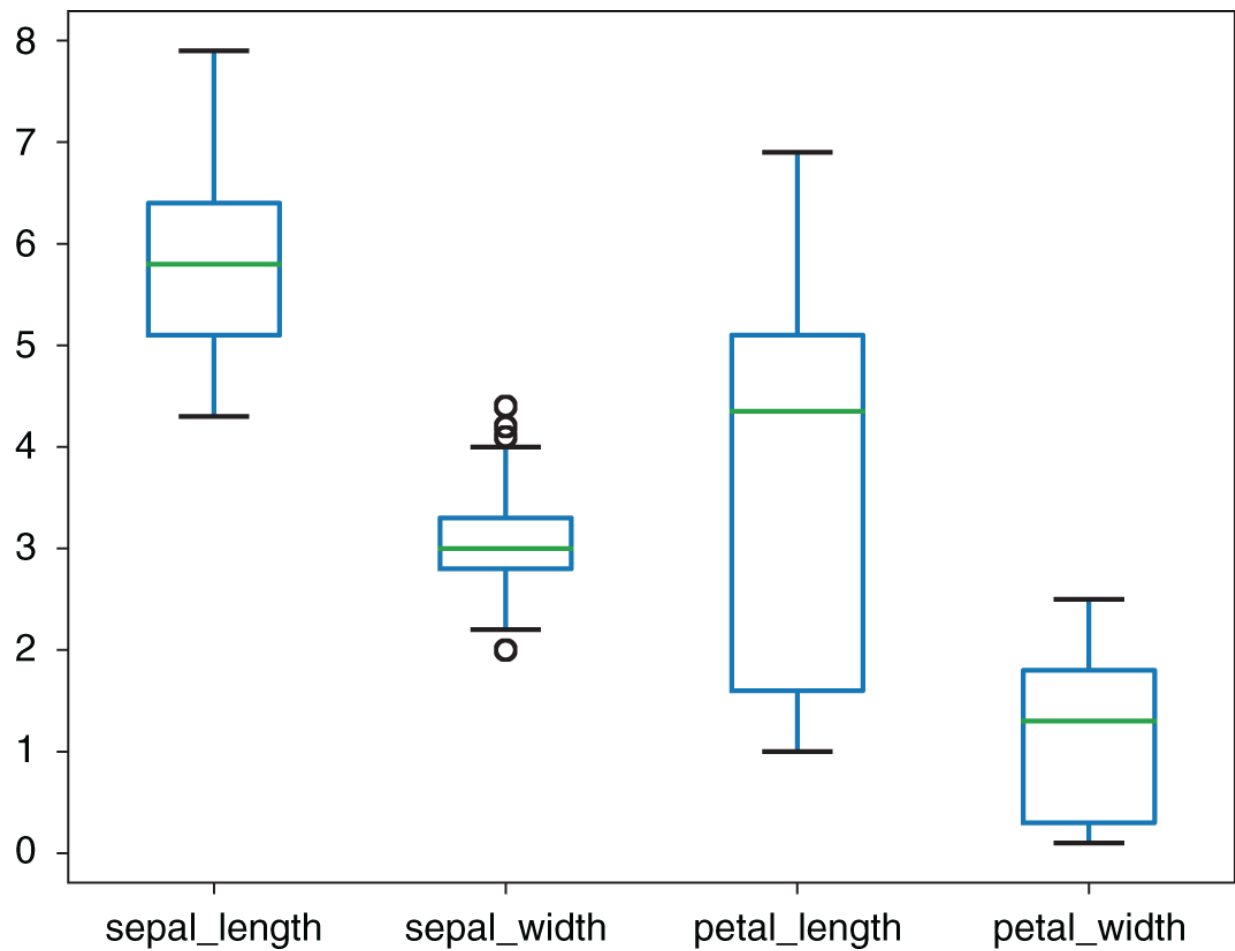
Figure 17.9 Box plot of iris data.

```
>>> plt.clf()
>>> iris.plot.density()
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f6b3f60>
>>> plt.show()

>>> plt.clf()
>>> iris.plot.line()
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f166668>
>>> plt.show()
```
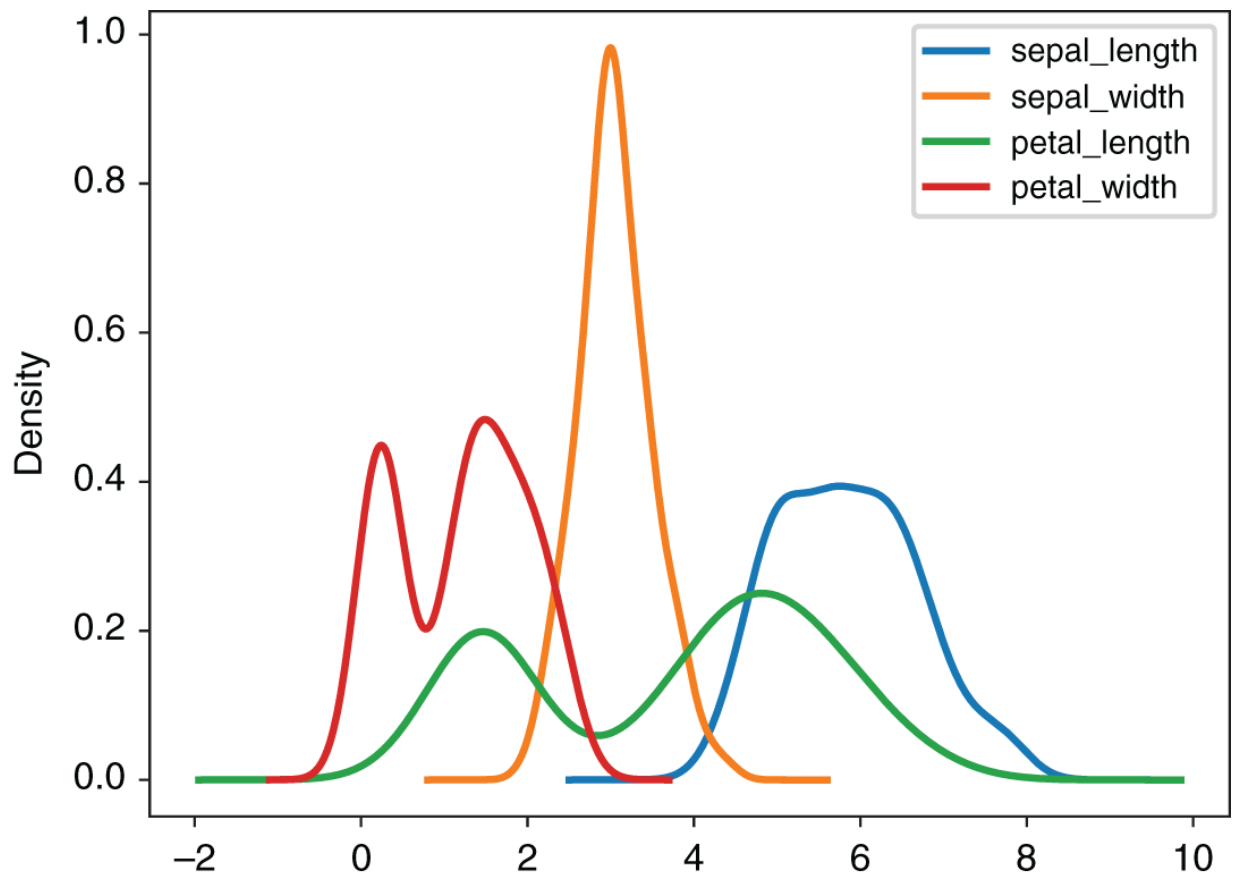
Figure 17.10 Density plot on iris data.

Next we will show a density and line plot, as shown in Figures 17.10 and 17.11 of the DataFrame and like with the box plot example we get a line for each variable where we can get one. In the same way, we have labels for each box we get a legend for the lines indicating what each colour refers to.

```
>>> plt.clf()
>>> iris.plot.scatter(x='sepal_length', y='sepal_width')
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f146a58>
>>> plt.show()
```
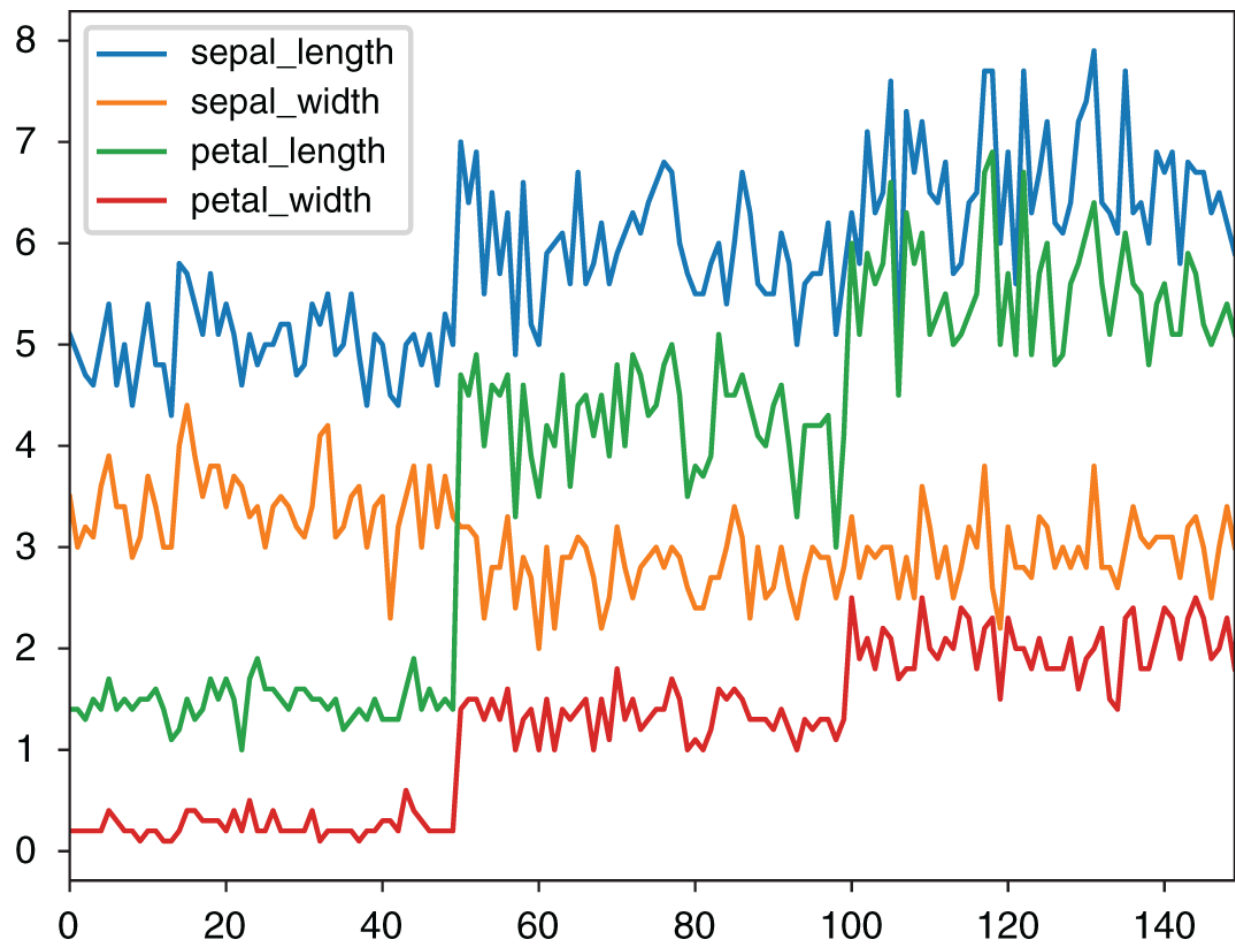
Figure 17.11 Line plot on iris data.
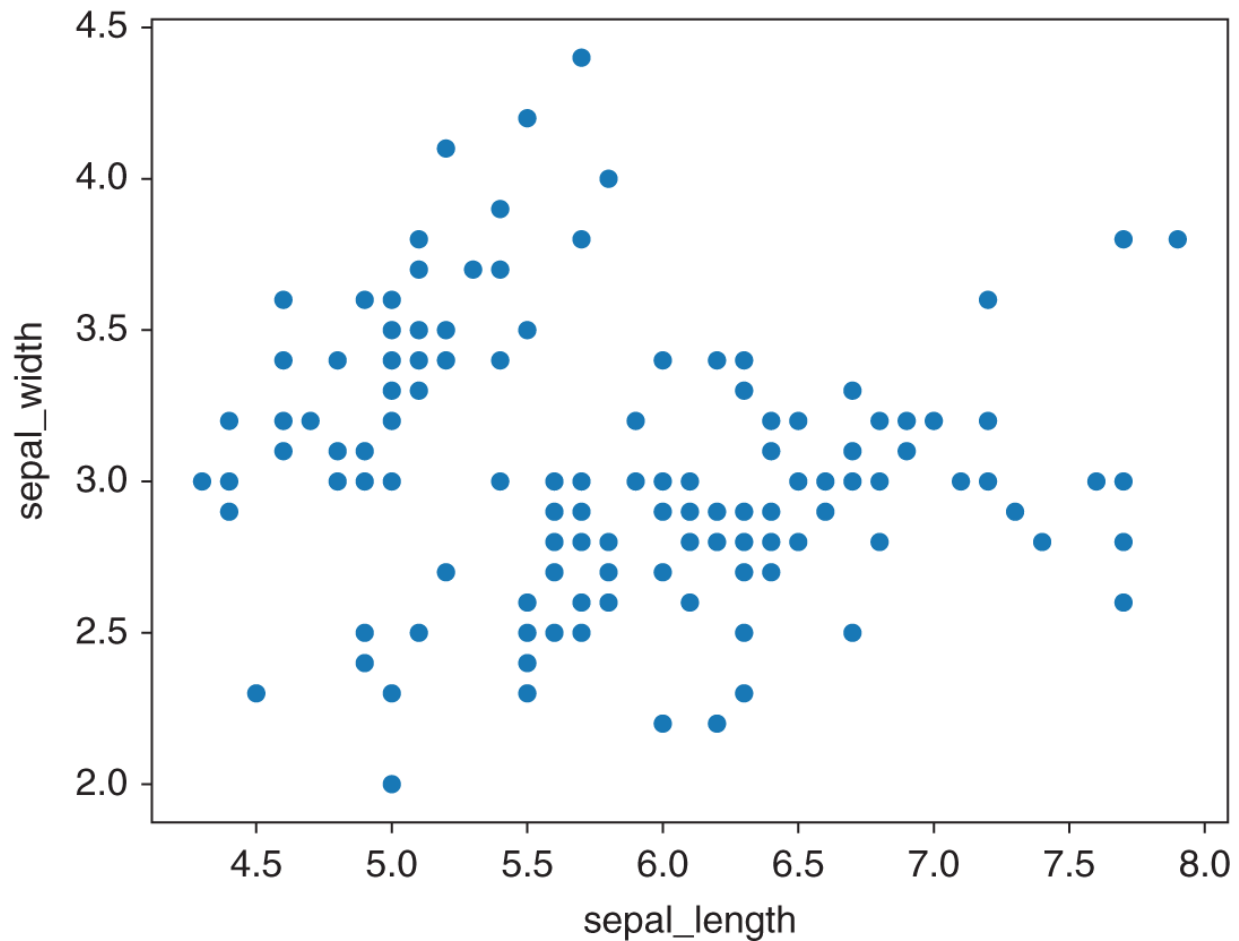
Figure 17.12 Scatter plot on pandas data frame.

In the last built-in plot, we shall consider a scatter plot on the Data-Frame as shown in [Figure 17.2](). Unlike the plots we have considered so far, a scatter plot requires an x and y variable as its plotting x against y. So, we pass in the specific column name from the Data-Frame and the method can pick up the data that it needs to giving us the scatter of sepal length against sepal width.

As before we have a number of other plot types that we can apply to the DataFrame, these are

- area as shown in Figure 17.13
- line as shown in Figure 17.14
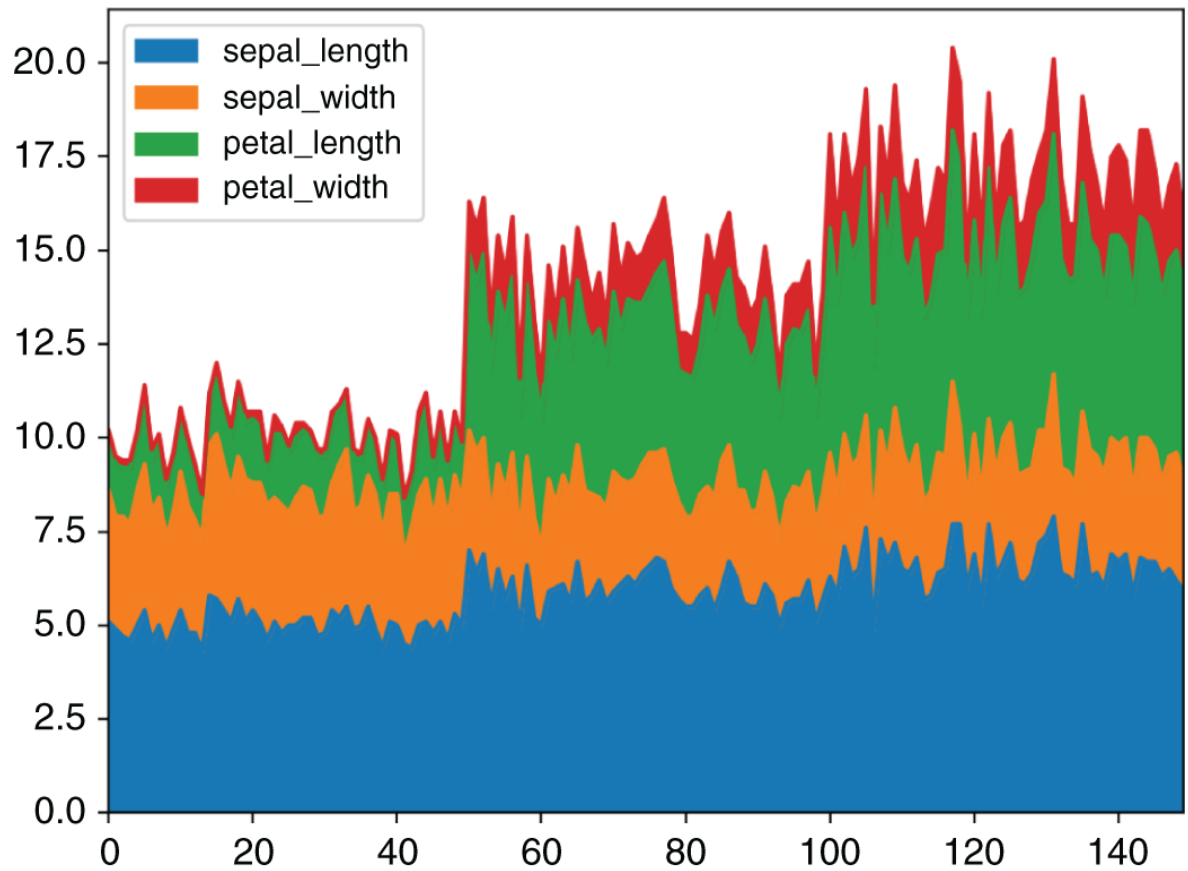
Figure 17.13 Area plot of iris DataFrame.
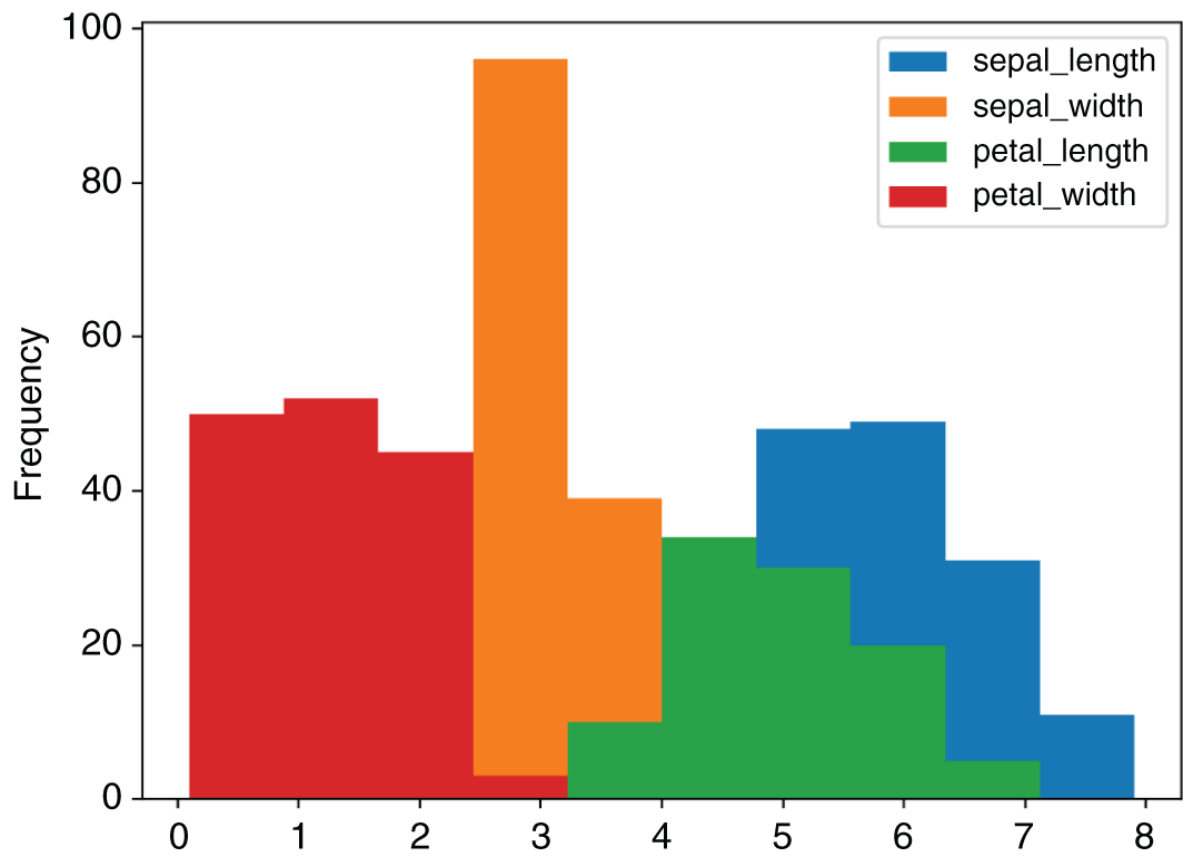
Figure 17.14 Histogram of iris DataFrame.
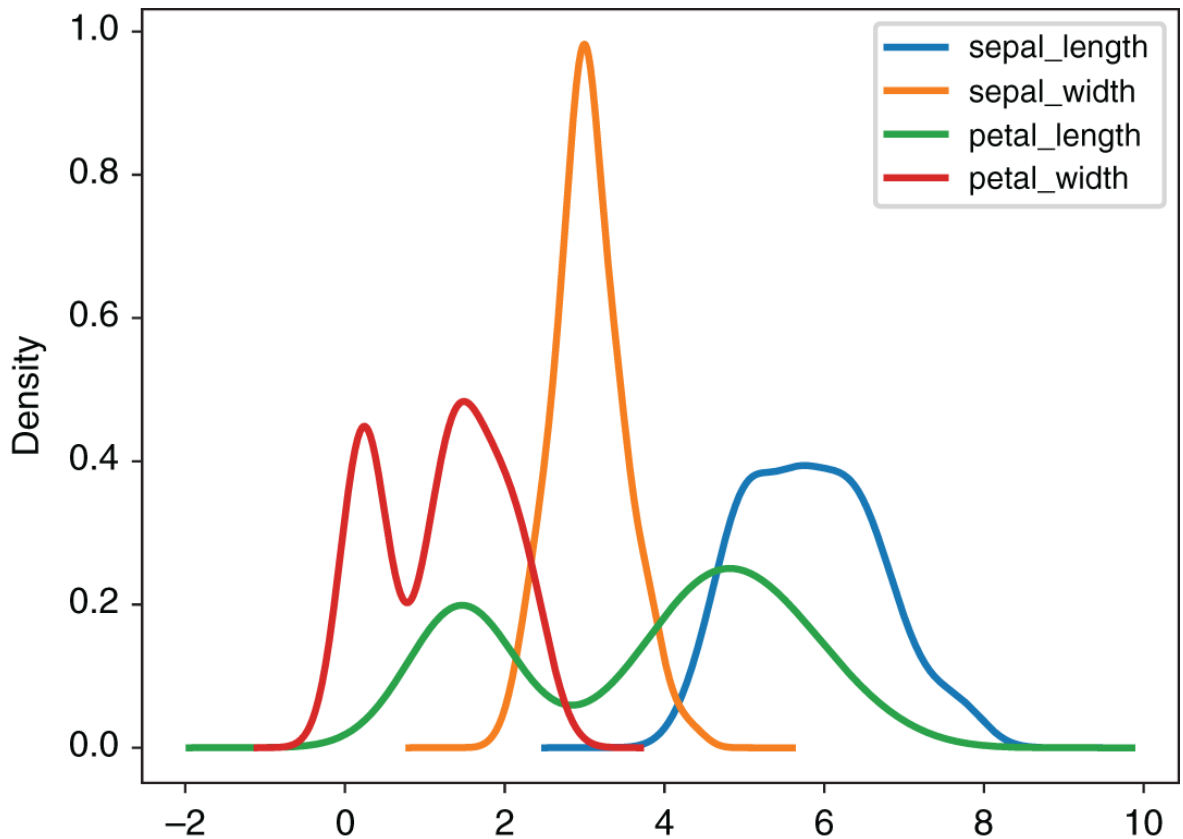
Figure 17.15 KDE of iris DataFrame.

- hist as shown in Figure 17.14
- kde as shown in Figure 17.15

Examples of these are shown below, with all being produced using iris.plot method as shown before.

There were a couple of plots that we did not include in the previous examples namely bar, barh, and pie (examples of a pie and barh plot are given in Figures 17.16 and 17.17). The reason being these plots require the data to be in the format suitable for the plot so we need data that has been manipulated to be in that form. Thus far all other

plots have been directly on the DataFrame but here we will group the tips DataFrame to get in a format that we can use.
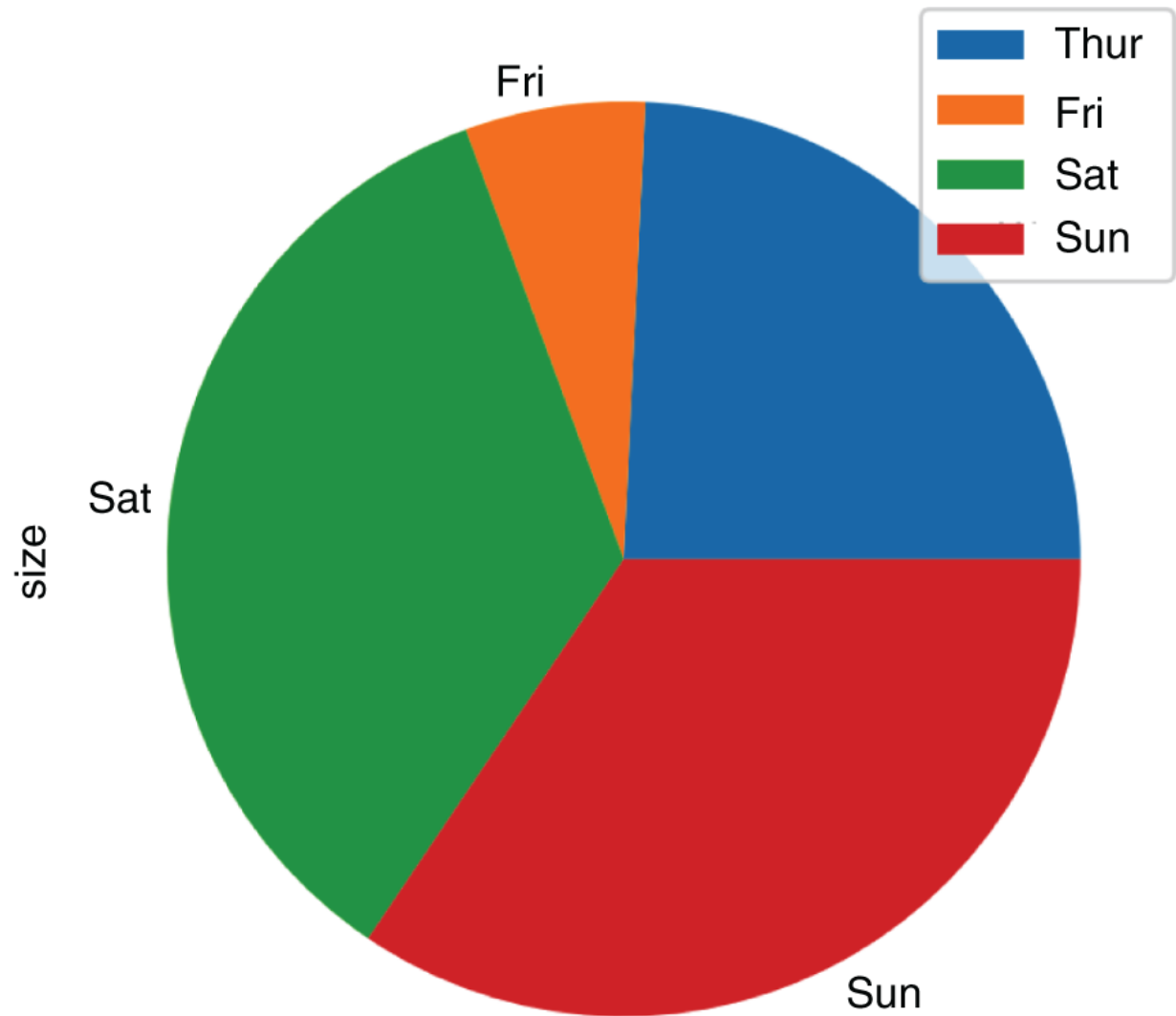
Figure 17.16 Pie plot of tip size by day.
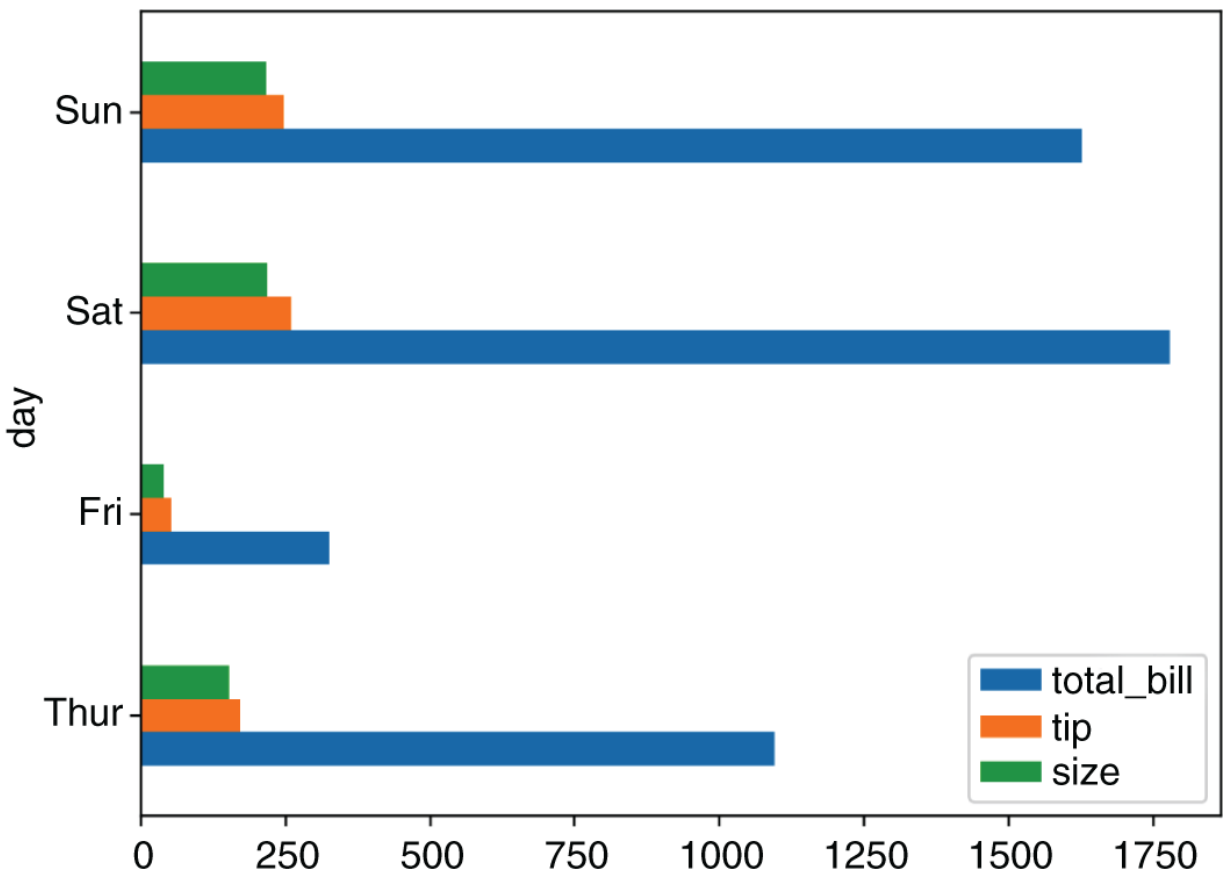
Figure 17.17 Barh plot of tips data by day.

```
>>> grouped = tips.groupby(['sex', 'smoker'])
>>> grouped.sum()
>>> data.plot.pie(y='size')

>>> grouped = tips.groupby(['sex', 'smoker'])
>>> grouped.sum()
>>> data.plot.bar()
```

17.2 Matplotlib

So here we have shown a number of plots that you can easily gener-
ate with pandas and matplotlib; next, we will look at how we can cus-

tomise them.

```
>>> fig = plt.figure()
>>> plt.plot(iris.sepal_length, '-')
[<matplotlib.lines.Line2D object at 0x1a1ed400b8>]
>>> plt.plot(iris.petal_length, '--')
[<matplotlib.lines.Line2D object at 0x1a1ed408d0>]
>>> fig.savefig('iris_plot.pdf')
```

Now in the above code, we use a different approach to creating our plot (Figure 17.18) using one which is more common from a matplotlib point of view. Here we setup a figure and plot lines on them using the plot method. In this example, we specify the line type by using an extra argument to determine if the line is solid or partially solid. Lastly, we save the plot using the savefig method, we can find what filetypes are supported by running the following command.

```
>>> fig.canvas.get_supported_filetypes()
{'ps': 'Postscript', 'eps': 'Encapsulated Postscript',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```
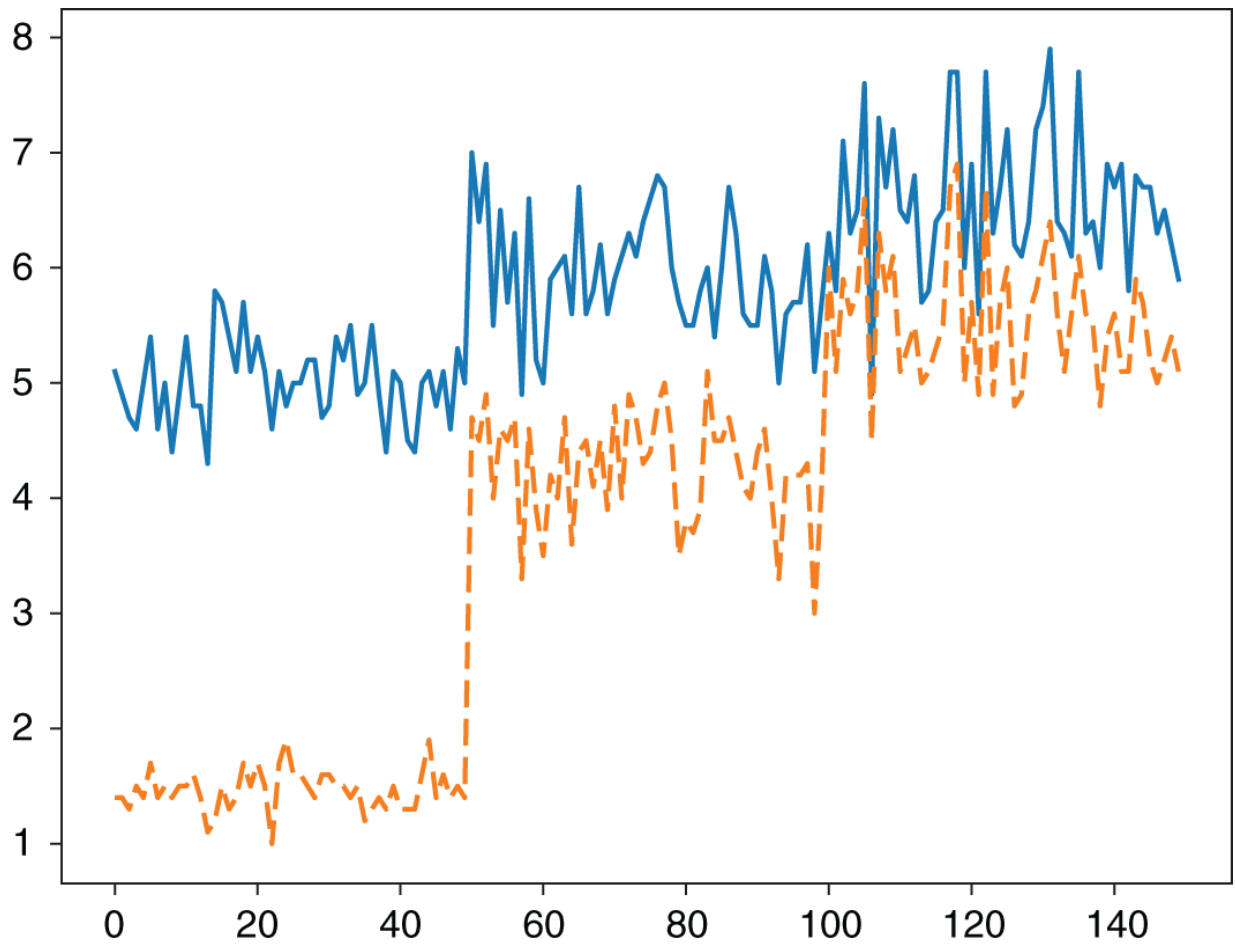
Figure 17.18 Iris plot.

The next example we look at is a panel plot as shown in Figure 17.19. This is where we have two plots on one figure as opposed to plotting two lines on one plot.

