A Masters Course in Big Data
(Representing and Manipulating Data Chapter)

Dr. Kevin Swingler

September 25, 2019

Contents

Chapter 1

Representing and Manipulating Data

What are *Data*? In the broadest sense, they represent observations, ideas, recordings, or measurements from something in the real world. Data might be stored on paper, but in this book we are concerned mainly with data stored in a computer. Data might represent measurements like temperature or height or scores in a test; photos taken on holiday or brain scans; sound recordings or vibration measurements; letters, emails or collections of literature; speeches, paintings or poems. They might represent a patterns of mouse clicks on a website, an insurance claim, or a history of credit card usage.

Data are about things in the real world and those things usually have some kind of structure. They are organised in a certain way, which needs to be reflected in their digital representation. Once the data have been collected and stored, they can be used to tell us something about the things in the real world that they measure. This might involve visualising the data, or performing statistical inference, or building models of the process that generated the data. This all needs the data to be represented in a consistent way and manipulated for analysis. This chapter describes how to do those things in a programming language called Python.

To try and bring some clarity, I'll use a consistent set of words (as best as I can!) to refer to the different aspects of data and the world they record. Things in the real world that are measured or recorded are called **entities** and entities can be organised into **entity types**. People, books, and cars are all entity types and Brad Pitt, Hamlet and the MX-5 are all entities. Some entities have **attributes**, which are things we can measure about them. People have attributes like height and weight and an entity type is defined to varying extents by what attributes the member entities can have. Single entities are defined by giving a **value** to each attribute. The value associated with the height attribute for the person entity that is me, for example, is 180.

Much of the work of a data scientist involves getting data into a suitable shape before the interesting work starts. Sometimes, manipulating the data *is* the interesting work! Data need to be extracted from their source, filtered and cleaned, explored and visualised, prepared and transmitted. Even with high speed internet connections, moving and accessing large quantities of data can be problematic. This makes decisions about which data are moved and in what format very important. If data are to be shared among users or applications, a common format is needed—one everybody can read and process. There are plenty of options for representing data and it is important to choose the right one for the job.

In the past few years, one programming language has emerged as a clear favourite for data processing and machine learning. Python is easy to learn, fast to program with, and has a wealth of powerful capabilities from spreadsheet like data tables to computer vision and natural language processing. It might not always produce the fastest running code but in situations where your code is written once and perhaps only run a few times to get a specific job done, the speed at which you can build something more than compensates.

This chapter will teach you how to access, transport, manipulate, represent and store data from a variety of sources using the most common data formats and the Python programming language. You'll learn how to access data on an incredible range of subjects, extract new findings from them, visualise them in new ways, and share your results with the world.

1.1 A Quick Introduction to Data Structures

Data come in many forms: measurements of variables like height or temperature, images like photographs or brain scans, text from tweets to text books, and sound and video recordings. You will learn more about representing data in memory for processing with a computer program later in this chapter, but a little context is needed before we start. This section introduces a number of the most common data structures used in computer programming as they are central to the rest of the chapter.

1.1.1 Variables

Variables are used to store data in memory and are referenced by a name. Variables generally refer to attributes that entities have, like height or temperature. They contain a **value**, which can vary as the program runs. For example, the variable named `age` could be assigned the value 50, then that value could be changed to 51. The name stays the same, but the value changes. The variable name does not mean anything to the computer, but should be chosen to help a human reader understand its role in the code. The value could be something simple like a number or a word, or a more complex data structure.

In a program, you cannot have two variables with the same name in the same place. If you want to store more than one value under the same name at the same time, you need a data structure like a list. Let's say we measure the age of all the students in a class, giving us 100 values for the variable `age`. We can store these in a **list** (also known as an **array**) called `age`. The list itself is referenced by its name: `age`, and individual entries in it are accessed by a number, known as an index, that specifies their location in the list. As entries are accessed by location, the order in which entries are stored is important. For this reason, lists are known as *ordered* structures.

A list is generally used to store several measurements of the same variable from different entities, which means that it can contain repeated values (if two students are the same height, for example). If you want to represent a set of distinct values, then rather than a list, you use a **set**. Unlike lists, sets do not always have a defined order and they cannot contain repeated values.

Having measured the heights of all the students in our class, we might realise that we want to store the student's name along with their height. A simple way to do this is to use what is known as a **map** (or in Python, a **dictionary**). A map is a set of named identifiers, known as **keys**, each with an associated value. The keys must be unique within a single map (so we might need to use student ID rather than name if our class contains students who share a name). A map is unordered, like a set, and values are accessed by their identifier, not their location. A map is a far more elegant way of storing an arbitrary number of identifiers and values than using a simple variable for each one, as you can process all the members of a map together.

Finally (for now) imagine you have measured several things about each student - their name, height, age, and nationality. To store the data about a single student, you need to define a structure that contains each of those variables. Such structures are complex data types that are often known as **classes**. A class represents what we have been calling an entity type. Once you have defined a class (*student*, for example) you can instantiate variables that belong to that class (student number 548223, for example). These variables are called **objects** and an object represents what we have been calling entities.

You can mix lists, sets, maps, and objects to construct a variety of data structures. For example, you could have a list of objects, or a map in which each value is a list. If we want to store a classroom's worth of students, for example, then we need to use a list of student objects (or a map in which student ID is the key and the value is a *student* object).

1.1.2 Variable Types

It is very helpful to know what type of data a variable should hold. Numbers and words should be processed very differently, for example. We will look in detail at data types in Python in section 1.3.4 but for now, we will define the few that we refer to in the following sections. A numeric variable can only represent numbers. Most programming languages make a distinction between integers (whole numbers) and floating point numbers (those with a decimal place). Boolean variables can take one of only two values: True or False. String variables represent text as a string of characters. Some programming languages also allow you to explicitly state that a variable currently has no associated value, using `null` (Javascript) or `None` (Python).

1.2 Representing Data

When we store data in computer memory or in a file or a database, we need to choose an appropriate representation format. The format should record what the data measure, how different parts of the data are related, and which parts of the data belong together. The representation will specify something about the type of data being stored (are they numbers, or text, for example). A data format often also comes with a set of rules for interpretation or conversion to a human readable form. For example, images may be represented as arrays of colour values and need an associated algorithm to present the data as an image. Choosing the right representation format for data is very important. It has an impact on how easy it is to store, search and analyse the data. It is important to understand the different data formats so that you can share data in a way that others can read and so that others can share their data with you.

1.2.1 Data Formats

Structured Data

Some data take the form of a set of variables that have been repeatedly measured (or sampled) to produce a set of observations. If the variables are the same in each observation, then the data are referred to as being **structured**. In structured data, each variable is uniquely identifiable—usually with a name, and takes values from a fixed domain (numbers or dates, for example). In structured data, the variables usually take a single value for each measurement, rather than a list or a more complex structure. For example, the variable `age` could take a numeric values, and the variable `email` could take a string value. In strictly structured data, it would not be permissible for the `email` variable to store two different email addresses for the same person, for example. Structured data are often represented in a tabular format.

Semi-Structured Data

If different observations can measure different variables, or the variables themselves have a more complex structure such as a list, then the data are known as **semi-structured**. The data are still organised into a set of observations, but the structure of each observation is not fixed. In semi-structured data, the values that a variables can take can be more complex. They might be lists, for example, or other data structures. The variables in semi-structured data still have named identifiers, but their structure is more flexible. For example, a variable called `phone_numbers` could take a list of numbers rather than just one. Some observations might have an `email` variable and others might have one called `twitter` instead. Such data do not sit well in a table as each row might need different columns.

Unstructured Data

Some data do not consist of values for named variables at all. Such data are often called **unstructured** because there are no identifiable variables with specific roles. Data describing images, text and sound are all known as unstructured because a specific value is not generally associated with a named variable. Take an image, for example. It is made up of values that represent the colour of each pixel in the image. Each pixel does not have a fixed meaning. The pixel at coordinates (100, 500) in one image might be part of a picture of a dog, and the same pixel, in the same location in another image might be part of a tree. Similarly, free text (like tweets, product reviews or books) is unstructured because each word only really makes sense in the context of the words around it, rather than its meaning being completely defined by its location in a structure.

1.2.2 Tabular Data

A simple way to represent structured data is in a table where the columns refer to variables and the rows contain values from a single entity. Tabular data is called *structured* because the variables are fixed by the column names and each cell can only take one value. Tabular data structures are widely used in spreadsheets and relational databases. Section 1.3.13 describes the use of data frames in Python, which are designed to process tabular data.

1.2.3 JSON

JavaScript Object Notation (JSON) is a popular format for representing unstructured data. Unlike tabular data, where the variable being measured is represented by the column in which its values are stored, variables in JSON format are named every time a value is recorded for them. This allows different entries to record different variables. What's more, the values in a JSON object can be values, lists, or even JSON objects themselves. This make the format very flexible.

JSON notation represents entities in an object like form in which each variable has a name (known as its **key**) and an associated value. The keys must be strings, but the flexibility of JSON comes from the fact that the values can be Boolean values, numbers, strings, arrays, JSON objects or arrays of JSON objects. JSON is reasonably human readable and takes the following form:

```
{ "name1" : number_value , "name2" : "string value" }
```

A JSON object is enclosed in curly braces: { } and the `"key": value` pairs are defined with the key in double quotes. A value is written in double quotes if it is a string and without quotes if it is a number or one of the values: `true`, `false` or `null`. The key:value pairs are separated by commas.

An array is specified using square brackets around a comma separated list of values like this: {"Pets":["Cat", "Dog", "Hamster"]}. Here is a more complete example, showing many of the ways you can mix values, objects and arrays:

```
{"Name":"Kevin", "Emails":["kms@cs.stir.ac.uk", "kevin.swingler@stir.ac.uk"],
"Teaches":[{"Code" : "ITNPBD2", "Name":"Python"},
                {"Code":"ITNPBD3", "Name":"Databases"}]}
```

I have used an array of strings for my email addresses as I have more than one. I have used an array of JSON objects to list the courses I teach - each one is a separate object and have key names in common (`"Code"` and `"Name"`).

JSON Schemata

JSON objects do not need to have a defined structure before they are used. You simply create one as you need it. In practice, however, if you want to represent a large number of objects, all of the same type (all the lecturers in the university, for example) then you might want to ensure that they all have the same structure. You might want to define that structure, share it with other people who are generating data, and enforce the validity of JSON objects in your code. You can do this with JSON schemata.

A JSON schema specifies a set of rules that define the valid content for a JSON object. You can specify things like the names of fields that must be present, data types, the length of arrays, and maximum and minimum values for numeric data. JSON schemata are defined in JSON format (of course). Here is a partial example for JSON objects that describe a university lecturer. For a full description of JSON schemata, see the website at `www.json-schema.org`.