```
>>> plt.clf()
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> plt.subplot(2, 1, 1)
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f6e0358>
>>> plt.plot(iris.sepal_length)
[<matplotlib.lines.Line2D object at 0x1a1b0b1e48>]
>>> plt.subplot(2, 1, 2)
<matplotlib.axes._subplots.AxesSubplot object at 0x1a1b0b1cc0>
>>> plt.plot(iris.petal_length)
[<matplotlib.lines.Line2D object at 0x1a1b1e7438>]
```

The first thing we do is create the figure in the usual way and we then use the subplot method to create two panels. The arguments for the subplot method is

- Number of rows
- Number of columns
- Index
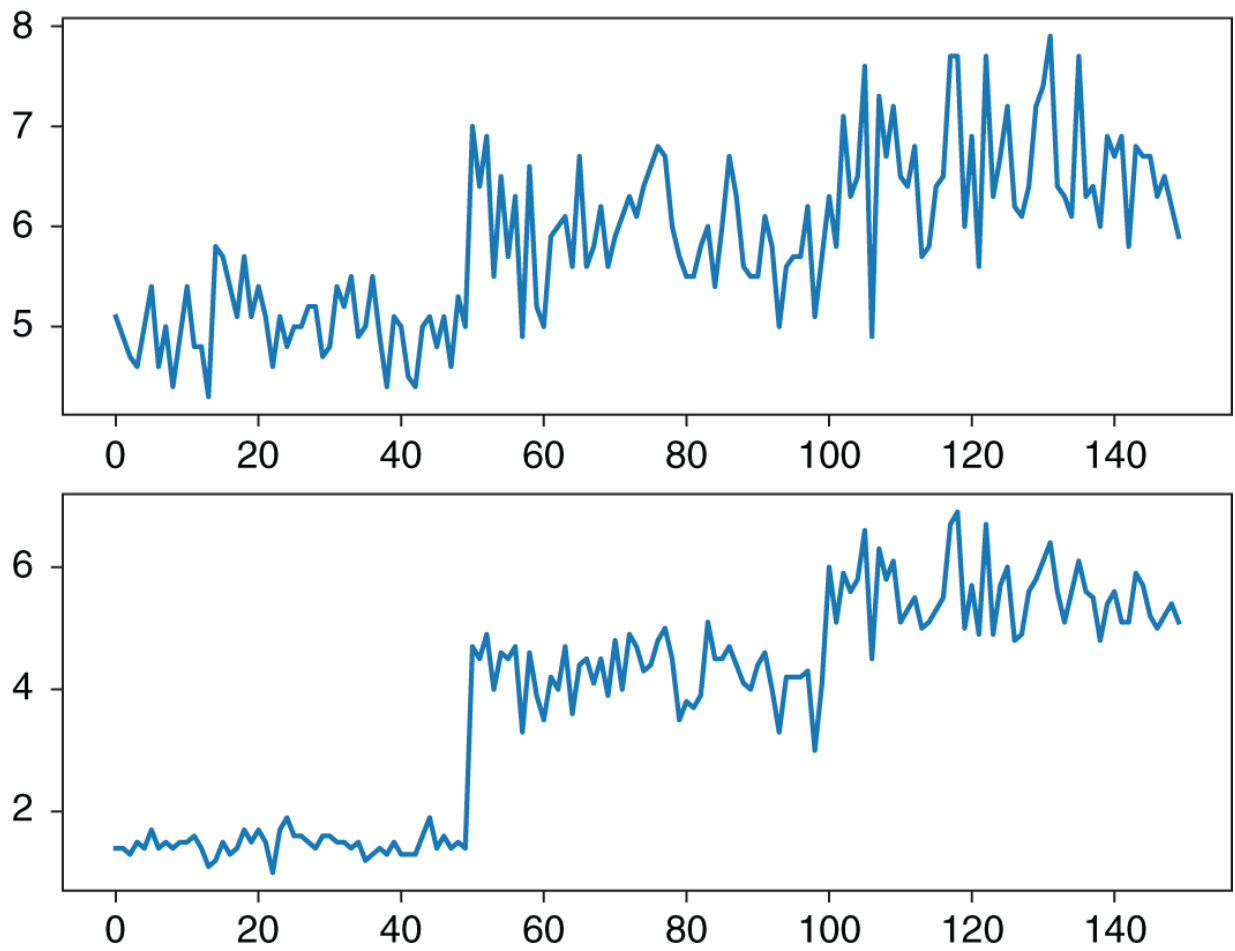
Figure 17.19 Panel plot example one.
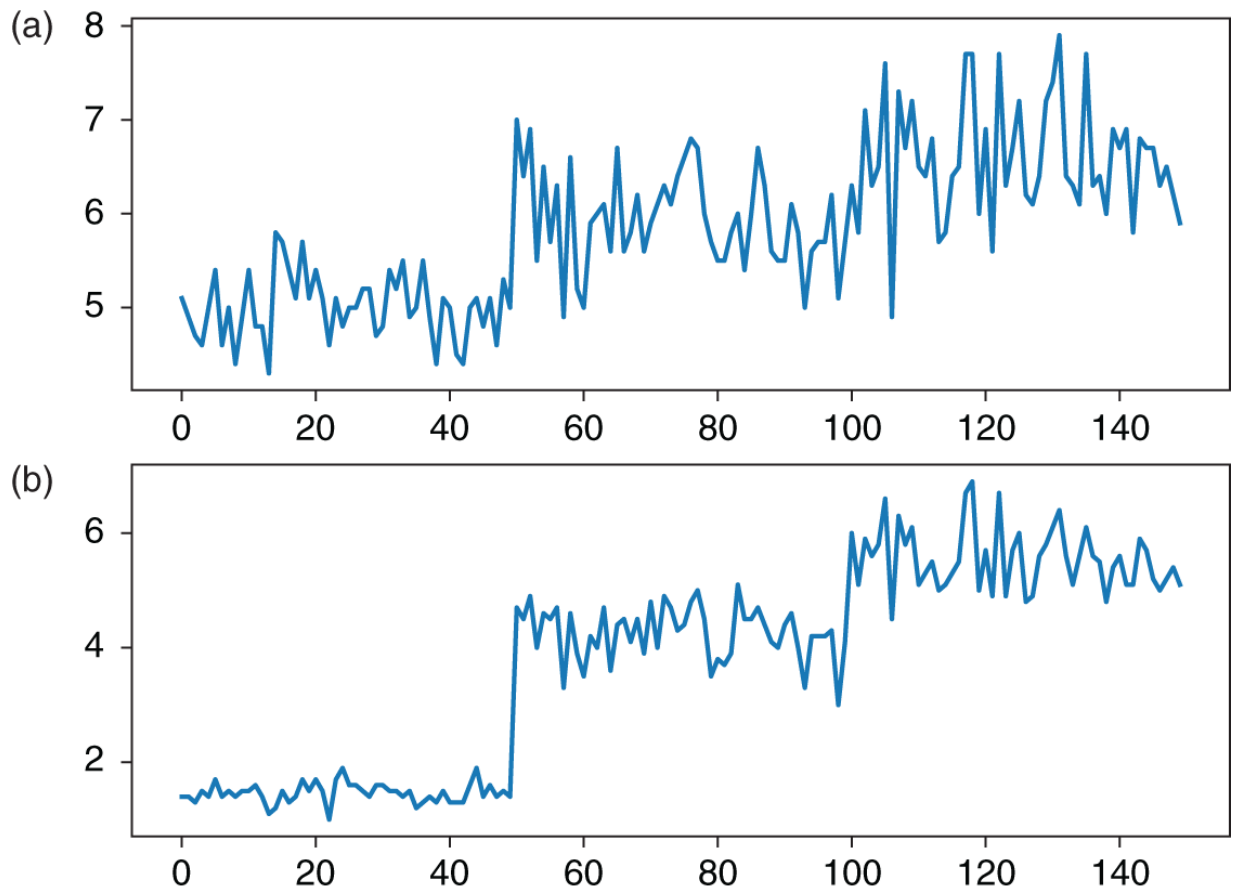
Figure 17.20 Panel plot example two.

So in this example, we have two rows with one column and initially we plot on index one the line of sepal length.

Next, we plot the line of petal length in the second panel and we now have an example of a panel plot as shown in Figure 17.20.

```
>>> plt.clf()
>>> fig, ax = plt.subplots(2)
>>> ax[0].plot(iris.sepal_length)
[<matplotlib.lines.Line2D object at 0x1a1c35e9e8>]
>>> ax[1].plot(iris.petal_length)
[<matplotlib.lines.Line2D object at 0x1a1f693358>]
```

We can achieve the same thing using the subplots method directly specifying we want two subplots and using the ax variable to control what goes into each subplot. Here we can use the plot method directly to plot sepal length in the first subplot and petal length in the second.

Next, we will look at how the plots look. In matplotlib, we can control the styling of the plot and by running the below command you get a list as follows

```
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks',
'fivethirtyeight', 'seaborn-whitegrid', 'classic',
'_classic_test', 'fast', 'seaborn-talk',
'seaborn-dark-palette', 'seaborn-bright',
'seaborn-pastel', 'grayscale', 'seaborn-notebook',
'ggplot', 'seaborn-colorblind', 'seaborn-muted',
'seaborn', 'Solarize_Light2', 'seaborn-paper',
'bmh', 'tableau-colorblind10', 'seaborn-white',
'dark_background', 'seaborn-poster', 'seaborn-deep']
```

To get a more detailed view of how these differ please refer to the documentation
https://matplotlib.org/3.1.1/gallery/style_sheets/style_sheets_reference.html where there are images relating to the specific types. What is important to note is that we can alter how a plot looks by just using a different style sheet.

```
>>> plt.clf()
>>> plt.style.use('seaborn-whitegrid')
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> plt.plot(tips.tip, color='blue')
[<matplotlib.lines.Line2D object at 0x1a1b0562e8>]
>>> plt.plot(tips.tip+5, color='g')
[<matplotlib.lines.Line2D object at 0x1a1b056240>]
>>> plt.plot(tips.tip+10, color='0.75')
[<matplotlib.lines.Line2D object at 0x1a1b056748>]
>>> plt.plot(tips.tip+15, color='#FFDD44')
[<matplotlib.lines.Line2D object at 0x1a1b056a20>]
>>> plt.plot(tips.tip+20, color=(1.0,0.2,0.3))
[<matplotlib.lines.Line2D object at 0x1a1b056f98>]
>>> plt.plot(tips.tip+25, color='chartreuse')
[<matplotlib.lines.Line2D object at 0x1a1b07c5c0>]
```

Here we import the seaborn whitegrid style to use in this example and we setup the figure and axes. Next, we plot the same tips line but in each instance we add 5 to each value so we can see the difference between each of the lines. Now for each line we use, a different approach to give it a colour. The first line using the colour argument set as blue and we can use the colour as a name. The second line uses the colour set as a single letter where in this case g refers to green. For the third line, we use the grayscale value between 0 and 1. The forth line shows how we can use hex codes to pass in the colour. The fifth line shows how you use the RGB tuple where each value can be between 0 and 1 with the first referring to red the second to green and third to blue. The result of this is shown in Figure 17.21.

Next, we consider different line types that we can use within mat-
plotlib where we use the linestyle argument to pass in different types
of lines as shown in .

```
>>> plt.clf()
>>> plt.style.use('fivethirtyeight')
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> plt.plot(tips.tip, linestyle='solid')
[<matplotlib.lines.Line2D object at 0x1a1f69bf98>]
>>> plt.plot(tips.tip + 5, linestyle='dashed')
[<matplotlib.lines.Line2D object at 0x1a1f6bf438>]
>>> plt.plot(tips.tip + 10, linestyle='dashdot')
[<matplotlib.lines.Line2D object at 0x1a1f6bf7b8>]
>>> plt.plot(tips.tip + 15, linestyle='dotted');
[<matplotlib.lines.Line2D object at 0x1a1f6bfb38>]
>>> plt.plot(tips.tip + 20, linestyle='-')
[<matplotlib.lines.Line2D object at 0x1a1f69bfd0>]
>>> plt.plot(tips.tip + 25, linestyle='--')
[<matplotlib.lines.Line2D object at 0x1a1f6bfe48>]
>>> plt.plot(tips.tip + 30, linestyle='-.')
[<matplotlib.lines.Line2D object at 0x1a1c352748>]
>>> plt.plot(tips.tip + 35, linestyle=':')
[<matplotlib.lines.Line2D object at 0x1a1f6c3978>]
```
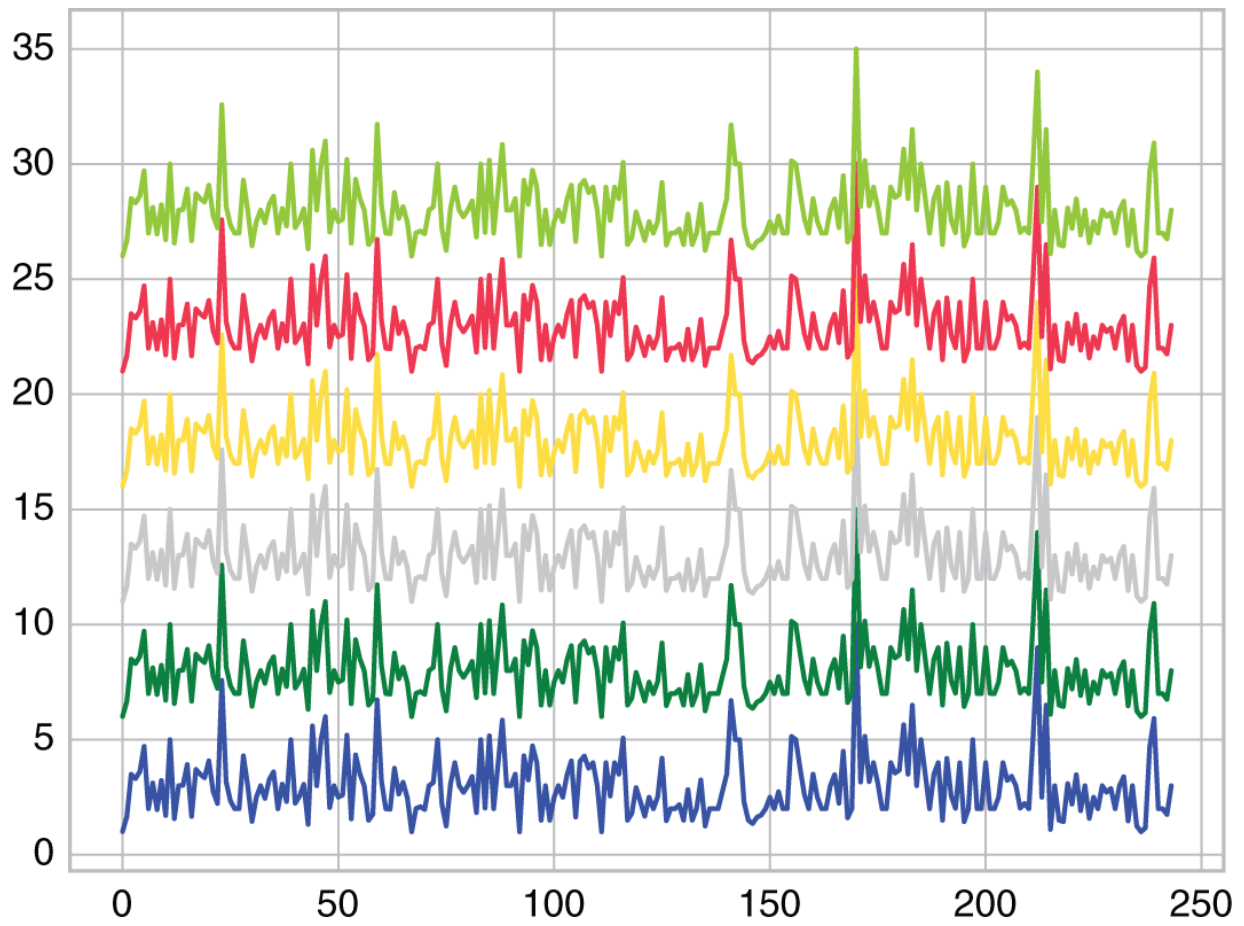
Figure 17.21 Plot with custom line colour.
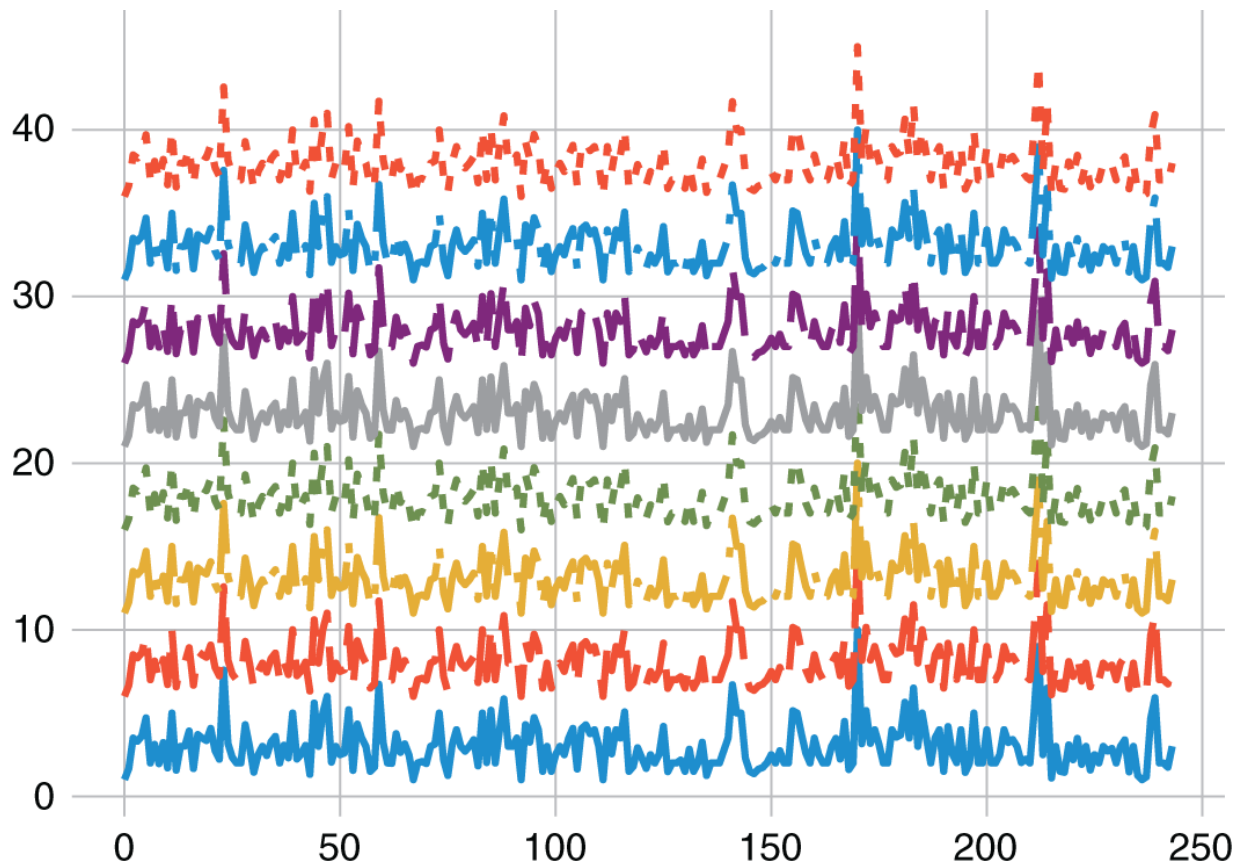
Figure 17.22 Plot with custom linetype.
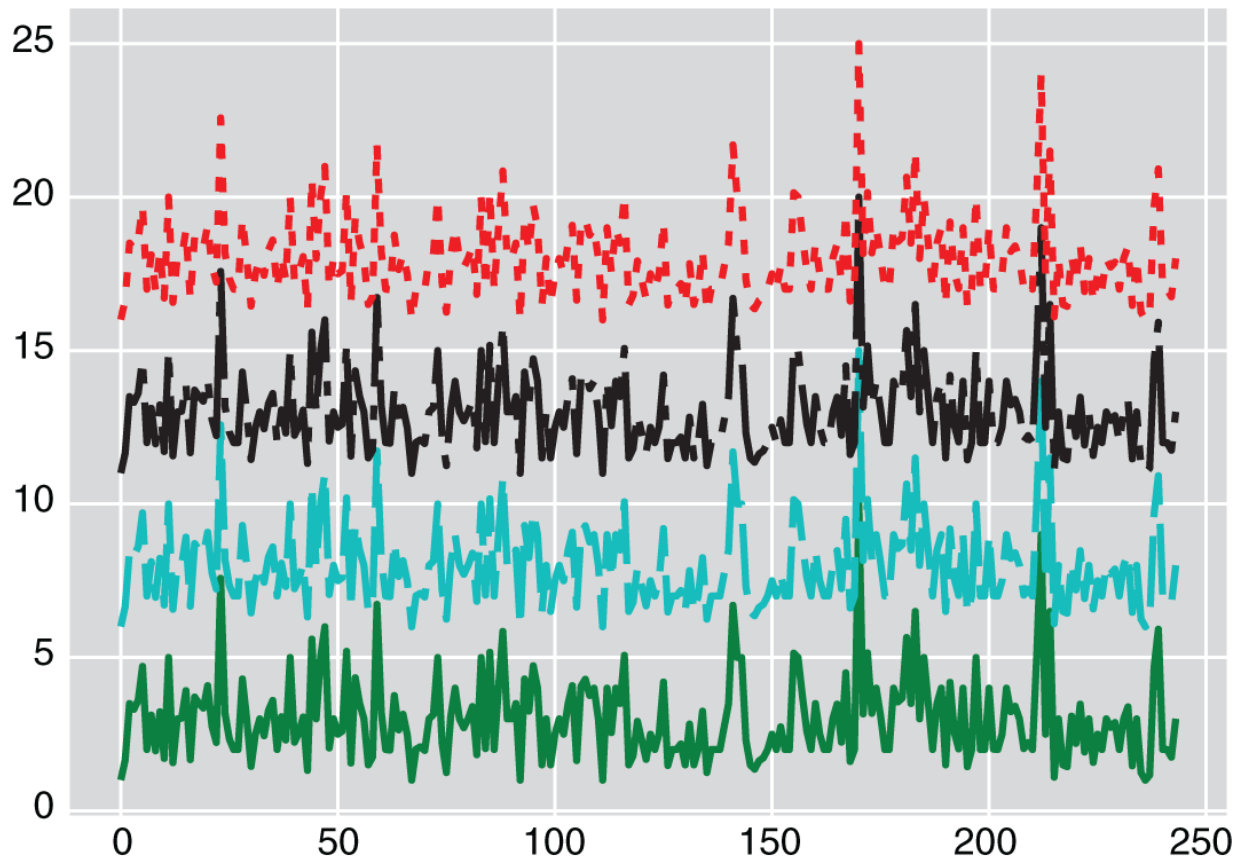
Figure 17.23 Plot with custom colour linetype.

We can also combine linestyle and colour into one argument to give different coloured lines on our plots. To do this, we use the short linestyle combined with the single letter colours the result of which is shown in Figure 17.23.

```
>>> plt.clf()
>>> plt.style.use('ggplot')
>>> fig = plt.figure()
>>> ax = plt.axes()
>>> plt.plot(tips.tip, '-g')   # solid green
[<matplotlib.lines.Line2D object at 0x1a1b087940>]
>>> plt.plot(tips.tip + 5, '--c') # dashed cyan
[<matplotlib.lines.Line2D object at 0x1a1b0876a0>]
>>> plt.plot(tips.tip + 10, '-.k') # dashdot black
[<matplotlib.lines.Line2D object at 0x1a1b087668>]
>>> plt.plot(tips.tip + 15, ':r')   # dotted red
[<matplotlib.lines.Line2D object at 0x1a1b045908>]
```

In the next examples, we will look at how we change the limits and add some labelling to plots, we begin by looking at setting the limits for the plot.

```
>>> fig = plt.figure()
>>> plt.plot(tips.tip)
[<matplotlib.lines.Line2D object at 0x1a20f81160>]
>>> plt.xlim(50, 200)
(50, 200)
>>> plt.ylim(0, 10)
(0, 10)
```

```
>>> fig = plt.figure()
>>> plt.plot(tips.tip)
>>> plt.xlim(200, 50)
>>> plt.ylim(10, 0)
```

In the first example shown in Figure 17.24, we plot the tip from the tips dataset and here we customise the x limits and y limits using xlim and ylim methods. To use these methods, you simply put in the start and end value where you want the axis to start and end with us using 50 to 200 for the x axis and 0 to 10 for the y axis.

In Figure 17.25, we reverse the axis by simply using 200 to 50 and 10 to 0 for the x axis and the y axis respectively.
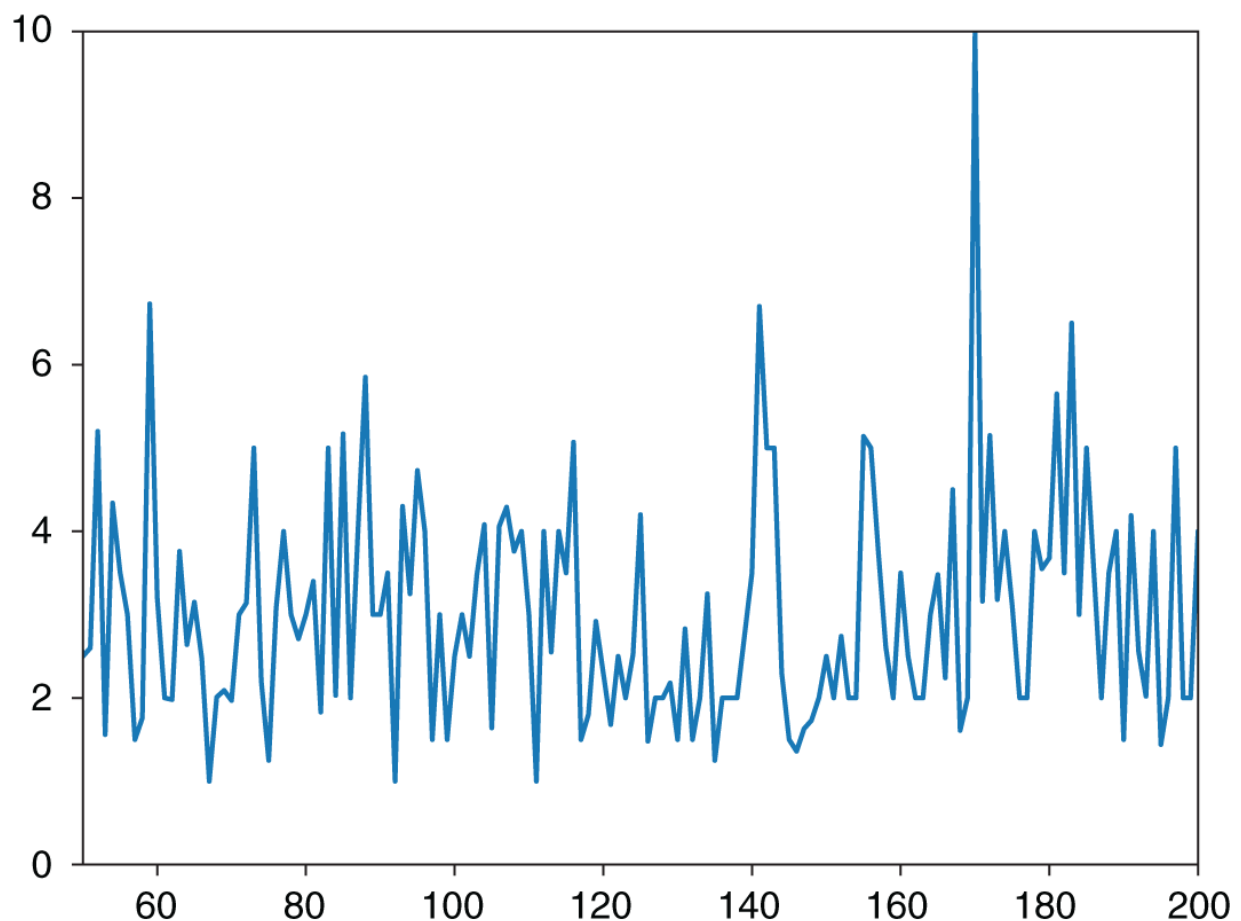
Figure 17.24 Plot with limits altered.
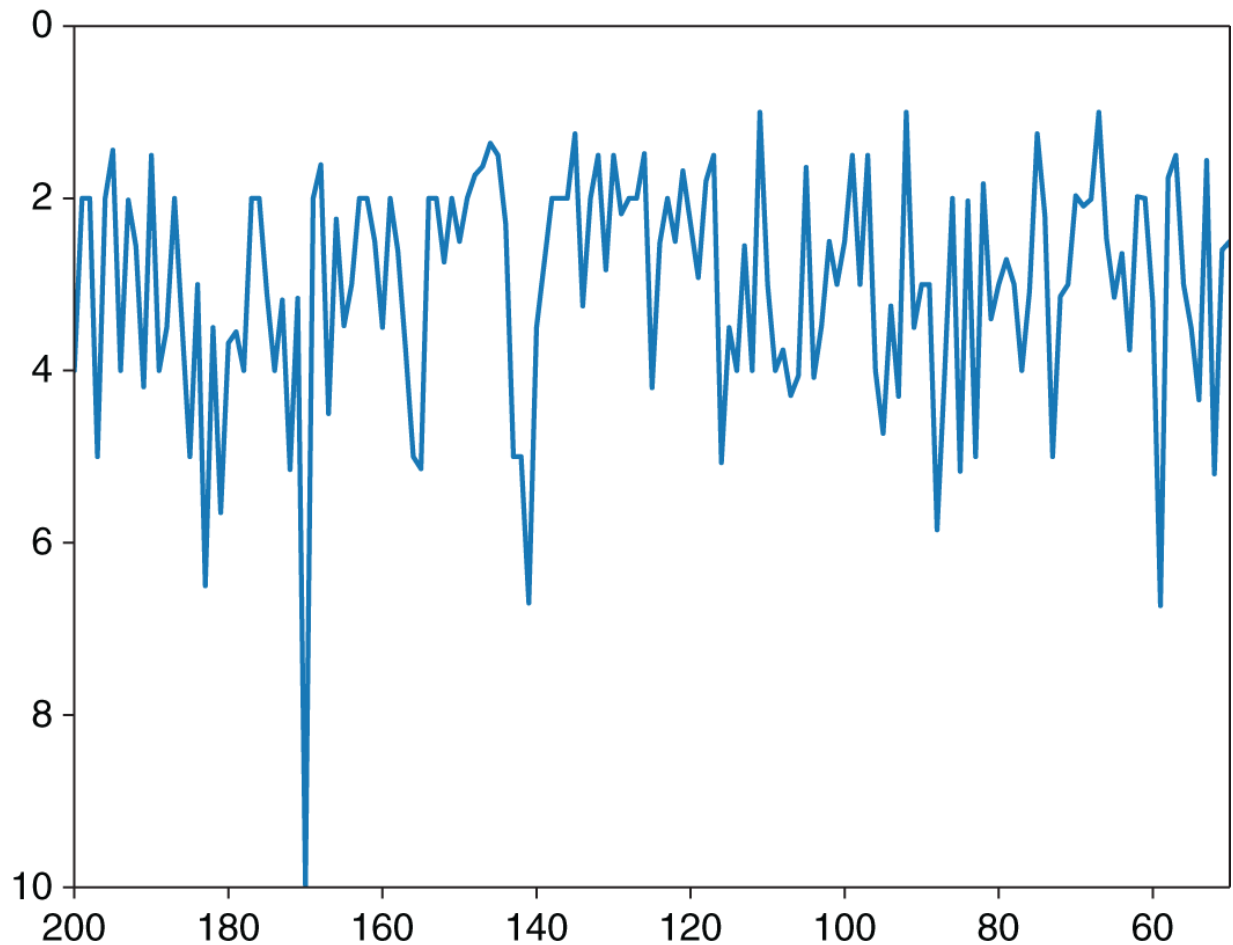
Figure 17.25 Plot with reverse limits.

```
>>> fig = plt.figure()
>>> plt.plot(tips.tip)
[<matplotlib.lines.Line2D object at 0x1a21149518>]
>>> plt.title("Tips from the seaborn tips dataset")
Text(0.5, 1.0, 'Tips from the seaborn tips dataset')
>>> plt.xlabel("Indiviual")
Text(0.5, 0, 'Indiviual')
>>> plt.ylabel("Tip (£)")
Text(0, 0.5, 'Tip (£)')
```

In the first example, shown in Figure 17.26, we label the plot to display titles and axis names by using the methods title, xlabel, and ylabel,

which populate the plot title, x axis, and y axis, respectively. We can also add a legend to the plot (shown in Figure 17.27) using the legend method. In using this we add the label variable to the plot where we specify the name we want to be shown in the legend.

We next consider some special plots showcasing what we can do beyond standard plots within matplotlib.
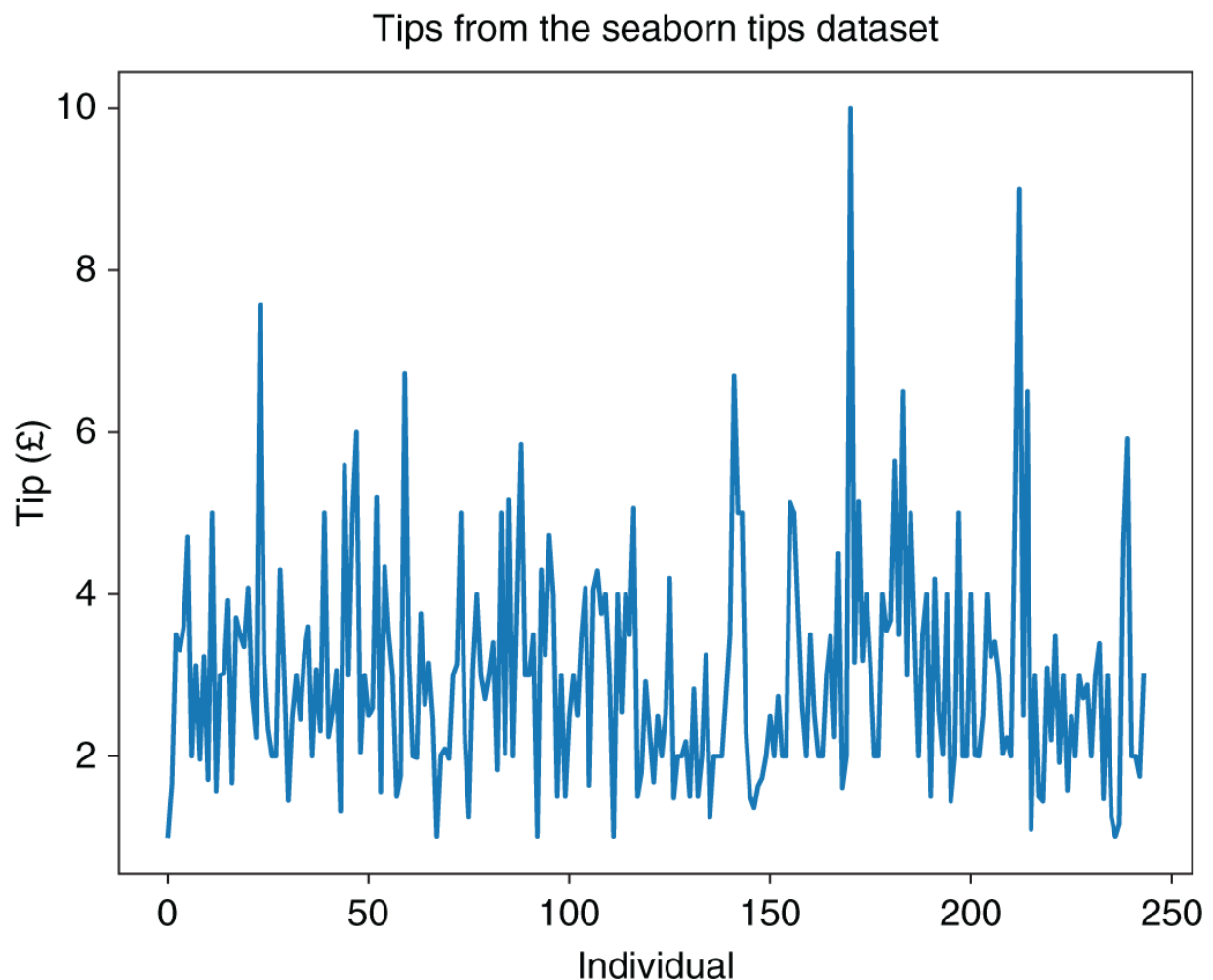
Figure 17.26 Plot with labels.
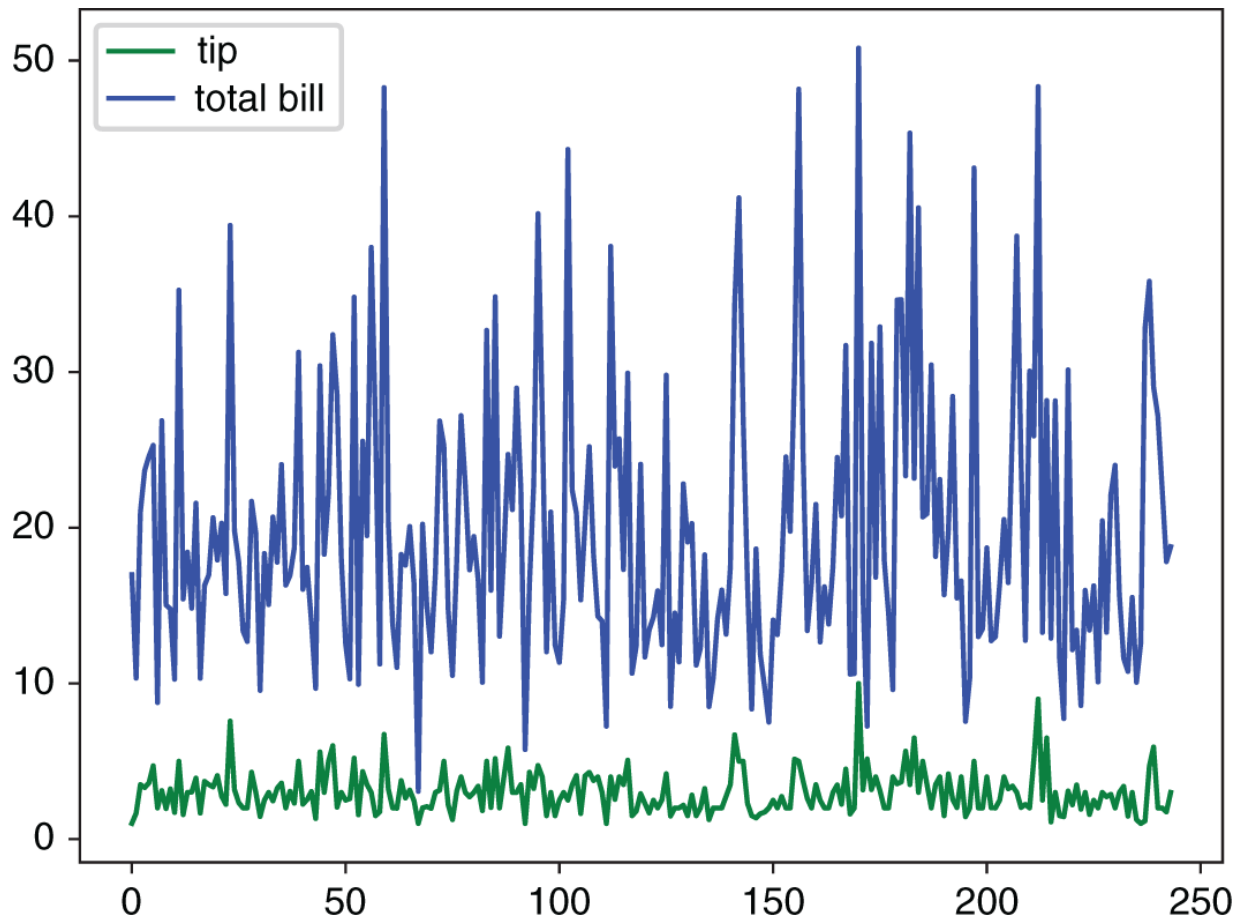
Figure 17.27 Plot with legend.

```
>>> fig = plt.figure()
>>> plt.plot(tips.tip, '-g', label='tip')
>>> plt.plot(tips.total_bill, '-b', label='total bill')
>>> plt.axis('equal')
>>> plt.legend()
```

```
>>> fig = plt.figure()
>>> rng = np.random.RandomState(0)
>>> for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>',
    's', 'd']:
...     plt.plot(rng.rand(5), rng.rand(5), marker,
        label="marker='{0}'".format(marker))
...
[<matplotlib.lines.Line2D object at 0x1a211880b8>]
[<matplotlib.lines.Line2D object at 0x1a211888d0>]
[<matplotlib.lines.Line2D object at 0x1a21188d30>]
[<matplotlib.lines.Line2D object at 0x1a20fe1860>]
[<matplotlib.lines.Line2D object at 0x1a21064160>]
[<matplotlib.lines.Line2D object at 0x1a210640f0>]
[<matplotlib.lines.Line2D object at 0x1a21064128>]
[<matplotlib.lines.Line2D object at 0x1a20ff9f98>]
[<matplotlib.lines.Line2D object at 0x1a210bfb00>]
[<matplotlib.lines.Line2D object at 0x1a20765cf8>]
[<matplotlib.lines.Line2D object at 0x1a2118f240>]
>>> plt.legend(numpoints=1)
<matplotlib.legend.Legend object at 0x1a2119c4a8>
>>> plt.xlim(0, 1.8)
(0, 1.8)
```

In this first example shown in Figure 17.28, we show how to use a different markers on a plot. To do so we use the numpy RandomState to generate arrays of random numbers. We then loop over the list of markers that we want to plot and generate a five value random array to plot all of these values and add a legend to show what each point in our code refers to what on the plot.
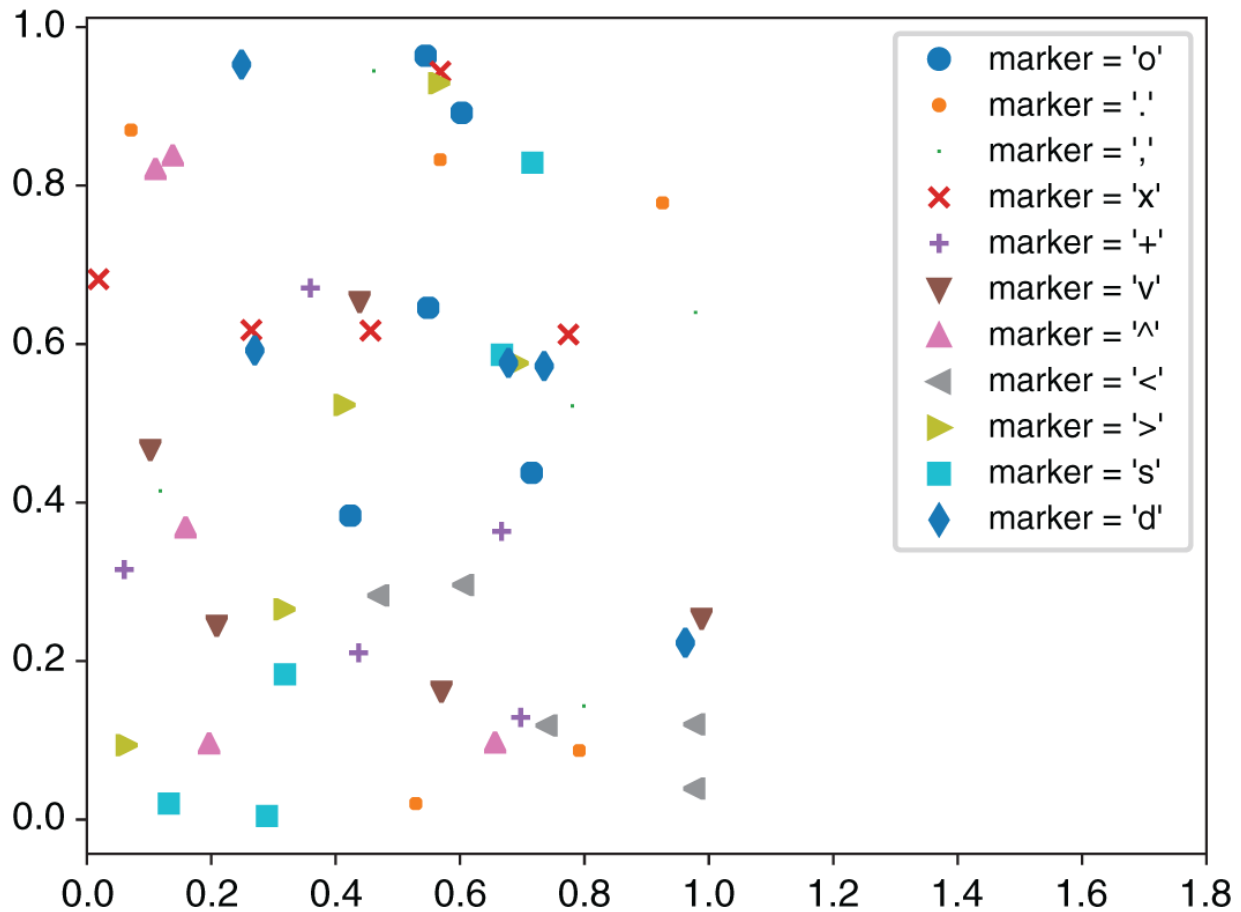
Figure 17.28 Scatter plots with different markers.