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Lecturer",
   "description": "A description of a university lecturer",
   "type": "object",

   "properties": {

      "name": {
         "description": "Lecturer name",
         "type": "string"
      },

      "Salary": {
         "type": "number",
         "minimum": 0
      }
```

```
        "Emails": {
          "type": "array",
          "contains":
              {
               "type": "string"
              }
        }
    },
    "required": ["name", "emails"]
}
```

1.2.4 XML

XML stands for eXtensible Markup Language. A markup language is a human readable method of tagging data in a document. Some markup languages (like HTML) are designed so that the tags in the data dictate how it is presented, for example on screen in a browser. In an XML document, tags define the structure and meaning of the data. XML is used primarily for storing and transporting data in a form that is both human and machine readable. XML offers a method of encoding data with a complex structure in a way that is independent of software or type of computer. It represents both the data and some rules about how the data are structured, both using the same format. As it is widely used, this makes XML a powerful and flexible choice for representing data.

XML represents data in a tree-like structure and defines elements, the attributes that elements possess, and data about the elements. Here is an example. We will start with some semi-structured data—a menu from a cafe—and show how it would be marked up in XML. Here is the (rather limited) menu:

Menu

Food

Breakfast
Light Breakfast 2.99: Toast, Butter, Jam
English Breakfast 5.99: Egg, Bacon, Sausage, Toast
Lunch
Ploughman's Lunch 6.99: Bread, Cheese, Ham, Salad, Pickle
Programmer's Lunch 6.99: Quiche, Salad, Pork Pie

Drinks

Hot
Coffee 2.99
Tea 1.99
Cold
Milk 1.99
Cola 1.99

This menu can be represented in a tree structure, shown in figure 1.1. The root node is called Menu and the main branches are Food and Drink, with branches in Food for Breakfast and Lunch and in Drink for Hot and Cold. The leaf nodes are the actual data—in this case the names and descriptions of the food and the prices.

Having identified the tree structure, the menu can be encoded in XML by identifying each element to be represented, its position in the tree and the attributes and data associated with it. Elements are items to be stored (they correspond to what we have been calling *entities* in the text so far). In our example they are items from the menu. Elements have data associated with them and those data can also have attributes that describe something about the data.

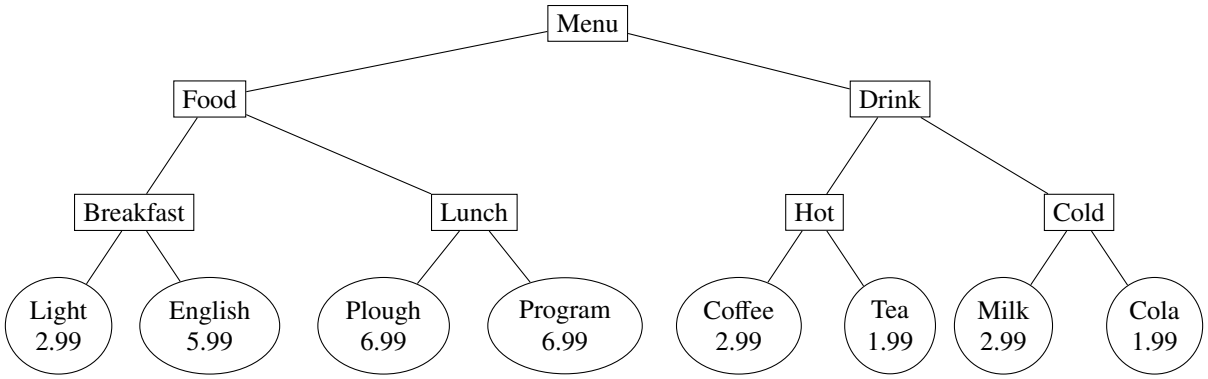

Figure 1.1: Our example menu represented as a tree with section headings in rectangles and items (the data) in ellipses.

The syntax for specifying XML involves a set of **tags**, which describe the data they enclose. Tags have a name and an optional list of attributes that are specified with an equals sign. Here is an example:

```
<drink price="2.99" temperature="Hot">Coffee</drink>
```

The tag is named `drink` and has an attribute called `price`, which has a value of 2.99. The word `Coffee` is the data in this example. Note the following about the syntax:

- The data element begins with a tag enclosed in angled brackets
- The data element ends with the same named tag in angled brackets, but with a / symbol in front of the name.
- The attributes are in a space separated list (not commas!) of `name` = `"value"` pairs.
- The attribute values are enclosed in quotes regardless of their type
- Data are not enclosed in quotes.

You might see other ways you could represent that same data with XML, removing the attributes and using data elements instead:

```
<drink>
<price>2.99</price>
<temperature>Hot</temperature>
<description>Coffee</description>
</drink>
```

That is also valid. In general, you should use attributes for meta data (data about the data). You can only use a named attribute once for each element and the value it takes can only be a simple string (no structures) whereas adding elements is more flexible.

XML documents have a few other requirements. They must have a root element, inside which everything else is put. For our current example, that would be `menu`. They should also contain what is known as a prolog, which specifies the XML version and the character encoding:

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

As an exercise, look at the cafe menu above and try to produce an XML document that specifies everything it contains. When you have one, compare it to the example below (which has some sections missing, but is enough to give you the idea). It might not be the same as your solution, as there is more than one way to specify things in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
<food>
```

```
<breakfast>
<item price="2.99">
  <title>Light Breakfast</title>
  <ingredient>Toast</ingredient>
  <ingredient>Butter</ingredient>
  <ingredient>Jam</ingredient>
</item>
<item price="5.99">
<title>English Breakfast</title>
  <ingredient>Sausage</ingredient>
  <ingredient>Egg</ingredient>
  <ingredient>Bacon</ingredient>
  <ingredient>Toast</ingredient>
</item>
</breakfast>

</food>
<drinks>
<hot>
  <item price="2.99">
    <title>Coffee</title>
  </item>
</hot>
</drinks>
</menu>
```

Validation and Correctness

There could be many ways to choose how to represent data in XML, so to facilitate sharing data in a common format, XML has methods for specifying what an XML document of a certain type can contain. A document that is written in well formed XML (the syntax is right) and conforms to a given set of definitions is considered **valid**. There are two ways to specify the requirements for an XML format: a Document Type Definition (DTD) and an XML Schema. We will just describe the XML schema here, as it is defined using XML itself and is more flexible than a DTD.

An XML schema defines what a valid XML document that adheres to that schema can contain, and how to interpret its contents. It defines the names of the tags and attributes that can be used and the data types they should contain. Imagine you want to share menus among different cafes using XML. You could define a schema that they must all conform to, which would make sure that everyone interprets the data in the correct way. Here is an extract from a definition for the menu XML:

```
<xs:element name="item">
<xs:attribute name="price" type="xs:decimal"/>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="ingredient" maxOccurs="unbounded" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The snippet above defines the menu item element, specifying that it has one attribute, called price which is of type decimal. It then specifies that the element can contain other elements (`complexType`) and that those elements must be in the sequence of a title and a list of ingredients. The `maxOccurs="unbounded"` specifies that any number of ingredients are allowed. If that is removed, the maximum number of elements with the same tag name defaults to 1.

There is a lot more to defining, validating and writing XML than we have room for here. This should be enough to allow you to start with and learn more about XML when you need to, however.

1.2.5 Web Services

Some data sets can be very large and it is best to avoid moving them if possible. It is far more efficient to send requests to a server where the data are stored so that the processing and selection can take place where the data are. The results of such processing are usually smaller and faster to transmit. It is not sensible to open up a database to allow users to simply type database commands (in SQL, for example) as that is neither secure nor user friendly. What is needed is a method for allowing controlled access to a data source via an existing data transfer protocol. One way to do that is via HTTP (the protocol used to deliver web pages to your browser) using what is known as a web service API. API means Application Programming Interface. An API provides access to a set of functions so that other programmers can use the raw functionality of some code, rather than running an application. For example, the Twitter API allows you to search and retrieve tweets in JSON format rather than using the Twitter app.

Most people's experience of the web involves using a browser to request web pages. Pages can be requested from a static URL, which identifies the page to be returned, or from a URL plus some additional data (the contents of a form that a user has filled in, for example), in which case those data are used to determine the contents of the page that is returned. The same process can be applied to access data rather than web pages. Consider a movie review service. The data accessed might be static (like a list of all the movies on the database) or it might be selected by passing search terms (a film or director name, for example). The key difference is that the data are not returned as a web page. They are returned in some format a program can use - often JSON or XML.

Many web APIs us an architectural style known as REST. REST stands for Representational State Transfer and a proper REST API (known as a RESTful API) satisfies a number of conditions about how it works. We won't worry about what those are here. For our purposes, a web API or a REST API are ways of accessing data or data related services by sending HTTP requests to a web address (known as a Uniform Resource Identifier, or URI).

Many web services require some form of authentication, which is proof that you are allowed to access the data you are requesting. Sometimes authentication is required to protect private data (using the GMail API, for example) but other times, it is just to monitor or limit how much data each user is accessing. Google offers lots of web based APIs for things like maps, search, and translation. They are free if you use them sparingly, and charged for otherwise. Either way, you need to identify yourself with an access key to use them. Many services allow you to generate an API key for free on their website.

The simplest way to experience a web services API is to use a browser. If you type a web service URI into your browser, it will display the data that are returned. Here is an easy example, which searches the Victoria and Albert Museum database and returns a JSON formatted response to (in this case) a search for objects that mention Turner:

```
http://www.vam.ac.uk/api/json/museumobject/search?q=Turner
```

The real intention for web APIs is that the data they provide will be used in a program, not viewed as raw data in a browser. Python has several modules for accessing data via HTTP and, of course, modules for processing JSON and XML once the data are received. Think of some of the websites you use and look to see if they offer an API—you will find many of them do. Here is a list of a few you could look at:

The online Movie API	`http://www.omdbapi.com/`
The V and A Museum	`https://www.vam.ac.uk/api`
Oxford Dictionaries	`https://developer.oxforddictionaries.com/`
Fun Translation	`https://api.funtranslations.com/api`

1.2.6 Data Sources

If you want to practice your data wrangling skills, then you need a variety of data sets to play with. Fortunately, there are thousands of data sets available online for you to download and explore. Many governments have open data initiatives that allow you to access data from public life. For data from a broader range of sources, the web site called **Kaggle** is a great place to look. It has a large repository of datasets that people have uploaded, It also has competitions, courses, examples of code, and discussion communities. Kaggle is a great place for any aspiring data scientist. The website is at `kaggle,com`. You will also find a similar collection of data and resources at the **KD Nuggets** web site at `kdnuggets.com`. Other places to look for good data sets include **Github** at `github.com`, **Data World** at `data.world` and the UCI machine learning data repository at `https://archive.ics.uci.edu/ml/`.

1.3 Scripting with Python

It is fair to say that Python is a pretty popular programming language, particularly for data science. It is easy to learn, easy to use and easy to read (if programmed well). It also has an excellent set of pre-made modules for doing great things with data. In just a few lines of code you can process a file, visualise its contents, plot locations on a map, perform speech recognition, do machine learning, analyse the sentiment of a written sentence, recognise a face or detect a street sign. You can generate music, solve equations, manipulate matrices, or scrape data from a website. The internet has a wealth of examples of code you can download and use, tutorials you can follow and advice you can take. I like Python a lot and I think you will too, so let's get started.

1.3.1 Some Practicalities

What do you need to write your first Python program, and how will you organise all these great projects you will build? Python programs are just plain text. You can simply type them into any text editor and save them — usually with the suffix `.py` — ready to be run. You run your script with what is known as a Python interpreter, which is a program that reads your Python text file and executes the commands it contains. The least you need, then, is a text editor and the Python interpreter.

You will quickly find you need more support than that, however. You will want an editor that knows the Python syntax and can help you type it correctly. You will want to organise your various projects so that they do not interfere with each other if they need different versions of any of the software they rely on. You will also want a program that can help you debug (fix errors in) your code. For all of this, I recommend a program called Anaconda Navigator. You can download it from:

```
anaconda.com/distribution
```

It looks like a serious piece of software, and it is, but you can download it and use it for free. It will take a while to get to know and appreciate all it can do for you, but let's start with the right tools rather than swap over later. Python can actually run in two different ways. I've described how you write code into a file and then run it through the interpreter. You can also use Python in interactive mode. So-called iPython allows you to type and run commands one (or a few) at a time to see intermediate results and make programming decisions as you go. This can be particularly useful for manipulating and analysing data or for learning Python.

A very nice way to run iPython is to use a program called Jupyter Notebooks. It comes as part of the Anaconda package, so once you have installed that, you will have Jupyter too. Jupyter runs in a web browser and allows you to type and run sections of Python code one after another. There are also many online repositories of ready made notebooks you can learn from. This chapter has an associated set of notebooks you can download and run yourself. A nice feature of the notebooks is that they contain a mixture of code and written document, so they can guide you through the steps a program is taking.

1.3.2 Python Versions and Environments

Python, like many software products, has grown through a number of versions. After version 2.7, a large change was made in the jump to version 3. The most recent version (in 2019, at least) is 3.8. This is important for two reasons. Many pre-built packages for useful things like performing speech recognition or connecting to Bluetooth have been written over the years. When first written, they were made to work with whatever was the current version of Python. Over the years, some have been updated and some have not. This means that the package you need for your project might need to run on some earlier version of Python. The distinction is often between versions 2 or 3, but some require a version after a certain point (say 3.3, for example).

The differences between version 2.7 and all the versions at 3 and above mean that you cannot necessarily run a Python 2.7 program using a version 3 interpreter. The consequences of this are that you may need more than one version of Python installed on your computer so that you can use the right version for the right code. You will need to keep the installed packages separate so that they are run in the correct version.

This is achieved in Python using something called **environments**. You can have any number of Python environments on your computer, each one running a different version of Python and each one with different packages

installed. The environments do not interfere with each other so you can install different packages with the right version of Python. Keeping track of your environments also helps you to share your Python code as you can specify what is needed to run it. You do not want too many environments set up. Try to avoid making a new one for each project you start, or you will spend your life installing modules you already have elsewhere. If you are using Python a lot, the chances are that you will need at least two environments—one for each major version: 2.7 and 3.6 or above.

The Anaconda program described above allows you to manage your Python environments. It also comes with a default environment called *Base*, which has a great many of the packages you will need already installed. Until you need to do more, simply use this base environment for learning Python. Software changes fast, so this book won't include click by click instructions for using software like Anaconda. It is simple to use and there is a lot of help online, so I am sure you will work it out.

1.3.3 Introduction to Python Programming

Let's write some Python code. You should now have Anaconda installed on your computer (or your preferred alternative if you like). All the examples in this section are short enough for you to type them in and run them. You can use a Jupyter notebook to do this. Run Jupyter from Anaconda—you should see a listing of the home directory in which Jupyter will store your programs. Create a new Python 3 notebook using the button marked `New`. A new tab should open in your browser with a blank notebook in it. You can give it a name by clicking where it says `Untitled` at the top. Pick a short name—let's say `myprogram`. This will produce a file called `myprogram.ipynb` and save it on your computer. Note that the `.ipynb` files are in a format that Jupyter can read - they are not Python programs in their own right.

This section assumes you are new to Python and to programming. If you are not, then please excuse the simple start. Things will speed up soon enough. In the first code box on your new notebook, type the following:

```
print("Here is my first line of code.")
```

You can run the code either by clicking the right facing triangle button marked **Run** at the top of the page or by pressing the `ctrl` and `enter` keys at the same time. The output from your code appears below the box containing the code (these boxes are called *cells*).

In what follows, you can start a new code cell for each new example or add them to the last box you used. When you run a cell of code, everything in it runs, so you should start a new cell if the previous one does anything that takes time. Anything that one cell does (importing modules, setting variables, etc.) persists into the following cells, so you can open a file and read its contents into memory once and then use that data in following cells without needing to re-run the file reading each time. We assume you are reading this while typing the examples at a computer, so there will be suggestions in the text for things for you to try. This is the best way to work through the sections.

You can add comments to your code so that other people who read it know what it does. The Python interpreter ignores the comments, which are specified using a hash character: #. Some of the code below will include comments to help illustrate its function. You do not need to type in the comments to run the code.

1.3.4 Python Data Structures

Programs store and manipulate data in memory in variables that are accessed by their name. Here is a simple example:

```
my_var = 10
print(my_var)
```

We have created a variable named `my_var` and assigned it the value of 10. Once we have a named variable, we can manipulate it:

```
my_var = my_var+20
print(my_var)
```

Remember, variable names should be meaningful to a person reading the code, but they have no meaning to the Python interpreter. Just because I used `my_var` here in an example, doesn't mean you should get into the habit of using that name for your variables.

Data Types

It is important for a computer program to distinguish among different possible data types. There are things that make sense for numbers that do not make sense for strings (like adding, for example). Python has basic data types, listed in table 1.1.

Name	Type	Note
`int`	Integer	Whole numbers, e.g. 1,10,100
`float`	Floating point	Numbers with a decimal place, e.g. 0.5, 234.43
`bool`	Boolean	Truth values, either `True` or `False`
`str`	String	String of characters, e.g. `'Hello'`, `"World"`
`complex`	Complex Number	Real and imaginary parts, e.g. 3+5j
`tuple`	Tuple	Fixed ordered list of values, e.g. (1,2, 1) or (1,'b')
`frozenset`	Frozen set	Fixed unordered set of values, e.g. (1,2) or (1,'b')

Table 1.1: The basic types in Python.

We do not have to tell Python the type of each variable that we use; the Python interpreter decides automatically. You can find out the type of a variable using the `type`() function:

```
print(type(my_var))
```

Try setting `my_var` to various values like 10, 2/3, `"Hello"` and `True` to see what type is reported. The meaning of `int`, `float`, `bool`, and `str` are probably obvious. The last two might need a bit more explanation, and are described in section 1.3.7. You can force a variable to be treated as a certain type if you need to. This is called **casting** a variable and each basic type has a function that shares its name and converts (casts) its argument to the appropriate type. Some operators, like + and ∗ have a different meaning depending on the variable types they operate on. For example, + means add two numbers (`int` or `float`) but concatenate two strings. Here are some examples:

`str`(3)	`'3'`
3 + 3	6
`'a'` + `'b'`	`'ab'`
3 + `'3'`	Error - you cannot add a number and a string
3 + `int`(`'3'`)	6
`str`(3) + `'3'`	`'33'`

Try some more for yourself. For example, try `bool`(1) and `bool`(0). Note that `tuple`(v) and `frozenset`(v) require v to be a list or some other iterable structure.

Constructing Strings

A common requirement when generating output from a script is to construct a string from a number of variables. In the example above, we used `str`(3) + `'3'` to concatenate the number 3 converted to a string with '3'. That is a bit clumsy and there is a better way. Python formatted strings allow you to include variables inside a string, to be converted to strings automatically in place. They are used by putting an `f` character before the opening quotation mark and putting variable names (or expressions to be evaluated) inside { } braces like this `f'The variable a has the value {a}'`. Here is a longer example:

```
r = "red"
b = "blue"
x = 1
y = 3

print(f"Roses are {r}, violets are {b}, {y} - {x} is equal to {y-x}")
```

These are known as f-strings and were added to Python in version 3.6.

Lists

You can give variable names to more complex structures too. Python has built in support for lists, sets and dictionaries, which we introduced in section 1.1.1.

A list (other programming languages call them arrays) is an ordered list of values that are accessed by an index specifying the location of an entry. List syntax uses square brackets: [] for defining and accessing the values. You can define and access a list like this:

```
my_list = ["Dog", "Cat", "Pig"]
print(my_list[0])
```

`my_list`[0] refers to the first item in the list (the index starts at zero). Try accessing the other values in the list. You can find the length of a list using **`len`**(`my_list`). What happens if you try to access a location past the end of the list? If you use negative numbers as the index, the counting starts from the end of the list, so `my_list`[−1] gives the last value in the list.

You can access parts of a list using the slice operator, which is a colon (:). The code below appends some further values to the list using `append`() and then demonstrates some slices.

```
my_list.append("Mouse")
my_list.append("Cow")
my_list.append("Hamster")
print(my_list[:3])          # The first three entries
print(my_list[2:5])         # Entries from positions 2,3,4
print(my_list[2:])          # Entries from position 2 to the end
```

You will have noticed that the range slice `my_list`[`s`:`f`] retrieves the entries starting at position `s` up to position `f`−1. A third argument is allowed when slicing arrays, which specifies the step size taken when extracting elements from the list. For example, `my_list`[0:9:2] extracts the entries at locations 0, 2, 4, 6, and 8. The full syntax for slicing is:

```
my_list[start:end:step]
```

where `end` is one greater than the final index to extract. You can access each entry in a list in turn like this:

```
for entry in my_list:
    print(entry)
```

There are a few things to note in this piece of code. Each entry in the list is assigned to the variable named `entry`, one at a time. The **`print`**(`entry`) code is run once for each value in the list. There is also some new Python syntax here. Note the : at the end of the first line. This means that what follows should be run inside the loop. The code to run inside the loop is indented four spaces. The indentation defines the scope of the loop. If you want to run more than one line of code in the loop, indent them all to the same level. The loop ends when the indentation level returns to the same level as the code that started the loop.

```
for entry in my_list:
    print(entry)
    print("Still in the loop")
print("Out of the loop, now")
```

This use of the colon and indentation is used throughout Python to define the scope of loops, conditions, functions and classes. You may have used other programming languages that use curly braces { } for this purpose. Python does not do this.

Higher Dimensions You can embed lists in lists, or define lists over several dimensions. For example, you might define this list of lists:

```
embedded = ["a","b",["c","d","e"],"f"]
```

Write some more code to print the values at different positions. What is special about position 2? See what happens if you print `embedded`[2][1]

A more common arrangement is to use a two dimensional list, like this:

```
my_list = [["a","b","c"],["d","e","f"],["g","h","i"]]
```

This is a list of length 3, where each entry is a list of length 3. You can treat this as a matrix by thinking of it as a list of rows. Define this matrix and then use the syntax you already know to print the second row, then the second entry in the second row (e).

Extracting a column is a little more involved. You have to extract the element in the chosen column one at a time: `[row[i] for row in my_list]` will extract column `i` from each row into a new array. Of course, if you want to deal primarily with columns, not rows, you can just treat the structure as a list of columns instead. Sections 1.3.12 and 1.3.13 describe two much better ways of manipulating tabular data: Numpy and Pandas. Here are a few more list manipulation tricks you can use:

- Concatenate lists with + like this [1,2,3] + [4,5,6] produces [1,2,3,4,5,6]
- Replicate lists with $*$ like this [1,2] $*$ 3 produces [1, 2, 1, 2, 1, 2]
- Create an empty list like this `my_list = []`

1.3.5 Dictionaries

Unlike entries in lists, which are identified by their position, entries in dictionaries are identified by their name. Dictionaries are defined as a set of key:value pairs like this:

```
my_dict = {"Name":"Kevin", "Age":50}
```

The keys (entry names) must be strings but the values can be anything you want. In the example above, the first is a string and the second is an integer. The values can also be lists or dictionaries themselves, or lists of dictionaries, and so on. Access an entry using its name instead of a position:

```
print(my_dict["Name"])
```

Iterate through a dictionary by iterating over its keys and then using the key to access its associated value:

```
for k in my_dict:
    print(k,my_dict[k])
```

Or iterate over both the keys and values at the same time, in pairs:

```
for k,v in my_dict.items():
    print(k,v)
```

Do these look familiar? They are almost identical to JSON objects and can be used in the same way.

1.3.6 Sets

A Python set is an unordered collection of unique values. That means that any value cannot appear in the set more than once and that elements cannot be accessed by their location. Create a set using the `set` function and a list of values:

```
my_set = set([1, 2, 3, 1])
```

The resulting set will contain {1, 2, 3}. Note that sets are shown in curly braces when printed, but defined with the `set` function to differentiate them from dictionaries. You can provide a list or a tuple to set. Sets you can add to a set using `set.add(element)`, remove from it using `set.remove(element)` and clear it with `set.clear()`. Use a set when collecting the unique values in a list. For example, you could build a set of all the keys in an array of dictionaries. The good thing about using a set is that you do not have to check whether an item is already in the set before you add it. It it is already present, it will not be added.

1.3.7 Tuples and Frozensets

Lists, sets and dictionaries can all be added to or changed once they have been created. For this reason, we say that they are **mutable**. Section 1.3.7 describes mutability in more detail. Sometimes you need a list or a set that is immutable: once it has been created it cannot be changed. Tuples and frozensets have this quality. A tuple is a way of keeping a list of values together in a single named entity and a frozen set is a similar way of keeping a set of values together. The difference is that a tuple can contain duplicates and has a defined order (being a list) but a frozen set cannot contain duplicates and has no order. Here is an example of using each:

```
my_tuple = (1, 2, 1, 'a')
print(my_tuple[2])

my_frozen_set = frozenset([1, 2, 1, 'a'])
for m in my_frozen_set:
    print(m)
```

Note the differences: a **tuple** is defined using round brackets but a frozen set needs the explicit use of the name **frozenset**. The second duplicate value, 1, has been removed from the frozen set, but the tuple contains 1 twice. You cannot use `my_frozen_set[1]` to access an entry in the frozen set as entries are not ordered, and so not indexed. There is one more thing to note about tuples. You can define a two (or more) element tuple like this: `a = (1, 2, 3)` but you cannot define a single element tuple like this `a = (1)` because `(1)` is the same as 1, so you need to add a comma with nothing after it to tell Python that you want a tuple: `a = (1,)` or you can be explicit and say: `a = tuple([1])`.

Mutability

Whenever you use a piece of data in a Python program, it is stored in memory for as long as it is needed. Python decides when it is no longer needed, you do not have to tell it. This is true of variables and objects of any type, and even of literals (like 5 or `"Hello"`). The location in memory uniquely identifies the data concerned. For example, `a=5` will cause the program to allocate some memory to store the number 5 and record the fact that this memory location has the name `a` in the program. You can find the id of a variable with **id**(a). It is far more efficient if the value associated with a given id never changes. If an existing variable is assigned a new value, then a new id is created for the new variable—the old one is not overwritten. So the code:

```
a=1000
a=a+1
```

causes the program to create a variable called `a` at a certain memory location and store the value 1 there. It then creates another new variable, also called `a`, at a new memory location and stores the value of the first `a` plus 1 (so 1001) there. The first `a` is no longer needed and is disposed of. The memory location of the first `a` is not overwritten with the new value. This means we could then write `a="Hello World` without having to worry that there wasn't enough space where the integer 1 was stored to put the string `"Hello World"`. This is true of the basic Python data types of **int**, **float**, **str**, **complex**, **bool**, **tuple** and **frozenset**. These are all known as **immutable** types—their values cannot be changed.

More flexible structures like lists, dictionaries and sets are known as **mutable** types—their values can be altered without destroying the old structure and creating a new one. This is essential for allowing items to be added to or removed from the structure, for example. Tuples and frozen sets are immutable but lists and sets are mutable (and so are dictionaries). This means, for example, that there is no `append()` function for tuples, but there is for lists. We will return to the question of mutability and some of its consequences later.

1.3.8 Functions and Generators

Now we can store some data, it is time to do something with it. Imagine you have stored some temperature readings in a list, something like this:

```
temperatures = [15, 16, 18, 24, 26, 23, 24, 18, 16]
```

These are in centigrade and you want to produce a new list of temperatures in Fahrenheit. You can do this;

```
fah = [32 + 1.8*t for t in temperatures]
print(fah)
```

That is fine, but you might want to use that temperature conversion formula again elsewhere in the code. Rather than write the same code again, you can write it once in what is known as a function. Here we define the function for converting from centigrade to Fahrenheit:

```
def cent_to_fah(cent):
    return 32 + 1.8*t
```

This code does several things. It defines a function called `cent_to_fah`, which expects a single value to be passed to it. This is known as the **argument** to the function. Arguments are discussed in more detail in section 1.3.8. The code specifies what operations should be performed using the value in the argument and what value should be returned. To make use of a function, you use its name and the values of the argument(s) you want to pass to it. This is known as calling the function. The value that the function returns is used in the place from which it is called. We can rewrite our conversion code now as:

```
fah = [cent_to_fah(t) for t in temperatures]
print(fah)
```

Note the names of the variables and where they are used. In the loop that is calling the function, the variable `t` is set to each entry in the array in turn. This value is then passed to the function, where it is treated as having the name `cent`. The function call `cent_to_fah(t)` causes the code in the function to run. Note the syntax—the function name followed by the argument inside round brackets.

Argument Passing

You can pass zero or more arguments to a function and there are a variety of ways of matching the values you send to the variables named by the arguments. The simplest method uses **positional** arguments. This means that the values passed in the function call are matched to variable names in the function based on their position in the argument list. The function definition `def my_func(a, b, c)` called with `my_func(1, 2, 3)` will result in `a` being assigned the value 1, `b` being assigned 2 and `c` being assigned 3. This relies on you knowing the correct order for the arguments when you call the function.

Another method, known as **keyword arguments**, or kwargs, allows you to name the arguments in the function call, effectively making a variable assignment when you make the call. The order of the arguments no longer matters because you are naming then explicitly. The names you use when you call the function must match the argument names in the function definition, of course. The function call `my_func(b=2, c=3, a=1)` is equivalent to the positional call above. In this method, the order of the arguments in the function definition is not important, but the argument names are.

Function arguments can be given default values in the function definition. For example, `def my_func(a, b=2, c=3)` defines a function that must be called with a value for `a` (as it has no default defined) but for which the arguments `b` and `c` are optional. If they are not given in a call to the function, they take the default values in the function definition. The function call `my_func(5)` would result in `a` being 5, `b` being 2 and `c` being 3. If you want to set values for `a` and `c` but leave `b` as its default, you can call the function using keyword arguments: `my_func(1, c=5)`, for example. Here a mixture of positional and keyword arguments are used - `a` is set by its position and `c` is set using a keyword.

Sometimes the number of arguments a function needs to take might vary depending on its use. Imagine we need a function that takes a student number and the grades the student has achieved as arguments, printing them out in a table. The problem is that we might need a different number of grades for different students. We would like to be able to call it like this `tab_grades(234512,'A','B')` or like this: `tab_grades(234512,'A','B','B')` How do we define the function to accept an arbitrary number of grades? We use a variable length argument. In the function definition, you write:

```
def tab_grades(student_id, *grades):
    for g in grades:
        print(str(student_id)+','+g)
```

We can now call it with `tab_grades(234512,'A','B')` or `tab_grades(234512,'A','B','B')` or any other number of grades. Note the use of the star `*` before the argument name, `grades`. The number of arguments can vary and

the function can cope. The arguments are actually converted into a tuple so you can access them with either an iteration or using an index. Let us extend the function further and allow it to be called with an arbitrarily long list of course codes and associated grades, `'ITNPBD2'='A'`, for example. Now we would like an argument for each course code, but we do not know how many courses there might be, and we will not know their course codes. A similar feature is available for keyword arguments, which uses two stars: ** like this:

```
def tab_course_grades(student_id, **course_grades):
    for course in course_grades:
        print(str(student_id)+','+course+','+course_grades[course])
```

and can be called like any of these examples:

```
tab_module_grades(43534)
tab_module_grades(43534,ITNPBD2='A',ITNPBD3='B')
tab_module_grades(43534,ITNPBD2='A',ITNPBD3='B',ITNPBD1='A')
```

This time, the argument names and values are accessed as if the argument were a dictionary. If you are wondering why we don't just pass a tuple or a list or a dictionary to the function, then well done for thinking it! One reason is that it is more convenient to write the argument values straight into the function call, rather than building a list first. The second is that the variable length arguments are immutable, whereas lists and dictionaries are mutable, which can cause problems in some circumstances. For example, when setting the default value of a list argument to the empty list `[]`. If that list is ever changed in the function, it will stay changed the next time the function is called as it is mutable. That might not be what you want!

Return Values

When the function has run, its result may need to be passed back to the part of the program that called it. In the function, this is done with the **`return`** command. It is used like this:

```
def add_up(x,y):
    return x+y
```

and used in the calling environment in the same way that a variable is accessed, for example:

```
result = add_up(1,3)
print(add_up(2,4))
```

A function can return any data type. It can also return from more than one possible place in the function:

```
def is_bigger(x,y):
    if x > y:
        return True
    else:
        return False
```

A function does not need to return the same variable type from each of its possible exit routes, but it is considered good practice to ensure that it does. Once exception is where a function returns either a value or `None`. For example, here is a function that returns the number of characters in a string, or `None` if the value passed to it is not a string.

```
def str_len(x):
    if type(x) == str:
        return len(x)
    else:
        return None
```

A return statement also has the effect of terminating the function. Any code that follows the return statement is not run if the return statement executes. So in the example above, the **`else`** term is not strictly necessary as the **`return`** `None` line would only be reached if the **`return`** **`len`**`(x)` command was not run. However, it is good practice and aids readability to include the **`else`**.

Talking of good practice, it is also desirable to keep the return statements at the extremes of the function (near the start or near the end). You should avoid sprinkling return statements throughout a function as it makes it harder for a

human reader to follow. Consider the following function that calculates the grade class from a score from 0 to 100, but returns `None` for values outside that range:

```
def number_to_grade(number):
    if number > 100 or number < 0:
        return None
    if number < 50:
        return "Fail"
    if number < 60:
        return "Pass"
    if number < 70:
        return "Merit"
    return "Distinction"
```

Note how the returns are spread through the code and that later code (like the final `return "Distinction"`) relies on the fact that all other possibilities would have caused a return already. This code is much better written like this:

```
def number_to_grade(number):
    if number > 100 or number < 0:
        return None
    if number < 50:
        grade = "Fail"
    elif number < 60:
        grade = "Pass"
    elif number < 70:
        grade = "Merit"
    else:
        grade = "Distinction"
    return grade
```

Now the return is at the top for the error condition and the bottom for everything else. See section 1.3.9 for a description of the use of `if`, `elif` and `else`.

Argument Scope and Mutability

You can skip this section on the first reading if you want to get to grips with functions for the first time. Come back and read it later though. If you are happy with functions and the way they handle arguments, read this section now—it is important.

Consider the function definition:

```
def a_function(func_arg):
```

and the code calling it:

```
a_function(calling_arg)
```

The variable `calling_arg` in the calling environment becomes the variable `func_arg` inside the function. Behind the scenes, it is actually the id of `calling_arg` that is sent to the function (see section 1.3.7) and inside the function, both `calling_arg` and `func_arg` share the same id, so they are effectively the same variable. See for yourself:

```
def a_function(func_arg):
    print(id(func_arg))

calling_arg=1
print(id(calling_arg))
a_function(calling_arg)
```

See how the ids are the same. So what happens if we change the value of `func_arg` inside the function? If it is actually the same variable as `calling_arg`, does that change too? Unfortunately, that depends on the variable type

of `calling_arg` and on its scope. `calling_arg` can have global or local scope and it can be of either a mutable or immutable type.

Imagine you have written a bank of useful functions that you will use in many of your Python programs. These functions may well contain variables that are used as part of the calculations they make. You do not want these variables to interfere with the variables you are using in the programs that call these functions. Otherwise, you would need to know the names of all the variables used in all the functions! To avoid this problem, variables have an operating **scope**, which defines where their name has meaning. Each scope has its own **namespace**, which keeps track of the variables it can see. Variables defined inside a function have their own namespace, and so operate in a different scope from those in the main body of the program (or in other functions). Here is an example:

```
def sum_range(low,high):
    sum=0
    for i in range(low,high+1):
        sum=sum+i
    return sum

for i in range(2,5):
    print(i,sum_range(1,i))
```

The code above defines a function that returns the sum of the integers from `low` to `high` inclusive. Note that it uses a variable called `i` to count from `low` to `high`. In the calling environment, another loop, also using a variable called `i`, loops from 2 to 4 and calls `sum_range` using `i` as the argument value for `high`. Note how the two versions of `i` do not interfere with each other. You can verify that they are different by adding `print(id(i))` to both the function body and the calling loop. See how they are different?

When Python interprets a variable it looks at the most local scope first and, if the variable is not defined there, works its way out looking for a variable of that name in broader scopes. It uses the version of the variable it finds in the scope closest to where it is used. In this simple example, there are only two places to look—inside the function and in the calling environment. If a variable is defined in the calling environment, but not defined again inside the function, its value is still available to the function. For example:

```
def look_at_x():
    print(x)

x=1
look_at_x()
```

will result in the function printing 1. However, if the function assigns a value to `x`, it does so to a new, local version of `x` and leaves the top level `x` unchanged. If you want the function to alter the value of the higher level `x`, you can tell it to use the **global** version of `x` like this:

```
def change_x():
    global x
    x=5

x=1
change_x()
print(x)
```

Try this code and see how removing the `global x` declaration changes what is printed. Verify what you have found using `print(id(x))` inside the function both with and without the `global x` declaration. One more thing is worth pointing out here. What do you think the following code will produce?