```
>>> fig = plt.figure()
>>> rng = np.random.RandomState(0)
>>> x = rng.randn(100)
>>> y = rng.randn(100)
>>> colors = rng.rand(100)
>>> sizes = 1000 * rng.rand(100)
>>> plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
<matplotlib.collections.PathCollection object at 0x1a22ae5c88>
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar object at 0x1a2115beb8>
```

The next plot Figure 17.29 that we create is a scatter plot where each element has a colour and size associated with it. To do this, we create a figure in the normal way and again use RandomState to allow us to

generate random numbers. In this case, we calculate the x and y co-ordinates for the scatter plot by generating 100 random numbers. We next generate another 100 random numbers to determine the colours. Lastly, we generate the size of the scatter by multiplying another 100 random numbers by 1000 to make the ball size varied enough so that we can see them. These then get passed into the scatter method setting the c value to be the colours array and s to be the sizes array. We use the alpha argument to set how opaque the plots are. The last argument that we set is the cmap value which sets the colour map that we want to use. Once we have plotted the scatter, we can add the colour bar to show what the colours refer to relative to the random numbers from the colour array.
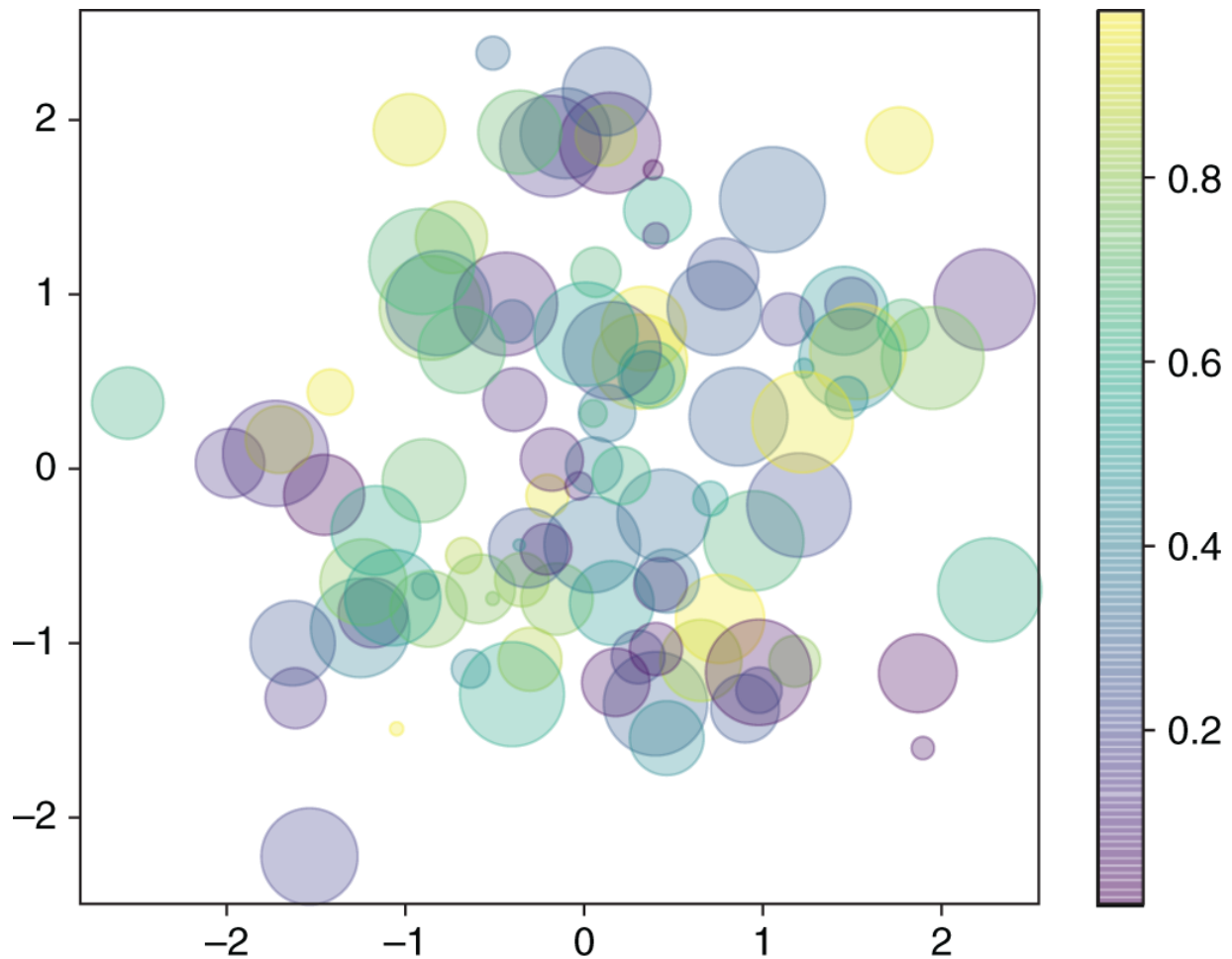
Figure 17.29 Scatter plot with different sizes.

17.3 **Seaborn**

Having covered direct plotting from pandas objects and how we can use matplotlib to create and customise plots, we next move onto Seaborn which is the natural progression as its built using matplotlib and is highly coupled with pandas DataFrames. In Seaborns own words, the functionality that it offers is as follows.

- A dataset-oriented application programming interface (API) for examining relationships between multiple variables.
- Specialised support for using categorical variables to show observations or aggregate statistics.
- Options for visualising univariate or bivariate distributions and for comparing them between subsets of data.
- Automatic estimation and plotting of linear regression models for different kinds dependent variables.
- Convenient views onto the overall structure of complex datasets.
- High-level abstractions for structuring multi-plot grids that let you easily build complex visualisations.
- Concise control over matplotlib figure styling with several built-in themes.
- Tools for choosing colour palettes that faithfully reveal patterns in your data.

What does this mean? In simple terms what Seaborn offers is some out of the box plots which look great and work well with DataFrames but can be configured to a high level.

The best way to show this is through example so let us dive in and get started

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style="darkgrid")
>>> tips = sns.load_dataset("tips")
>>> tips.columns
Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time',
       'size'], dtype='object')
>>> sns.relplot(x="total_bill", y="tip", data=tips)
```

The above code loads up the tips dataset in a manner similar to what we have done previously in the book. We then use the replot method and pass in the name of the DataFrame as the argument to the data option and set x and y to the column names from the DataFrame.

The result of this is a scatter plot of the x and y variables shown in Figure 17.30. Now we can extend this by adding a third dimension. In this case, we want to plot the same scatter plot but now segment the data by whether they are a smoker or not. To do this we use the hue option and pass in the name smoker which relates to the column within the DataFrame. Note here the values of smoker are yes and no which mimics what we see in the plot
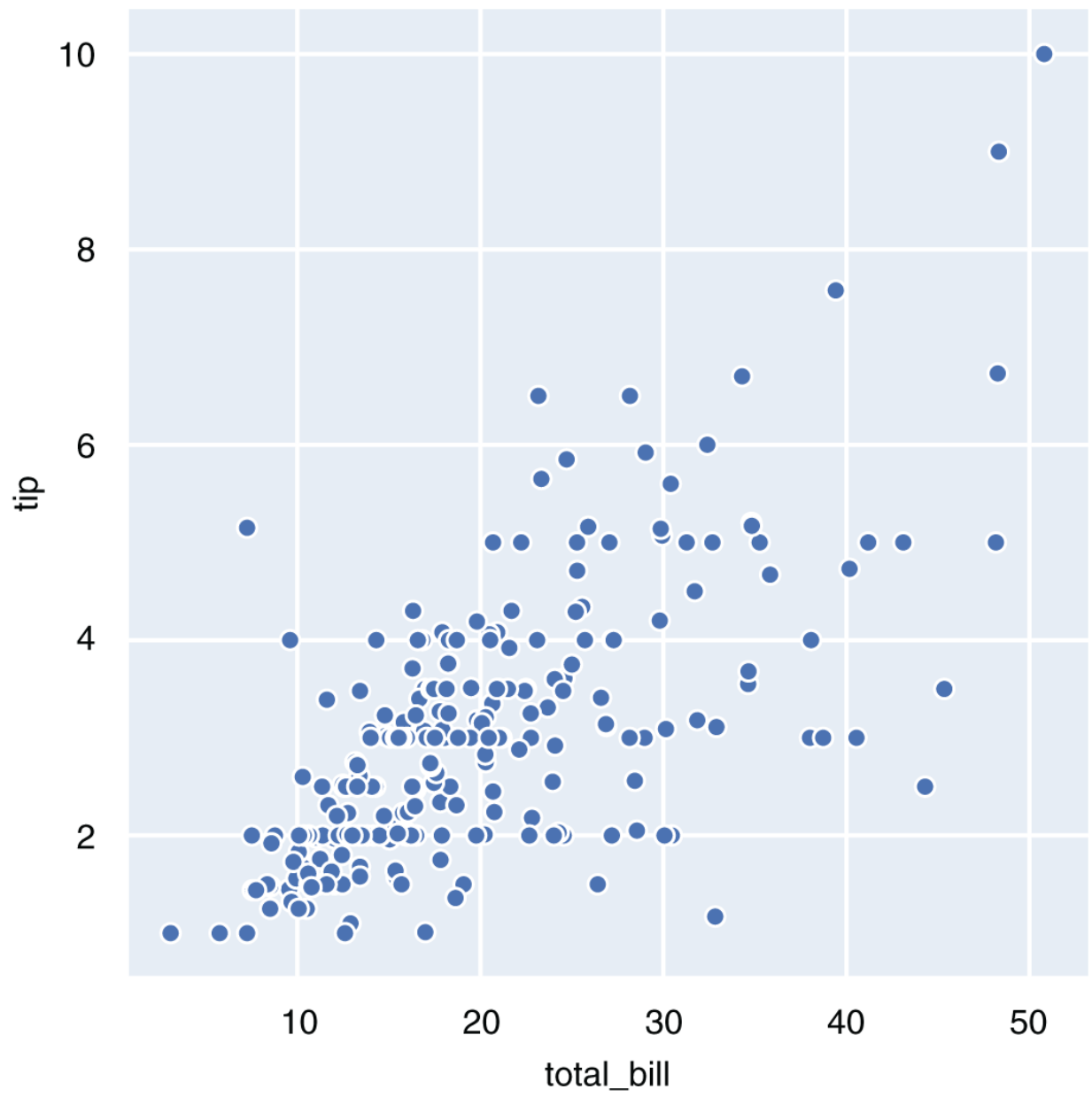
Figure 17.30 Scatter using replot in seaborn.

```
>>> tips.head()
   total_bill    tip      sex smoker    day     time   size
0       16.99   1.01   Female     No    Sun   Dinner      2
1       10.34   1.66     Male     No    Sun   Dinner      3
2       21.01   3.50     Male     No    Sun   Dinner      3
3       23.68   3.31     Male     No    Sun   Dinner      2
4       24.59   3.61   Female     No    Sun   Dinner      4
>>> tips.smoker.unique()
[No, Yes]
Categories (2, object): [No, Yes]
>>> sns.relplot(x="total_bill", y="tip", hue="smoker", data=tips);
```

The plot Figure 17.31 conveniently colours the values related to whether they are a smoker or not however we can extend this by adding the variable style and setting it to smoker which will now change what is plotted for each category in smoker, which in this case is yes or no, this is shown in Figure 17.32.

```
>>> sns.relplot(x="total_bill", y="tip", hue="smoker",
    style="smoker", data=tips);
```

The hue value previously was a categorical variable which makes sense when applying but what happens if your hue value is a numeric value? To demonstrate this, we change the hue value to use the size variable in the dataset. Initially we used the dot syntax to obtain the unique value but we get an error and this is due to the fact that dot size on any DataFrame brings back the actual size of the DataFrame which is rows multiplied by columns. So to obtain the unique values we use the square bracket notation. The size value is then passed into the hue to give us a scatter plot using a numeric hue.
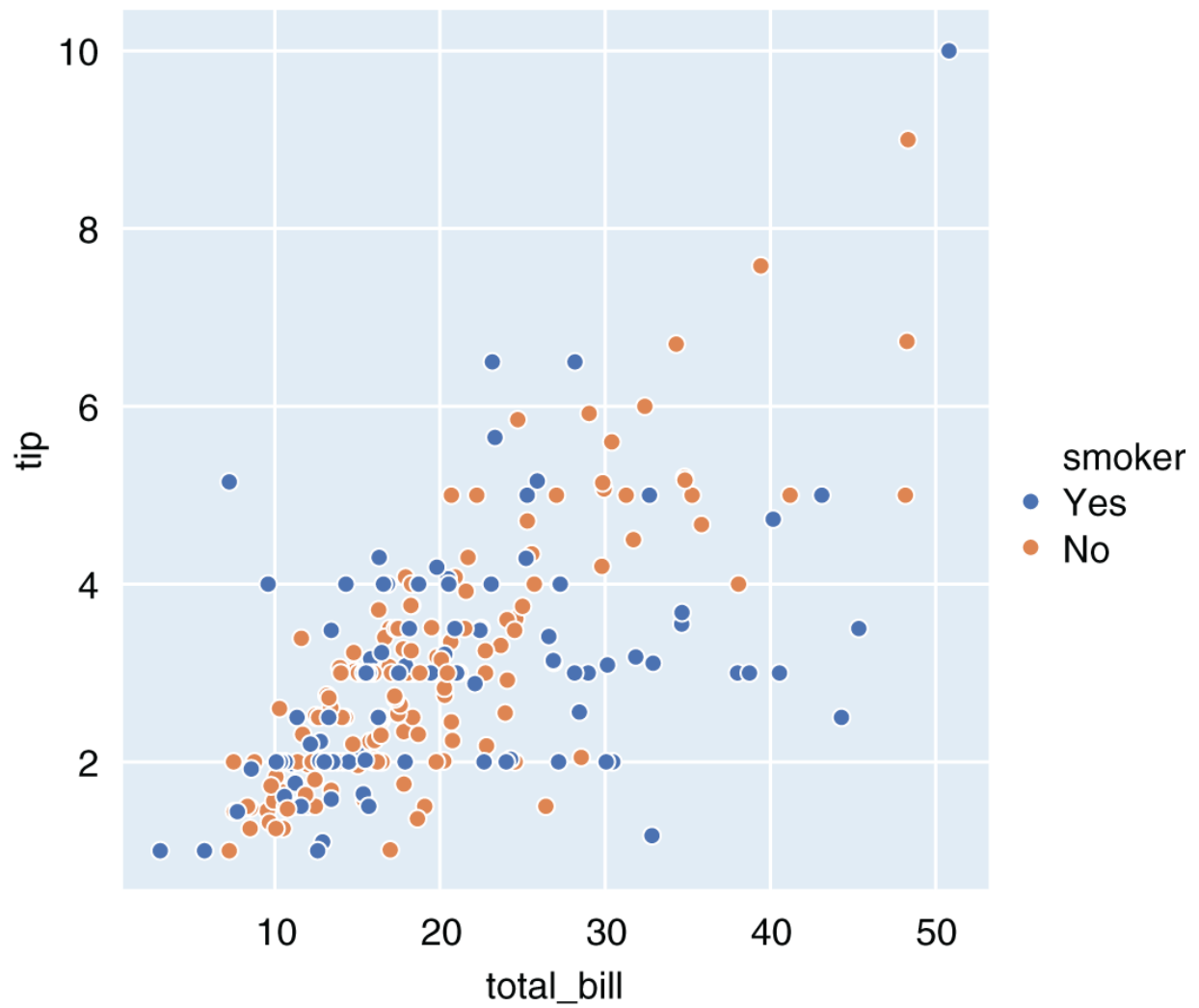
Figure 17.31 Scatter plot using replot in seaborn with a third variable.

Figure 17.32 Scatter plot using replot in seaborn with a third variable and styling.

```
>>> tips.dtypes
total_bill      float64
tip             float64
sex            category
smoker         category
day            category
time           category
size              int64
dtype: object
>>> tips.head()
   total_bill    tip      sex smoker   day    time   size
0       16.99   1.01   Female     No   Sun  Dinner      2
1       10.34   1.66     Male     No   Sun  Dinner      3
2       21.01   3.50     Male     No   Sun  Dinner      3
3       23.68   3.31     Male     No   Sun  Dinner      2
4       24.59   3.61   Female     No   Sun  Dinner      4
>>> tips.size.unique()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.int64' object has no attribute 'unique'
>>> tips.size
1708
>>> len(tips)*len(tips.columns)
1708
>>> tips['size'].unique()
array([2, 3, 4, 1, 6, 5])
>>> sns.relplot(x="total_bill", y="tip", hue="size", data=tips);
```
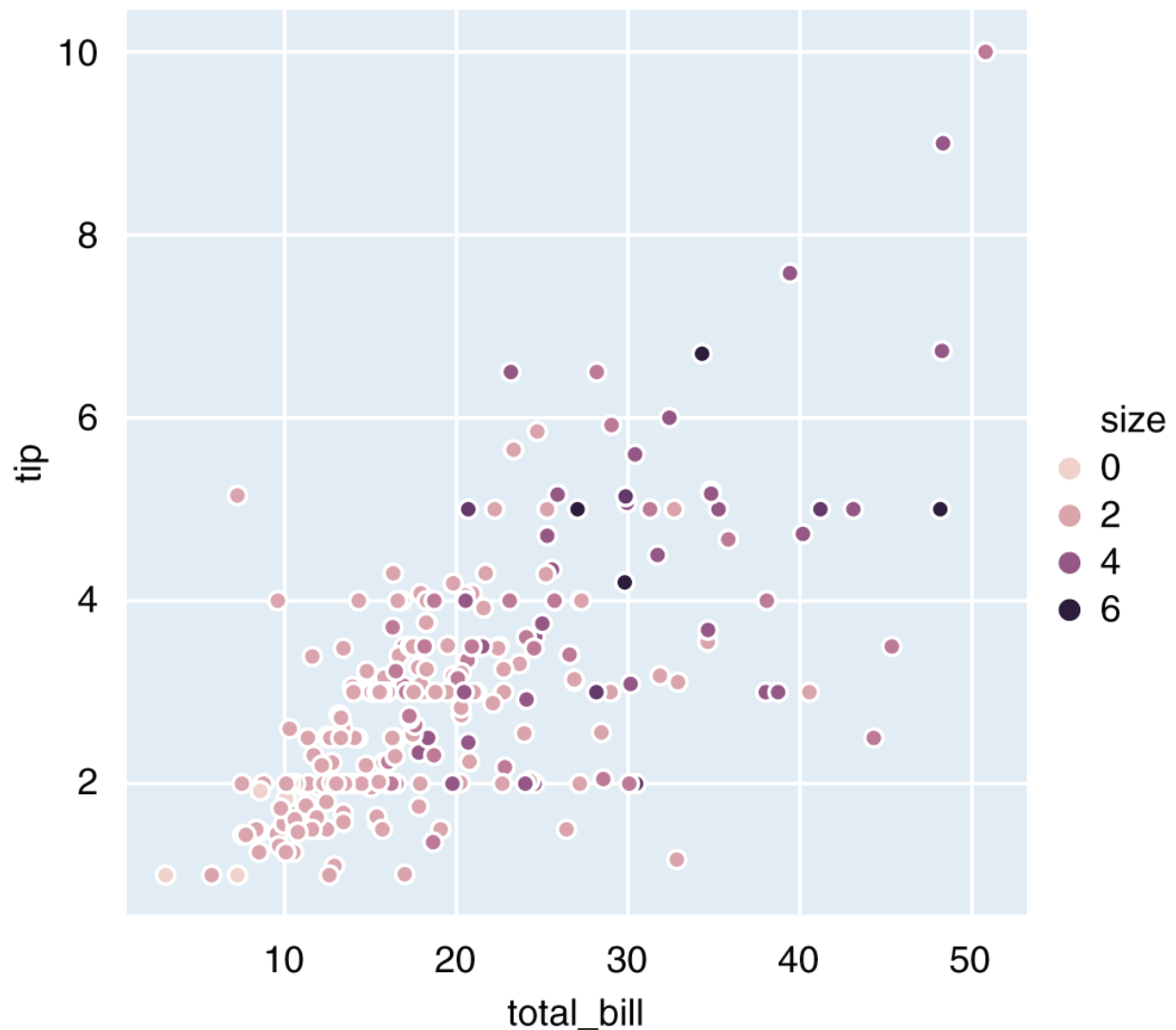
Figure 17.33 Scatter plot using replot in seaborn with hue on size.

What we see in the Figure 17.33 is that the size colours get darker as the value increases and seaborn casts the value to a float and uses a sequential palette taking care of this for you.

Having shown how to apply a third variable using hue we can also use the size option. As in the previous example, we will use the size

variable as the input to size and what this does is change the size of the point based on this third variable, this is shown in Figure 17.34.

```
>>> sns.relplot(x="total_bill", y="tip", size="size", data=tips);
```

We can expand upon the previous example by using the sizes variable this can be in the form of list, dictionary, or tuple. In the event of a categorical variable, a dictionary would contain the levels with the relevant values so for the smoker variable we could pass a dictionary as follows.

```
>>> tips.smoker.unique()
[No, Yes]
Categories (2, object): [No, Yes]
>>> sizes_dict = {'No': 100, 'Yes': 200}
>>> sns.relplot(x="total_bill", y="tip", size="smoker",
    sizes=sizes_dict, data=tips)
```

We can do the same with a list replacing the dictionary

```
>>> sns.relplot(x="total_bill", y="tip", size="smoker",
    sizes=[100,200], data=tips)
```
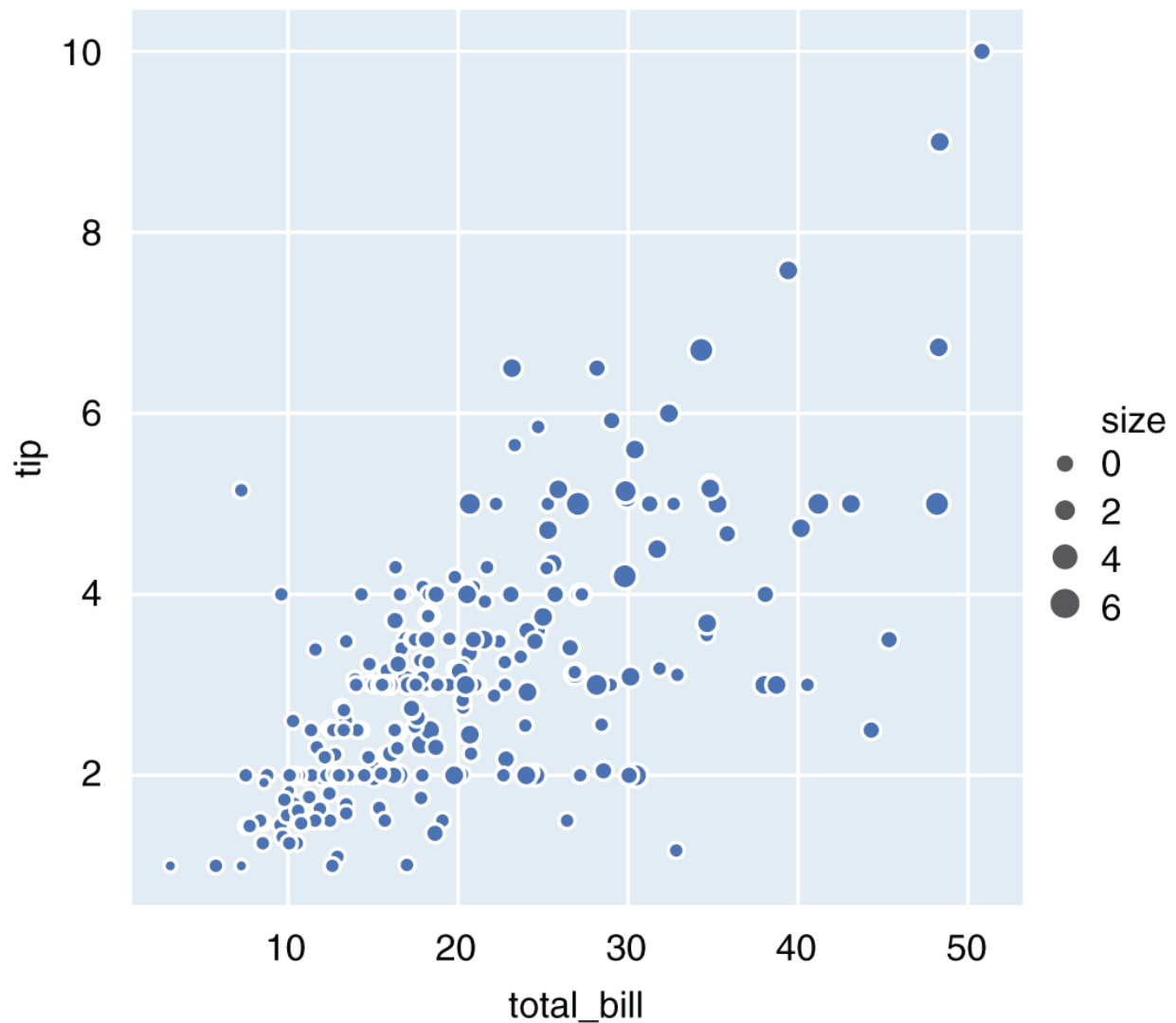
Figure 17.34 Scatter plot using replot in seaborn with a third variable using the size command.

If the size used is numeric, we can pass a tuple with the min and max values which is then applied across the different numeric values to give appropriate sizes of each point, this is shown in Figure 17.35.

```
>>> sns.relplot(x="total_bill", y="tip", size="size",
    sizes=(15, 200), data=tips);
```

Next, we will look at line plots in seaborn. Initially we will setup a DataFrame using two methods in numpy. The first is arange which creates a list of ascending integer values from 0 to $n-1$, which we demonstrate below.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

We then also generate a set of random numbers using the randn method of random and them apply the cumsum method to cumulatively sum these.

```
>>> np.random.randn(10)
array([ 0.75458636,  0.97439336,  0.60599301, -0.43901759, -2.09960322,
       -0.98797121,  0.49168321, -0.67410672,  0.31230068,  0.30193896])
>>> np.random.randn(10).cumsum()
array([ 0.90373977,  1.33607406, -0.14939861, -0.02879636, -0.72357103,
       -0.17421058, -0.64378907,  0.19505236,  0.10861358, -1.86160312])
```

Note the values in the two examples will not add up as everytime we generate a set of random numbers we get a different value as they do not have memory. We could have cast the first value to a variable as follows and you can see the effect of cumsum.
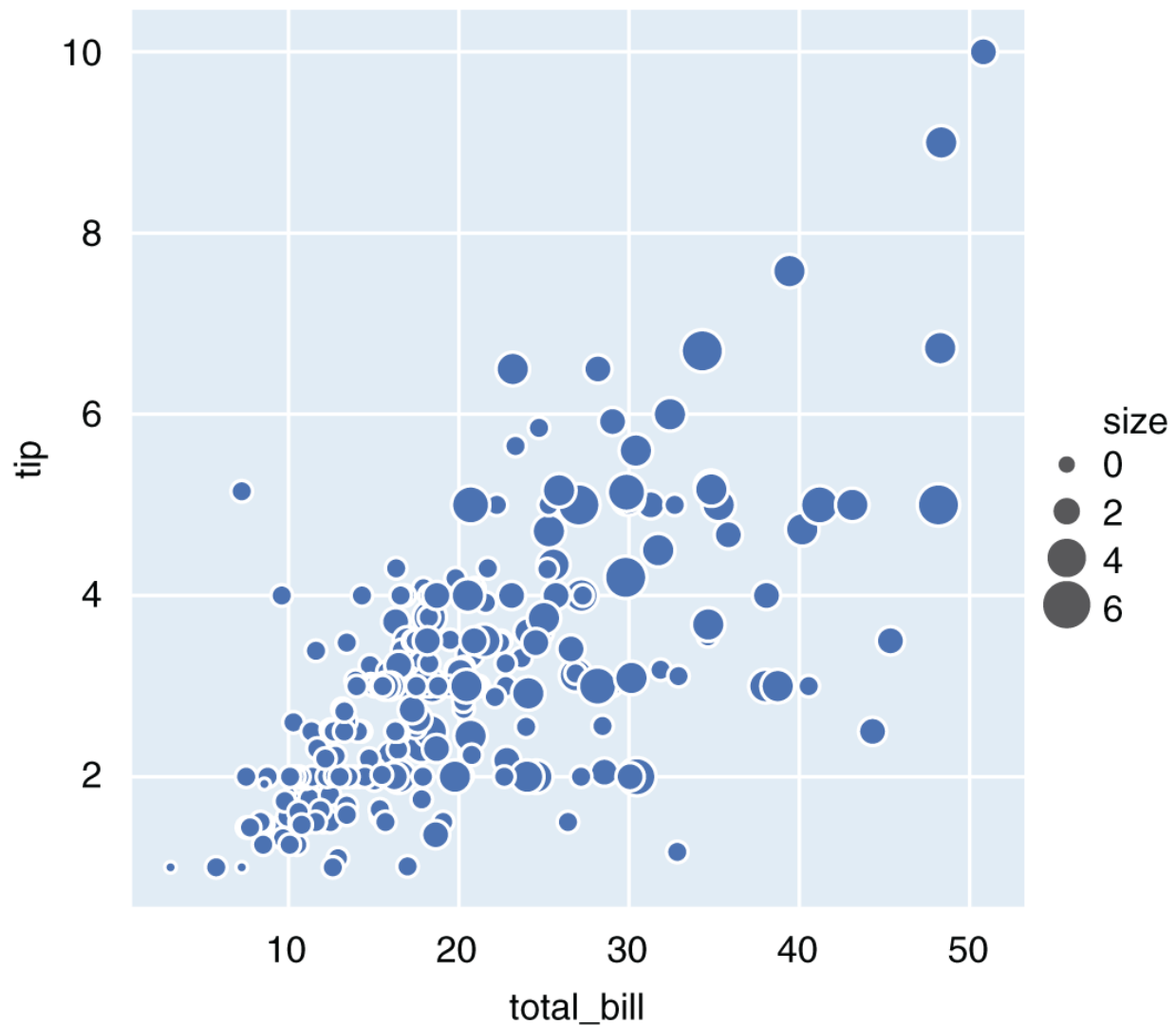
Figure 17.35 Scatter plot using replot in seaborn with different size of points.

```
>>> x = np.random.randn(10)
>>> print(x)
array([-0.66468216, -0.64194921,  0.4189947, -1.76462566,  0.60687855,
       -1.46429832, -0.01150336,  1.97159021,  0.93025079, -0.47068112])
>>> x.cumsum()
array([-0.66468216, -1.30663136, -0.88763666, -2.65226232, -2.04538377,
       -3.5096821, -3.52118546, -1.54959525, -0.61934447, -1.09002559])
```

Using the above we can create a DataFrame with some random values as our value column. To plot the data, we can use the lineplot or

with replot setting the kind to be line as is shown below with the resulting graph in .

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style="darkgrid")
>>> df = pd.DataFrame(dict(time=np.arange(500),
                           value=np.random.randn(500).cumsum()))
>>> df.head()
   time      value
0     0 -1.189914
1     1 -1.744486
2     2 -2.182634
3     3 -2.321100
4     4 -0.503165
>>> g = sns.relplot(x="time", y="value", kind="line", data=df)
>>> g.fig.autofmt_xdate()
```
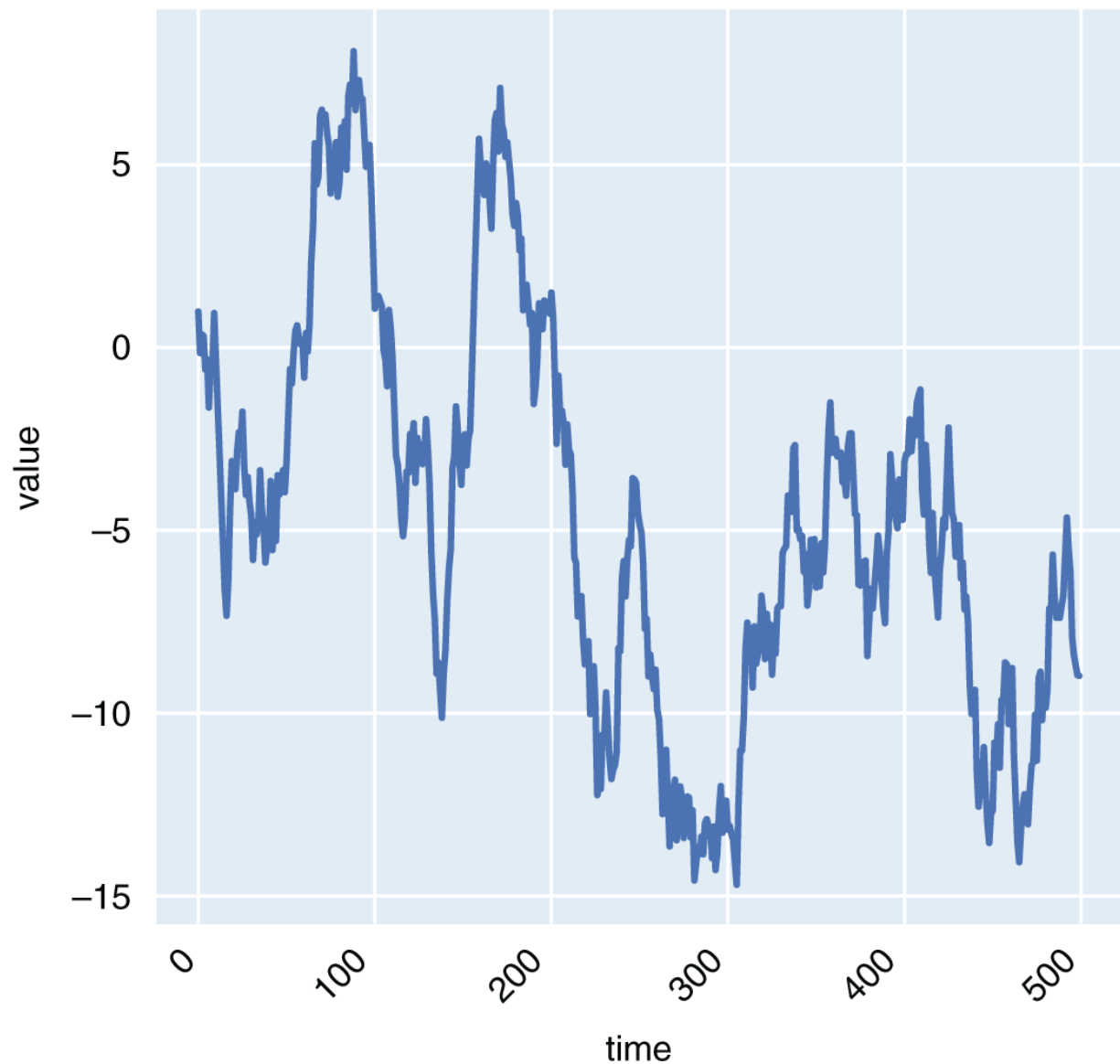
Figure 17.36 Line plot in Seaborn using replot.

Next, we look at the fmri dataset and how we can use replot to pro-
duce a line plot of timepoint by signal. Looking at the dataset, we can
see for each timepoint we have multiple measurements.

```
>>> fmri = sns.load_dataset("fmri")
>>> fmri[fmri['timepoint']==18].head()
  subject  timepoint event     region    signal
0     s13         18  stim  parietal -0.017552
2     s12         18  stim  parietal -0.081033
3     s11         18  stim  parietal -0.046134
4     s10         18  stim  parietal -0.037970
5      s9         18  stim  parietal -0.103513
>>> fmri.groupby('timepoint').count()['signal']
timepoint
0      56
1      56
2      56
3      56
4      56
5      56
6      56
7      56
8      56
9      56
10     56
11     56
12     56
13     56
14     56
15     56
16     56
17     56
18     56
```

So for each timepoint, we have 56 values; now passing these values into replot aggregates over the data with the line representing the

mean of that timepoint and a 95% confidence interval around that point, this is shown in Figure 17.37.

```
>>> sns.relplot(x="timepoint", y="signal", kind="line", data=fmri);
```

We can repress the confidence interval by passing the ci to be None as shown in Figure 17.38.

```
>>> sns.relplot(x="timepoint", y="signal", ci=None,
        kind="line", data=fmri);
```
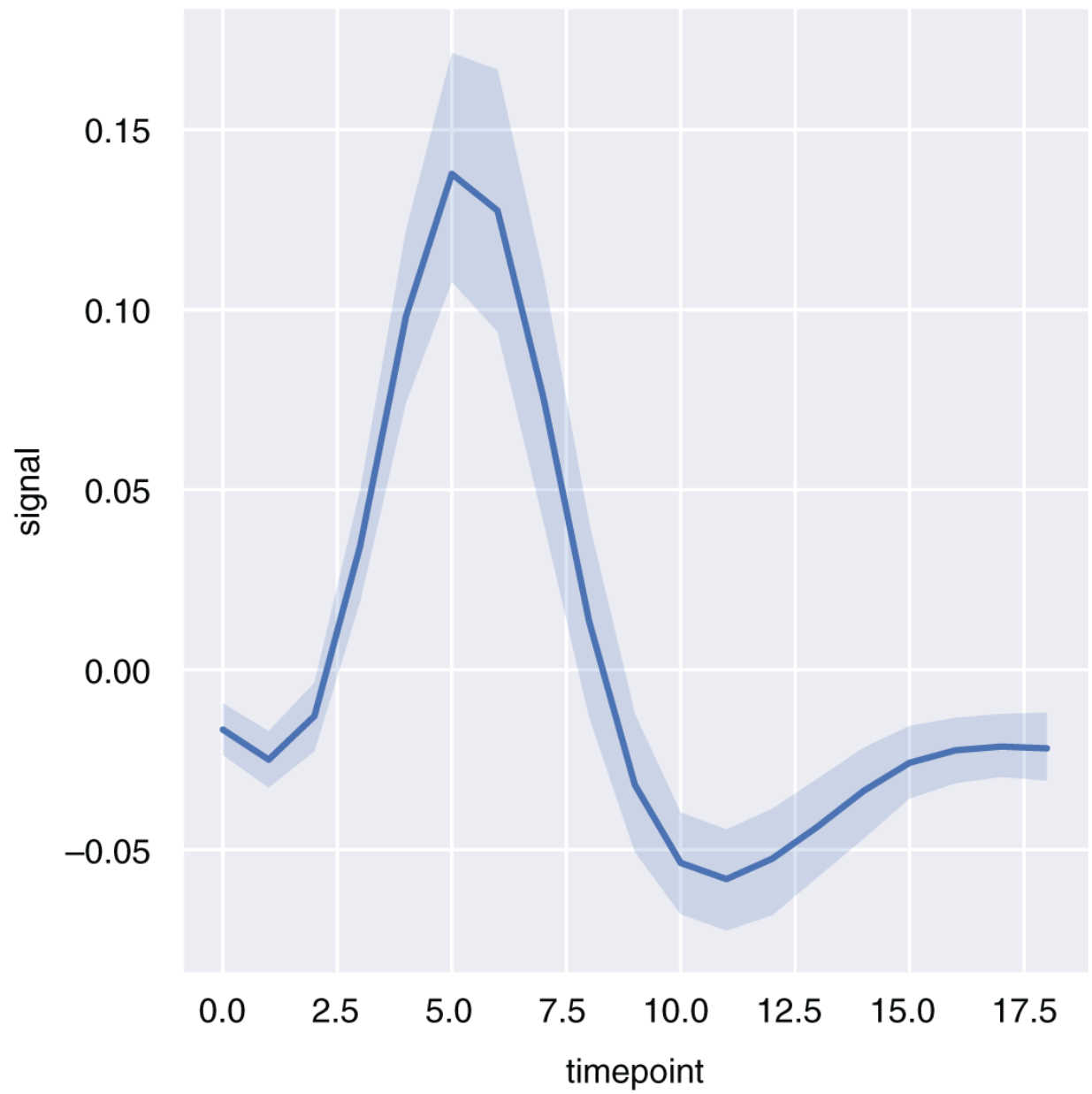
Figure 17.37 Line plot in replot with mean and confidence interval.
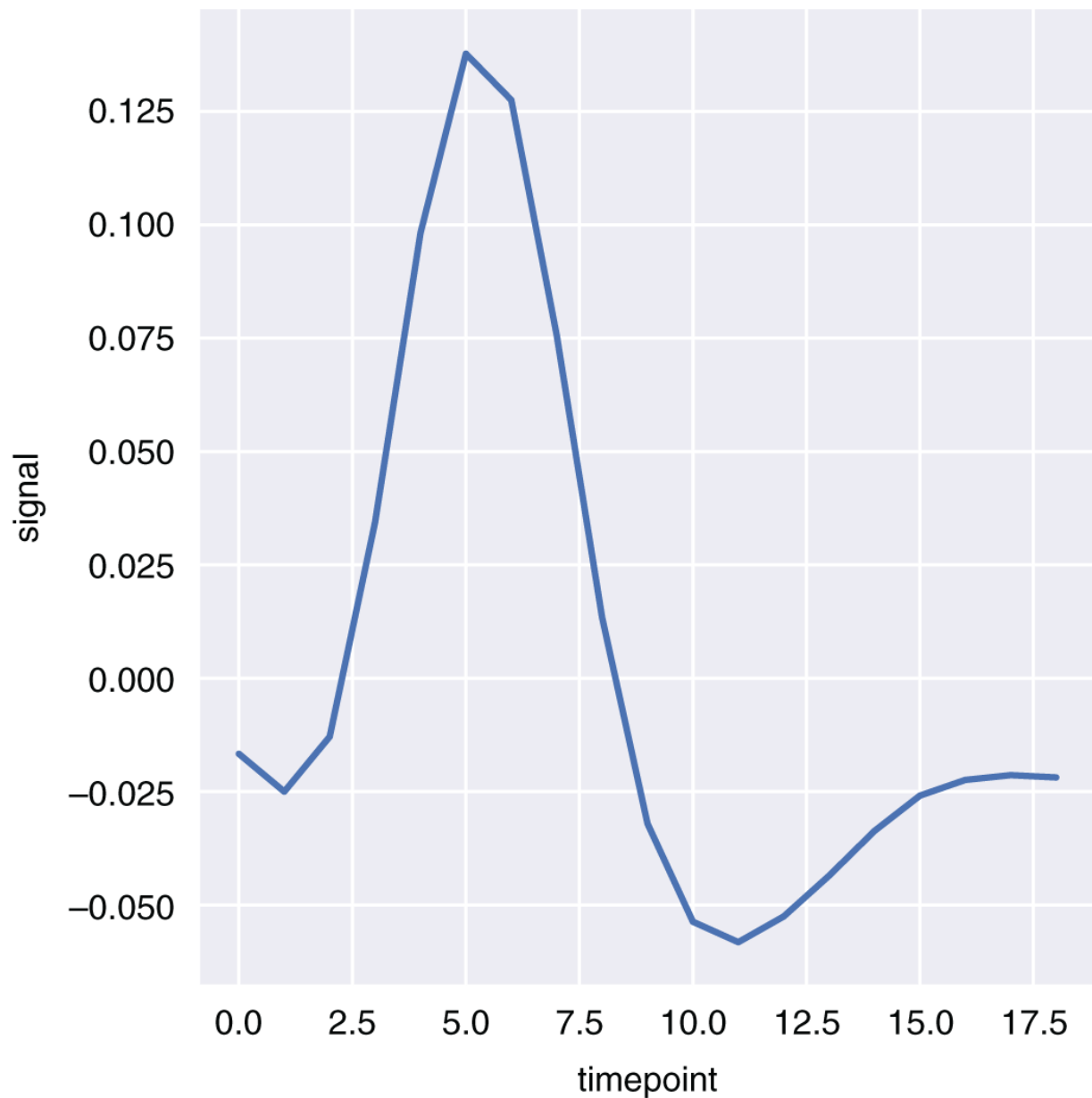
Figure 17.38 Line plot in replot with mean and no confidence interval.

We can change what the interval around the mean by passing sd to it which then calculates the standard deviation around the mean which is given in Figure 17.39.

```
>>> sns.relplot(x="timepoint", y="signal", kind="line",
      ci="sd", data=fmri);
```

Like we saw before with the scatterplot, we can use the hue to pass in a third variable to group our data by, in the below code we choose the event variable which splits our data in two as shown in Figure 17.40.

```
>>> fmri.head()
  subject  timepoint event    region    signal
0     s13         18  stim  parietal -0.017552
1      s5         14  stim  parietal -0.080883
2     s12         18  stim  parietal -0.081033
3     s11         18  stim  parietal -0.046134
4     s10         18  stim  parietal -0.037970
>>> fmri.event.unique()
array(['stim', 'cue'], dtype=object)
>>> sns.relplot(x="timepoint", y="signal", hue="event",
    kind="line", data=fmri);
```

We can expand on the previous example by setting the hue to be region and the style to be event. In doing so we have the different colours representing the regions and the line type representing the event. This allows us to represent more variables in the line plot as shown in Figure 17.41.
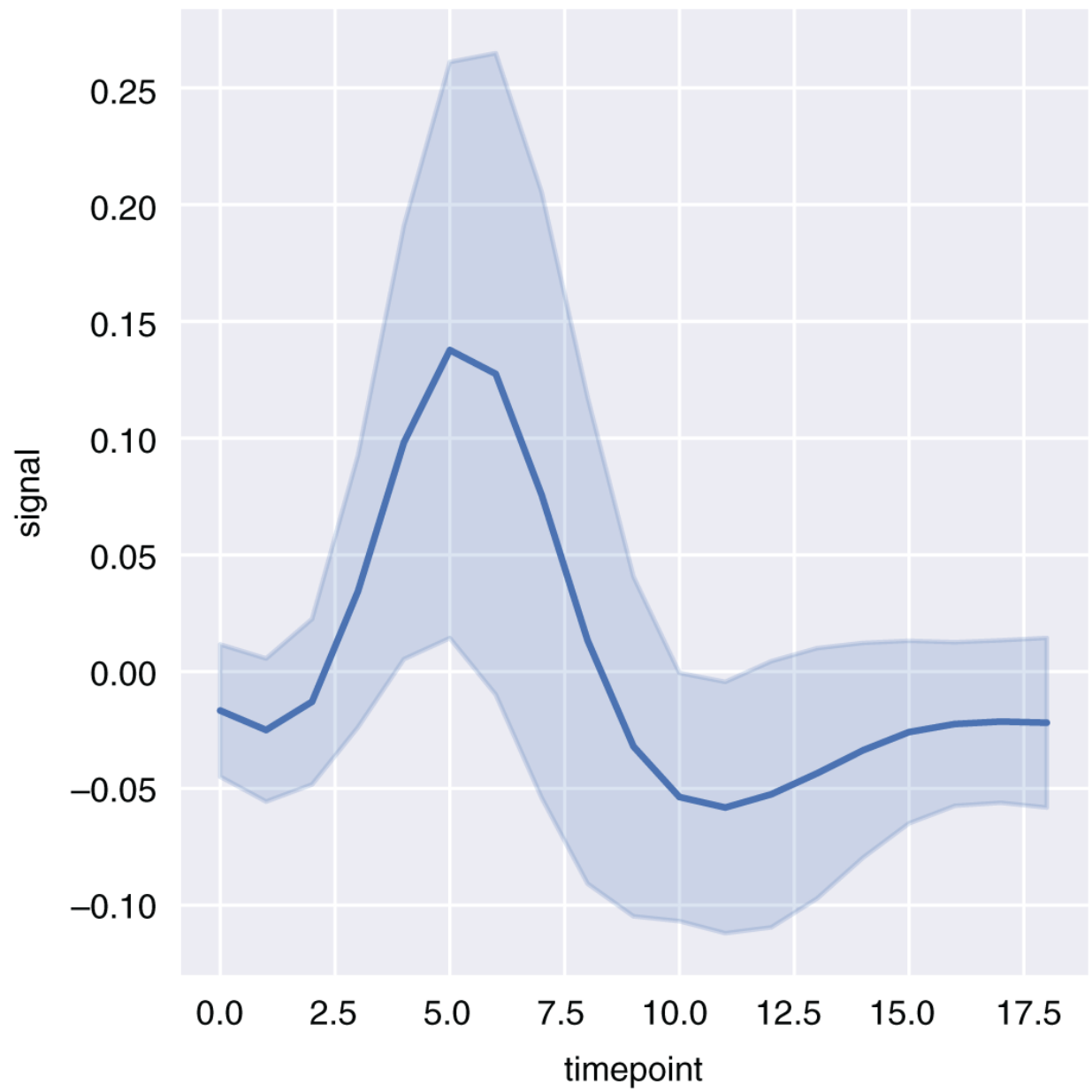
Figure 17.39 Line plot in replot with mean and standard deviation.
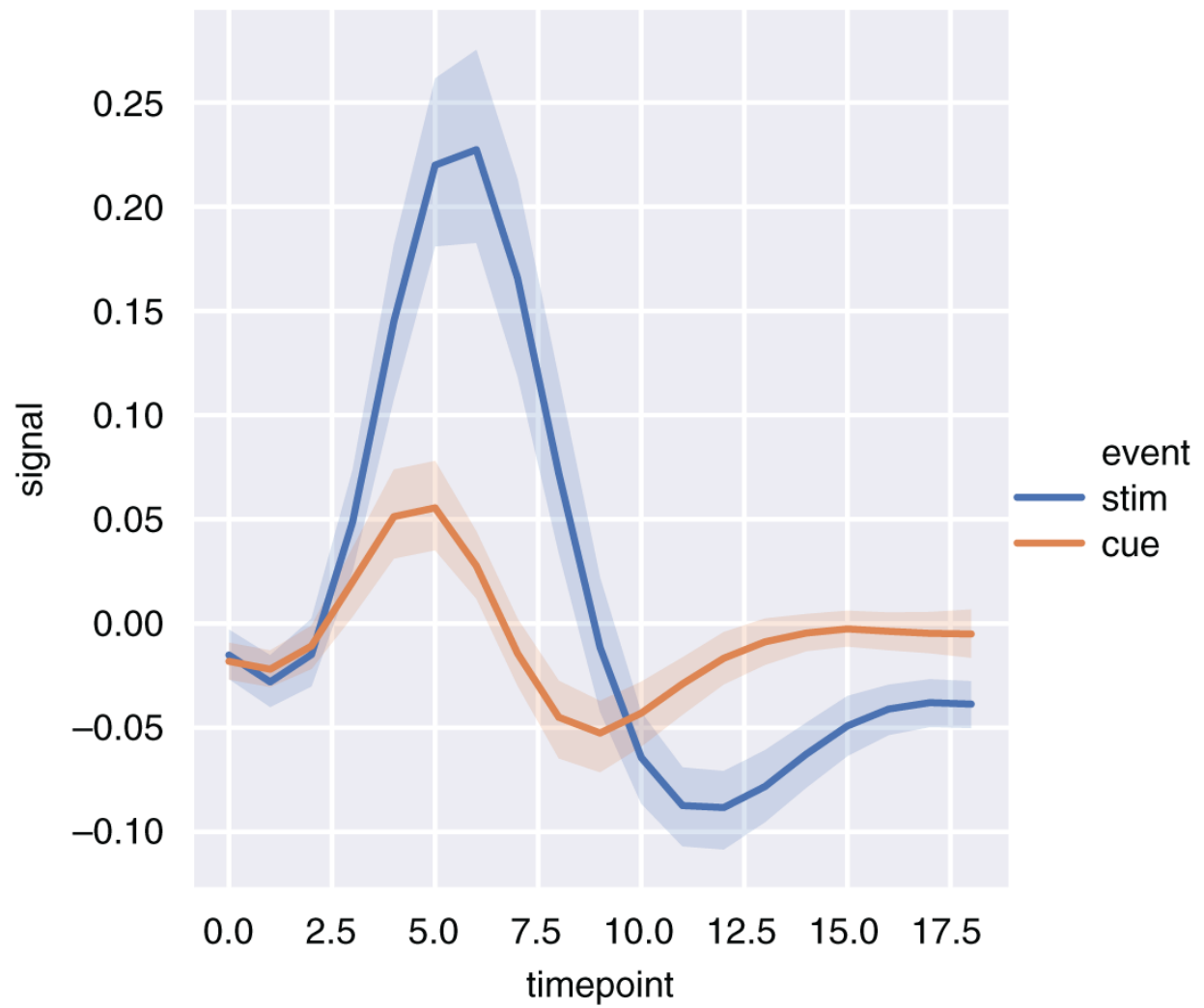
Figure 17.40 Line plot in replot with hue applied.
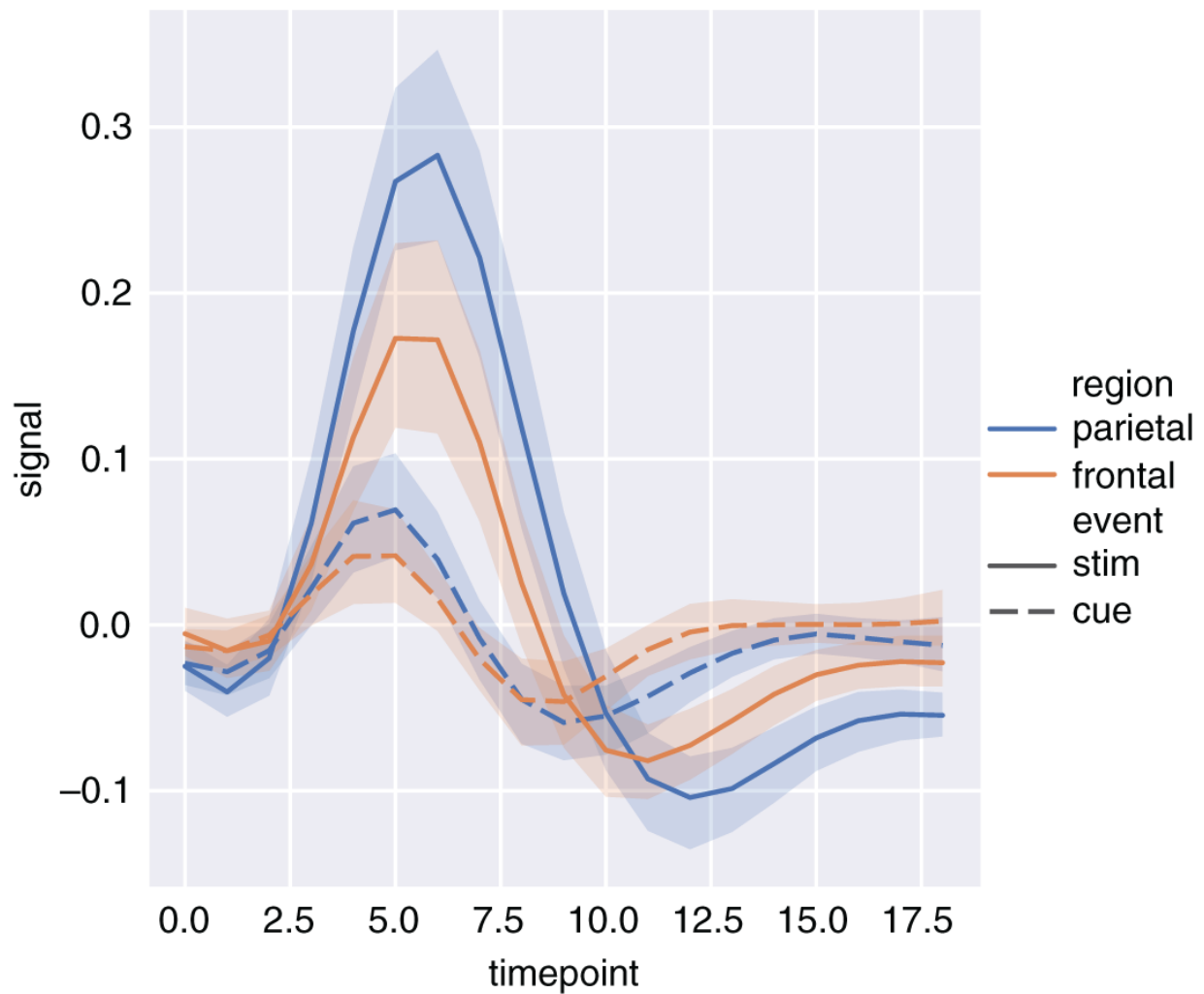
Figure 17.41 Line plot in replot with hue and style applied.

```
>>> fmri.region.unique()
array(['parietal', 'frontal'], dtype=object)
>>> sns.relplot(x="timepoint", y="signal", hue="region", style="event",
        kind="line", data=fmri);
```

Let us change the example and use a different dataset. Here we load in the dots dataset.

```
>>> dots = sns.load_dataset("dots").query("align == 'dots'")
>>> dots.head()
  align choice  time  coherence  firing_rate
0  dots     T1   -80        0.0    33.189967
1  dots     T1   -80        3.2    31.691726
2  dots     T1   -80        6.4    34.279840
3  dots     T1   -80       12.8    32.631874
4  dots     T1   -80       25.6    35.060487
>>> dots.tail()
     align choice  time  coherence  firing_rate
389   dots     T2   680        3.2    37.806267
390   dots     T2   700        0.0    43.464959
391   dots     T2   700        3.2    38.994559
392   dots     T2   720        0.0    41.987121
393   dots     T2   720        3.2    41.716057
>>> dots.coherence.unique()
array([ 0.,   3.2,   6.4, 12.8, 25.6, 51.2])
>>> dots.choice.unique()
array(['T1', 'T2'], dtype=object)
>>>sns.relplot(x="time", y="firing_rate",
            hue="coherence", style="choice",
            kind="line", data=dots);
```
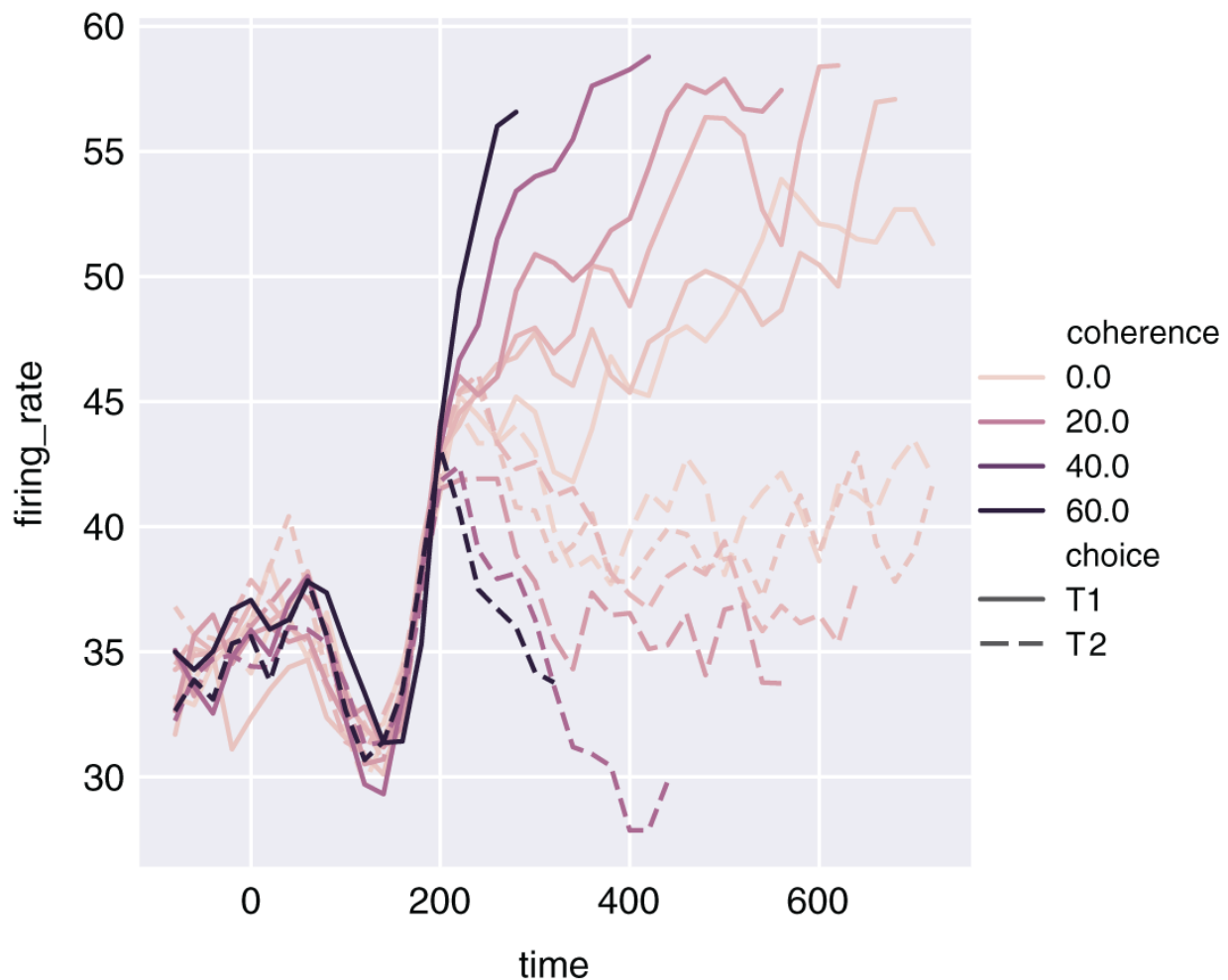
Figure 17.42 Line plot in replot with hue and style applied on the dots dataset.

In the plot shown in Figure 17.42, we use time and firing rate as our x and y variables, we then use the coherence and choice variables to provide the hue and the styling. Using both hue and style allows us to combine colour and style at each x and y combination. So in essence we can display four variables on one plot. It should be noted that the makeup of this dataset is particularly suited to doing this as we have six unique values for coherence and two for choice resulting in 12 lines. When considering using this type of plot, it is important to make

sure that your dataset is suitable and that the plot is a better representation of the data, as opposed to something that visually offers little.

So far we have considered single plots but what if we want to compare the relationship between multiple variables on the same plot. Let us look at the tips dataset and the relationship between total bill and tip

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style="darkgrid")
>>> tips = sns.load_dataset("tips")
>>> tips.head()
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
>>> tips.time.unique()
[Dinner, Lunch]
Categories (2, object): [Dinner, Lunch]
>>> tips.smoker.unique()
[No, Yes]
Categories (2, object): [No, Yes]
>>> sns.relplot(x="total_bill", y="tip", hue="smoker",
            col="time", data=tips);
```

In this example, we have looked at the relationship between total bill and tip which has been covered before. Now looking at other variables within the dataset that we could use time and smoker to drill further into the relationship between total bill and tips. Setting the hue to be smoker breaks our data into those who are and who are not a smoker; however, we can expand on this further by using the col argument. In this case, we set col to time, which like with smoker contains only two categories, and results in our initial plot by hue being replicated on the variable passed through col. This is useful in showing us the relationship side by side and is shown in Figure 17.43.

We can expand upon the previous example by producing a multi plot using rows and columns. Here we load the fmri dataset from seaborn.

```
>>> fmri = sns.load_dataset("fmri")
>>> fmri.head()
  subject  timepoint event      region    signal
0     s13         18  stim    parietal -0.017552
1      s5         14  stim    parietal -0.080883
2     s12         18  stim    parietal -0.081033
3     s11         18  stim    parietal -0.046134
4     s10         18  stim    parietal -0.037970
>>> fmri.dtypes
subject        object
timepoint       int64
event          object
region         object
signal        float64
dtype: object
>>> fmri.event.unique()
array(['stim', 'cue'], dtype=object)
>>> fmri.region.unique()
array(['parietal', 'frontal'], dtype=object)
>>> fmri.subject.unique()
array(['s13', 's5', 's12', 's11', 's10', 's9', 's8', 's7',
       's6', 's4', 's3', 's2', 's1', 's0'], dtype=object)
>>> sns.relplot(x="timepoint", y="signal", hue="subject",
            col="region", row="event", height=3,
            kind="line", estimator=None, data=fmri);
```
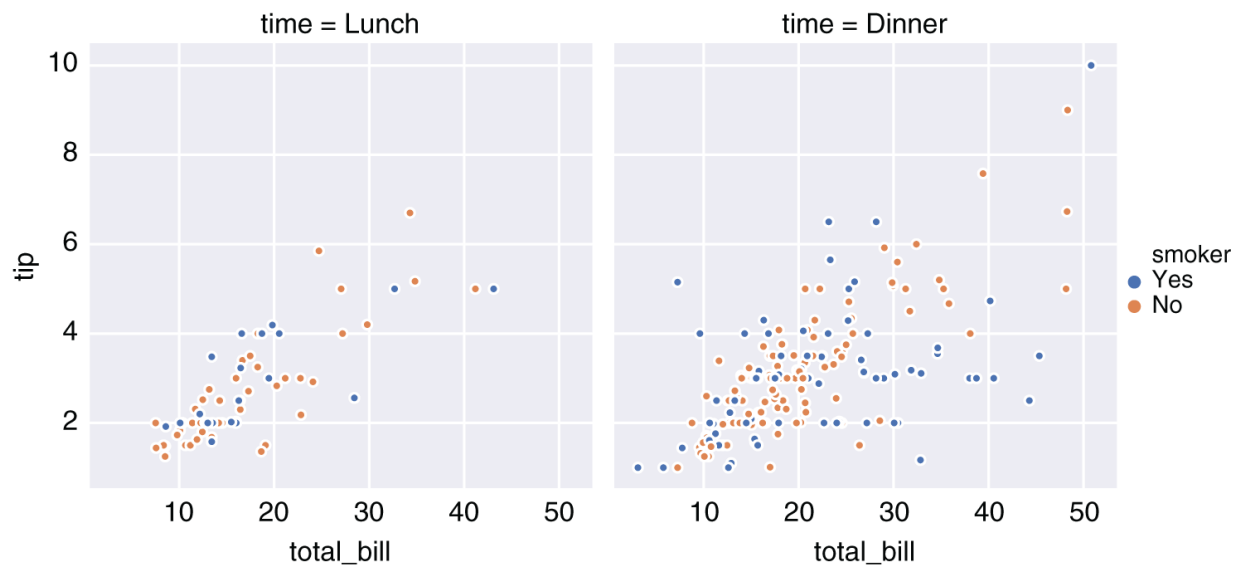
Figure 17.43 Multi scatter plot on tips data.

As usual we examine the dataset using head; however, in this example we use the dtypes method to display the types of each column. What this shows is that timepoint and signal are of types int64 and float64 respectively which make them ideal candidates as our x and y variable. Looking at the other variables, we see that both event and region have only two distinct values and the subject has 14 different values. So in putting together our plot subject is best suited to be the hue, and we can use event and region to be set as the row and col variables. In the code, we set the col as region and row as event. The result of this is that the headers for our 2 by 2 plot contains the distinct combinations of the variables as shown in Figure 17.44.

Now this plot is fine however you could argue that having 14 variables in the hue makes it hard to distinguish what's going on. We can quite

easily reduce this down by looking at selecting a subset of data as shown in Figure 17.45.

```
>>> fmri['subject'].isin(['s0','s1','s2']).head()
0    False
1    False
2    False
3    False
4    False
Name: subject, dtype: bool
>>> fmri_red = fmri[fmri['subject'].isin(['s0','s1','s2'])]
>>> sns.relplot(x="timepoint", y="signal", hue="subject",
                col="region", row="event", height=3,
                kind="line", estimator=None, data=fmri_red);
```
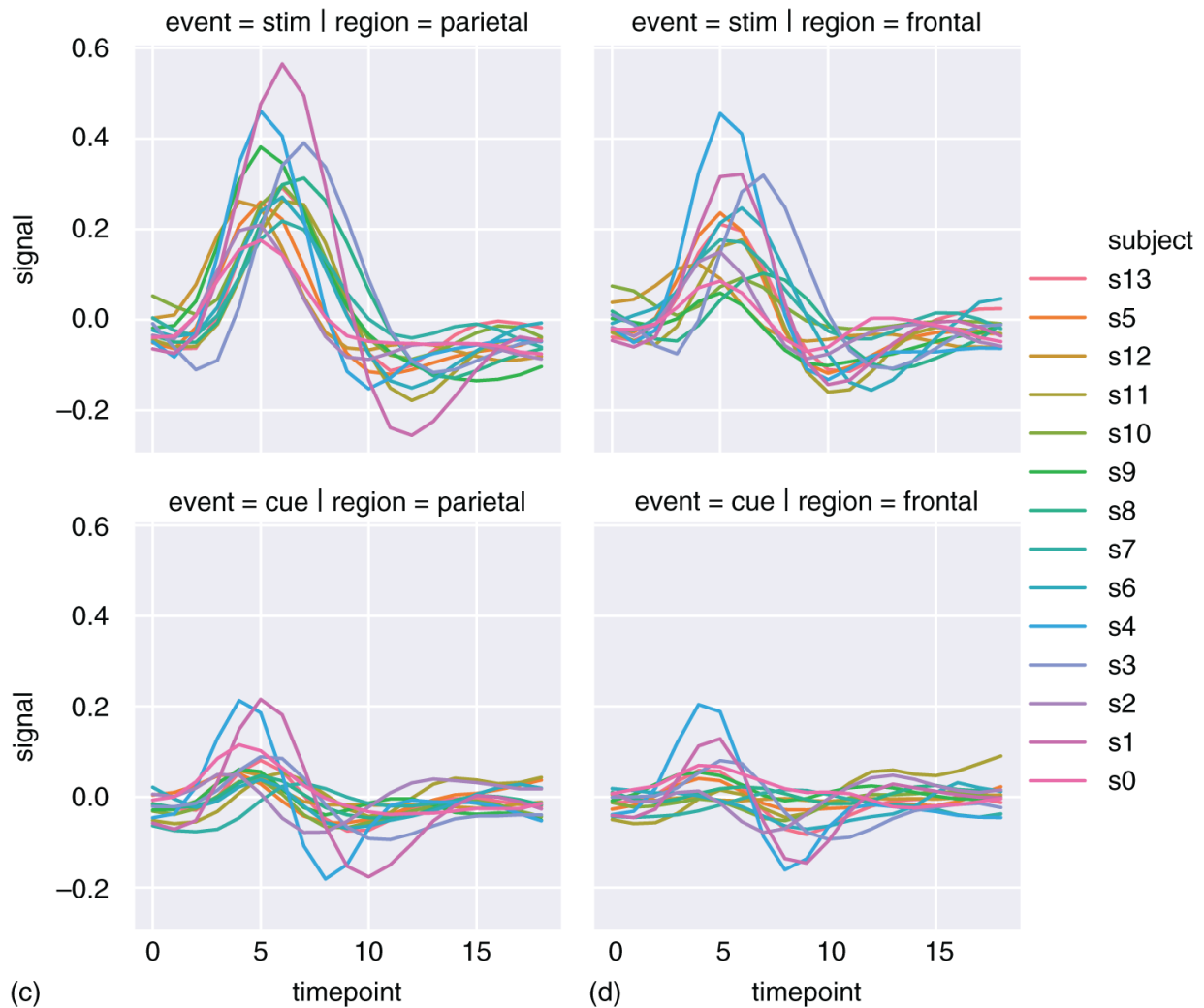
Figure 17.44 Multi line plot with rows and columns using fmri dataset.

If we only want to look at one variable passed through col, we can set a col_wrap which will have a max number of plots side by side. In this case, setting the value to 5 wraps the plots to 5 per line giving three rows to fit our 14 different subjects. This gives us the effect of setting rows and columns using just a single variable as shown in Figure 17.46.

```
>>> sns.relplot(x="timepoint", y="signal", hue="event", style="event",
        col="subject", col_wrap=5,
        height=3, aspect=.75, linewidth=2.5,
        kind="line", data=fmri.query("region == 'frontal'"));
```

We next consider plotting categorical data and start by looking at using the catplot method with the tips dataset.

```
>>> tips = sns.load_dataset("tips")
>>> tips.head()
   total_bill    tip      sex smoker  day    time  size
0       16.99  1.01   Female     No  Sun  Dinner     2
1       10.34  1.66     Male     No  Sun  Dinner     3
2       21.01  3.50     Male     No  Sun  Dinner     3
3       23.68  3.31     Male     No  Sun  Dinner     2
4       24.59  3.61   Female     No  Sun  Dinner     4
>>> tips.dtypes
total_bill        float64
tip               float64
sex              category
smoker           category
day              category
time             category
size                int64
dtype: object
>>> tips['day'].unique()
[Sun, Sat, Thur, Fri]
Categories (4, object): [Sun, Sat, Thur, Fri]
>>> sns.catplot(x="day", y="total_bill", data=tips);
```
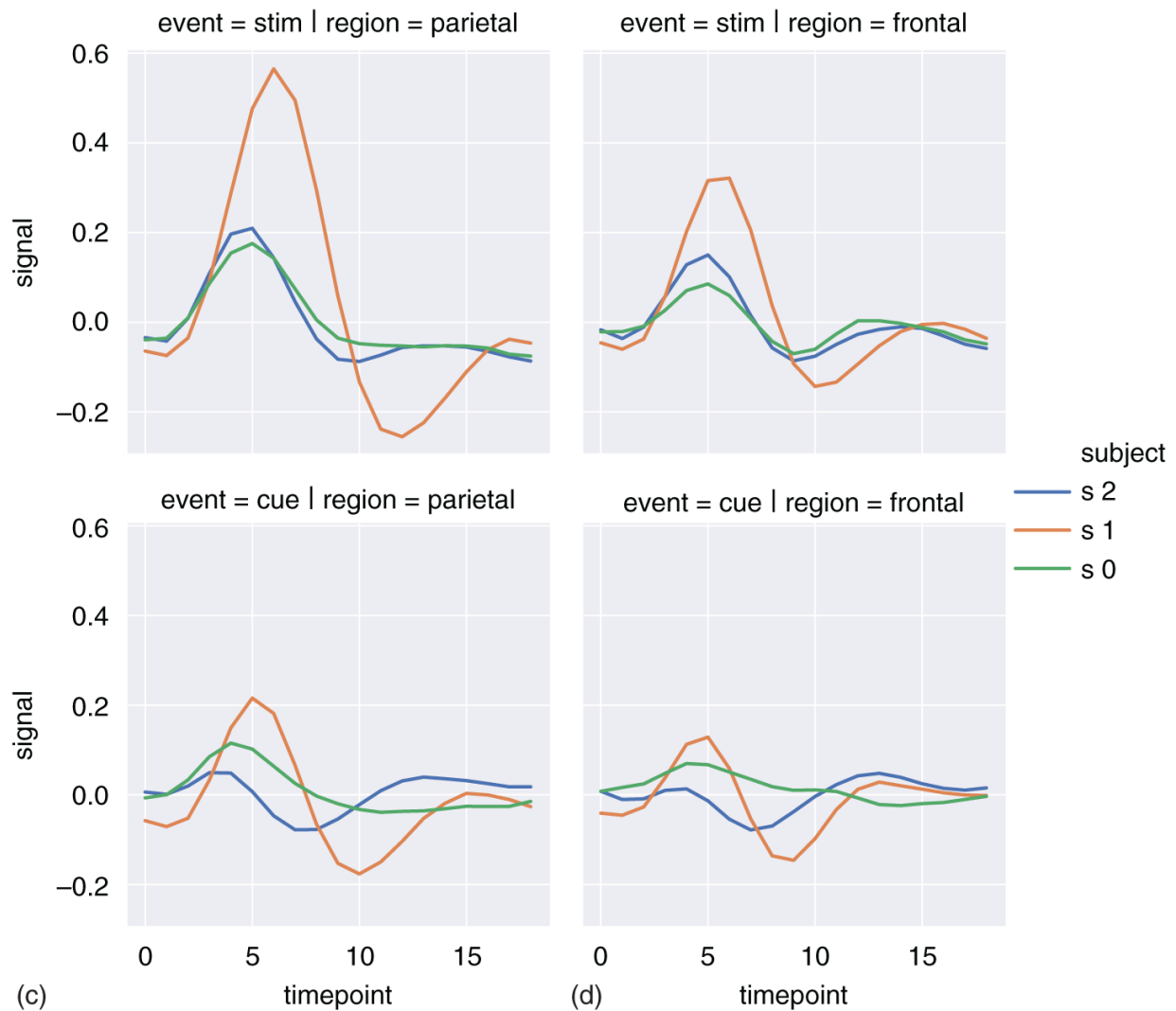
Figure 17.45 Multi line plot with rows and columns using reduced fmri dataset.