```
def look_at_x():
    print(x)
    x = 2
    print(x)

x=1
```

```
look_at_x()
```

You might think it would print 1 (the value of the high level `x`) at the start of the function, then print 2 after x is re-assigned. This is **not** the case. The line `x = 2` causes a variable called `x` to be created in the scope of the function and trying to print it before it is declared produces an error. Try it and see.

To summarise so far: all functions can read the value of variables defined in the function itself or at a higher level, but not those defined in other functions. Declaring a new immutable variable inside a function creates a new local variable that is distinct from any variables with the same name outside the function. That means that the function can not change the value of any existing immutable variable. Declaring variables as global inside a function means that the function can both read and write a value to the variable of the same name in the main program code.

You should take great care when using global declarations. They mean a function can have side-effects that programmers using it might not know about. Avoid them in functions you might share or use again in a different program.

It gets slightly more complicated yet. Section 1.3.8 describes how some data types are mutable and some are immutable. Variables of an immutable type are given a value when they are first created and that value cannot be changed. Mutable type variables can have their values altered—usually by adding data to or removing from an existing structure. Basic types like strings and numbers are immutable so if a function assigns a new value to such a variable, it actually discards the old variable and creates a new one with the same name but a different value (and a different id). When this is done inside a function, the new variable is created with a scope defined by whether it is global or not. If it is global, the variable that shares its name in the global scope is discarded and a new variable, also with global scope, is created in its place. Otherwise, the higher level variable is left untouched and a new local variable is created. That is all consistent with what we know of how local and global variables work.

Now consider mutable objects like lists. Appending to a list does not discard the old list and create a new one. It just adds to the existing list. This is because the list is a mutable type. A function can append to a list using its name without the need to declare it as global because it is mutable. Try this:

```
def append_y(x):
    y.append(x)

y=[1, 2]
append_y(3)
print(y)
```

See how the function `append_y` has changed the value of `y` even though it is not declared as global. This is because `y` is of a mutable type. If you want to use a mutable type in a function, you can ensure that it does not interfere with mutable variables with the same name outside of that function by declaring a new version of it inside the function before you use it. If you add the code `y=[5, 6]` as the first line of the function `append_y` above, you will see that it creates a new, local `y` and does not append to the list declared outside the function.

A similar logic applies to the arguments passed to a function. Each parameter shares an id with the variable that is passed to it. For immutable types like integers and strings that doesn't matter because if they are assigned a new value, that creates a new local variable and the variable in the calling environment is left unchanged. For mutable types, this means that a function can change the contents of an argument it is sent. Try this better version of `append_y`:

```
def append_y(x,y):
    y.append(x)

y=[1, 2]
append_y(3, y)
print(y)
```

See how `y` is passed as an argument and, being mutable, is appended in the function. As a final exercise, try a few combinations of mutable and immutable arguments and see how you can (or cannot) change their values inside a function. Remember that tuples and frozensets are immutable—now you can start to see why they are useful.

Python Iterables

One reason for Python's popularity as a data wrangling language is its ability to manipulate objects that represent a data set. I've used the term *data set* vaguely, but will clarify right away. Python is great for manipulating lists, sets, ranges, sequences, counters, or anything that can be enumerated or iterated across. It is this last word, *iterate*, that we use a lot in Python. An object that can be processed one element at a time is known as an **iterable** and Python is great at generating and processing iterables. Python has a simple syntax for processing the elements of an iterable:

```
for element in my_object:
    print(element)
```

The code above iterates over an iterable object called `my_object`, assigning each of its members in turn to the variable `element`. Here we just print its value, but you can do whatever you want with it. `my_object` can be a data structure like a list or a set, or an object that allows iteration, or a function that generates the next element in a sequence each time it is called (see section 1.3.8 for a description of generator functions). We have already met lists and sets and iteration over them is simple:

```
pets = ['Cat', 'Dog', 'Rabbit']
for p in pets:
    print(p)
```

`range` is a commonly used class in Python, which produces iterable objects. The code below creates a `range` object and iterates over it.

```
my_range = range(0,5)
for i in my_range:
    print(i)
```

The easiest way to print the contents of an iterable (that is not a list) is to convert it to a list like this: `list(my_iterable)`. We will meet iterables again in several sections to come.

High Order Functions

In Python, functions are treated like any other data object and as such they can be passed as a parameter to another function and returned from a function in the same way you can return a value. Imagine you have written a function to sum the square root of all integers in a range. It would have two arguments: the first and the last integer in the range. Your function would call the `sqrt` function for each integer in the range and return their sum. Now imagine you need the same function, but for cube roots, or for each number squared. Instead of writing each of those functions separately, it would be much nicer to have a single function that took three arguments: the start and end of the range to process and the name of the function to process them with. Python allows you to do that. Functions that accept or return functions themselves are called high order functions. Here is the code for our first example:

```
import math

def sum_func_range(start, end, func):
    sum=0
    for v in range(start, end+1):
        sum+=func(v)
    return sum

sum_func_range(1,3,math.sqrt)
```

The function `sqrt` is part of the `math` module so we can pass it by name to our new function. We can also define our own functions and pass them by name.

```
def sqr(x):
    return x*x

sum_func_range(1, 3, sqr)
```

We can define all the functions we like and pass them like that, but sometimes we do not want to go to the lengths of defining a named function just to pass it to another function. For simple functions like `x*x`, it can be simpler to use what is known as an anonymous, or **lambda** function. A lambda function is defined by an argument list and an expression that is evaluated to provide a return value separated by a colon. The square function we defined above can be replaced with `lambda x: x*x`, where `x` is the single argument and `x*x` is the expression to be evaluated. We can use that in place of a function name when we call `sum_func_range` like this:

```
sum_func_range(1, 3, lambda x: x*x)
```

There is a strict limit on what can be included in the lambda expression. You cannot perform assignment or execute loops, for example. Anything like that needs to be written in a proper function. You can assign a name to a lambda function and use it in the same way as you would any other function:

```
add_up = lambda x,y: x+y
print(add_up(2,3))
```

Python also allows functions (or lambda functions) to be returned from functions. Here is an example of the syntax for each:

```
def make_func():
    def my_func(x):
        return x+1
    return my_func
```

Or the same thing using a lambda function

```
def make_func():
    return lambda x: x+1
```

Now we can call the function we have called `make_func` and receive a function as the return value. This new function can then be used in code that follows:

```
f = make_func()
print(f(5))
```

Map, Reduce and Filter

There are some common tasks that programmers need to perform that work very well with high order functions. They are known as **map**, **reduce** and **filter** and they each operate on iterable objects like lists. A map process takes an iterable and produces a new iterable, which is the result of passing each value in the first iterable through the same function. A reduce processes each entry in an iterable to produce a single value. A filter takes an iterable and produces a new list consisting only of the members of the first iterable that pass a given test. There are three functions in Python to run these processes and you will not be surprised to hear they are called `map`, `reduce`, and `filter`. Each has a similar syntax: the function takes two parameters—the name of the function to call and the iterable data structure to process. They are all examples of high order functions as their first argument is, itself, a function. Here are examples of each.

The `map` example squares each item in an iterable and produces a new iterable containing those squared values.

```
a = [2,3,6,8]
b = map(sqr, a)
print(list(b))
```

We can do the same thing with a lambda function, of course:

```
a = [2,3,6,8]
b = map(lambda x: x*x, a)
print(list(b))
```

The result of performing the `map` process is actually a map object. If you want to see the objects it contains (the items in the list) then you can convert it to a list, as we do above with `list(b)` or iterate over its contents like this:

```
for r in b:
    print(r)
```

The **reduce** example needs a little more explanation. The reduce process accumulates the final result by repeatedly calling a function that takes the current accumulated value and the next value in the list and produces a new accumulated value to be passed in with the next item in the list. Given a list to reduce, the first call takes the first and second items as the two parameters. Here is an example that adds up the elements in the list, note we need to import **reduce**:

```
from functools import reduce

def reduce_add(tot, next_val):
    return tot + next_val

a = [2,4,6,8]
b = reduce(reduce_add, a)
print(b)
```

The function that we pass to **reduce**, which we have called `reduce_add` above, takes two parameters. The first, (`tot` in this example) keeps a running total and the second (`next_val`) is the next value in the list. The reduce function is called once for every item in the list (with the first call having `tot` as the first item in the list and `next_val` as the second).

The **filter** function also takes a function as its first argument and an iterable as its second. In this case, the function must return a Boolean type (`True` or `False`). The filter process produces an iterable that contains only those members of the input iterable that cause the function to return `True`. Here is an example that selects only the square numbers from a list.

```
def is_square(x):
    return math.sqrt(x) == int(math.sqrt(x))

print(is_square(9))
print(is_square(10))

candidates=range(1,101)
squares=filter(is_square,candidates)
print(list(squares))
```

The map and reduce functions can be combined, for example here we square each element before summing the result:

```
sumsq = reduce(reduce_add, map(sqr,a))
print(sumsq)
```

Generator Functions

Sometimes you have more time than memory. Imagine you need to find the sum of all the integers from one to a billion. You would not create an array containing each integer and pass it to a function that sums array contents—that would be wasteful of memory. You would start a counter at 1 and a sum at zero and keep incrementing the counter and adding it to the sum until the counter reached a billion. Now consider what you would do if the sequence you needed to generate was more complicated than a simple counter. Perhaps you often need a sequence of prime numbers or even numbers, or all the combinations of eight letters. It would be useful if you could write a function that iterated over a loop, producing the next item in the sequence one at a time. Python does this using **Generator Functions**.

The key difference between a generator function and the functions we have met already is that generators do not return a value—they yield one. To yield a value means to return the next value in a sequence, but with a memory that means the function yields subsequent values on subsequent calls. The function itself is written as if it is required to iterate over the entire sequence at once, but does not actually do so. It actually takes one iteration step each time it is used, until it reaches the end.

Generator functions are called and used in a slightly different way from other functions too. When you call a generator function and assign it to a variable, the variable does not contain a return value, rather it contains a generator object, which is treated like an iterable. Here is a simple example, which defines a generator function that produces a sequence of square numbers. The function takes two arguments: a start and end value of the integers to be squared.