The code shown loads the seaborn dataset tips and we look at its content using the head and dtypes methods. What we see is that there are some options for categorical data and we look at the individual values of the day column which unsurprisingly contain some of the week of the year names (albeit abbreviated). It is also worth noting that the type of this column is category. We can then plot the values of the total bill split by the day of the week which is in the form of

points grouped by the day and also coloured to distinguish them with the resulting plot shown in .
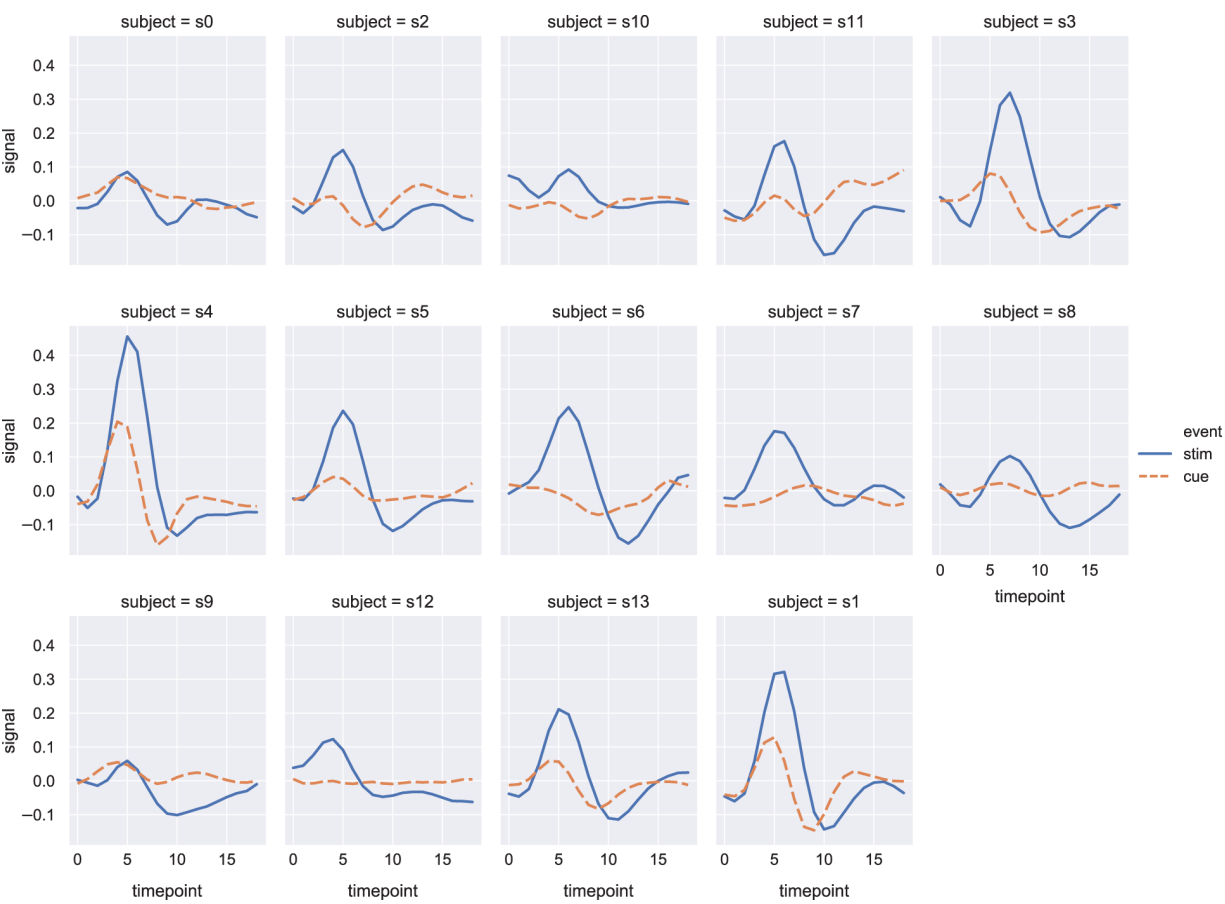
Figure 17.46 Multiline plot using col wrap.
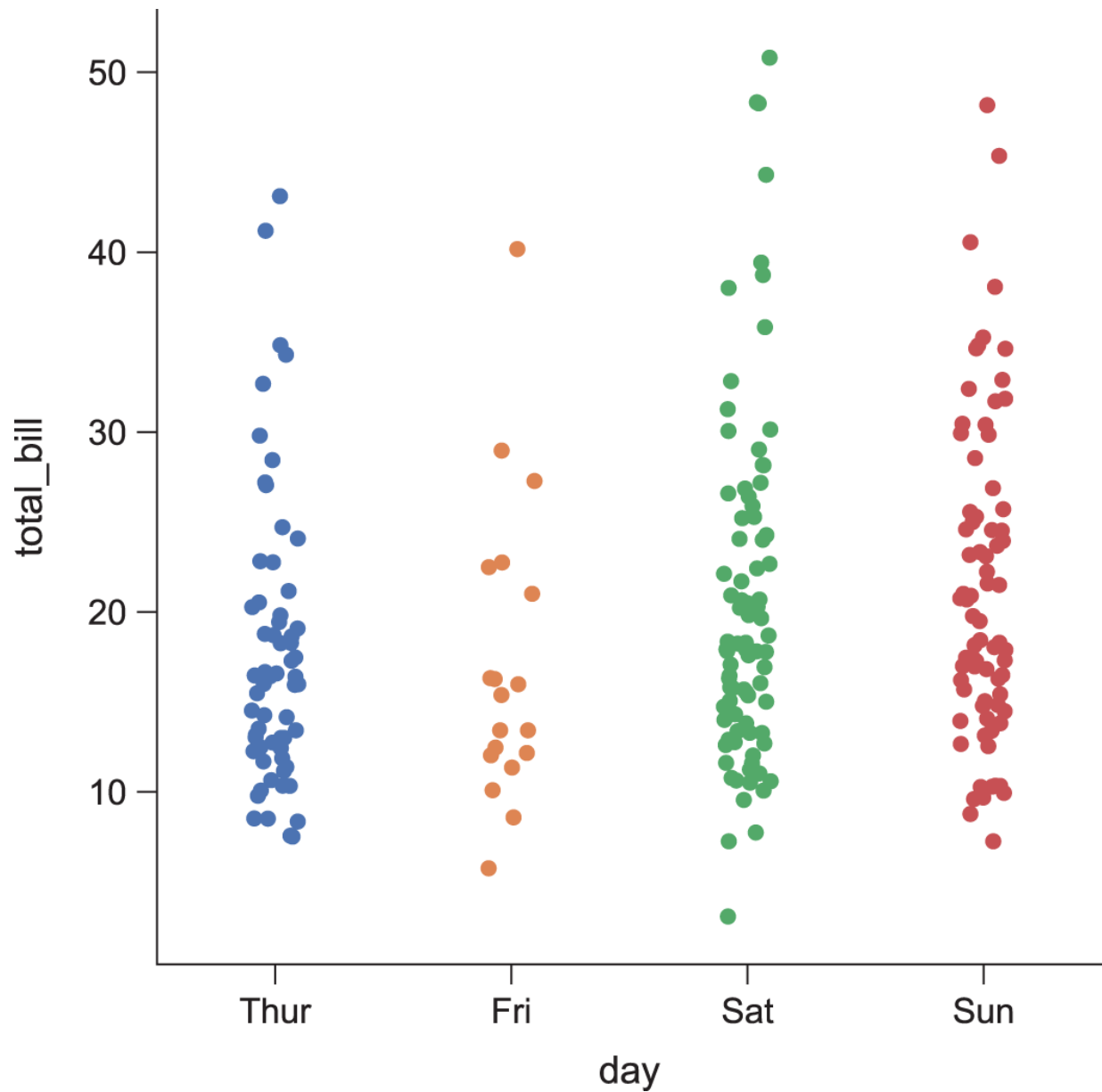
Figure 17.47 Catplot of day against total bill from the tips dataset.

Now if we consider the data, we can see some shared values

```
>>> tips[tips['day']=='Thur'].sort_values(by='total_bill').head(20)
     total_bill   tip     sex smoker    day    time  size
149        7.51  2.00    Male     No   Thur   Lunch     2
195        7.56  1.44    Male     No   Thur   Lunch     2
145        8.35  1.50  Female     No   Thur   Lunch     2
135        8.51  1.25  Female     No   Thur   Lunch     2
126        8.52  1.48    Male     No   Thur   Lunch     2
148        9.78  1.73    Male     No   Thur   Lunch     2
82        10.07  1.83  Female     No   Thur   Lunch     1
136       10.33  2.00  Female     No   Thur   Lunch     2
196       10.34  2.00    Male    Yes   Thur   Lunch     2
117       10.65  1.50  Female     No   Thur   Lunch     2
132       11.17  1.50  Female     No   Thur   Lunch     2
128       11.38  2.00  Female     No   Thur   Lunch     2
120       11.69  2.31    Male     No   Thur   Lunch     2
147       11.87  1.63  Female     No   Thur   Lunch     2
133       12.26  2.00  Female     No   Thur   Lunch     2
118       12.43  1.80  Female     No   Thur   Lunch     2
124       12.48  2.52  Female     No   Thur   Lunch     2
201       12.74  2.01  Female    Yes   Thur   Lunch     2
202       13.00  2.00  Female    Yes   Thur   Lunch     2
198       13.00  2.00  Female    Yes   Thur   Lunch     2
```

So we can see that we have a shared value on Thursday for the total bill value of 13.00. In the previous plot, our scatter does not take account of that so we need a way to deal with this. Luckily, we have a setting that can be applied to change this. By passing the kind variable and setting it to swarm applies an algorithm to prevent the overlapping of the variables and this gives us a better representation of the distribution as shown in Figure 17.48. It should be noted that the default value of kind is strip which gives the plot shown in the previous example.

```
>>> sns.catplot(x="day", y="total_bill", kind="swarm", data=tips);
```

Previously we had shown how we could add a hue to a plot to group it by that value and the same is true when we produce a catplot. Looking through the variable list, we can see that sex is another categorical variable that would work well passed to the hue. The difference that we see between this and the last plot is that the colour is now driven by the hue which is to be expected as our other categorical variable is passed through the x variable. This is shown in Figure 17.49.

```
>>> tips.dtypes
total_bill      float64
tip             float64
sex             category
smoker          category
day             category
time            category
size            int64
dtype: object
>>> tips['sex'].unique()
[Female, Male]
Categories (2, object): [Female, Male]
>>> sns.catplot(x="day", y="total_bill", hue="sex",
    kind="swarm", data=tips);
```
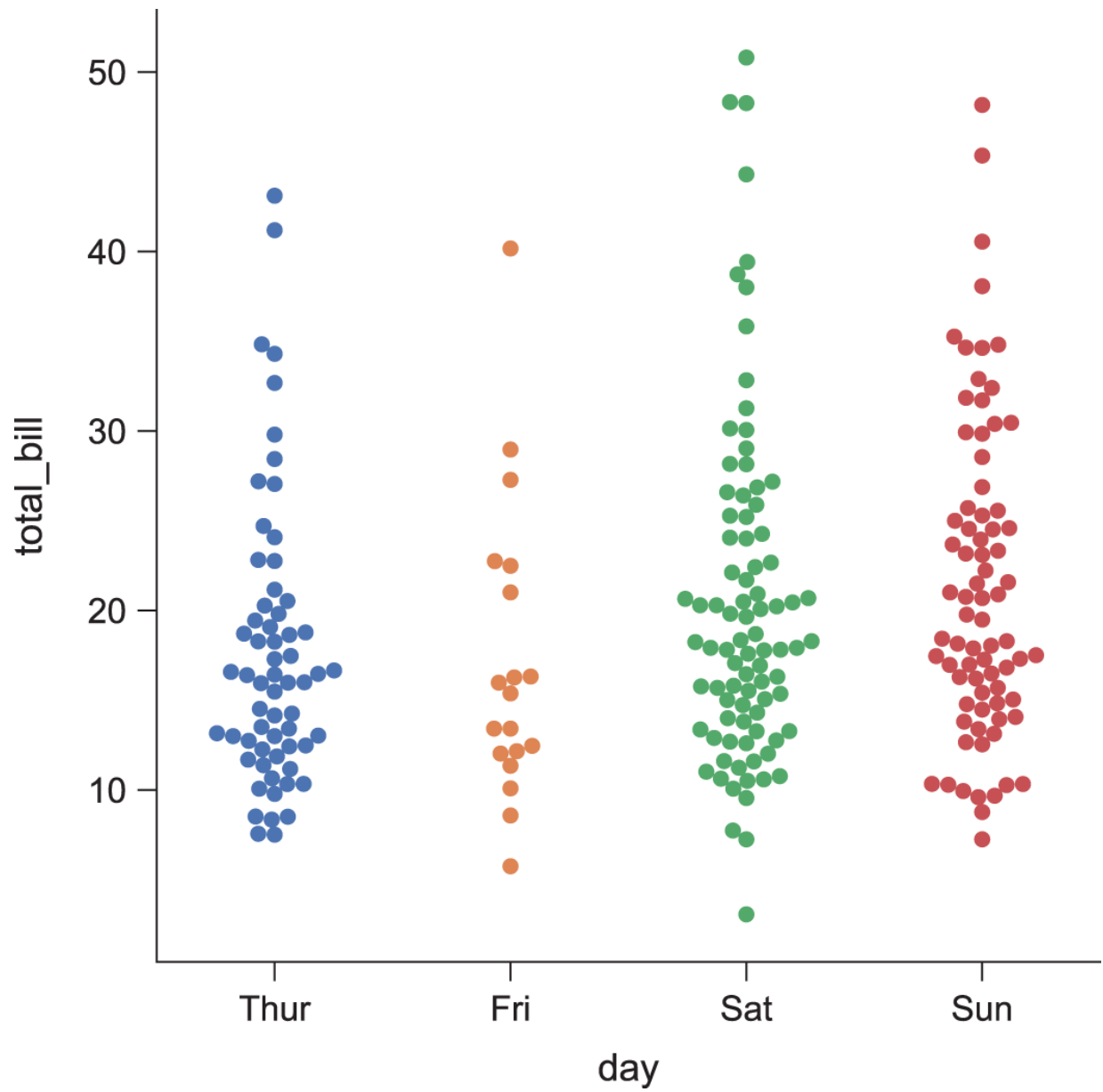
Figure 17.48 Catplot of day against total bill from the tips dataset with kind set to swarm.
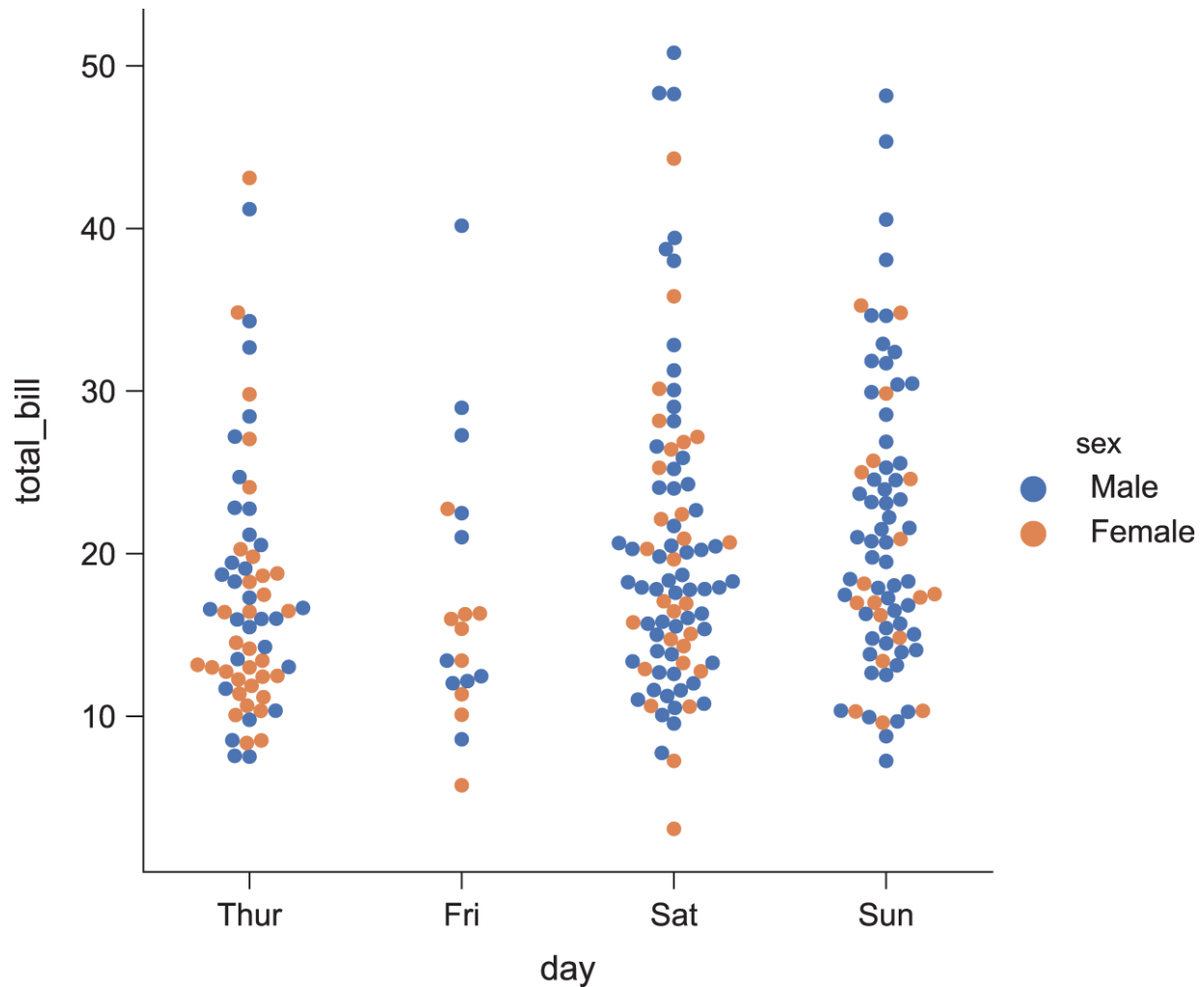
Figure 17.49 Catplot of day against total bill from the tips dataset with kind set to swarm with hue set to sex.

So far we have dealt with categorical variables in the form of text data like sex or day, but what if the category could be numerical. In this next example, we will set the x value to be size which is an integer and what we see is that the value is treated as a category in a similar way to what we have seen in the previous example. In the code, you will also see that we have introduced a new DataFrame method query which works by passing a query to apply on the DataFrame in

the form of a string. So here we have passed in the query size != 3 so what we are saying is that we want to get the data where size is not equal to 3. This could also have been written using Boolean series where we would look for tips[tips['size']!=3]. What should be noted is that catplot will order the x value appropriately based on the numerical values as shown in Figure 17.50.

```
>>> sns.catplot(x="size", y="total_bill", kind="swarm",
            data=tips.query("size != 3"));
```
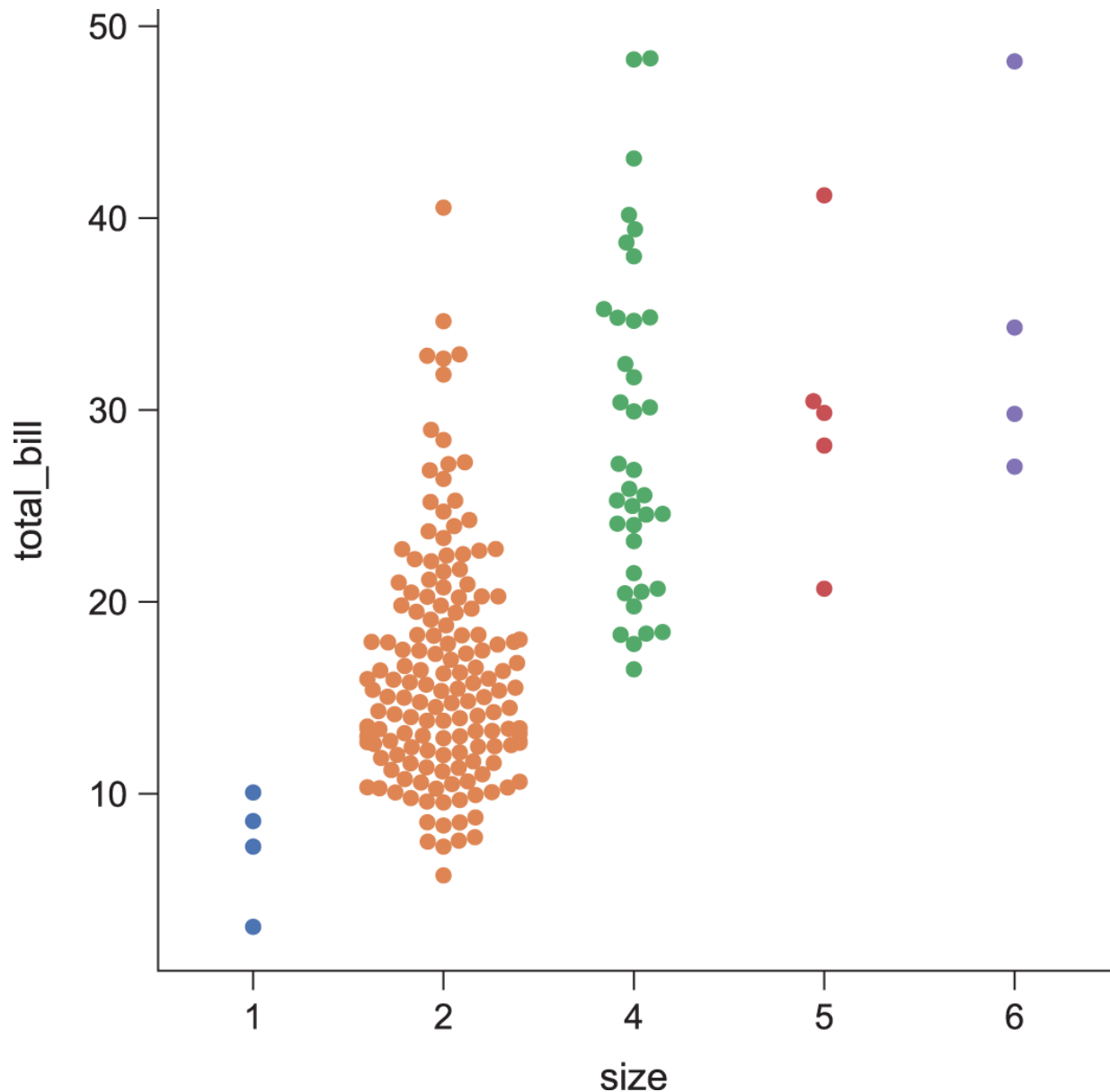
Figure 17.50 Catplot of size against total bill.

The question of ordering a numerical value is relatively simple as you would expect seaborn to order it using the ascending numerical value, but for a categorical variable, it is not quite as straight forward. If we take the example of plotting smoker against tip how do we control the ordering of the smoker values which are yes and no. To do this

we can use the order argument and set it to be how we want the responses to be shown as demonstrated in Figure 17.51.

```
>>> tips.smoker.unique()
[No, Yes]
Categories (2, object): [No, Yes]
>>> sns.catplot(x="smoker", y="tip", order=["No", "Yes"],
    data=tips);
```

We can change the example by using the x axis as total bill and setting the y axis to be the categorical variable day to invert the plot with the swarm being horizontal as opposed to vertical. In this example, we also apply the hue and swarm which shows we can achieve the same result horizontally or vertically as shown in Figure 17.52.

```
>>> sns.catplot(x="total_bill", y="day", hue="time",
    kind="swarm", data=tips);
```
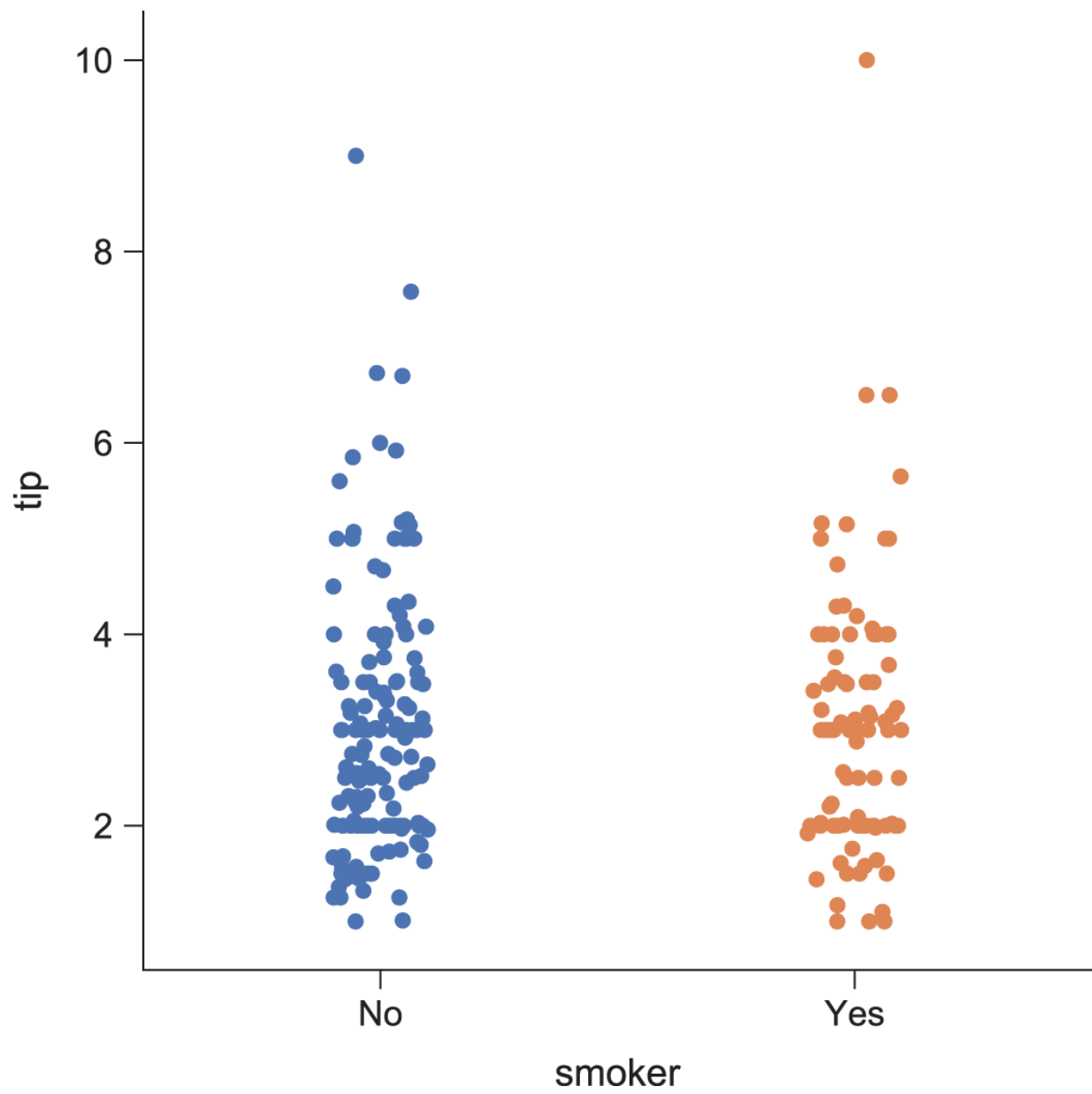
Figure 17.51 Catplot of smoker against tip using order argument.
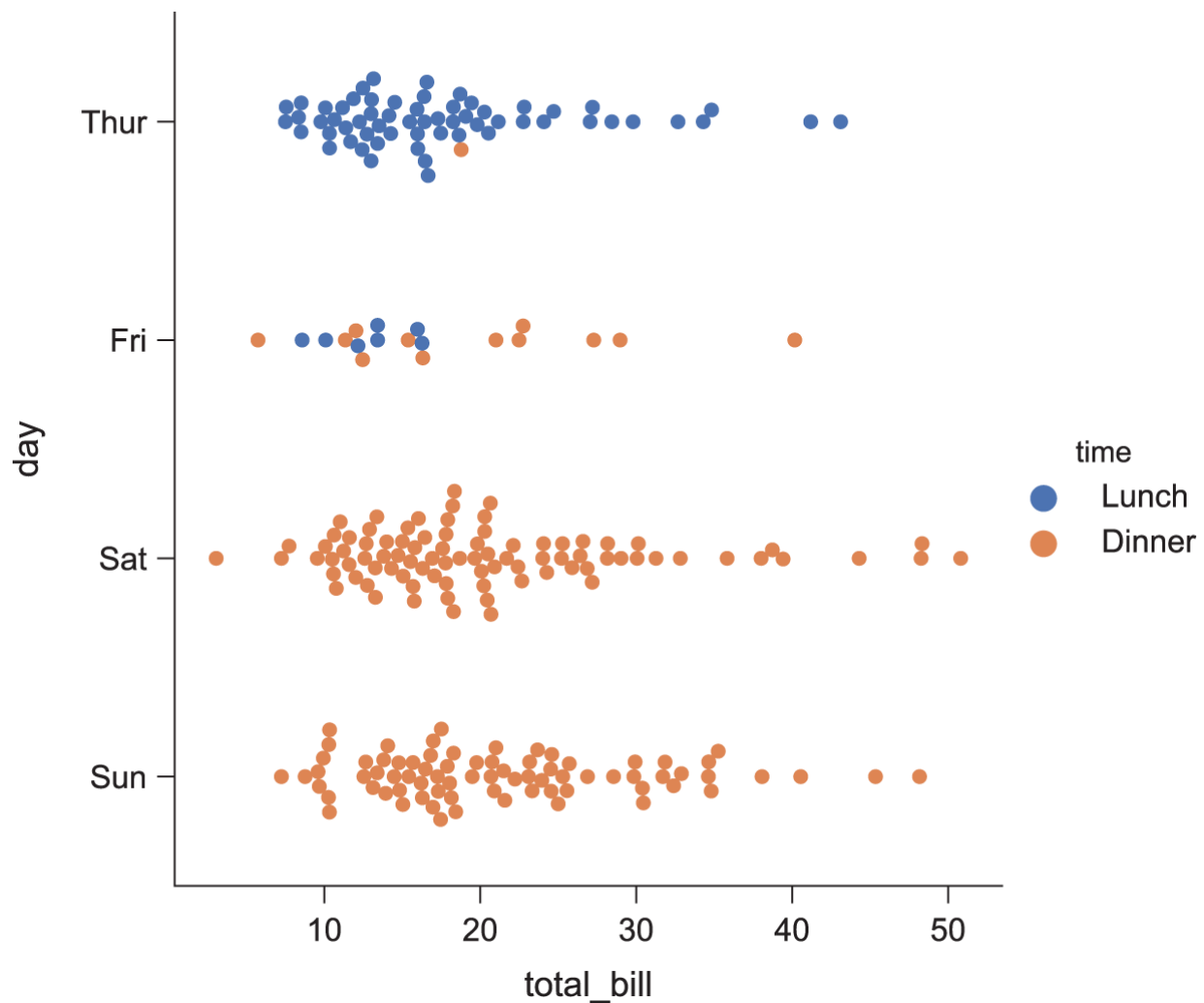
Catplot of total bill against day with swarm and hue of time.

The previous examples have all looked at scatterplots but we can produce different plots by changing what we pass as the kind argument. The first such example that will be shown is how we produce a box plot of our data. To do this, we specify the kind to be box and that will move us from the default of a scatter plot to a box plot representation of the data. In the example below, we set the y axis to be the total bill and the x axis to be day.

```
>>> sns.catplot(x="day", y="total_bill", kind="box", data=tips);
```

The result of this is a standard box plot but what we see is that data outside of the whiskers are shown as data points on the plot so every point of data is represented on it as shown in Figure 17.53.

As we have shown earlier with the scatterplot example, we can add a hue to our data which then gives us multiple box plots per category using the colour to distinguish the levels of the hue, which we demonstrate in Figure 17.54.

```
>>> sns.catplot(x="day", y="total_bill", hue="smoker",
    kind="box", data=tips);
```

The next type of plot we consider is a boxen plot, this is produced where the kind argument is set to boxen. The example below uses the diamond dataset and plots the variable colour against price. This plot is similar to the box plot except for the fact that the data is grouped beyond the quartiles that a box plot shows as shown in Figure 17.55. In doing this we can get a better picture of the distribution of the data given the larger number of groups.
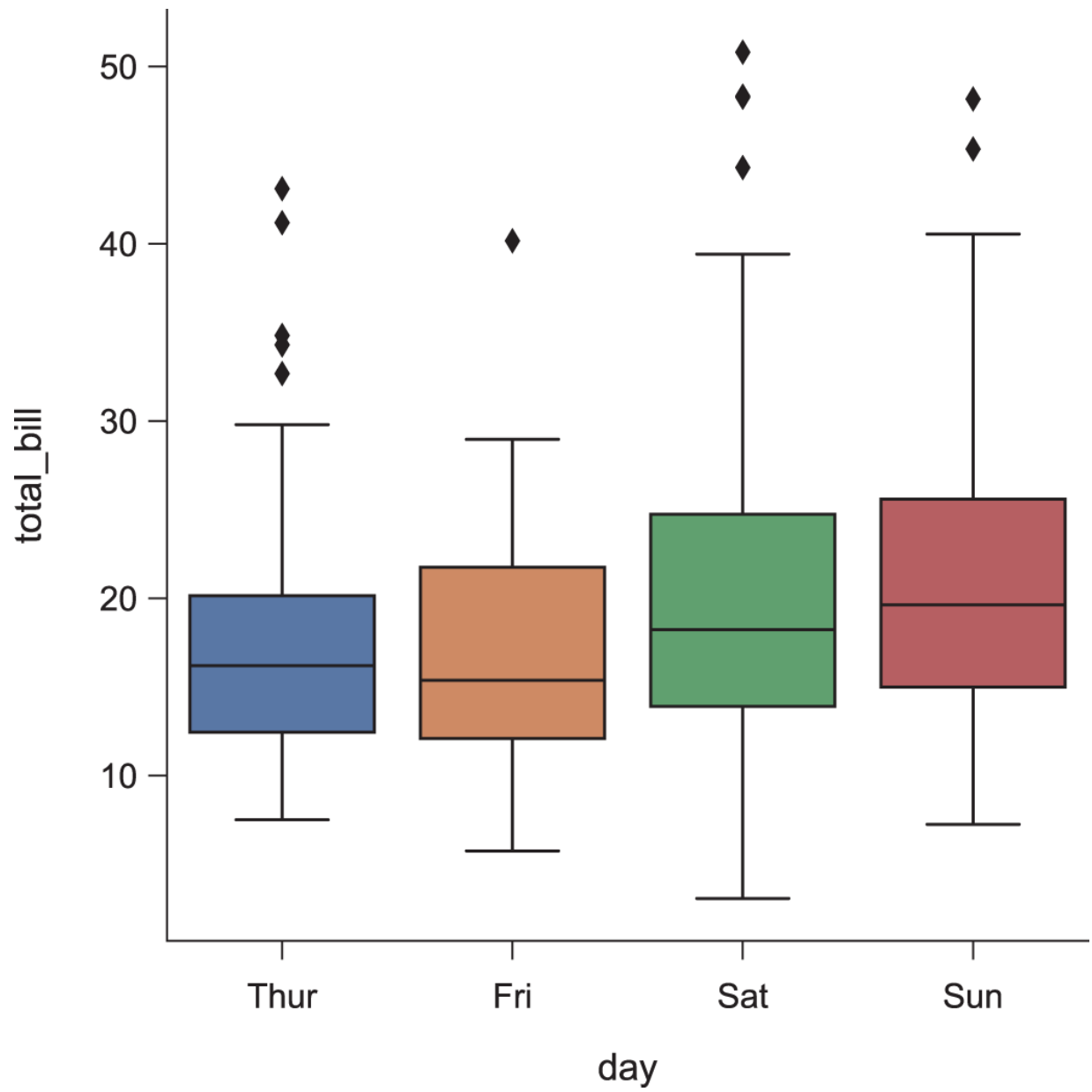
Figure 17.53 Boxplot using catplot.
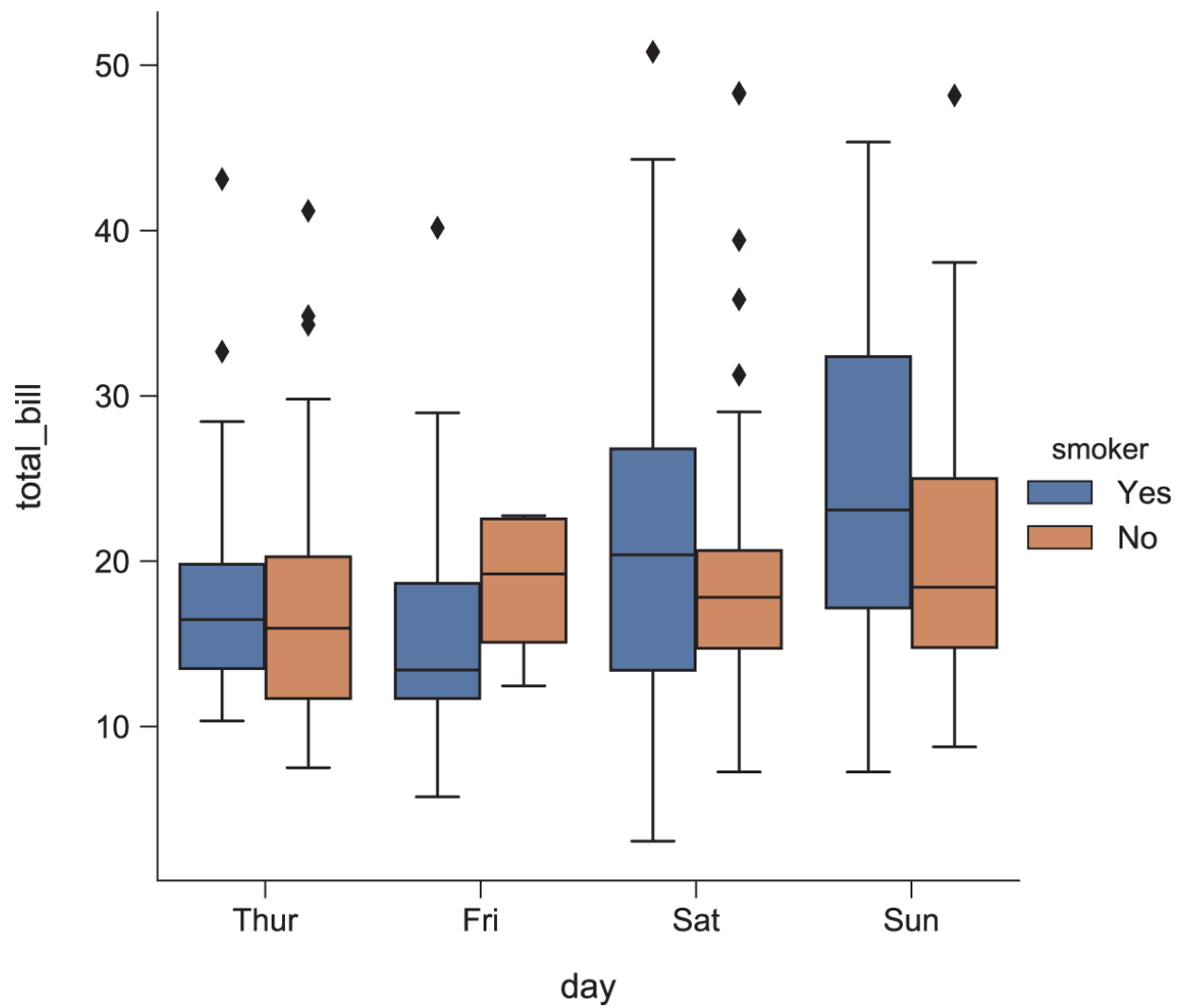
Figure 17.54 Boxplot using catplot with a hue.
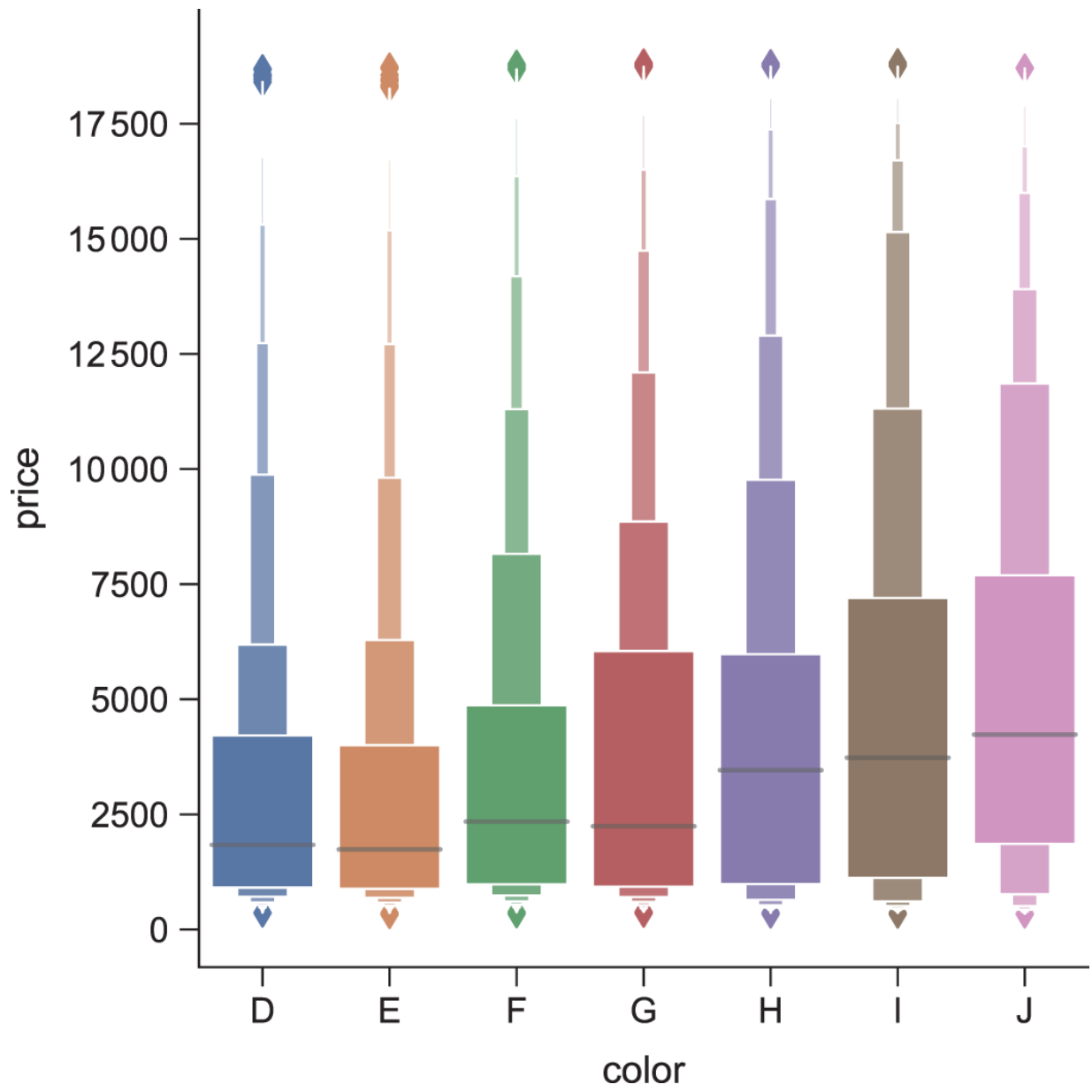
Figure 17.55 Boxen plot using catplot.

```
>>> diamonds.head()
    carat        cut color clarity  depth  table  price     x     y     z
0    0.23      Ideal     E     SI2   61.5   55.0    326  3.95  3.98  2.43
1    0.21    Premium     E     SI1   59.8   61.0    326  3.89  3.84  2.31
2    0.23       Good     E     VS1   56.9   65.0    327  4.05  4.07  2.31
3    0.29    Premium     I     VS2   62.4   58.0    334  4.20  4.23  2.63
4    0.31       Good     J     SI2   63.3   58.0    335  4.34  4.35  2.75
>>> diamonds.dtypes
carat        float64
cut           object
color         object
clarity       object
depth        float64
table        float64
price          int64
x            float64
y            float64
z            float64
dtype: object
>>> diamonds.color.unique()
array(['E', 'I', 'J', 'H', 'F', 'G', 'D'], dtype=object)
>>> sns.catplot(x="color", y="price", kind="boxen",
            data=diamonds.sort_values("color"));
```

The previous example of the boxen plot allowed us to get a better picture of the distribution with more groups than a box plot; however, our next example allows us to get the overview given by the box plot whilst also displaying the information of a box plot. To do this, we set the kind to violin which gives us a violin plot. In our example, we use the common total bill and day values that we have used previously; we apply a hue to the data using the variable time which has the unique values of Dinner and Lunch. The plot calculates the kernel density estimate (KDE) with the box plot representation of the data inside it. The KDE aspect of this plot is something that we will go into more detail. A KDE looks to represent the data in much the same way

as histogram summarising the distribution. For a histogram, we would look to set the bin numbers to determine how the plot would look, now we cannot do this for the KDE aspect of the violin plot instead we have to set a smoothing parameter. In the example given, you can see that this is set to 0.15, this is in contrast to the default value 1. The choice of this parameter is key to how your plot looks as over smoothing may remove aspects of the dataset. The other argument that we pass here which we have not seen before is the cut parameter. This is much more straightforward as it determines where we extend the plot when used with the smoothing value. In this case, we set the value to 0 which means we truncate at the end of the data.

In the example, we can see that we do not have data points for each combination of day and time; therefore, for Thursday and Friday, we have two violins and for Saturday and Sunday we only have the one. Another interesting consequence is how a single data point is dealt with as we have only one for Thursday Dinner. In this case, we have a single line at the point 18.78 representing that total bill value. If we did not want this, we could remove it from the data before plotting it. The result of this is shown in Figure 17.56.

```
>>> tips.head()
   total_bill    tip      sex smoker    day     time   size
0       16.99   1.01   Female     No    Sun   Dinner      2
1       10.34   1.66     Male     No    Sun   Dinner      3
2       21.01   3.50     Male     No    Sun   Dinner      3
3       23.68   3.31     Male     No    Sun   Dinner      2
4       24.59   3.61   Female     No    Sun   Dinner      4
>>> tips.time.unique()
[Dinner, Lunch]
Categories (2, object): [Dinner, Lunch]
>>> tips.groupby(by=['day', 'time']).count()
             total_bill     tip     sex    smoker    size
day   time
Thur  Lunch        61.0    61.0    61.0      61.0    61.0
      Dinner        1.0     1.0     1.0       1.0     1.0
Fri   Lunch         7.0     7.0     7.0       7.0     7.0
      Dinner       12.0    12.0    12.0      12.0    12.0
Sat   Lunch         NaN     NaN     NaN       NaN     NaN
      Dinner       87.0    87.0    87.0      87.0    87.0
Sun   Lunch         NaN     NaN     NaN       NaN     NaN
      Dinner       76.0    76.0    76.0      76.0    76.0
>>> thursday = tips[(tips['day']=='Thur')]
>>> thursday.head()
    total_bill    tip    sex smoker     day    time   size
77       27.20   4.00   Male     No    Thur   Lunch      4
78       22.76   3.00   Male     No    Thur   Lunch      2
79       17.29   2.71   Male     No    Thur   Lunch      2
80       19.44   3.00   Male    Yes    Thur   Lunch      2
81       16.66   3.40   Male     No    Thur   Lunch      2
>>> thursday[thursday['time']=='Dinner']
     total_bill    tip      sex smoker     day      time   size
243       18.78   3.0   Female     No    Thur    Dinner      2
>>> sns.catplot(x="total_bill", y="day", hue="time",
          kind="violin", bw_adjust=.15, cut=0,
          data=tips);
```
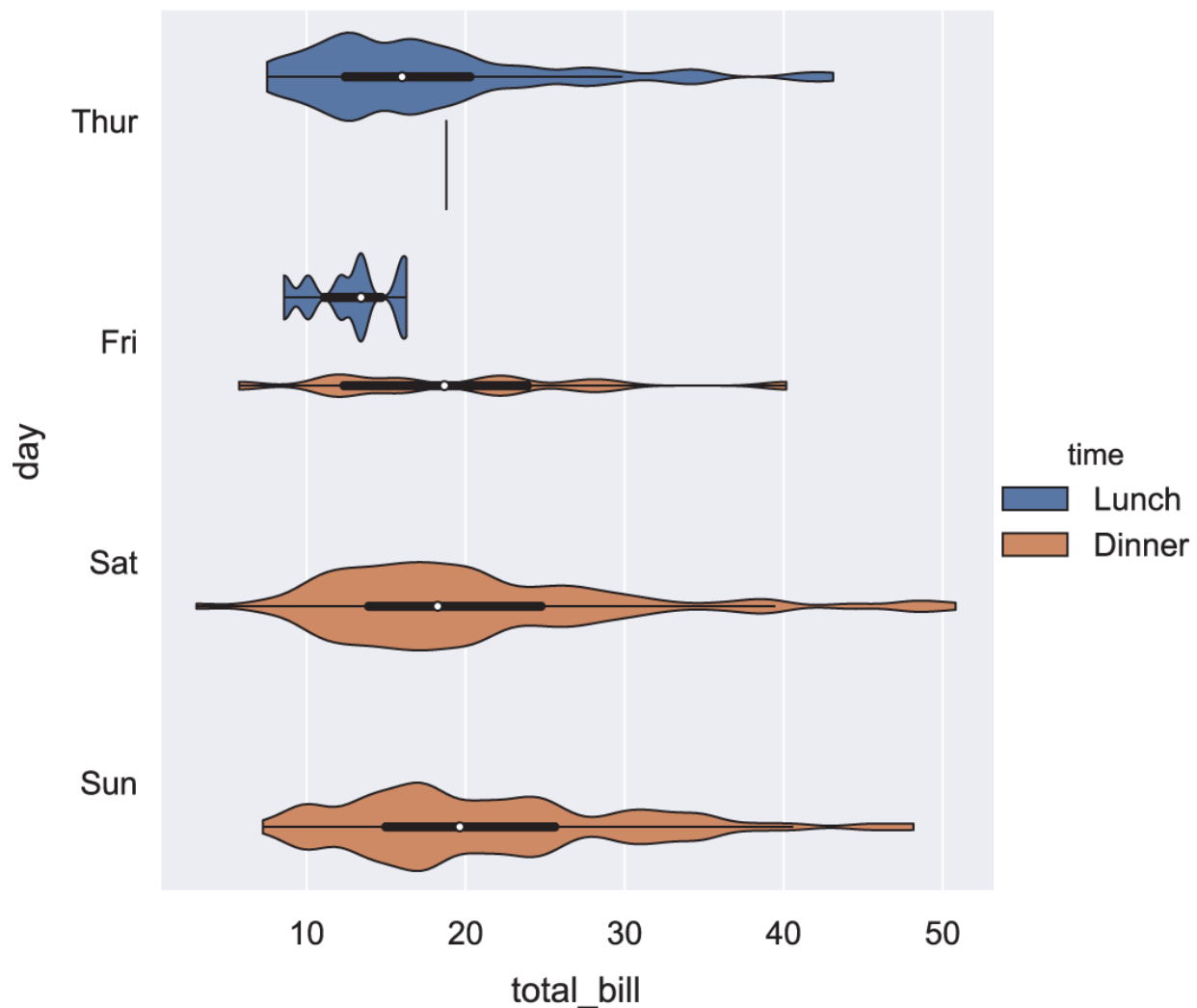
Figure 17.56 Violin plot using catplot.

An interesting variation of this is to set the split value to True which when applied with a hue shows both sets of values in the same violin. So in this example, we have one violin per day as opposed to two that we might expect when we normally use hue. In doing this, our box plot is done on the data as a whole with the split on the violin so you can compare the overall distribution to the distribution of each category of the hue. This example is also good to compare the effect of

the smoothing parameter which is set to the default here compared to the adjustment applied in the previous example. The resulting plot is shown in Figure 17.57.

```
>>> sns.catplot(x="day", y="total_bill", hue="sex",
                kind="violin", split=True, data=tips);
```
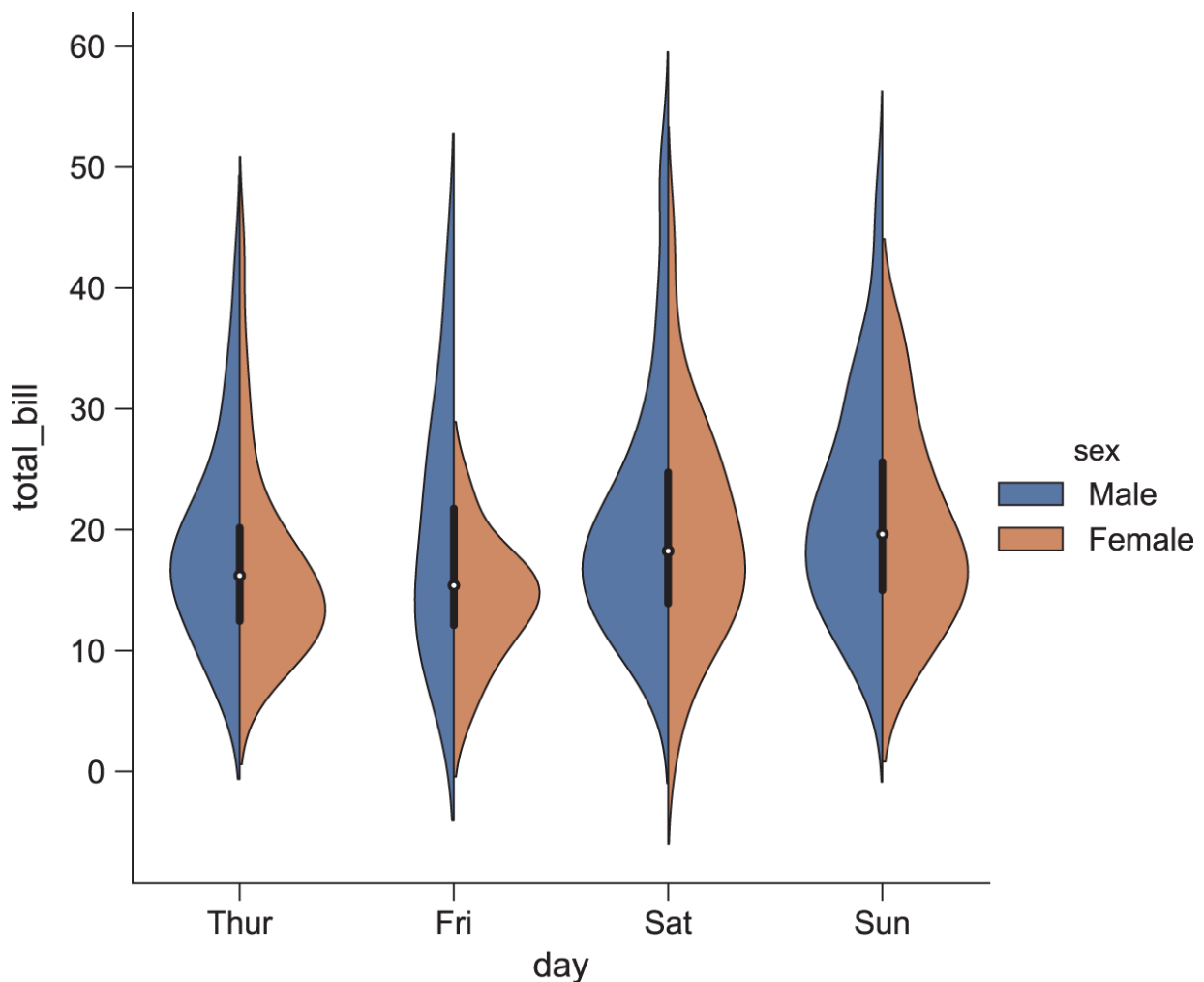

Figure 17.57 Violin plot using catplot using a split on the hue.

Next we look at how we can capture variability within our dataset over categorical data using a bar plot. To achieve this, we pass bar as the

argument to kind which produces a bar plot based on our x and y variables which are survived and sex with the hue class as shown in . The default operation that the bar plot applies to the data is the mean and as such if we have multiple observations a confidence interval is bootstrapped from the data and shown as a vertical line at the top of the bar.

```
>>> import seaborn as sns
>>> import matplotlib.pyplot as plt
>>> sns.set(style="ticks", color_codes=True)
>>> tips = sns.load_dataset("tips")
>>> titanic = sns.load_dataset("titanic")
>>> titanic.head()
   survived  pclass     sex   age  ...  deck  embark_town  alive  alone
0         0       3    male  22.0  ...   NaN  Southampton     no  False
1         1       1  female  38.0  ...     C    Cherbourg    yes  False
2         1       3  female  26.0  ...   NaN  Southampton    yes   True
3         1       1  female  35.0  ...     C  Southampton    yes  False
4         0       3    male  35.0  ...   NaN  Southampton     no   True
[5 rows x 15 columns]
>>> titanic.dtypes
survived          int64
pclass            int64
sex              object
age             float64
sibsp             int64
parch             int64
fare            float64
embarked         object
class          category
who              object
adult_male         bool
deck           category
embark_town      object
alive            object
alone              bool
dtype: object
>>> sns.catplot(x="sex", y="survived", hue="class", kind="bar",
    data=titanic);
```
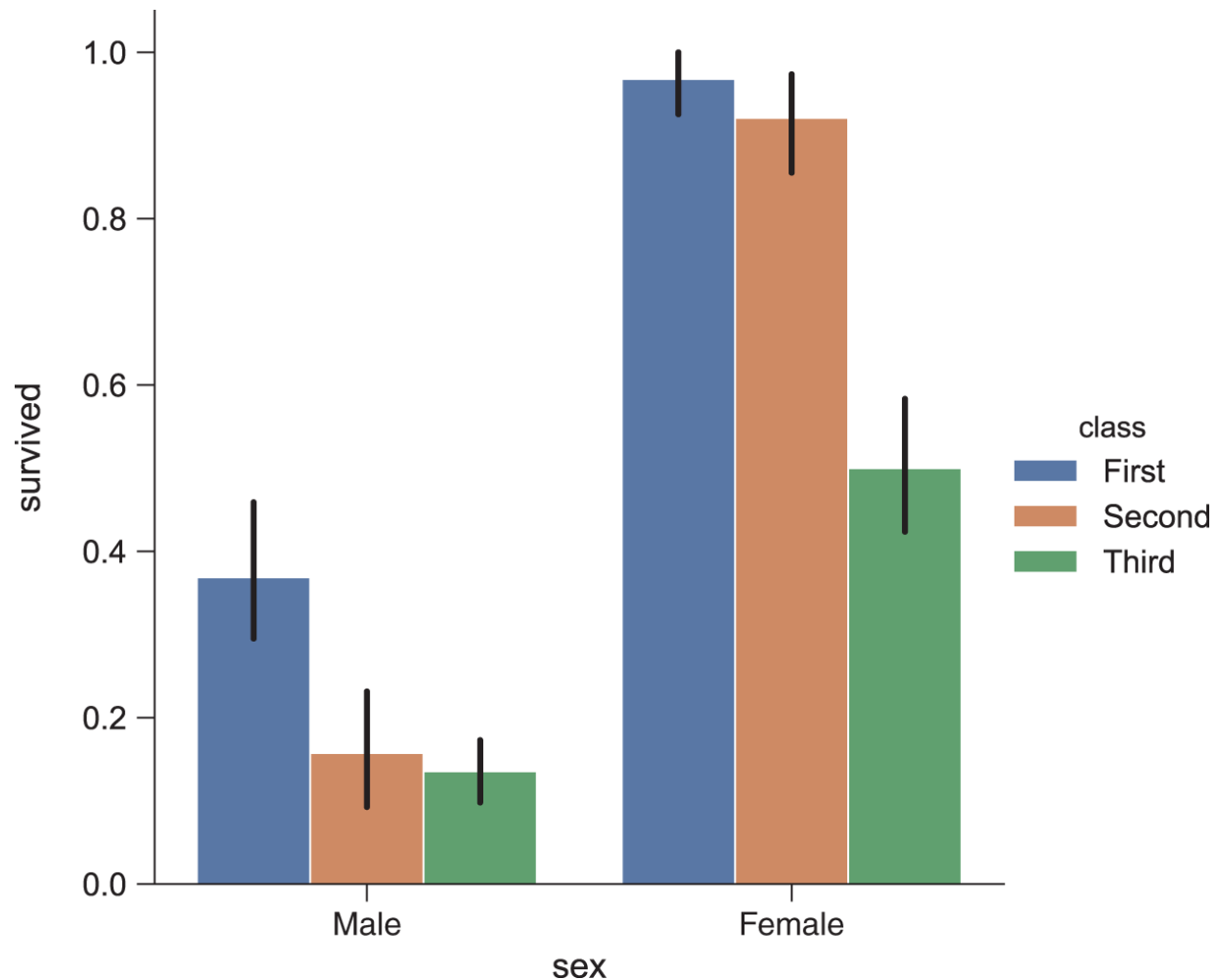
Figure 17.58 Bar plot using catplot.

If we are interested in the frequency and not the mean, we can pass count to the kind argument which gives us a countplot as shown in Figure 17.59. Here we have not passed an x value and instead use the hue argument to group by class with the y value set to deck. What happens here is that as we are looking to obtain the counts of the deck variable grouped by class which means the output is the count per grouping which then becomes the x axis.

```
>>> sns.catplot(y="deck", hue="class", kind="count",
          palette="pastel", edgecolor=".6",
          data=titanic);
```

The examples we have looked at so far have concentrated on rela-tionships between variables that are explicitly passed to be plot. If we take the example of the iris dataset, we can be much more loose and pass the data to the method to get it applied across the dataset. In this example, the data is of type float for four of the five columns and passing this into the catplot with the type set as box results in box plots for these four variables of type float. So we can use seaborn to be more exploratory when producing plots of our datasets as shown in Figure 17.60. It should also be noted that we achieved the horizon-tal box plots by passing the orient argument set to h.
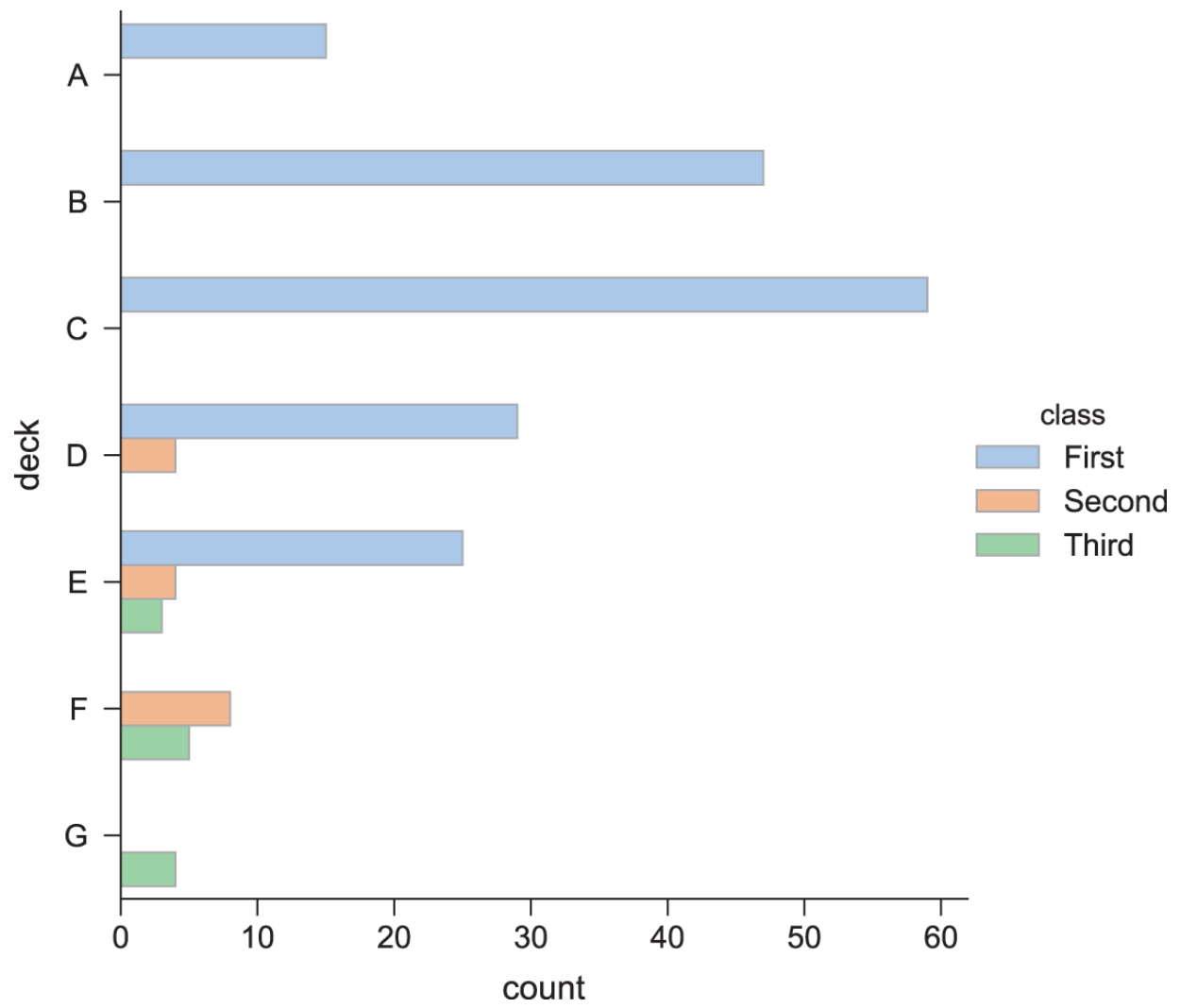
Figure 17.59 Count plot using catplot.
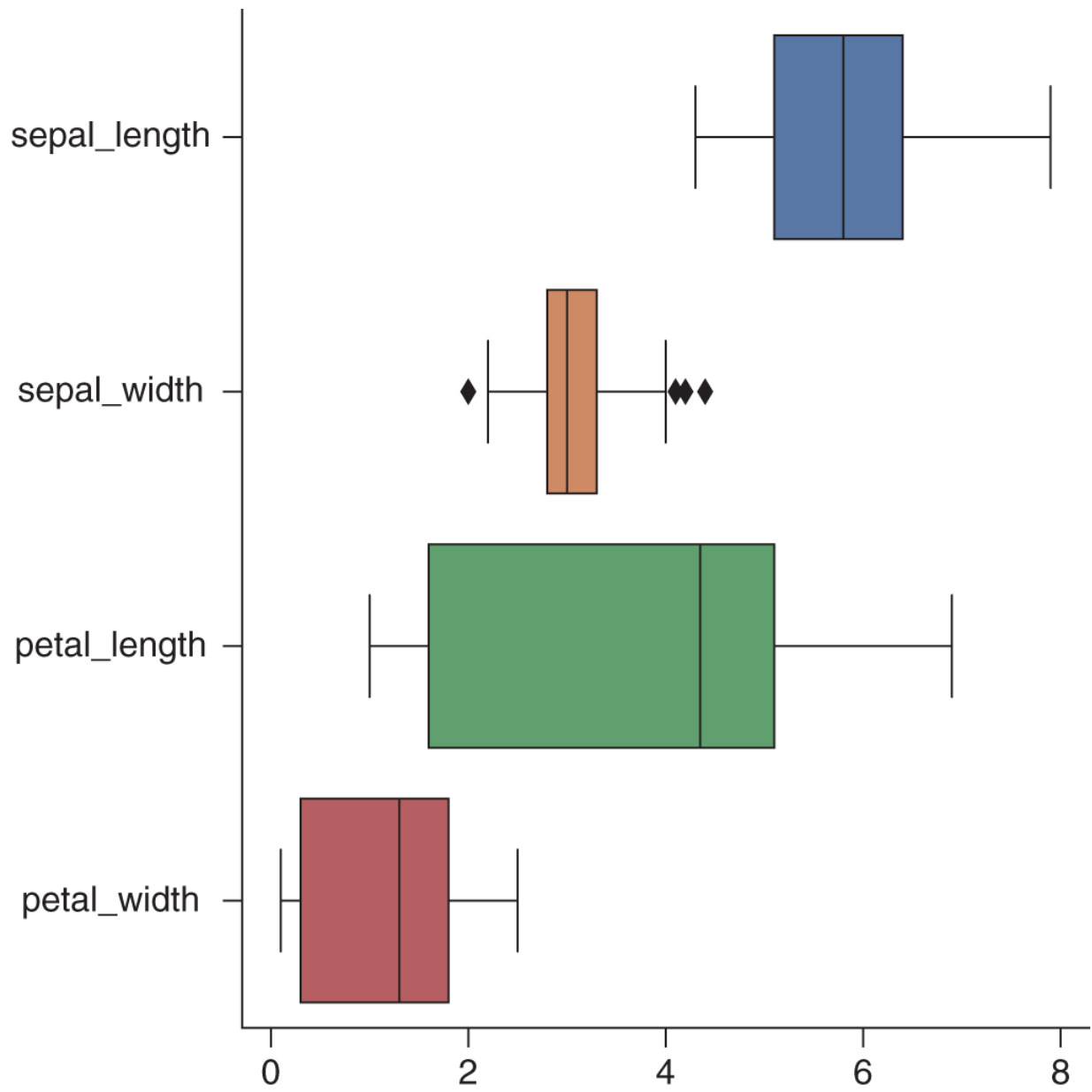
Figure 17.60 Boxplot of iris data.

```
>>> iris = sns.load_dataset("iris")
>>> iris.head()
   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2   setosa
1           4.9          3.0           1.4          0.2   setosa
2           4.7          3.2           1.3          0.2   setosa
3           4.6          3.1           1.5          0.2   setosa
4           5.0          3.6           1.4          0.2   setosa
>>> iris.dtypes
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species          object
dtype: object
>>> sns.catplot(data=iris, orient="h", kind="box");
```
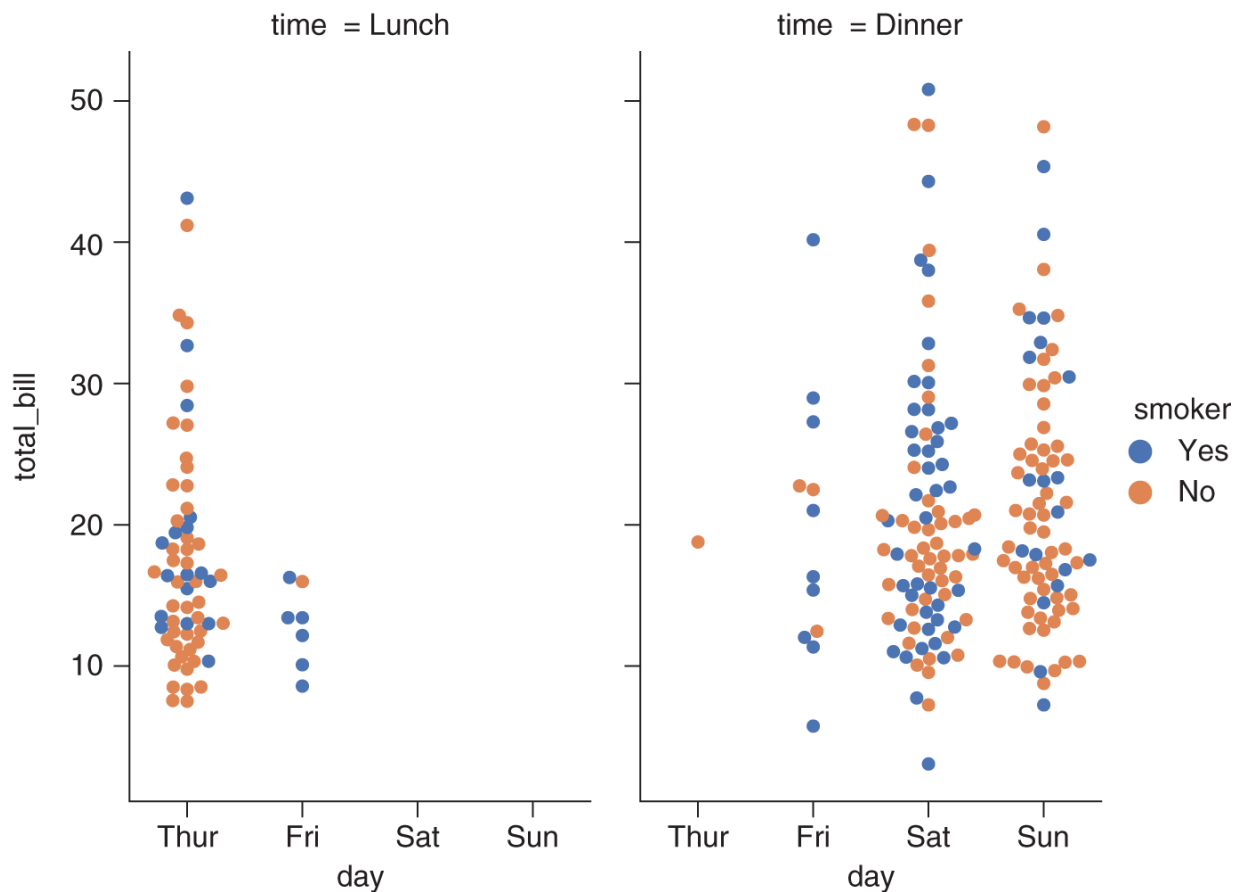

Figure 17.61 Multiple plots with col in catplot.

Having shown how to apply hue on catplot, we can also use the col argument to create multiple plots. This is demonstrated using the tips dataset and plotting a swarm plot of day against total bill with the hue being smoker, but to extend this we add the col as time which gives us two plots one for time set to Lunch and one for time set to Dinner. This is shown in Figure 17.61.

```
>>> sns.catplot(x="day", y="total_bill", hue="smoker",
          col="time", aspect=.6,
          kind="swarm", data=tips);
```

Next, we look at plotting a single set of data as opposed to one value against another. We call this a univariate distribution, and for this type of data, we may want to use a histogram. To demonstrate this, we can simply generate some random data and then pass this into distplot to give us a histogram with a KDE fit to our data as shown in Figure 17.62.

```
>>> import numpy as np
>>> import pandas as pd
>>> import seaborn as sns
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> sns.set(color_codes=True)
>>> x = np.random.normal(size=100)
>>> sns.distplot(x);
```

We can customise the histogram and next we look to remove the KDE by setting the kde option to False and add a rugplot by setting the rug option to True as shown in Figure 17.63. A rugplot shows every single value of data as lines along the x axis giving us a representation of the data.
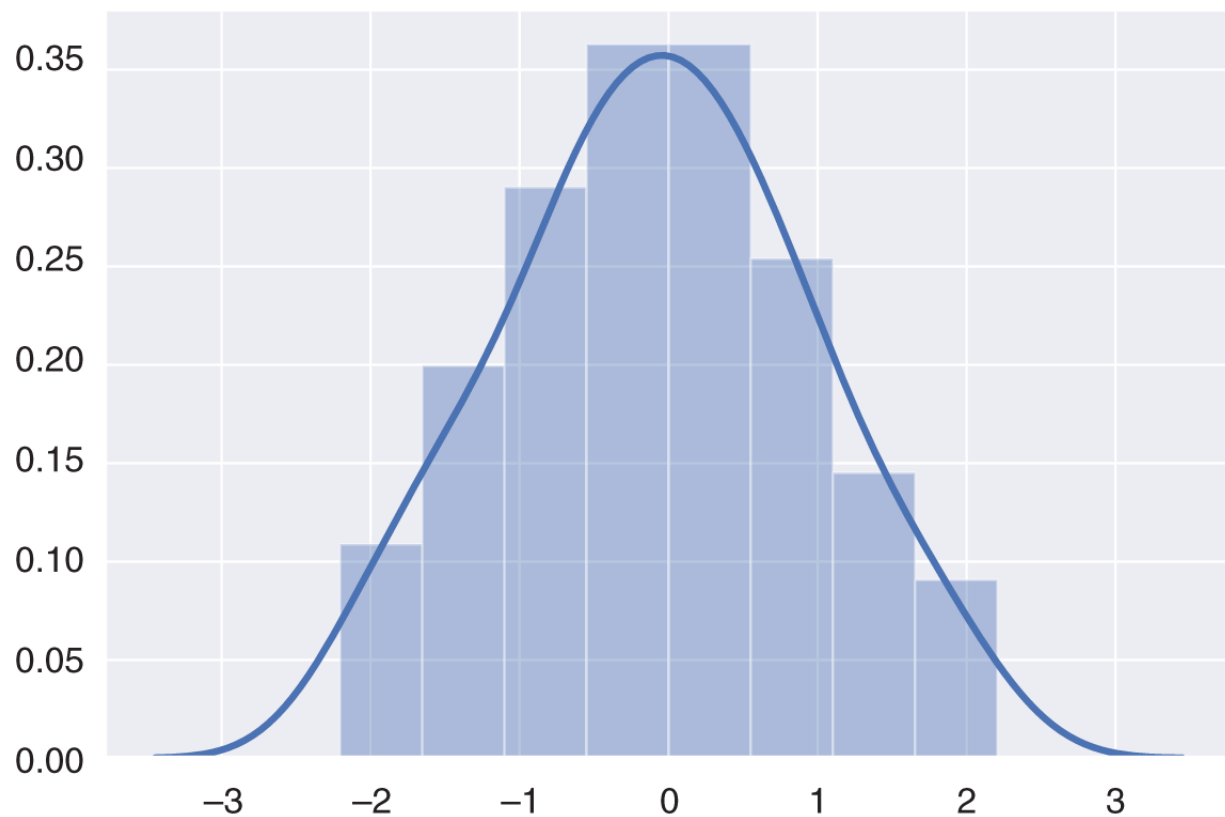
Figure 17.62 Histogram with KDE.
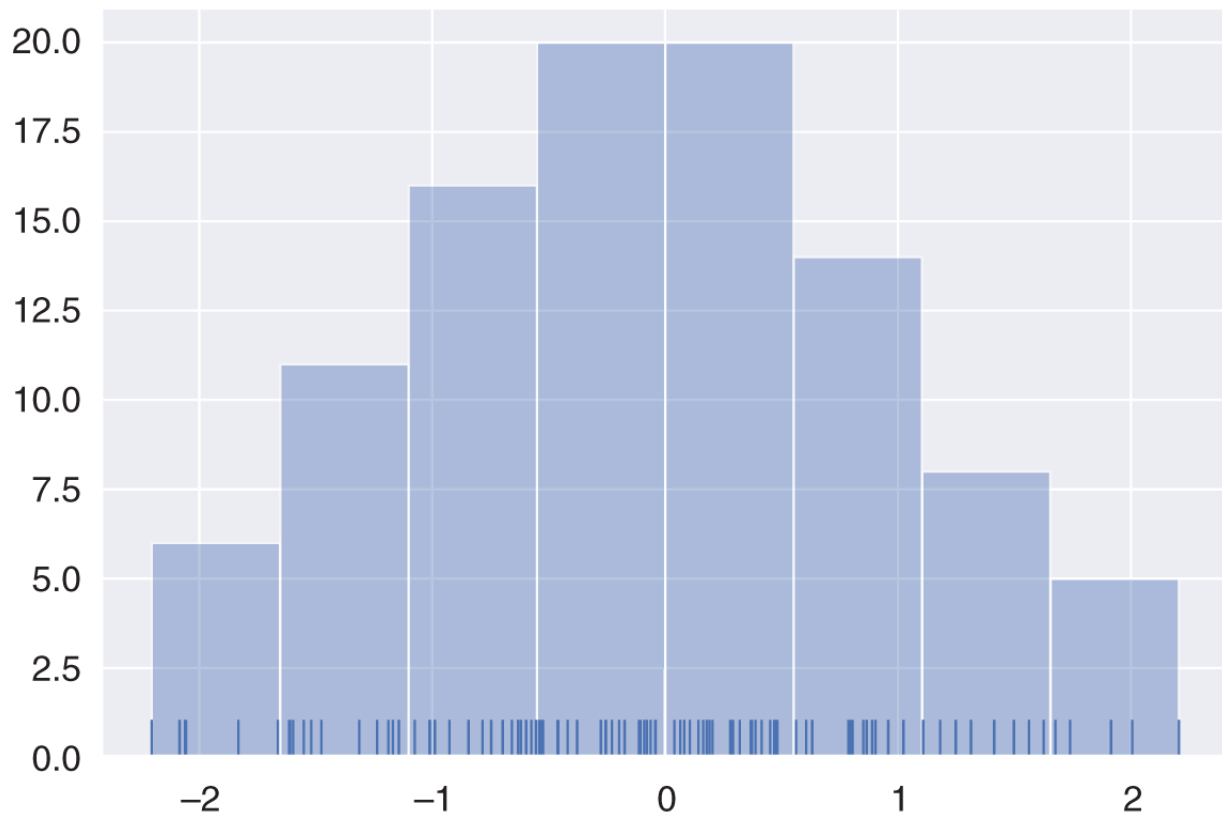
Figure 17.63 Histogram with ruglplot.

```
>>> sns.distplot(x, kde=False, rug=True);
```

The previous histogram bin values have been the default values that distplot have used with the dataset. If we want to specify the number of bins to use, we can just set the bin argument to the value that we want and the plot will divide into the number of bins requested as shown in Figure 17.64.

```
>>> x = np.random.normal(size=100)
>>> x[0:10]
array([-0.4087231,  0.6615343, -1.57704264,  0.56074442,  0.75721634,
       -0.18845652,  1.89587154, -0.8917129, -0.48700585, -0.69029233])
>>> sns.distplot(x, bins=20, kde=False, rug=True);
```
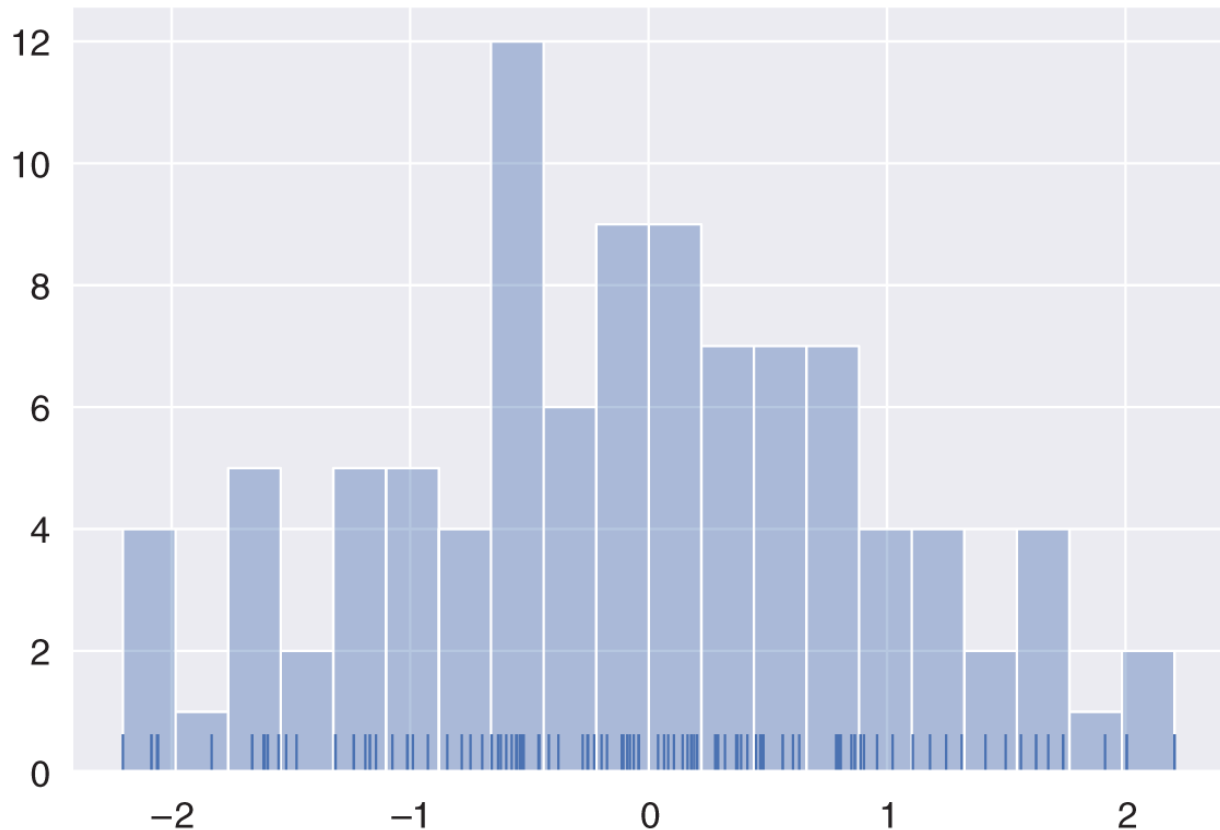
Figure 17.64 Histogram with bins option set.