```
def square_gen(s,f):
    for a in range(s,f+1):
        yield a*a

sq=square_gen(0,5)    # Set up the generator
for i in sq:          # This repeatedly calls the function
    print(i)
```

In the code above, you can see that the call to our function, `square_gen` returns a generator object (you could insert `print(sq)` into the code to verify that). The function looks like it executes the whole loop, but it doesn't, it prepares to execute the loop, and then yields one value at a time as required. We iterate over `sq` as if it were any other Python iterable object, using `for i in sq`: in this case. You can produce a list containing all the elements in the sequence by calling `list(sq)`. Note, however, that once the generator has been iterated once, it is finished. You need to call it again if you want to repeat the sequence.

Generators can be very useful if the sequences you need are long and should not be stored in memory. They are useful if the process of generating the next item in a sequence requires sufficient lines of code to warrant the use of a function. For generating sequences that require a simpler expression, it can be more efficient to use list comprehension, which is described next.

Comprehension

Another, simpler way to define a sequence is to apply an expression to every element of another iterable. This approach, known as **comprehension**, is often used to define a sequence based on a range of integers that form the inputs to the function. For example, we can define the sequence of numbers starting at zero and increasing in steps of 0.5 to 5. Here is the code for that:

```
halfs = [x/2 for x in range(0,11)]
print(halfs)
```

The variable, `x` is used to iterate over the range and is also used in the expression to produce the members of a list. The square brackets specify that the result is to be stored in a list. To produce a generator instead, use round brackets:

```
halfs = (x/2 for x in range(0,11))
print(halfs)
```

You can now iterate through `halfs` in the normal way: `for h in halfs`.

You can generate lists, as above, which is known as list comprehension, or you can generate sets, which is known as set comprehension! You can also include a selection term using an `if` statement. Here we select the four letter words from a set into a new set:

```
animal_set = {'Cat', 'Dog', 'Horse', 'Goat', 'Fish'}
len4 = {x for x in animal_set if len(x) == 4}
print(len4)
```

This time we select all the words with more than three letters plus their letter count into a set of tuples:

```
animal_set = {'Cat', 'Dog', 'Horse', 'Goat', 'Fish'}
len4 = {(x,len(x)) for x in animal_set if len(x) > 3}
print(len4)
```

I find myself using list comprehension all the time. To me, it is preferable to using `map` or `filter` and somehow, more Pythonic. These two lines of code are equivalent, for example:

```
nums = [1, 2, 3, 4]
xsq = map(sqr, nums)
xsq = [sqr(x) for x in nums]
```

Recursive Functions

List comprehension and generator functions are useful for generating sequences of values. Another way to use a function repeatedly to calculate a result is to allow a function to call itself. This is known as **recursion**. Recursive functions are often used to traverse a sequence, for example to explore a tree structure or calculate a value that has a recursive definition. A good example of a recursive function definition is the factorial function. n factorial, often written as $n!$ is the result of multiplying together all the integers from 1 to n, so $n! = \prod_{i=1}^{n} i$. The recursive definition of $n!$ is

$$1! = 1$$

$$n! = n(n-1)!$$

The definition has two parts, a base case for $n = 1$ and a recursive case for every other positive integer. Recursive functions in Python require the same two parts: a base case and a recursive call. Each time the function is called from inside its own body, the argument is updated and passed in again. The function must also check for the base case, which is the point at which the recursive calls stop and a return statement is reached. Here is the Python code for recursively calculating $n!$:

```
def recur_factorial(n):
    # Base case: 1! = 1
    if n == 1:
        return 1
    # Recursive case: n! = n * (n-1)!
    else:
        return n * recur_factorial(n-1)

print(recur_factorial(5))
```

In the example, the function is first called with the value 5 for the argument. The function then returns a value that requires it to call itself again with the argument $n = 4$. So rather than returning, it calls another instance of the function—so we now have the same function running twice—to calculate $(n-1)!$. That continues until the base case is reached, when $n = 1$ and the function returns the value 1. That is not the value that is returned to the main body of code that made the function call in the first place (otherwise, the function would always return 1). The value 1 is returned to the instance of the function that was called when $n = 2$, where it can be multiplied by n and returned to the previous instance of the function, where $n = 3$ and so on until finally the instance of the function that was first called with $n = 5$ can return the answer, 120.

When you design a recursive function, you must identify the base case, which allows the function to return a value, and the recursive call, which calls the same function again with an argument that moves closer to the base case. In the factorial example, we moved closer to the base case by subtracting one from n in the function call.

It is easier to think about recursion if you think of each call to the function as a call to a new function with the same name. Each new function has its own scope and so the variables it uses have their own values in each. The following code illustrates the point by recursively counting down from a starting integer to zero. It also assigns a random value to a variable called `my_rnd` each time the function is called. Run the code and look to see how the random numbers and their IDs correspond before and after the recursive function call. Note that the function does not return anything, so is not useful for anything other than demonstrating the process of recursion.

```
import random

def recur_track(n):
    a = random.randint(300, 500)
    print("Called with", n, "rand=", my_rnd, id(my_rnd))
    if n==0:
        return 0
    else:
        recur_track(n-1)
        print("After recursion with",n,"rand=",a,"id=",id(my_rnd))
```

```
        return 0

recur_track(4)
```

The contexts for each call to the function are stored on what is known as a **stack**. They are added to the top of the stack as the function is called and removed again (in reverse order) when the function returns. You can see this when you run the code above, as the numbers first decrease towards zero as the recursive calls are made, but then increase again as the program returns to each earlier call.

Perhaps these are not very satisfactory examples. You could compute the factorial iteratively just as easily and with code that is probably easier to read. Here, then, is an example that makes use of recursion in a more satisfying way. In this example, we want to list all the keys in a dictionary. That would be simple if the dictionary were flat (had no values that were, themselves, dictionaries) but for a nested dictionary, there may be an arbitrary depth of embedded objects for each element. Dictionaries can also contain lists of dictionaries in their values, so we need to be able to traverse lists too. As with all recursive design, we need a base case and a way of moving towards it. The base case is when a value is neither a list nor a dictionary, in which case no recursion is required and the current key set is returned. If the value is a list or a dictionary, it must be recursively processed by the same function. Examine the code to achieve this:

```
def recur_get_keys(d,keyset):
    '''Recursively get all the keys from a dictionary and
    all sub docs, including arrays of sub docs'''
    if isinstance(d,dict):    # If this value is a dictionary
        for k in d.keys():
            keyset.add(k)
            recur_get_keys(d[k],keyset)   # Here is the recursion
            if isinstance(d[k],list):
                for e in d[k]:
                    recur_get_keys(e,keyset) # and here
    return keyset

d1 = {'a':1, 'b':2, 'c':3}
k = recur_get_keys(d1,set())
print(k)

d2 = {'a':1, 'b':{'c':1}, 'd':[1,2,3], 'e':[{'f':1}, {'g':1,'h':1}]}
k = recur_get_keys(d2,set())
print(sorted(k))
```

The recursive function is a very elegant solution. The base case is simple: list the keys at the current level, and the recursive reduction is a step towards finding the lowest level where no further embedding is found. It is not easy to see how you would do this without recursion. Recursive functions have an overhead of memory as each new call to the function is added to the stack. There is also a little extra time required to run the function, compared to iteration, so recursion isn't often the first tool in the box you would reach for, but when it is appropriate, it can be a joy to use.

1.3.9 Conditional Statements

Previous sections have already used some examples of conditional statements, but this section describes them in more detail. A conditional statement runs some code only if a given condition is true. You can also provide code to be run if the condition is false. The basic syntax is:

```
if condition
   Thing to do if the condition is true
else
    Thing to do if the condition if false
```

That is not Python code, of course, so here is an example you can run and experiment with:

Operator	Meaning
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
! =	Not equal
is	Object identity
is not	Negated object identity

Table 1.2: Python operators for comparing two variables. Note that equal uses a double equals sign: ==. Use **is** to test whether two differently named variables refer to the same object (have the same id). You should also use **is** to test whether a value is None.

```
a = 2
b = 3
if a > b:
    print(f"{a} is greater than {b}")
else:
    print(f"{a} is not greater than {b}")
```

As you know, Python uses indentation to define the scope of blocks of code. In the code below, if a is greater than b then the two lines of code that follow will be executed because they are indented to the same level. The final print("Done") will execute regardless of the values of a and b because that code is not indented.

```
if a > b:
    c = 1
    print(c)
print("Done")
```

Python has a shorthand method for cascading if statements following an else, called **elif**. Here is an example of its use:

```
if a > b:
    rel = "gt"
elif a == b:
    rel = "eq"
else:
    rel = "lt"
```

The operators you can use to compare expressions are listed in table 1.2. The main thing to notice is that testing whether two expressions evaluate to the same value is done using a double equals sign: ==. The rule is = for assignment and == for comparison.

```
a = 1    # Assign 1 to the variable a
a == 1   # Evaluates to True if a equals 1, and False otherwise
```

You can compare values, variables, or expressions, so the following are all valid:

```
if 1 == 2:
    print("You should never see this!")

a = 3
b = 3
if a == b:
    print("You will see this if a and b have the same values")

if a+b > 4:
    print("You will see this if a+b is greater than 4")
```

The condition can be any statement that evaluates to `True` or `False`. You can construct logical conditions using the keywords **`and`**, **`or`** and **`not`** and use round brackets to specify the order in which operators are evaluated. When evaluating a statement containing no brackets, Python will evaluate the **`not`** operators first, then the **`and`**s and then, last of all, the **`or`**s. You can change that order using brackets. Anything inside brackets will be evaluated first. Table 1.3 shows a few examples to allow you to test your understanding.

Condition	Evaluation
`True and False`	`False`
`False or True`	`True`
`not False or True`	`True`
`not (False or True)`	`False`
`True or False and False`	`True`
`(True or False) and False`	`False`

Table 1.3: Some examples of the result of evaluating various clauses joined by **`and`**, **`or`**, **`not`** and brackets.

Note that some other, non boolean values also evaluate to false. They include 0 and `None`. Python supports a shorthand way of writing an expression that evaluates to one value if a condition is true and another if the condition is false. They are called **conditional expressions**. Here is an example:

```
grade = 60
result = "Pass" if grade > 49 else "Fail"
print(result)
```

This does not alter the flow of control like the **`if else`** statement, it just produces one of two possible values based on a condition. In the example above the result is assigned to the variable `result` but it can also be directly printed or used as part of a larger expression, for example:

```
overtime = True
base = 100
pay = base + (100 if overtime else 50)
print(pay)
```

1.3.10 Python Style - PEP 8

Python code should be written in a way that makes it easy for a human reader to understand. This is achieved in two ways: by writing clear consistent code, and by using comments correctly. The standard for writing Python in a style that is consistent across all programmers is known as PEP 8. It sets out guidance on how Python programs should be formatted and presented. Here are the main points of importance from PEP 8:

- Code style should be consistent across all your programs unless there is good reason for it not to be, for example to conform to a style from some old legacy code
- Code indentation should be four spaces deep. Do not use tabs
- Lines of code should not be longer than 79 characters
- Use longer indents to break long lines of code. Line up brackets and quotation marks if appropriate:

  ```
  print("This is a very long piece of text, well, at least long enough",
        "that it would be better to split it over two lines")
  ```

- Put binary operators (+, - etc) at start of broken lines:

  ```
  numbers = [1, 2, 3]
  res = (numbers[0]
      + numbers[1]
      - numbers[2])
  print(res)
  ```

- Put a space either side of the = in variable assignments
- Put a space after the comma in lists, but not before
- Use spaces to clarify operator precedence (this does not change the way the code runs, but makes it easier to read:

```
my_var = 1
x = [1, 2, 3]
x = a*b + c*d
```

- Variables should be named descriptively. Single letter variable names can be acceptable (iterating over an index with `i`, for example), but don't be lazy.
- Variable and function names should be lower case with underscores for spaces: `my_variable`
- Class names should be first letter upper case with no spaces: `MyClass`
- Constants should be upper case with underscore spaces: `MY_CONSTANT`

Comments Comments are added to code using the hash symbol: *#*. Anything on the same line following the *#* is ignored by the Python interpreter and is there only to aid the human reader. Comments should explain what the code does and, more importantly, why it is doing it. Comments should add something that the average Python programmer would not find obvious just by looking at the code. Compare the two commented lines of code below, for example:

```
bonus = bonus + 100     # Add 100 the bonus

bonus = bonus + 100     # The bonus is 100 more here because it is Saturday
```

When adding a descriptive comment to a module, function, method or class, do not use the *#* symbol, use what is known as **docstring** format, in which a longer string (over several lines if needed) is delimited by triple quotes: *'''*. These comments should describe the purpose of the function or class and how it should be used. If you share your code with others (or have a short memory for the functions you have written in the past), the docstring is a useful guide to using your code. You can see the docstring associated with any module, function, method or class with the **help** function. The following code defines a function and then prints the docstring associated with it:

```
def str_int_concat(st, num):
    '''Concatenate an integer and a string to produce a new string

    Args:
        num (int): The integer to append to the string
        st (str): The string to append to

    Returns:
        str_int_concat (string): A string that is the result
        of appending the number argument to the end of the
        string argument
    '''

    return st+str(num)

conc = str_int_concat("Cat",2)
print(conc)
help(str_int_concat)
```

There are a number of different styles for formatting docstrings. If you get a job somewhere that produces Python code, they should have a house style—ask them what it is.

1.3.11 Reading and Writing Data

A chapter on manipulating data wouldn't be complete without a section on reading and writing data in various forms. Here it is. This section describes how to read and write data using files, and across the internet.

Files

To open a file for reading or writing, use the **open**() function. In its simplest form, it requires a file path name and an argument that specifies whether to read from or write to (or append) the file. Each file that is opened should be closed later once processing has completed. This is done with `close`()

```
f = open("my_data.txt","r")     # Open the file called my_data.txt for reading
data = f.read()
print(data)
f.close()
```

In the code above, we open a file for reading (using `"r"` as the second argument) and read its entire contents into a variable called `data` as a string. To try this code for yourself, create a text file called my_data.txt and save it on your computer. If it is not in the same directory as your Python code, you will need to specify the full path in the file open statement: `f = open("C:\users\me\data.txt","r")`, for example. To read one line at a time from a file, use `readline`(), which returns a string containing the next unread line from a file or `None` if there are no more lines to read:

```
f = open("my_data.txt","r")
line = f.readline()
while line:
    print(line)
    line = f.readline()
f.close()
```

Here is a a more elegant way to do this:

```
f = open("my_data.txt","r")
for line in f:
    print(line)
f.close()
```

Files can be written to by either opening a new, empty file, which overwrites any existing file, or by appending to the end of an existing file. Apart from appending to a file, all other manipulation of data in a file is done in memory by reading from the file first. Open a new, empty file for writing like this: `f = open("my_new_file.txt","w")`. Note the `"w"` specifies that the file is to be written to. Open a file for appending with `"a"`: `f = open("my_new_file.txt","a")`. Regardless of how a file is opened (for writing or appending), each time you write data to it until it is closed, that data is appended to the end of the file. Run the following and examine the file it produces.

```
f = open("my_new_file.txt","w")
f.write("Here is a string I have written to a file")
f.write(" Here is another string.")
f.close()
```

Python has an alternative syntax for opening and using files, which automatically closes the file when you have finished using it:

```
with open("my_data.txt","r") as f:
    a = f.readline()
    b = f.readline()
print("The file is closed now we are out of the indented block")
```

You will see in sections 1.3.12 and 1.3.13 that other Python data manipulation packages have their own dedicated file reading and writing functions.

Directory Listing You may need your script to look for a file in one or more folders (or directories) or list the contents of a folder from a location on disk. You can access folder data including file names and sub folders using the Python package called `os`. To find the location of the current working directory (the one that Python will read from or write to if you only give a file name and no path), use:

```
import os

print(os.getcwd())
```

File systems are organised in a tree-like structure with a root directory that has sub directories below it. Each directory can contain files (which are leaves in the tree) and directories (which are branches). To list all the files and sub directories (and the contents of the sub directories, and so on) in a given directory, use `os.walk`, which produces a generator function for files and directories. Here we pass it the current working directory:

```
root = os.getcwd()

dir_content = os.walk(root)
# dir_content is a generator, so we iterate:
# r=root, d=directories, f = files
for r, d, f in dir_content:
    print(r, d, f)
```

The `walk` method traverses the file tree from the current location down each branch in turn, yielding results for one folder at a time. Each iteration of the generator, `dir_content` yields three values: the location of the folder at the current location of the walk, a list of sub directories contained by that folder, and a list of filenames in that folder. To access the individual filenames in a given folder, for example, you need to iterate over the files list. Here is an example that looks for `.txt` files in the current working directory.

```
dir_content = os.walk(root)
_, _, f = next(dir_content)
for fn in f:
    if fn[-4:] == '.txt':
        print(fn)
```

In the example above, we do not need to look in sub directories, so rather than iterating over `dir_content`, we just use the **`next`**() function to take the first result yielded by the generator (that for the current working directory). We do not need the name of the current directory, nor the list of folders it contains, so we use `_, _, f = next(dir_content)`. The underscores tell Python that we know there is a variable there, but we don't need it. Having extracted the file names list (`f`), we iterate over it looking for names that end in `'.txt.'` and print them. We could do the same thing with list comprehension (note how we needed to call `os.walk` again to restart the iteration:

```
dir_content = os.walk(root)
_, _, f = next(dir_content)
txts = [fn for fn in f if fn[-4:] == '.txt']
print(txts)
```

Data From a URL

If you want to read data from an online source, you cannot just use `'open()'`. Some packages, like Pandas and NumPy are able to open files directly from a URL, but in general, you need a package that can handle moving data over HTTP. We will describe one such package: **requests**. Here is a simple example, which requests JSON data from the Victoria and Albert Museum API. We are searching for entries in the museum's catalogue that mention Turner. The code prints the status code given in response to the HTTP request (200 means success) and the data received.

```
import requests

url="https://www.vam.ac.uk/api/json/museumobject/search?q=Turner"
resp = requests.get(url)
```

```
print (resp.status_code)
print (resp.content)
```

1.3.12 NumPy

Python has two very widely used data manipulation packages, called NumPy and Pandas. This section describes the use of Numpy and the next covers Pandas. NumPy is a module for manipulating array data in Python. We have already met Python lists, but NumPy extends the functionality of Python lists considerably. It is also one of the best examples of doing things in a Pythonic way, as you will see as we progress.

If you are using Anaconda Base, then Numpy will be already installed. If not, you will need to install it with `pip install numpy`. To use it in your code, you will need to import it using `import numpy as np`. The `as np` part is optional, but it allows you to refer to all the numpy functions with the two letter name, `np` rather than typing `numpy` every time. This has become a common convention now. This first example shows the modules being imported. The rest of the examples will skip the import statement, assuming you have already done it.

```
import numpy as np

x = np.array([1,2,3])
print(x)
```

Numpy arrays can have one or more dimensions but we will restrict most of our examples to one or two dimensions as they are most common. Many of the useful operations that can be performed on a NumPy array operate on every element in the array with a syntax that does not need to specify a loop or iteration over its elements. Here is some code that adds 5 to every element in a numpy array. This code is elegant and readable.

```
x = np.array([1,2,3])
print(x)
x = x + 5
print(x)
```

Note that the numpy function `array()` is called using the prefix `np` to tell the interpreter to use the NumPy function called `array`. We will use this `np` prefix whenever we use a NumPy function.

NumPy Indexing A two dimensional array is constructed as an array of rows, each of which is an array itself. Rows are accessed by their index, starting at zero. Single elements are accessed by their row, column coordinates. Columns are extracted using the slice operator `:`. A colon in place of the first coordinate means *the whole column*. Which column is extracted is specified in the second coordinate. Here are some examples:

```
y = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Array", y)
print("Row 1", y[1])
print("Row 1, Column 2 intersect", y[1, 2])
print("Column 1", y[:,1])
```

The slice operator is even more useful than that. You can specify a start and end index around the `:` to specify a subrange of data to extract. The syntax is `s:e` where `s` is the index of the start of the array you want to extract and `e` is one more than the last index. Here are some more examples using the array `y` defined in the previous example.

```
print("Rows 0 to 1", y[0:2])
s = 0
e = 2
print("The same, but with variables defining start and end",y[s:e])
print("Columns 1 and 2", y[:,1:3])
print("The top left square of 2 by 2", y[0:2,0:2])
```

It is also possible to extract those data from a NumPy array that satisfy a condition. Try this simple version first:

```
gr_five = y[y>5]
print(gr_five)
```

Selections can be done with a binary mask like this:

```
my_sel=np.array([[True, False, False], [False, True, False], [False, False, True]])
print(y[my_sel])
```

or with an array of index arrays, one for each dimension. In 2 dimensions, that might look like this:

```
# Extracts locations [0, 0], [1, 1], [2, 2]
print(y[[0, 1, 2], [0, 1, 2]])
```

The locations that are selected are those at the intersection of the coordinates in the index arrays. So the first array specifies the row indexes and the second specifies the columns. The selection is where they intersect, which in the example above is at the locations [0, 0], [1, 1], [2, 2]. Selection and elementwise operations can be combined to operate on only a selection from an array. Here we add 100 to all the elments in y that are greater then 3:

```
y[y>3] += 100
print(y)
```

NumPy Array Shapes The shape of a NumPy array is defined as a tuple listing the number of rows and columns it has. Try this:

```
z = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(np.shape(z))
```

You can reshape an array using the reshape() function. For example, this is how to reshape the array z to be 4 rows by 2 columns.

```
z = np.reshape(z, (4, 2))
print(z)
```

Arrays of a chosen shape can be defined in a number of ways:

```
# A 2D array containing 1.0 in each location with 2 rows and 3 columns
ones = np.ones((2, 3))
print(ones)

# Now the 1s are integers
int_ones = np.ones((2, 3), dtype=np.int)
print(int_ones)

# An array of integers from 0 to 26 in a 3 by 3 by 3 array
a = np.arange(27).reshape((3, 3, 3))
print(a)
```

NumPy and Files NumPy arrays can be filled directly from a file in a single command, assuming the file contains tabular numeric data with each row of the file corresponding to a row of data and each column separated by a special character (comma and tab are common).

```
dat = np.loadtxt("for_numpy.csv")
```

There are some useful options you can specify when you load the data from a file. The most useful are the delimiter used to separate columns in the file and the number of rows to skip (often one if the file contains a header row). Here we load a file while specifying that it is comma separated and that the first row should be skipped:

```
dat = np.loadtxt("for_numpy.csv", delimiter=',', skiprows=1)
```

Once the data are loaded, NumPy offers a number of built in functions for making calculations and manipulating the data it contains. Here are some examples of those you might find most useful.

Calculate the sum of all the rows, or all the columns or all the entries in an array:

```
my_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
add_col = np.sum(my_array, axis = 0)     # axis = 0 means sum columns
add_row = np.sum(my_array, axis = 1)     # axis = 1 means sum rows
add_all = np.sum(my_array)     # No axis given, so sum all entries
print(add_col, add_row, add_all)
```

Round all the elements in an array to a given number of decimal places:

```
div_3 = my_array/3
rounded = np.round(div_3, 2)
print(div_3,rounded)
```

Reverse the order of elements in an array along the given axis:

```
flip_c = np.flip(my_array, axis = 0)     # Flip columns
flip_r = np.flip(my_array, axis = 1)     # Flip rows
print(flip_c, flip_r)
```

Find the unique elements of an array, flattened into a one dimensional list:

```
my_array = np.array([[1, 2, 3], [3, 4, 5], [5, 6, 7]])
unq = np.unique(my_array)
print(unq)
```

Split an array into *n* equal parts:

```
rows = np.split(my_array, 3, axis=0)
print(rows)
cols = np.split(my_array, 3, axis=1)
print(cols)
```

Flatten an array into one dimension:

```
flat_ar = my_array.flatten()
print(flat_ar)
```

1.3.13 Pandas

As you have learned about NumPy, you have possibly thought about how you could do spreadsheet-like or even database-like operations with it. The data are in something like a table, after all. You can perform some simple spreadsheet like operations on a NumPy array (calculate the sum of a column, for example) but it is not designed to do many of the other things you could do with a spreadsheet or a database. If you want to load data into memory and perform tabular operations using Python, then you need Pandas.

Pandas (the name is a contraction of 'Panel Data') is built on top of NumPy and offers a wealth of tabular data manipulation tools. You can load data into memory, analyse it, plot it, add and remove rows and columns, you can even perform relational database style aggregation and joins across multiple tables. I hope that by learning Pandas, you will be tempted away from spreadsheets forever!

The two main data structures used by Pandas are the **series**, which is a single column of data, and the **data frame**, which is a set of series joined to create a table of rows and columns. Columns are identified by labels, much like the field names in a database table. If you don't provide labels, Pandas uses the integers starting at zero. You can also label rows, with what is known as an index. You can actually have more than one index label for each row, indexes do not have to be unique, and rows also have a number identifying them, whether you define an index or not.

Columns are accessed by name and rows can be accessed by either their index value or their numbered location. There are two different syntax patterns for accessing columns: the array-like square bracket notation or a notation that uses a dot (like in SQL). The syntax for accessing rows by name is `loc['name']` and by row number is `iloc[rownumber]`

An example will help. Here we define a data frame, name the columns and the indexes and select a single column by name (using each possible syntax) and a single row, first by name using `loc` and then by row number using `iloc`.