Now having covered scatterplots and histograms, we now look at method which plots both; by using the jointplot method, we get a scatterplot of x against y as well as the histograms for each variable in the x and y axis. In our example, we create two random variables and put these into a DataFrame. This DataFrame is then passed in as the data argument and we set x and y arguments to refer to the columns in our DataFrame with the resultant plot shown in Figure 17.65.

```
>>> x = np.random.normal(size=100)
>>> y = np.random.normal(size=100)
>>> df = pd.DataFrame({'x':x, 'y': y})
>>> df.head()
          x           y
0 -0.926842   0.927292
1  1.402616   0.056310
2 -1.101788  -1.164577
3 -0.068730   0.469220
4  0.708112   1.861127
>>> sns.jointplot(x="x", y="y", data=df);
```
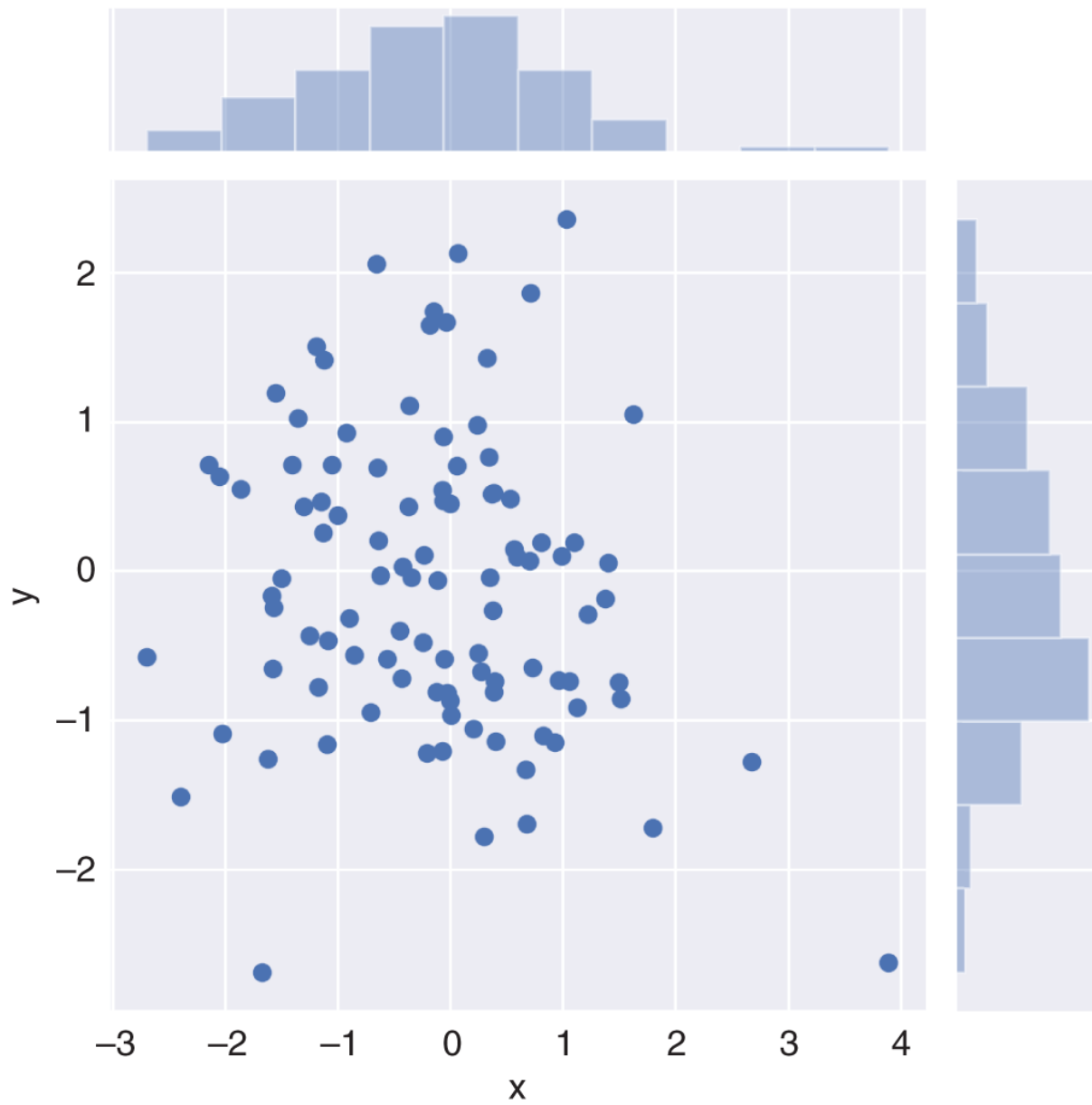
Figure 17.65 Joint plot.
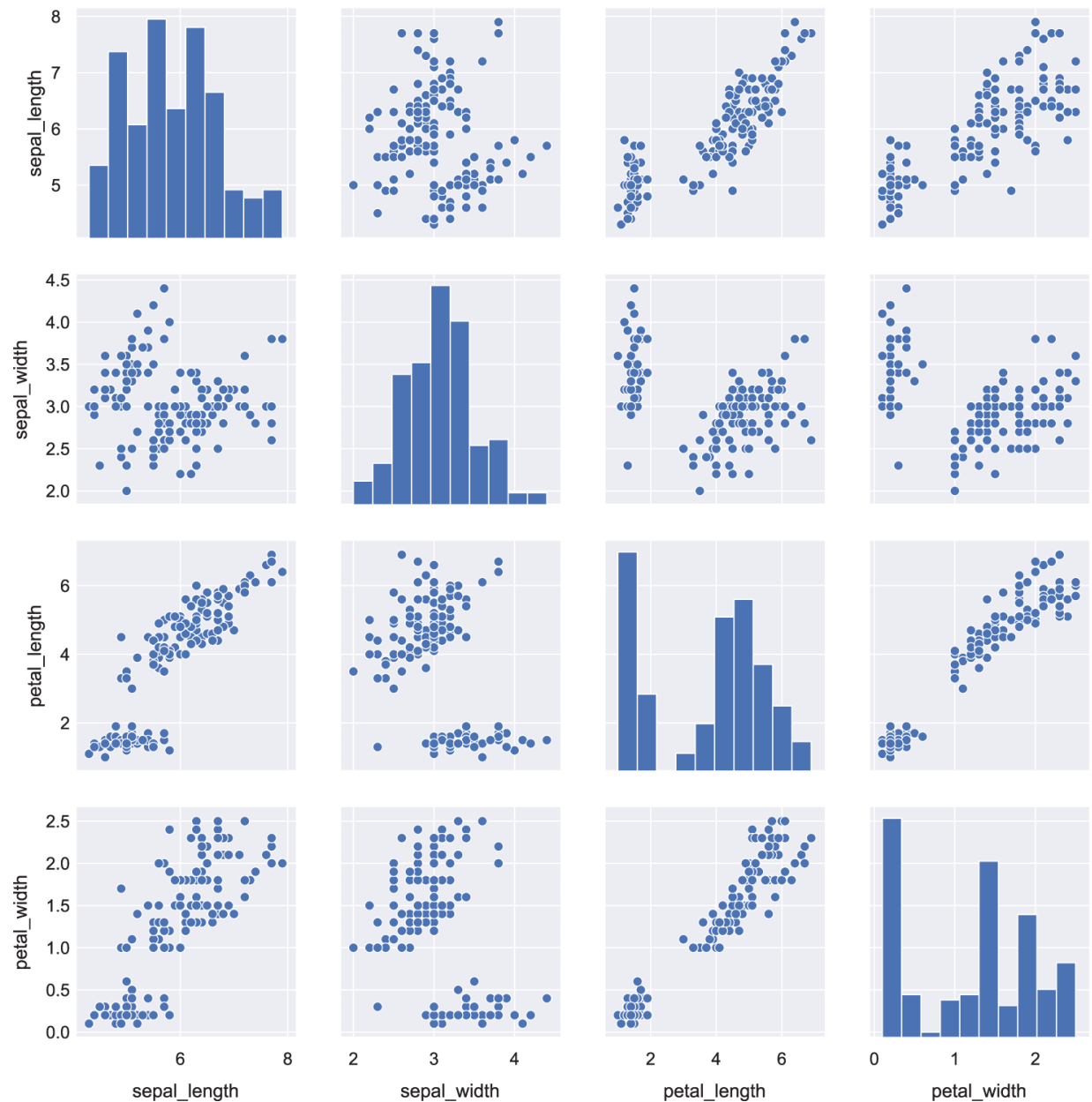
Figure 17.66 Pairplot example using iris data.

The last plot that we will consider in this chapter is the pairplot. In the example shown, we simply pass in the iris DataFrame and the result is a plot which shows every value plotted against every other one as a scatter plot with the distribution of each variable given as a histogram where the x and y names are the same as shown in Figure

17.66. Note in the example we have categorical data within the iris DataFrame yet the pairplot ignores this column.

```
>>> iris = sns.load_dataset("iris")
>>> iris.head()
   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2   setosa
1           4.9          3.0           1.4          0.2   setosa
2           4.7          3.2           1.3          0.2   setosa
3           4.6          3.1           1.5          0.2   setosa
4           5.0          3.6           1.4          0.2   setosa
>>> sns.pairplot(iris);
```

In this chapter, we have looked at plotting in Python and seen how we can produce simple easy to use through to very complicated customisable plots. What this hopefully demonstrates is the power of Python when it comes to producing plots. The examples shown should act as a reference for you to refer back to and give you a document of some of what can be achieved in Python.

18
APIs in Python

In this chapter, we will cover how to deal with APIs (application programming interfaces) using Python. To do this we are going to cover both how to create and access an API and build examples to do both. Before we get to writing any code we need to cover what an API is and why its useful. An API is a mechanism that allows communication between software applications and in this case will cover communication between an application and a web user. The uses of APIs have become increasingly popular allowing users to access data or communicate with services. They give a unified approach to doing so and therefore have become an important aspect to become familiar with, understanding how to communicate with.

We begin by creating our own API to do this. We are going to use the Python packages flask as well as the package flask-restful. To see if you have the packages you can try and import them.

```
>>> from flask import *
>>> from flask_restful import *
```

Now, flask comes by default with the Anaconda distribution of Python but you may not have flask_restful, if that is the case you will need to

install it. To do so go to https://anaconda.org/conda-forge/flask-restful to find something like the one shown in Figure 18.1.

Now, while this book has intended to be self-contained and not rely on many things outside the Anaconda distribution of Python, the url around things like the Anaconda website and subsequent links within it may change. If at the time of reading this is the case, then you just need to search the package list of Anaconda to get this. You could also do a simple search of conda flask restful using your favourite search engine and you should find the relevant web pages. You can then install flask-restful from the command line using one of the conda commands given.

With all the packages installed we can then look to create our very first API, to do so we will work within the confines of a script, so create a file called my_flask_api.py and add the following in the file.

```python
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

Figure 18.1 Example of flask-restful download page.

Let's first run this and then explain what is going on. To do so open up a terminal or command prompt and change directory to where your file is living.

Once there run the command Python my_flask_api.py and you will see something as shown in Figure 18.2.

What this is doing is starting up your API and you are now running it locally on machine. This means that it is accessible by you on your

machine but not available on the world wide web. To demonstrate this if we open up a web browser we can go to the ip address http://127.0.0.1:5000/ then we see what is shown in Figure 18.3.

```
(base) MacBook-Pro-3:23-apis rob$ python my_flask_api.py
 * Serving Flask app "my_flask_api" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 307-239-802
```

Figure 18.2 Display of terminal window upon starting up API.

Figure 18.3 Display of API from browser.

How has this all happened? Let's go back and look at the original code.

```python
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)
```

We initially import the relevant objects from both flask and flask_restful, and using this we create a flask app using the Flask method with the name of the current module passed in via __name__. This gives us an app object, and we can then create an api object by passing the app into the API method. This is our setup stage, next we want to add an endpoint to it.

```python
class HelloWorld(Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')
```

In this code snippet, we create a class named HelloWorld using the argument of Resource. Within this class we create a method called get, which simply returns the dictionary of hello with the key world.

```python
if __name__ == '__main__':
    app.run(debug=True)
```

Lastly, show the code that is executed to start the api up. In relation to the app object we use its run method with the argument of debug set as True. This makes the api available locally for the user. This block of code does raise an interesting line of code, namely,

```python
if __name__ == '__main__'
```

This is common place within Python but many use it without under-standing it, and we will attempt to resolve that now. Using an if state-ment with the == between two variables is pretty straightforward but what do __name__ and '__main__' mean? As shown when the Flask object was created, we used the __name__ variable which gives us the name of the current module. But how does this work, we will show by an example creating two files that each call the __name__ variable to show how it behaves. Let us call our first file file_one.py, we put the following code in there:

```
out_str = 'file one __name__ is {}'.format(__name__)
print(out_str)
```

If we run this code, then we see the output:

```
file one __name__ is __main__
```

Now, we create file_two.py and put in the following code:

```
out_str = 'file two __name__ is {}'.format(__name__)
print(out_str)
```

If we run this code, then we see the output:

```
file two __name__ is __main__
```

All sounds sensible but if we now import file_two.py into file_one.py as follows:

```
from file_two import *
out_str = 'file two __name__ is {}'.format(__name__)
print(out_str)
```

We get the following output:

```
file two __name__ is file_two
file one __name__ is __main__
```

What does this mean in relation to the original code snippet?

```
if __name__ == '__main__'
```

Essentially in having that bit of code, it means that if we run the code from the script that it is in, then we can execute what lives within the if statement.

So we have our API running on http://127.0.0.1:5000/ if we maintain the persistence of the script my_flask_api.py. So, keep that running in the same window you had it going in before. Now if we want to programmatically get the data from our own API, we need to use the requests package and access the data. If we work in the console we can interactively access the API running on our own machine using the following code:

```
>>> import requests
>>> data = requests.get('http://127.0.0.1:5000/')
>>> data
<Response [200]>
>>> type(data)
<class 'requests.models.Response'>
```

Now, if you refer back to the window where you are running your API from, you will notice that a get request was made to the endpoint / at the time you ran the requests.get method with the url as the argument. You'll notice that the data variable from the requests.get doesn't give us the json data that we saw from the website but instead gave us a response of 200. If we use the dir method around the data object, then we see the following:

```
>>> dir(data)
['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
 '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', '_content', '_content_consumed',
 '_next', 'apparent_encoding', 'close', 'connection', 'content', 'cookies',
 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect',
 'is_redirect', 'iter_content', 'iter_lines', 'json', 'links', 'next', 'ok',
 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text', 'url']
>>> data.json()
{'hello': 'world'}
>>> data.status_code
200
>>> data.reason
'OK'
>>> data.text
'{\n    "hello": "world"\n}\n'
>>> data.url
'http://127.0.0.1:5000/'
```

So, here we can see the methods and attributes of the data object and we look at a few of these. The json method of the object unsurprisingly gives the json that we saw from the endpoint via the web browser. The status_code attribute returns the status of the web request that we made here 200,which is a successful request. We will not cover all status codes within this book, however if you are interested then they are easily accessible online. Alongside a status code we also have a reason, here we would expect an informative message to alongside our status code. The text representation of what is returned is also available alongside the url we used. What is clear is that we get a lot of information back from our request.

Now this is a good first example, but it only demonstrates the get request and actually the information isn't that useful as we only get back some simple json. What we will do next is create an API that allows us to get, post, and delete information from a small movie database that we will add to the code. The full code is shown below and as usual we will step through it line by line:

```python
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

film_dict = {
    '1': {'Name': 'Avengers: Infinity War', 'Year': 2018, 'Month': 'March'},
    '2': {'Name': 'Ant Man and the Wasp', 'Year': 2018, 'Month': 'August'},
}


def abort_if_todo_doesnt_exist(film_id):
    if film_id not in film_dict:
        abort(404, message="Film {} doesn't exist".format(film_id))

parser = reqparse.RequestParser()
parser.add_argument('name')
parser.add_argument('year')
parser.add_argument('month')


class Films(Resource):
    def get(self, film_id):
        abort_if_todo_doesnt_exist(film_id)
        return film_dict[film_id]

    def delete(self, film_id):
        abort_if_todo_doesnt_exist(film_id)
        del film_dict[film_id]
        return '', 204

    def put(self, film_id):
        args = parser.parse_args()
        task = {'Name': args['name'],
                'Year': args['year'],
                'Month': args['month']}
        film_dict[film_id] = task
        return task, 201


class FilmDict(Resource):
    def get(self):
        return film_dict


api.add_resource(FilmDict, '/films')
api.add_resource(Films, '/films/<film_id>')


if __name__ == '__main__':
    app.run(debug=True)
```

The imports that we use within this API are as follows:

```python
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource
```

This is similar to what we used in the hello world example. However, now there are two more imports from flask_restful, namely reqparse and abort. Reqparse is an argument parser that we can make use of to process arguments sent with the web request to the API as shown below:

```python
parser = reqparse.RequestParser()
parser.add_argument('name')
parser.add_argument('year')
parser.add_argument('month')
```

Here, we create a RequestParser and then add arguments, name, year, and month, which we will pass when we need to add a film. Adding this doesn't allow us to get the arguments, to do this we need to parse them out and later in the code we have the following snippet of code:

```python
args = parser.parse_args()
```

What this does is parse the arguments from the RequestParser and then store them in a dictionary.

The abort import is used in a custom function that we use to send an appropriate message if the film:id doesn't exist.

```python
def abort_if_film_doesnt_exist(film_id):
    if film_id not in film_dict:
        abort(404, message="Film {} doesn't exist".format(film_id))
```

Here, we pass the film:id as an argument and if the id doesn't exist in the dictionary then a 404 is given with a custom message relating to the film that doesn't exist. We use this function in a number of places within the code as we will show.

Next, we consider the two classes that are in the code namely Films and FilmDict. The first of these looks as follows:

```python
class Films(Resource):
    def get(self, film_id):
        abort_if_film_doesnt_exist(film_id)
        return film_dict[film_id]

    def delete(self, film_id):
        abort_if_film_doesnt_exist(film_id)
        del film_dict[film_id]
        return '', 204

    def put(self, film_id):
        args = parser.parse_args()
        task = {'Name': args['name'],
                'Year': args['year'],
                'Month': args['month']}
        film_dict[film_id] = task
        return task, 201
```

This class defines get, delete, and put methods which do, as you would think, what it says on the tin and gets a film, deletes a film, and puts a film in our dictionary. Notice that in the get and delete methods, we use our abort_if_film:doesnt_exist function to first check if the film exists and sends the appropriate error message and status code. Note that we could have done it in each method but if we ended up with 20 different methods that would be lots of repeated code. The actual nuts and bolts of what this code does is pretty straight forward and uses dictionary methods covered within this book. Note that for the delete and put methods we return a status code alongside an empty string for the delete method or the task for the put method.

```
class FilmDict(Resource):
    def get(self):
        return film_dict
```

The next class only has a single get method and returns our entire database of films, but why have this as a separate class? This is because we add this to a separate url as shown in the following code snippet:

```
api.add_resource(FilmDict, '/films')
api.add_resource(Films, '/films/<film_id>')
```

Now, the FilmsDict class be accessed at the films url, if we want to get, delete, or put a film based on an id and we do so via the /films/ ¡film:id¿.

```
if __name__ == '__main__':
    app.run(debug=True)
```

And as before we run the app in debug mode. To get this running we change directory to where the code lives and run the script at which time we will have the API available on http://127.0.0.1:5000/ so if we navigate to http://127.0.0.1:5000/films which shows us the list as shown in Figure 18.4.

We can pass an id of a film that exists and get the result shown in Figure 18.5.
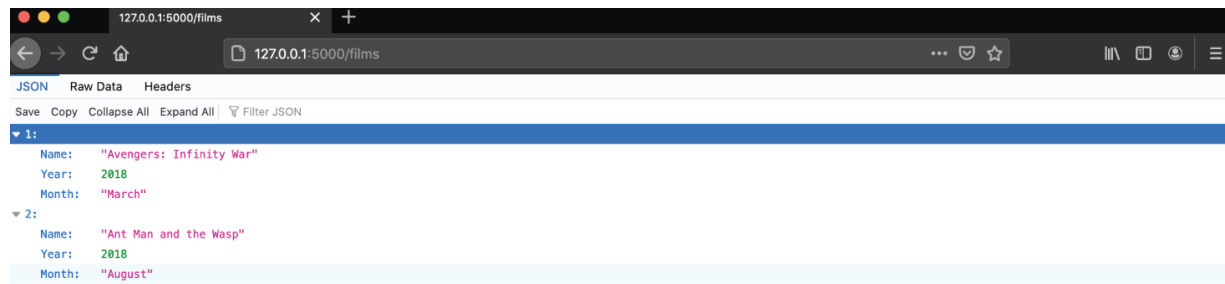
Figure 18.4 Display of API from browser getting all films.
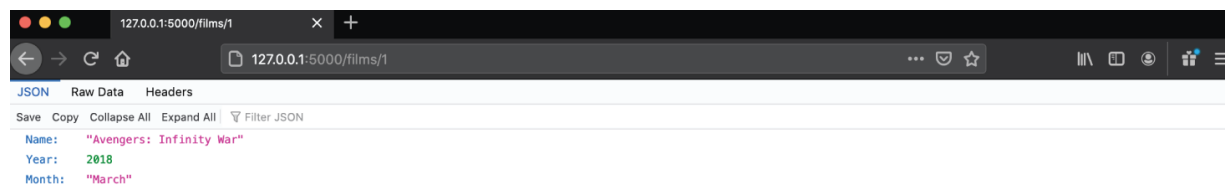
Figure 18.5 Display of API from browser getting film id 1.
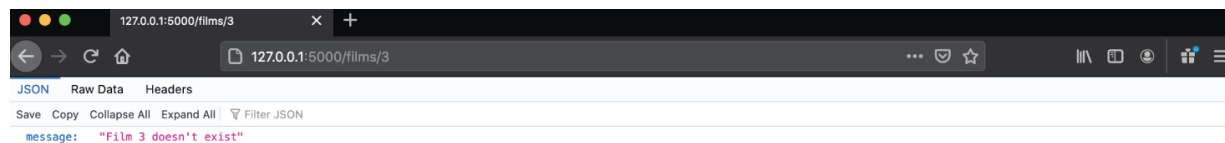
Figure 18.6 Display of API from browser getting film id 3.

If we enter the id for a film that isn't already in our database we get the message shown in Figure 18.6.

Now, if we want to use the API to add or delete films we can do so using the requests library. So, let's show some code examples.

```
>>> import requests
>>> r = requests.get('http://127.0.0.1:5000/films')
>>> r.json()
{'1': {'Name': 'Avengers: Infinity War', 'Year': 2018, 'Month': 'March'},
'2': {'Name': 'Ant Man and the Wasp', 'Year': 2018, 'Month': 'August'}}
>>> r = requests.get('http://127.0.0.1:5000/films/1')
>>> r.json()
{'Name': 'Avengers: Infinity War', 'Year': 2018, 'Month': 'March'}
```

So, we can call the same urls that we did before from the browser using requests. Now, let's show how to add and delete using the api.

```
>>> r = requests.delete('http://127.0.0.1:5000/films/1')
>>> r.text
''
>>> r.status_code
204
```

To delete a film based on the id, we use the delete method and pass in the url and note that the text and status code match that mentioned within the API code. Now, to add that record back in we can use the put method.

```
>>> r = requests.get('http://127.0.0.1:5000/films')
>>> r.json()
{'2': {'Name': 'Ant Man and the Wasp', 'Year': 2018, 'Month': 'August'}}
>>> r = requests.put('http://127.0.0.1:5000/films/1',
...                     data={'name': 'Avengers: Infinity War',
...                           'year': 2018,
...                           'month': 'March'})
>>> r.json()
{'Name': 'Avengers: Infinity War', 'Year': '2018', 'Month': 'March'}
>>> r.status_code
201
>>> r = requests.get('http://127.0.0.1:5000/films')
>>> r.json()
{'2': {'Name': 'Ant Man and the Wasp', 'Year': 2018, 'Month': 'August'},
'1': {'Name': 'Avengers: Infinity War', 'Year': '2018', 'Month': 'March'}}
```

Now, we can see that we had only the film with id 2 available so using the put method with the same arguments as the reqparser means we can use the data argument to populate id 1. In sending the put request we get back the data in the form it has been added with the variable names matching what we have in the API. Lastly, we check that it was added correctly by calling the http://127.0.0.1:5000/films and getting all the films we have, which shows the film with id 1 was added back in.

So far our API has been public but what if we want to secure it in some way? To do so we need some authentication on our API, and below we discuss some of the options available to us.

Basic Authentication Mentioned above is the definition of basic authentication (Basic Auth) which is a server asking for a username and a password (e.g. to a personal social media account) to allow access

to private data (though there are potential security risks to be aware of if this data is not encrypted as it is transferred). Basic Auth is performed over HTTP and is encrypted using SSL to protect the username and password being transmitted. Note that Basic Auth can be done without SSL but sensitive data will be transmitted over HTTP unencrypted which is an extreme security risk. Basic Auth is a simple authentication technique which makes coding it into scripts a relatively straightforward process. However, since it relies on the use of a username and password to access the API and manage the account associated with it, this is not ideal. It is like you lending your car keys to your friend and the keys can open everything in your house and workplace as well. Put differently, if you give your social media usernames and passwords out to scripts, those scripts will end up having a far greater access to your personal social media accounts than you might like!

API Key Authentication API key authentication is a technique that overcomes the weakness of Basic Auth by requiring the API to be accessed with a unique key. The key is usually a long series of letters and numbers that is completely separate from the user's login details (e.g. username and password). As such, API keys can be intentionally limited for security reasons, so that they provide access only to the bits of data and services users we need, rather than granting access to everything.

OAuth Token OAuth is a prevailing standard that applications can use to provide client applications with secure access that operates using the principles of API key authentication. OAuth authorises devices, APIs, servers and applications with access tokens rather than credentials. When OAuth is used to authenticate a connection to a server an authentication request is sent from the client application (in the present case, a Python script that we build) to an authentication server. It is the authentication server that generates the OAuth token. The token is returned to the client application over HTTPS, which then passes it to the API server. You may have come across websites or apps that ask you to login with your Google, Facebook, or Twitter account. In these cases Google, Facebook, or Twitter are acting as an authentication server. Note that an authentication server doesn't need to be a third-party server, but will generally be a different server to the one providing data.

In the last example of this chapter, we will create an API that uses basic authentication to authenticate a web request. The full code is shown as follows:

```python
from flask import Flask
from flask_restful import Resource, Api
from flask_httpauth import HTTPBasicAuth

app = Flask(__name__)
api = Api(app)
auth = HTTPBasicAuth()

USER_DATA = {
    "admin": "atCh_5K}?g"
}


@auth.verify_password
def verify(username, password):
    if not (username and password):
        return False
    return USER_DATA.get(username) == password


class HelloWorld(Resource):
    @auth.login_required
    def get(self):
        return {"Hello": "World"}


api.add_resource(HelloWorld, '/hello_world')

if __name__ == '__main__':
    app.run(debug=True)
```

Now much of this code is similar to what we have seen before, so we will just go over the new elements of which there are two.

```python
from flask import Flask
from flask_restful import Resource, Api
from flask_httpauth import HTTPBasicAuth

app = Flask(__name__)
api = Api(app)
auth = HTTPBasicAuth()

USER_DATA = {
    "admin": "atCh_5K}?g"
}
```

In the above block of code we add in the HTTPBasicAuth from flask_httpauth, you may need to install this package so refer back to earlier in the book where we showed you where to find the command for a specific package. With this imported we create a HTTPBasic-Auth() object and assign it to auth. We then create a dictionary containing user data which has a username and password as key and value.

```python
@auth.verify_password
def verify(username, password):
    if not (username and password):
        return False
    return USER_DATA.get(username) == password


class HelloWorld(Resource):
    @auth.login_required
    def get(self):
        return {"Hello": "World"}
```

Next, we create a verify function which takes the username and password as arguments and returns True if the value of the dictionary call using the username gives the password we have. Note that we use get method of the dictionary so we don't get a key error. This is decorated this with the auth verify_password. We can then use the decorator auth.login_required on our get method that means we must be logged in to get the return of the HelloWorld class. The resource is added to the endpoint /hello_world and we run the API in the way we have done in the previous examples.

With the API running we can then attempt to access the endpoint http://127.0.0.1:5000/hello_world using the requests library.

```
>>> r = requests.get('http://127.0.0.1:5000/hello_world')
>>> r.status_code
401
>>> r = requests.get('http://127.0.0.1:5000/hello_world',
                     auth=('admin', 'atCh_5K}?g'))
>>> r.status_code
200
>>> r.json()
{'Hello': 'World'}
```

In the first example, we use the standard get method and get back the status code of 401 which refers to us being unauthorised. So to become authorised in the second example we pass a tuple of user-name and password to the auth argument and we get access to the endpoint and see we get back the expected json response.

In this chapter, we have covered API's how to create then and how to access them. All examples have been run locally on our machine however Python is a great tool for production quality APIs. The examples relating to how to access APIs are particularly useful for those wishing to work with different data sources as APIs are common place as a solution to allow users to interact with. While requests are great for interfacing directly with APIs you may find packages that wrap up some of the complexity associated with providers APIs.

19
Web Scraping in Python

The last chapter of the book covers the concept of web scraping. This is the programmatic process of obtaining information from a web page. To do this we need to get up to speed on a number of things:

- html
- obtaining a webpage
- getting information from the webpage

To do this we will create our own website using Python that we will scrape with our own code.

19.1 An Introduction to HTML

HTML stands for Hyper Text Markup Language and is the standard markup language for creating web pages. It is essentially the language that makes up what you see on the internet. An HTML file tells a web browser how to display the text, images, and other content on a webpage. The purpose of HTML is to describe how the content is structured and not how it will be styled, and rendered within a web browser. To render the page you need to use a cascading style sheet (CSS) and an HTML page can link to a CSS file to get information on

colours, fonts, and other information relating to the rendering of the page.

HTML is a markup language, so in creating HTML content you are embedding the text to be displayed alongside how the text should be displayed. The way this is done is by using HTML tags which can contain name-value pairs which are known as attributes. Information within a tag is known as an HTML element. Well-formed HTML should have an open and a close tags, and before you start a new tag you should close off your old tag.

Now, that we have described what HTML is we will give some examples of how you create elements within it and show how to put together a page. Let's start by looking at some tags. It is important to remember that when we open a tag we close it with a / (forward slash). Let's demonstrate with a header tag.

Header

This is how we define a header

```
<h>This is how we define a header</h>
```

Here, we see that to open the tag we have

```
<h1>
```

and to close it we have

```
</h1>
```

Paragraph

Next, we show how to tag a paragraph:

```
<p>This is how we define a paragraph</p>
```

Define

Here, we show how to define a tag, which is how we have embedded a hyperlink:

```
<a href='https://www.google.com">This is how we define a link</a>
```

Table

The next HTML tag is for a table which is a bit more complex than what we have covered before.

```
<table>
  <tr>
    <th>Name</th>
    <th>Year</th>
    <th>Month</th>
  </tr>
  <tr>
    <td>Avengers: Infinity War</td>
    <td>2018</td>
    <td>March</td>
  </tr>
  <tr>
    <td>Ant Man and the Wasp</td>
    <td>2018</td>
    <td>August</td>
  </tr>
</table>
```

So the first thing we need to define for a table is the table tag:

```
<table></table>
```

Next, we need to define the rows in the table using the table row tag this is denoted using:

```
<tr></tr>
```

Here, we have three rows defined.

In each row we need to have some data so you'll see we use

`<th>`

and

`<td>`

tags. The

`<th>`

tags refer to the table header and

`<td>`

the table data. So here we have the headers being Name, Year, and Month and then the next two rows are the table data.

Thead and Tbody

There are two other tags that we can add to this table and that is the thead and tbody tag. Within a table these can separate out the head and body of the table. They are used as follows:

```
<table>
<thead>
  <tr>
    <th>Name</th>
    <th>Year</th>
    <th>Month</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>Avengers: Infinity War</td>
    <td>2018</td>
    <td>March</td>
  </tr>
  <tr>
    <td>Ant Man and the Wasp</td>
    <td>2018</td>
    <td>August</td>
  </tr>
</tbody>
</table>
```

Div

The last tag we will introduce is a div tag. This is a tag that defines a section in the html. So linking back to the previous table example we can put a div tag around it. In putting html within a div we can apply the format to the whole section covered by it.

```
<div>
<table>
<thead>
  <tr>
    <th>Name</th>
    <th>Year</th>
    <th>Month</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>Avengers: Infinity War</td>
    <td>2018</td>
    <td>March</td>
  </tr>
  <tr>
    <td>Ant Man and the Wasp</td>
    <td>2018</td>
    <td>August</td>
  </tr>
</tbody>
</table>
</div>
```

HTML Attributes

Having defined lots of the tags, we will now discuss the attributes of these tags. Attributes provide us with additional information about the elements. We will show how these relate back to the tags we defined earlier. Let's begin by showing an example of an attribute applied to an

<a>

tag.

```
<a href='https://www.google.com">This is how we define a link</a>
```

Here, the href is the attribute and it specifies what the url is.

We could also add a title to a tag which would result in the value being displayed as a tooltip (when you hover over it).

```
<p title="It will show when you hover over the text">
This is how we define a paragraph.</p>
```

Id and Classes

Having introduced attributes, we will now look at two important ones which can help us locate elements within html, these are the id and class attributes. An id element is unique to that html element whereas a class can be used in multiple elements. Let's demonstrate this by looking at three headers with associated information.

```
<!-- A unique element -->
<h1 id='myHeader'>MCU Films</h1>

<!-- Multiple similar elements -->
<h2 class='film'>Avengers: Infinity War</h2>
<p>Can Earth's mightiest heroes protect us from the threat
   of Thanos?</p>

<h2 class='film'>Ant Man and the Wasp</h2>
<p>Following the events of Civil War will Scott Lang don the
   Ant Man suit again?</p>

<h2 class='film">Captain Marvel</h2>
<p>Plot Unknown.</p>
```

Here, we have one header with an id attribute. This unique attribute specifically identifies that header. The remaining headers all have the same class film which has been applied to each one. This is only intended as a brief introduction to html so while this will help us in the remainder of the chapter it is not a comprehensive exploration of all things html so if you are interested there are lots of resources online which cover html.

19.2 Web Scraping

Having introduced html and how it works, we now move onto how we obtain that data using Python. When we think of web scraping, we generally think of the process of getting and processing data from a website. Actually this can be broken down into two distinct processes: web crawling and web scraping.

- Web crawling is the process of getting data from one or more urls which can be obtained by traversing through a websites html. For example say a website has a front page with lots of links to other pages and you want to get all the information from all the links, you would traverse through all the links programmatically and then visit all the relevant pages and store them.
- Web scraping is the process of getting the information from the page in question so in the previous part you would have to scrape the page to get the links that you want to traverse. When you scrape the page you would programmatically get the information from the page and when you have that you would be able to store or process the data.

Given scraping plays an important part in the whole process the combination of crawling and scraping will be referred to as web scraping. In scraping a page there is no code of conduct that you sign up to. However, as soon as you try to get data from a website there are some things to note that are important.

- Check if you are allowed to get and use data from a given website: While you may think that any data on the website is fair game, it is not always the case. Check the website's terms of use as while they may not be able to stop you obtaining the data via web scraping they may have something to say that if they see the data used in any research so be careful. The issues around legality of getting

data from the web are really important and if you are in any doubt please get legal advice. The examples used in this book will involve creating our own web page locally and getting the data from it so we are covered.

- Check if there is a fair usage policy: Some websites are happy for you to scrape the data as long as you do it in an appropriate way. What do we mean by that? Well, every time you make a call to a website you are providing traffic to that site. By doing this via a computer you can send a lot of traffic to a site very quickly and this can be problematic to the site. If you are seen to do this your IP address can be blocked from the site which would mean you wouldn't be able to access the site in question. So what you need to consider is how often you plan to run code to hit certain websites and what is appropriate and necessary for you and whether the site will allow you to do. For code that does a call to a single url, it is just about how often you run it, however, if you wrote a code that crawled across lots of urls and brought back the data from them, then you would need to ensure that your code is running at an appropriate speed. To do this you would need to consider adding time delays to what you do to ensure that you are not sending excessive traffic to the site.
- Robots.txt: Again, linked to the above points, if you go to the websites url/robots.txt you will get information on what web crawlers can and can't do. If present, then it's expected that this is read to

understand what can and can't be scrapped. If there is no robots.txt, then there is no specific instruction on how the site can be crawled. However, don't assume you can scrape everything on the site.

Ultimately you need to take care when scraping a website and if they have an application programming interface (API) available then you should be using that. If you are unsure please get the appropriate advice. Before you can start crawling the site and scraping the data you need to understand the page. Python cannot just get you the data you want, instead you need to tell it how to find the data you are after. So you need to understand how the html works and where to look.

To inspect the page you have a couple of options.

Through a Web Browser

You can use the tools of the web browser to inspect what HTML refers to what elements of the page. The manner in which you can inspect is specific to the browser itself. Ultimately, it involves you selecting the element of the page and inspecting the corresponding HTML and then it showing what the html refers to that element. Different browsers have different ways of inspecting the pages they show so refer to the documentation around the specific browser that you are using.

Saving the Page and Physically Searching

You can physically save the page and then search for the name or value of certain text and in the same way determine what html refers to that element. Ultimately what you are trying to do is learn what html refers to what values in the website. This isn't an exact science due to the fact html can be written in different ways. The key is understanding the definitions of html and how they fit together and then use this to understand what you need to access in the html. With any piece of code you need to plan ahead and with parsing html you need to develop a plan for how you want to get the data from the html.

So going back to what we mentioned at the start, we described web scraping and web crawling. Web crawling is the process of actually accessing the data from a url or multiple urls. To do this we need a mechanism to do this, luckily making a web request can be in the same way we accessed an API endpoint in the Chapter so we will use the requests library. This will be demonstrated later in the chapter where we setup our own webpage.

Having a mechanism to get the data is great but we also need to process what we get back so we need a Python library that can do this. Python has many options and this book isn't intended to be a review of the best packages for processing html as the landscape is

constantly changing. Instead we will cover one specific parser namely BeautifulSoup.

BeautifulSoup

BeautifulSoup is not only an html parser but can also parse xml by using the lxml library which we covered earlier in the book. The way that BeautifulSoup works is that it takes advantage of other parsers. So, to run BeautifulSoup you would:

```
>>> from bs4 import BeautifulSoup
>>> BeautifulSoup(content_to_be_parsed, "parser_name")
```

Here content_to_be_parsed is the content from the site which could have been obtained using requests as shown before and the 'parser name' is the name of the parser to use. The four examples of these parsers are:

- html.parser: This is the default Python html parser. It has decent speed and is lenient in how it accepts html as of 3.2.2.
- lxml: This is built upon the lxml Python library which is built upon the C version. It's very fast and very lenient.
- lxml-xml or xml: Again built upon lxml so similar to above. However, this is the only xml parser you can use with BeautifulSoup. So, while we introduced how to parse XML with lxml, you could also do the same in BeautifulSoup.

- htmllib5: This is built upon the Python html5lib and is very slow, however, it's very lenient and it parses the page in the same way a web browser does to create valid html5.

Now, for the rest of this section we will concentrate on using the html.parser. So to create soup we can do as follows:

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> url = "some_url"
>>> r = requests.get(url)
>>> response_text = r.text
>>> soup = BeautifulSoup(response_text, "html.parser")
```

Now, this will have transformed the html into a format where we can access elements within it. What we will show now are the methods available to us.

```
>>> text = '<b class="boldest">This is bold</b>'
>>> soup = BeautifulSoup(text,"html.parser")
>>> soup
<b class="boldest">This is bold</b>
```

Now we can access this tag as follows:

```
>>> tag = soup.b
>>> tag
<b class="boldest">This is bold</b>
```

Now, if we had multiple b tags using soup.b would only return the first one.

```
>>> text = """<b class="boldest">This is bold</b>
<b class="boldest">This also is bold</b>"""
>>> soup = BeautifulSoup(text ,"html.parser")
>>> tag = soup.b
>>> tag
<b class="boldest">This is bold</b>
```

So, we won't get all the b tags back only the first one. The tag itself has a name which can be accessed as follows:

```
>>> tag.name
b
```

The tag also has a dictionary of attributes which are accessed as follows:

```
>>> tag.attrs
{"class": ["boldest"]}
>>> tag["class"]
["boldest"]
```

Now, let's have a look in a more complicated example. If we look at something like a table we can parse it as follows:

```
>>> text = """ <table>
...     <tr>
...         <th>Name</th>
...         <th>Year</th>
...         <th>Month</th>
...     </tr>
...     <tr>
...         <td>Avengers: Infinity War</td>
...         <td>2018</td>
...         <td>March</td>
...     </tr>
...     <tr>
...         <td>Ant Man and the Wasp</td>
...         <td>2018</td>
...         <td>August</td>
...     </tr>
... </table>"""
>>> text
' <table>\n    <tr>\n        <th>Name</th>\n        <th>Year</th> \n        <th>Month</th>\n
</tr>\n    <tr>\n        <td>Avengers: Infinity War</td>\n        <td>2018</td> \n
<td>March</td>\n    </tr>\n    <tr>\n        <td>Ant Man and the Wasp</td>\n
<td>2018</td> \n        <td>August</td>\n    </tr>\n</table>'
>>> soup = BeautifulSoup(text, "html.parser")
>>> soup
 <table>
<tr>
<th>Name</th>
<th>Year</th>
<th>Month</th>
</tr>
<tr>
<td>Avengers: Infinity War</td>
<td>2018</td>
<td>March</td>
</tr>
<tr>
<td>Ant Man and the Wasp</td>
<td>2018</td>
<td>August</td>
</tr>
</table>
```

Now, we can access elements of the table by just traversing down the tree structure of the html.

```
>>> soup.table
<table>
<tr>
<th>Name</th>
<th>Year</th>
<th>Month</th>
</tr>
<tr>
<td>Avengers: Infinity War</td>
<td>2018</td>
<td>March</td>
</tr>
<tr>
<td>Ant Man and the Wasp</td>
<td>2018</td>
<td>August</td>
</tr>
</table>
>>> soup.table.tr
<tr>
<th>Name</th>
<th>Year</th>
<th>Month</th>
</tr>
>>> soup.table.tr.th
<th>Name</th>
```

Now, in each example we get the first instance of the tag that we are looking for. Assume we want to find all tr tags within the table we can do so as follows using the findAll method.

```
>>> soup.find_all("tr")
[<tr>
<th>Name</th>
<th>Year</th>
<th>Month</th>
</tr>, <tr>
<td>Avengers: Infinity War</td>
<td>2018</td>
<td>March</td>
</tr>, <tr>
<td>Ant Man and the Wasp</td>
<td>2018</td>
<td>August</td>
</tr>]
```

So, here we get back a list of all the tr tags. Similarly, we can get back the list of td tags using the same method.

```
>>> tags = soup.find_all("td")
>>> tags
[<td>Avengers: Infinity War</td>, <td>2018</td>, <td>March</td>,
<td>Ant Man and the Wasp</td>, <td>2018</td>, <td>August</td>]
```

So with regards to getting data out if we look back and consider our original table, we can use the find all method to get all the tr tags and then loop over these and get the td tags.

```
>>> table_rows = soup.find_all("tr")
>>> table_rows
[<tr>
<th>Name</th>
<th>Year</th>
<th>Month</th>
</tr>, <tr>
<td>Avengers: Infinity War</td>
<td>2018</td>
<td>March</td>
</tr>, <tr>
<td>Ant Man and the Wasp</td>
<td>2018</td>
<td>August</td>
</tr>]
>>> headers = []
>>> content = []
>>> for tr in table_rows:
...     header_tags = tr.find_all("th")
...     if len(header_tags) > 0:
...         for ht in header_tags:
...             headers.append(ht.text)
...     else:
...         row = []
...         row_tags = tr.find_all("td")
...         for rt in row_tags:
...             row.append(rt.text)
...         content.append(row)
...
>>> headers
['Name', 'Year', 'Month']
>>> content
[['Avengers: Infinity War', '2018', 'March'],
['Ant Man and the Wasp', '2018', 'August']]
```

What we are doing here is looping over the high level tr tags to get every row and then looking for the th tags and if we find them, we know its the table header and if not we get the td tags and associate both with the appropriate list namely headers or content. The important thing to note here is that we know the structure of the data as we will have inspected the html so we build the parsing solution knowing what we will get.

So, we have introduced the find_all method in a single table. But if we had two tables and the table we wanted had a specific id we could use the find method as follows:

```
>>> text = """<table id="unique_table">
...     <tr><th>Name</th><th>Year</th><th>Month</th></tr><tr>
...     <td>Avengers: Infinity War</td><td>2018</td>" \
...         "<td>March</td></tr><tr>
...         <td>Ant Man and the Wasp</td><td>2018
...         </td><td>August</td></tr>\n</table>" \
...         "<table id="second_table">
...         <tr><th>Name</th><th>Year</th><th>Month</th></tr><tr>
...         <td>Avengers: End Game</td><td>2019</td>" \
...         "<td>April</td></tr>
...         <tr><td>Spider-man: Far from home</td>
...         <td>2019</td><td>June</td></tr>\n</table>
...         <table id="other_table"><tr><th>Name</th>"""
>>> soup = BeautifulSoup(text, "html.parser")
>>> soup
<table id="unique_table">
<tr><th>Name</th><th>Year</th><th>Month</th></tr><tr>
<td>Avengers: Infinity War</td><td>2018</td>"       "<td>March</td></tr><tr>
<td>Ant Man and the Wasp</td><td>2018
        </td><td>August</td></tr>
</table>"       "<table id="second_table">
<tr><th>Name</th><th>Year</th><th>Month</th></tr><tr>
<td>Avengers: End Game</td><td>2019</td>"       "<td>April</td></tr>
<tr><td>Spider-man: Far from home</td>
<td>2019</td><td>June</td></tr>
</table>
<table id="other_table"><tr><th>Name</th></tr></table>
>>> table = soup.find("table", id="second_table")
>>> table
<table id="second_table">
<tr><th>Name</th><th>Year</th><th>Month</th></tr><tr>
<td>Avengers: End Game</td><td>2019</td>"       "<td>April</td></tr>
<tr><td>Spider-man: Far from home</td>
<td>2019</td><td>June</td></tr>
</table>
```

And we can get the data from this table in a similar way as before but again we can take advantage of the find method to find the text in the specific element.

```
>>> table_rows = table.find_all("tr")
>>> table_rows
[<tr><th>Name</th><th>Year</th><th>Month</th></tr>, <tr>
<td>Avengers: End Game</td><td>2019</td>"            "
<td>April</td></tr>, <tr><td>Spider-man: Far from home</td>
<td>2019</td><td>June</td></tr>]
>>> headers = []
>>> content = []
>>> for tr in table_rows:
...     header_tags = tr.find_all("th")
...     if len(header_tags) > 0:
...         for ht in header_tags:
...             headers.append(ht.text)
...     else:
...         row = []
...         row_tags = tr.find_all("td")
...         for rt in row_tags:
...             row.append(rt.text)
...         content.append(row)
...
>>> headers
['Name', 'Year', 'Month']
>>> content
[['Avengers: End Game', '2019', 'April'],
 ['Spider-man: Far from home', '2019', 'June']]
```

This shows that when we have multiple tables we can obtain the information from a specific one, this is really dependent on the table having an id attribute which made the process much easier.

So, we now know how to process html, and the next stage is to grab data from a website and then parse that data using Python. To do this we will build our own website locally which we will grab data from and

parse the results. Given we have covered the libraries to get and process the data how do we go about creating a website?

As in the Chapter , we will use the package Flask to create a simple website that we can run locally and then scrape the data from. Let's just get started and write a hello world example to show how it will work. Here, we will create a file called my_flask_website.py and put the following code in it:

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
```
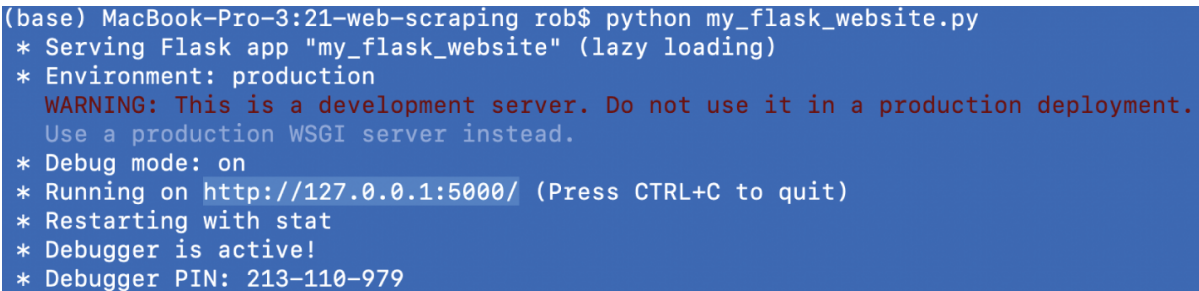
Now, if you think back to the Chapter what we have here is a reduced down version of what we used to create our API. We import Flask from the flask package and then create ourselves an app. Unlike with the API where we created a class we simply define a hello_world function which returns the string Hello World.

```python
@app.route('/')
def hello_world():
    return 'Hello World'
```

Again we use the syntax:

```
if __name__ == '__main__':
    app.run(debug=True)
```

to run our application. As with the API we built if we open a terminal or command prompt and move to the location of the file and run the code using Python my_flask_website.py then we will get a web page as shown in Figure 19.1.


```
(base) MacBook-Pro-3:21-web-scraping rob$ python my_flask_website.py
 * Serving Flask app "my_flask_website" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 213-110-979
```

Figure 19.1 Display of website from terminal.

Figure 19.2 Display of website from browser.

If we then go to the address on a web browser we will see a web page as shown in Figure 19.2.

Now, one part of the code that we didn't cover was the use of @app.route this is an example of a decorator. In this is an example its purpose is to bind a location to a function. So when we apply the following:

```python
@app.route('/')
def hello_world():
    return 'Hello World'
```

What we are doing is mapping any call of http://127.0.0.1:5000/ to the function hello_world so when that url is called the hello_world function is executed and the results displayed. This is a specific use of a decorator in general decorators are functions that can take functions as an argument. The best way to explain is by demonstration so we could decorate the hello_world function with a decorator that make a string all lowercase.

```python
def make_uppercase(function):
    def wrapper():
        func = function()
        lowecase = func.lower()
        return lowercase

    return wrapper
```

What this function does is take in another function as an argument and then run the wrapper function in the return statement. The function wrapper then runs the function that is passed in to make_upper-

case and take the output from it and make it lowercase and return that value. Let's demonstrate with one example (Figure 19.3).
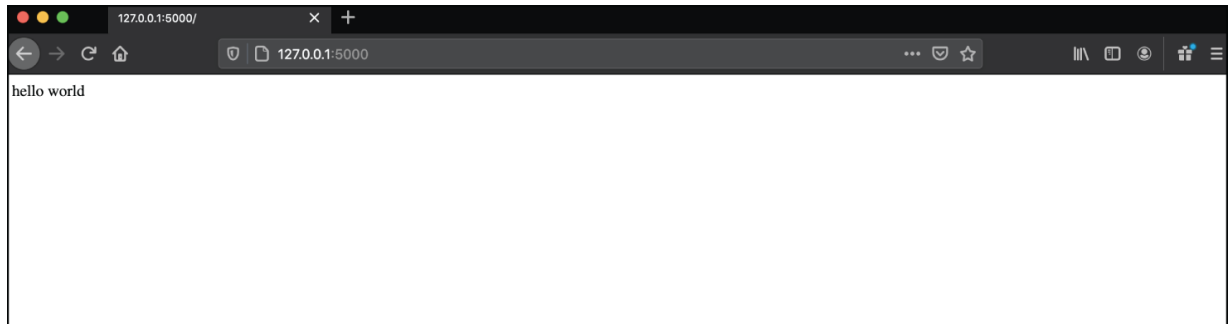
Figure 19.3 Display of website from with lowercase decorator applied.

```python
from flask import Flask
app = Flask(__name__)

def make_lowercase(function):
    def wrapper():
        func = function()
        lowercase = func.lower()
        return lowercase

    return wrapper

@app.route('/')
@make_lowercase
def hello_world():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
```

We have our website up and running, so let's programmatically get the data from it. To do this we can use requests in the same way we did in the API chapter to obtain a get request.

```
>>> import requests
>>>
>>> r = requests.get('http://127.0.0.1:5000/')
>>> r.text
'Hello World'
```

Notice that this time we looked at the text attribute as opposed to the json method and that is because the content of our website is not json. This is all well and good but its not much of a challenge to process the data mainly because its not in html format (Figure 19.4). We can change that pretty easily by just modifying the code in our flask application so let's change the output hello world as follows:
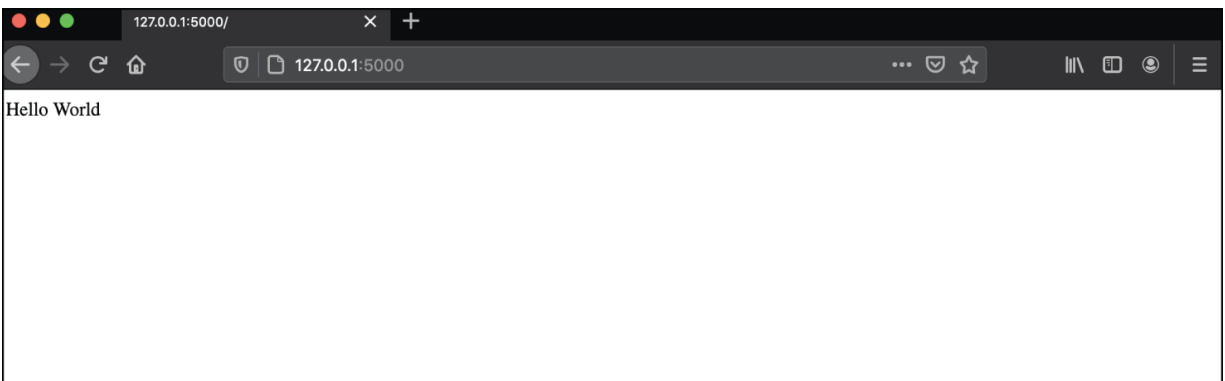
Figure 19.4 Display of website from browser with html.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h>Hello World</h>'

if __name__ == '__main__':
    app.run()
```

With our flask application running we can see the following on the web browser:

It looks pretty similar to what we saw before, so what has changed? If we run the code to get the data from the webpage we get the following:

```
>>> import requests
>>>
>>> r = requests.get('http://127.0.0.1:5000/')
>>> r.text
'<h>Hello World</h>'
```

You can now see that instead of just the text representation, we have some html around that with the h tags.

Let's modify the code once more and change the h tags to h1 tags, so our flask application now looks like so

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h1>Hello World</h1>'

if __name__ == '__main__':
    app.run()
```

With our flask application running we can see the following on the web browser:
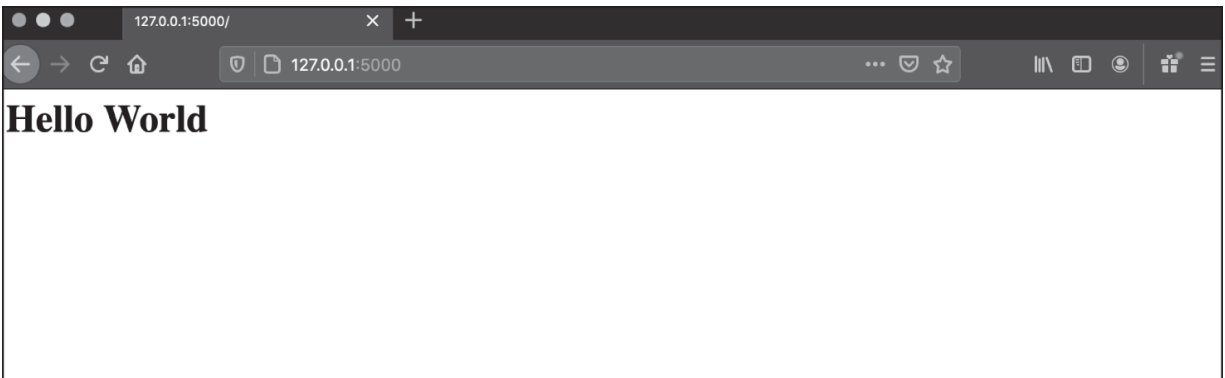
Figure 19.5 Display of website from browser with h1 hello world.

Again running the same requests code on this website bring back the h1 tags (Figure 19.5).

```
>>> import requests
>>>
>>> r = requests.get('http://127.0.0.1:5000/')
>>> r.text
'<h1>Hello World</h1>'
```

So now we have our website running lets add something a little harder to parse and create a table that we can look to programmatically obtain. To do this we will add a new route to the flask application and look to add a html table. To do this we will make use of some existing data from the seaborn package namely the tips data.

```
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")
>>> tips.head().to_html()
'<table border="1" class="dataframe">\n  <thead>\n
    <tr style="text-align: right;">\n
<th></th>\n      <th>total_bill</th>\n       <th>tip</th>\n        <th>sex</th>\n
<th>smoker</th>\n      <th>day</th>\n       <th>time</th>\n        <th>size</th>\n
</tr>\n  </thead>\n  <tbody>\n     <tr>\n      <th>0</th>\n        <td>16.99</td>\n
<td>1.01</td>\n       <td>Female</td>\n      <td>No</td>\n        <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n     </tr>\n     <tr>\n       <th>1</th>\n
<td>10.34</td>\n      <td>1.66</td>\n       <td>Male</td>\n        <td>No</td>\n
<td>Sun</td>\n       <td>Dinner</td>\n      <td>3</td>\n     </tr>\n     <tr>\n
<th>2</th>\n       <td>21.01</td>\n       <td>3.50</td>\n        <td>Male</td>\n
<td>No</td>\n       <td>Sun</td>\n       <td>Dinner</td>\n      <td>3</td>\n
</tr>\n     <tr>\n       <th>3</th>\n       <td>23.68</td>\n       <td>3.31</td>\n
<td>Male</td>\n       <td>No</td>\n       <td>Sun</td>\n       <td>Dinner</td>\n
<td>2</td>\n     </tr>\n     <tr>\n       <th>4</th>\n        <td>24.59</td>\n
<td>3.61</td>\n       <td>Female</td>\n      <td>No</td>\n        <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n     </tr>\n  </tbody>\n</table>'
```

Now, we have imported the tips dataset and we can make use of the to_html method from pandas, which takes the DataFrame and give us back html that we could put on our website. If we look back to our previous table example, we might want to add an id to the table to allow us to access the table and we can do that using to_html by passing in the table_id argument and setting it to the name that we want our table to have. So, let's apply it by setting the name to be tips.

```
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")
>>> tips.head().to_html(table_id='tips')
'<table border="1" class="dataframe" id="tips">\n  <thead>\n
<tr style="text-align: right;">\n      <th></th>\n
<th>total_bill</th>\n      <th>tip</th>\n       <th>sex</th>\n
<th>smoker</th>\n      <th>day</th>\n      <th>time</th>\n
<th>size</th>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n
<th>0</th>\n      <td>16.99</td>\n      <td>1.01</td>\n
<td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n
<th>1</th>\n      <td>10.34</td>\n      <td>1.66</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n
<th>2</th>\n      <td>21.01</td>\n      <td>3.50</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n
<th>3</th>\n      <td>23.68</td>\n      <td>3.31</td>\n
<td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n
<tr>\n      <th>4</th>\n      <td>24.59</td>\n
<td>3.61</td>\n      <td>Female</td>\n       <td>No</td>\n
<td>Sun</td>\n      <td>Dinner</td>\n      <td>4</td>\n
</tr>\n  </tbody>\n</table>'
```

So, we can now see we have added the id attribute with the name tips. Our next step is to add this to our website and we can alter the code as follows to do so:
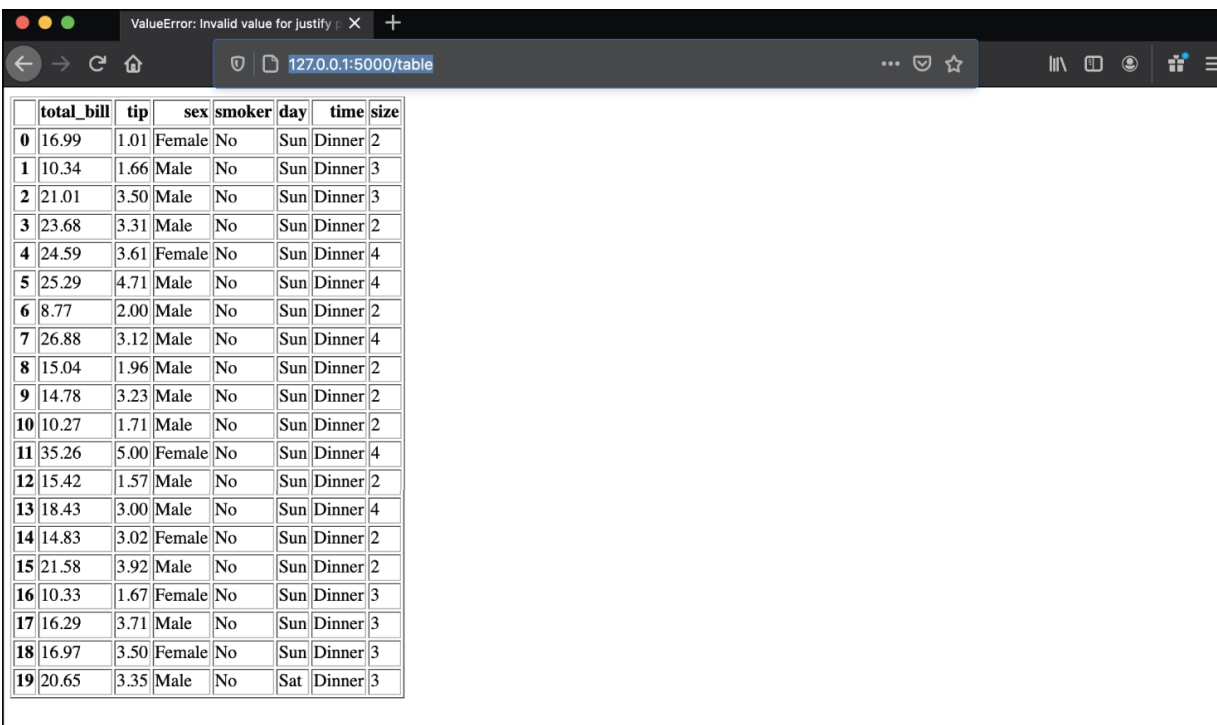
```python
from flask import Flask
import seaborn as sns
app = Flask(__name__)

tips = sns.load_dataset("tips")

@app.route('/')
def hello_world():
    return '<h1>Hello World</h1>'

@app.route('/table')
def table_view():
    return tips.head(20).to_html(table_id='tips')

if __name__ == '__main__':
    app.run(debug=True)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
5	25.29	4.71	Male	No	Sun	Dinner	4
6	8.77	2.00	Male	No	Sun	Dinner	2
7	26.88	3.12	Male	No	Sun	Dinner	4
8	15.04	1.96	Male	No	Sun	Dinner	2
9	14.78	3.23	Male	No	Sun	Dinner	2
10	10.27	1.71	Male	No	Sun	Dinner	2
11	35.26	5.00	Female	No	Sun	Dinner	4
12	15.42	1.57	Male	No	Sun	Dinner	2
13	18.43	3.00	Male	No	Sun	Dinner	4
14	14.83	3.02	Female	No	Sun	Dinner	2
15	21.58	3.92	Male	No	Sun	Dinner	2
16	10.33	1.67	Female	No	Sun	Dinner	3
17	16.29	3.71	Male	No	Sun	Dinner	3
18	16.97	3.50	Female	No	Sun	Dinner	3
19	20.65	3.35	Male	No	Sat	Dinner	3

Figure 19.6 Display of website from browser showing a table.

Now, the difference here is that we have added the imports for seaborn and then imported the tips dataset. To display this we then create another function called table_view and in it return 20 rows of the DataFrame and convert it to html with the id of tips. A decorator then defines the route of this to be /table which means when we go to the http://127.0.0.1:5000/table we will see the result of this function. Let's do that and go to the url and see what is shown (Figure 19.6).

Now, we can see the table but it doesn't look great, we can customise this using some of the options that come with pandas. First, we will remove the index from the table as you normally wouldn't see this on a website. Next, we will centre the table headings and we will also make the borders more prominent. So our flask application is now modified to this.
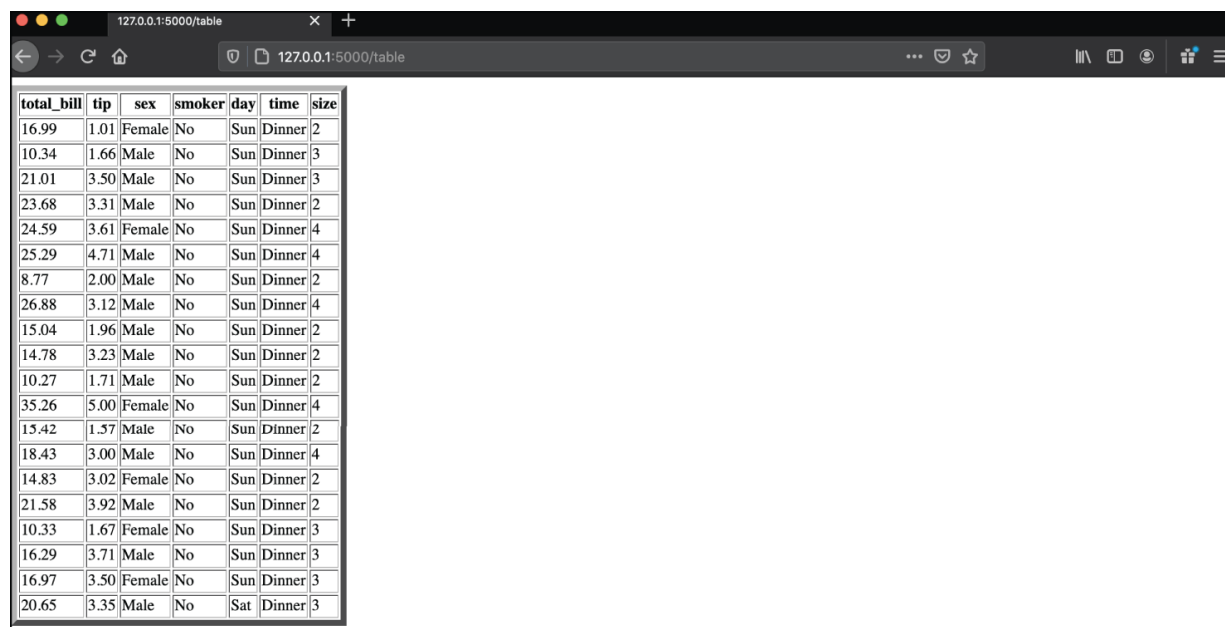
```python
from flask import Flask
import seaborn as sns
app = Flask(__name__)

tips = sns.load_dataset("tips")

@app.route('/')
def hello_world():
    return '<h1>Hello World</h1>'

@app.route('/table')
def table_view():
    return tips.head(20).to_html(table_id='tips', border=6,
    index=False, justify='center')

if __name__ == '__main__':
    app.run(debug=True)
```

The browser view now looks like the one shown in Figure 19.7.

Figure 19.7 Display of website from browser showing a table with customisation.

If we use some of what we covered earlier, we can add a title and some information about the website in a paragraph. To do that we can use the h1 and p tags to create a header and paragraph, respectively, and to show that everything belongs together let's put this all within a div tag so it resembles what you might find on a production web page. The flask application now looks like the following:

```python
from flask import Flask
import seaborn as sns
app = Flask(__name__)

tips = sns.load_dataset("tips")

@app.route('/')
def hello_world():
    return '<h1>Hello World</h1>'

@app.route('/table')
def table_view():

    html = '<div><h1>Table of tips data</h1>' + \
            '<p>This table contains data from the seaborn tips dataset</p>' + \
            tips.head(20).to_html(table_id='tips', border=6,
            index=False, justify='center')
            + '</div>'
    return html

if __name__ == '__main__':
    app.run(debug=True)
```

Our webpage now looks as the one shown in Figure 19.8.
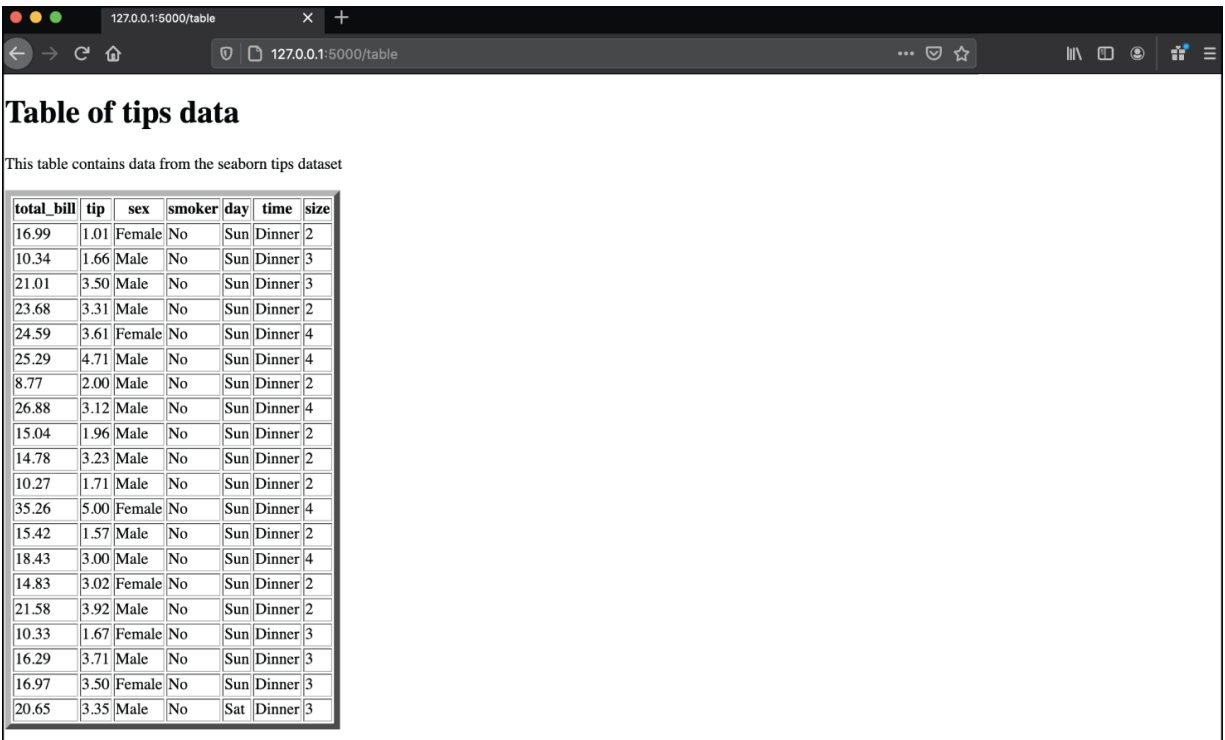
Figure 19.8 Display of website from browser showing a table with header and paragraph.

Ok so now we have a website we want to scrape it so let's use requests to get the html that we will look to obtain.

```
>>> import requests
>>>
>>> r = requests.get('http://127.0.0.1:5000/table')
>>> r.text
'<div><h1>Table of tips data</h1><p>This table contains data from the seaborn
tips dataset</p><table border="6" class="dataframe" id="tips">\n  <thead>\n
 <tr style="text-align: center;">\n      <th>total_bill</th>\n      <th>tip</th>\n
<th>sex</th>\n      <th>smoker</th>\n      <th>day</th>\n      <th>time</th>\n
<th>size</th>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n      <td>16.99</td>\n
<td>1.01</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>10.34</td>\n
<td>1.66</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n      <td>21.01</td>\n
<td>3.50</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n      <td>23.68</td>\n
<td>3.31</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>24.59</td>\n
<td>3.61</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n    <tr>\n      <td>25.29</td>\n
<td>4.71</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n    <tr>\n      <td>8.77</td>\n
<td>2.00</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>26.88</td>\n
<td>3.12</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n    <tr>\n      <td>15.04</td>\n
<td>1.96</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>14.78</td>\n
<td>3.23</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>10.27</td>\n
<td>1.71</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>35.26</td>\n
<td>5.00</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n    <tr>\n      <td>15.42</td>\n
<td>1.57</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>18.43</td>\n
<td>3.00</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>4</td>\n    </tr>\n    <tr>\n      <td>14.83</td>\n
<td>3.02</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>21.58</td>\n
<td>3.92</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>2</td>\n    </tr>\n    <tr>\n      <td>10.33</td>\n
<td>1.67</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n      <td>16.29</td>\n
<td>3.71</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n      <td>16.97</td>\n
<td>3.50</td>\n      <td>Female</td>\n      <td>No</td>\n      <td>Sun</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n    <tr>\n      <td>20.65</td>\n
<td>3.35</td>\n      <td>Male</td>\n      <td>No</td>\n      <td>Sat</td>\n
<td>Dinner</td>\n      <td>3</td>\n    </tr>\n  </tbody>\n</table></div>'
```

So, we can see that it was relatively straight forward to get the data but unlike with our static table example before the data from the web-page is more than just table data. The next step is to pass this into BeautifulSoup to parse the html.

```
>>> soup = BeautifulSoup(r.text, "html.parser")
>>> soup.find('table',id='tips')
<table border="6" class="dataframe" id="tips">
<thead>
<tr style="text-align: center;">
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>
<tr>
<td>10.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>16.97</td>
<td>3.50</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>
<tr>
<td>20.65</td>
<td>3.35</td>
<td>Male</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
<td>3</td>
</tr>
</tbody>
</table>
```

$$$$$$$$$$$$$$

In using the table id we can go directly to the table within the html and we then have access to all the rows within it just like before. Note that we have only shown a subset of this data as we have 20 rows. Now, if we want to parse the data from the html we can use something like we used on the dummy data.

```
>>> table = soup.find('table',id='tips')
>>> table_rows = table.find_all("tr")
>>> table_rows[0:3]
[<tr style="text-align: center;">
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
</tr>, <tr>
<td>16.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
</tr>, <tr>
<td>10.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
</tr>]
>>> headers = []
>>> content = []
>>> for tr in table_rows:
...     header_tags = tr.find_all("th")
...     if len(header_tags) > 0:
...         for ht in header_tags:
...             headers.append(ht.text)
...     else:
...         row = []
...         row_tags = tr.find_all("td")
...         for rt in row_tags:
...             row.append(rt.text)
...         content.append(row)
...
>>> headers
['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size']
>>> content
[['16.99', '1.01', 'Female', 'No', 'Sun', 'Dinner', '2'],
['10.34', '1.66', 'Male', 'No', 'Sun', 'Dinner', '3'],
['21.01', '3.50', 'Male', 'No', 'Sun', 'Dinner', '3'],
['23.68', '3.31', 'Male', 'No', 'Sun', 'Dinner', '2'],
['24.59', '3.61', 'Female', 'No', 'Sun', 'Dinner', '4'],
['25.29', '4.71', 'Male', 'No', 'Sun', 'Dinner', '4'],
['8.77', '2.00', 'Male', 'No', 'Sun', 'Dinner', '2'],
['26.88', '3.12', 'Male', 'No', 'Sun', 'Dinner', '4'],
['15.04', '1.96', 'Male', 'No', 'Sun', 'Dinner', '2'],
['14.78', '3.23', 'Male', 'No', 'Sun', 'Dinner', '2'],
['10.27', '1.71', 'Male', 'No', 'Sun', 'Dinner', '2'],
['35.26', '5.00', 'Female', 'No', 'Sun', 'Dinner', '4'],
['15.42', '1.57', 'Male', 'No', 'Sun', 'Dinner', '2'],
['18.43', '3.00', 'Male', 'No', 'Sun', 'Dinner', '4'],
['14.83', '3.02', 'Female', 'No', 'Sun', 'Dinner', '2'],
['21.58', '3.92', 'Male', 'No', 'Sun', 'Dinner', '2'],
['10.33', '1.67', 'Female', 'No', 'Sun', 'Dinner', '3'],
['16.29', '3.71', 'Male', 'No', 'Sun', 'Dinner', '3'],
['16.97', '3.50', 'Female', 'No', 'Sun', 'Dinner', '3'],
['20.65', '3.35', 'Male', 'No', 'Sat', 'Dinner', '3']]
```

As we can see we have now pulled the data from the html and got it into two separate lists bit to go a step further we can put it back into a DataFrame pretty simply by using what we have covered earlier in the book.

```
>>> data = pd.DataFrame(content)
>>> data
        0      1       2    3    4       5  6
0   16.99   1.01  Female   No  Sun  Dinner  2
1   10.34   1.66    Male   No  Sun  Dinner  3
2   21.01   3.50    Male   No  Sun  Dinner  3
3   23.68   3.31    Male   No  Sun  Dinner  2
4   24.59   3.61  Female   No  Sun  Dinner  4
5   25.29   4.71    Male   No  Sun  Dinner  4
6    8.77   2.00    Male   No  Sun  Dinner  2
7   26.88   3.12    Male   No  Sun  Dinner  4
8   15.04   1.96    Male   No  Sun  Dinner  2
9   14.78   3.23    Male   No  Sun  Dinner  2
10  10.27   1.71    Male   No  Sun  Dinner  2
11  35.26   5.00  Female   No  Sun  Dinner  4
12  15.42   1.57    Male   No  Sun  Dinner  2
13  18.43   3.00    Male   No  Sun  Dinner  4
14  14.83   3.02  Female   No  Sun  Dinner  2
15  21.58   3.92    Male   No  Sun  Dinner  2
16  10.33   1.67  Female   No  Sun  Dinner  3
17  16.29   3.71    Male   No  Sun  Dinner  3
18  16.97   3.50  Female   No  Sun  Dinner  3
19  20.65   3.35    Male   No  Sat  Dinner  3
>>> data.columns = headers
>>> data
    total_bill   tip     sex smoker  day    time size
0        16.99  1.01  Female     No  Sun  Dinner    2
1        10.34  1.66    Male     No  Sun  Dinner    3
2        21.01  3.50    Male     No  Sun  Dinner    3
3        23.68  3.31    Male     No  Sun  Dinner    2
4        24.59  3.61  Female     No  Sun  Dinner    4
5        25.29  4.71    Male     No  Sun  Dinner    4
6         8.77  2.00    Male     No  Sun  Dinner    2
7        26.88  3.12    Male     No  Sun  Dinner    4
8        15.04  1.96    Male     No  Sun  Dinner    2
9        14.78  3.23    Male     No  Sun  Dinner    2
10       10.27  1.71    Male     No  Sun  Dinner    2
11       35.26  5.00  Female     No  Sun  Dinner    4
12       15.42  1.57    Male     No  Sun  Dinner    2
13       18.43  3.00    Male     No  Sun  Dinner    4
14       14.83  3.02  Female     No  Sun  Dinner    2
15       21.58  3.92    Male     No  Sun  Dinner    2
16       10.33  1.67  Female     No  Sun  Dinner    3
17       16.29  3.71    Male     No  Sun  Dinner    3
18       16.97  3.50  Female     No  Sun  Dinner    3
19       20.65  3.35    Male     No  Sat  Dinner    3
```

Now, we have gone full circle and used a DataFrame to populate a table within our website and then scraped that data and converted it back into a DataFrame.

This chapter has covered a lot of content from introducing html to parsing it out to building our own website and scraping from there. The examples have been focussed on table data but they can be applied to any data we find within html. When it comes to web scraping Python is a powerful and popular choice to interact and obtain data from the web.

20
Conclusion

This book has given you an introduction into Python covering all the basics right up to some complex examples. However, there is much more that could have been covered and much more for you the reader to learn. Python has many more packages and changes are always being made, so its important to keep up to date with the trends within the language. From a development point of view, we have kept things simple working in the shell or writing basic scripts, however Python can be so much more than an exploratory language. Python can be used in a production environment and given its adoption by many big tech firms it works very well with a lot of cloud computing solutions and is an excellent choice for everything from web applications to machine learning. This book is a gateway to give you the tools to follow your own Python journey and with a community as big as Pythons there is always something to learn as well as new things to be aware of. Good luck!

Index

e

f

g

m

q

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.