```
import pandas as pd

d = pd.DataFrame([[1,2,3], [4,5,6]], index=['a', 'b'], columns=["q", "w", "e"])
print(d)
print("Column w", d['w'])
print("Column q" ,d.q)
print("Row 1", d.iloc[1])
print("Row a", d.loc['a'])
```

Here is the output from some of the code above. Having declared the data frame, `d`, we print it and get the data and the column and row names:

```
   q  w  e
a  1  2  3
b  4  5  6
```

When we print a single column, we are given the row index names too:

```
   w
a  2
b  5
```

and when we print a single row, we are still given the column names:

```
q  1
w  2
e  3
```

You can also create a dataframe from a JSON object, with the key names in the JSON forming the column names in the data frame. Column data are specified in arrays:

```
d = DataFrame({'q':[1,4], 'w':[2,5], 'e':[3,6]})
```

Notice how the data are given in columns, rather than rows. It is, of course, far more common to read data from a file, so let us construct a data frame from a file and conduct some initial analysis. The file contains fictional data from a fitness tracker and measures the time and distance of dated cycle rides, swims and runs. Here we open the file and look at the first five rows.

```
df = pd.read_csv("exercise.csv")
print(df)
```

```
   Activity  Distance  Date        Time
0  Cycle     40000     01/01/2017  88
1  Swim      1500      02/01/2017  36
2  Run       5000      21/01/2017  22
3  Swim      1000      08/01/2017  21
4  Swim      1500      10/01/2017  29
```

The `read_csv` function assumes that the data are comma separated unless you specify a different separator, for example `sep = '\t'` for tab separation. It also infers the column names from the file. You can specify which column(s) to index with the `index_col` argument.

```
df = pd.read_csv("exercise.csv", index_col='Date')
print(df.head())
```

```
              Activity  Distance  Time
Date
01/01/2017    Cycle     40000     88
02/01/2017    Swim      1500      36
21/01/2017    Run       5000      22
08/01/2017    Swim      1000      21
10/01/2017    Swim      1500      29
```

To get an initial idea of the contents of the file, use the `describe` function:

```
df.describe()
```

which produces a table giving the count, mean, standard deviation, minimum, maximum and quartiles of the numeric variables in the file. Here is the output from our file:

```
          Distance       Time
count   301.000000   301.000000
mean   11627.906977   39.671096
std    12306.140145   27.005705
min      500.000000    7.000000
25%     1500.000000   21.000000
50%    10000.000000   30.000000
75%    15000.000000   51.000000
max    40000.000000  130.000000
```

Find the unique values that the string type variables can take using the `unique` function:

```
print(df.Activity.unique())
```

The statistics for the whole file are not very meaningful because they cover different activities. Cycle rides tend to be longer than swims, for example, so the average distance across all activities is not very helpful. This is where using an index on the rows is useful. By setting the values in the Activity column as an index, it is possible to analyse the different activities separately,

```
df.set_index('Activity')
df.loc['Swim'].describe()
```

```
          Distance      Time
count   100.000000  100.000000
mean    950.000000   20.400000
std     385.992097    8.902389
min     500.000000    7.000000
25%     500.000000   11.000000
50%    1000.000000   20.000000
75%    1125.000000   26.000000
max    1500.000000   41.000000
```

More generally, you can split the data into groups based on the value of one variable and calculate statistics of a second variable for each of those groups. This works whether the field being grouped is an index or not. Here we calculate the mean distance by activity:

```
print(df.groupby(['Activity'])['Distance'].mean())
```

Adding and Removing Data Add a new column to a data frame by allocating it a name. In this example, we calculate the average speed of each activity by dividing the distance by the time taken, creating a new column called `av_speed`:

```
exercise['av_speed']=exercise['Distance']/exercise['Time']
```

That gives us the average speed in meters (the unit used in the distance column) per minute (the unit of time used). To convert that column to kilometers per hour, we divide by 1000 and multiply by 60. This code overwrites the `av_speed` column with a new value.

```
exercise['av_speed']=exercise['av_speed']/1000*60
print(df.head())
```

```
            Distance  Date        Time  av_speed
Activity
Cycle      40000     01/01/2017  88    27.272727
Swim       1500      02/01/2017  36    2.500000
Run        5000      21/01/2017  22    13.636364
Swim       1000      08/01/2017  21    2.857143
Swim       1500      10/01/2017  29    3.103448
```

Rows of data can be appended to a data frame with `append`. The data to be appended must be a data frame too. The column names of the new data should match those of the data frame being appended. The indexes of the new data will be used if given in the new data frame. If the new data frame does not have any indexes defined, it will use integer values starting at zero. You can specify that all indexes should be ignored by adding the argument `ignore_index=True`. Here is an example you can experiment with:

```
d = pd.DataFrame([[1, 2, 3], [4, 5, 6]], index=['a', 'b'], columns=["q", "w", "e"])
d2 = pd.DataFrame([[7, 8, 9], [10, 11, 12]], index=['c', 'd'],columns=["q", "w", "e"])
d3=d.append(d2)
print(d3)
```

Multi Indexes An index is a label for a row of data, rather than being data itself. For example, you might have sales data by region and by month. In this case, region and month are labels and sales figures are the data. It is a loose distinction—you could argue that region and month are data too, but calling them labels can be useful when organising or grouping data. Imagine some sales data, which we load into a Pandas dataframe with the code below. Column zero contains a region and column one contains a month.

```
sales=pd.read_csv('sales.csv',index_col=[0,1])
sales.head()
```

The argument `index_col`=[0,1] specifies that both the first and second columns in the file contain data that should become an index. If, in that file, those columns are called `Region` and `Month`, for example, then we can use `sales.loc['USA']` to access all the rows where the `Region` column contains `'USA'` or `sales.loc['USA','August']` to access the August figure for the USA. We can calculate aggregations by index like this:

```
sales.groupby(['Region']).mean()
```

1.3.14 SciPy

There is a package for Python called SciPy, and there is also the SciPy project, which includes some of the other packages described in this chapter like NumPy, Matplotlib and Pandas. The SciPy package provides many useful scientific numerical analysis tools for signal processing, optimisation, and statistics among others. SciPy is integrated with NumPy and the two work together. Here is a high level overview of the SciPy capabilities:

- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fft)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems with ARPACK
- Compressed Sparse Graph Routines (scipy.sparse.csgraph)
- Spatial data structures and algorithms (scipy.spatial)
- Statistics (scipy.stats)
- Multidimensional image processing (scipy.ndimage)
- File IO (scipy.io)

1.3.15 Data Visualisation and Plotting

There are a few different choices for plotting graphs and charts from data in Python. They differ in their complexity and in the format in which they expect data. We will take a look at three of them: Matplotlib, Plotly and Seaborn.

Matplotlib

Matplotlib is an easy to use and versatile plotting library that can produce a nice range of graphs from list or NumPy array data. To use it, first import the pyplot module—here we give it a short name of `plt`. If you are using a Jupyter notebook, you may also need to tell it to display the plots inline on the notebook - that is what the last line below does. Do not use that if you want the plots to appear in a separate window.

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Here we present a few examples to give you a feel for using matplotlib. For more detail, read the documentation at https://matplotlib.org/. In the following example we create two NumPy arrays, `x` and `y` and plot them as a line graph. Note how we build up the single plot by adding the data, then the axis labels and then a title. The first line, creates a NumPy array of 20 evenly spaced values between -4 and 4.

```
x = np.linspace(-4,4,20)
y = x*x
plt.plot(x, y)
plt.xlabel("X")
plt.ylabel("X Squared")
plt.title("X squared from -4 to 4")
plt.show()
```

In that way, we can add further lines and other things like a legend:

```
plt.plot(x, y, 'r', label='Squared')
plt.plot(x, x*x*x, 'b', label="Cubed")
plt.xlabel("X")
plt.ylabel("f(X)")
plt.legend()
plt.title("X Squared and Cubed")
plt.show()
```

In the code above, the colour of the line was specified by a letter: `r` for red and `b` for blue. That same argument can be used to specify whether the plot uses lines or points or both, the style of those lines or points, and their colour. The argument is a string and has components to define the plot style concatenated into a single code. The format of the string is `[color][marker][line]` where [] indicates the contents are optional. You can see the full set of codes on the matplotlib website, but here are some examples to give you the idea:

'r–'	Red Dashed line
'bo-'	Blue line with filled round points
'+'	+ shaped markers
'g:'	green dotted line
'#EE00EE'	Solid line in the defined colour

You can see these styles together in one graph by running this code. They are shown in a plot in figure 1.2.

```
yset = [x/i for i in range(1,6)]
fmt = ['r--', 'bo-', '+', 'g:', '#EE00EE']
for i in range(0,5):
    plt.plot(x,yset[i],fmt[i],label=fmt[i])
plt.legend()
plt.show()
```

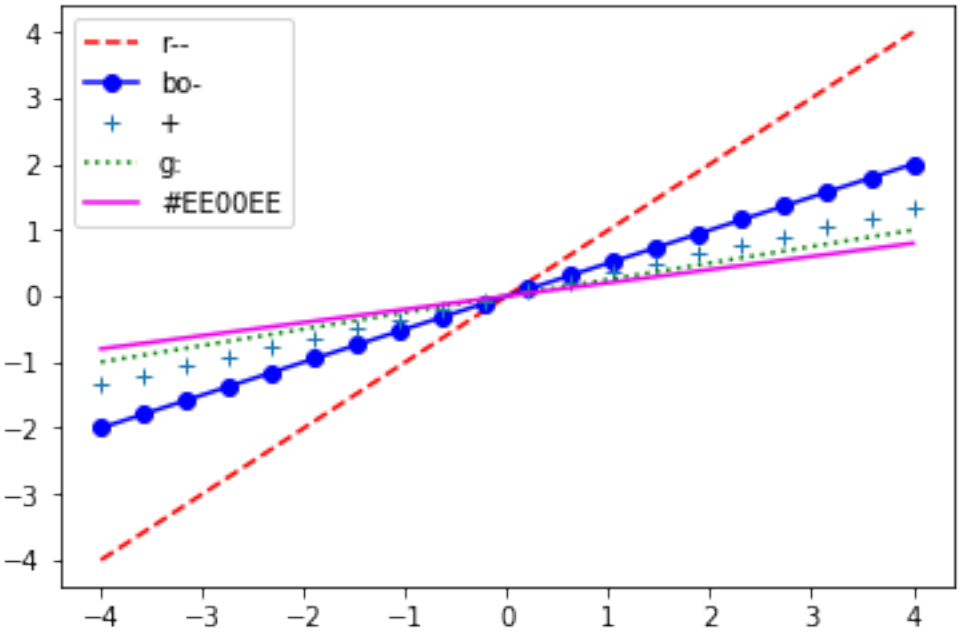

Figure 1.2: Some examples of different plot styles in Matplotlib

Matplotlib will plot a histogram for you without you needing to count the frequencies yourself. Given an array of numbers and a number of bins to split the data range into, matplotlib will construct the histogram for you. In this example we create an array of 1000 values sampled from a normal distribution with a mean of zero and variance of one and plot its histogram in 20 bins.

```
data = np.random.randn(1000)
plt.hist(data, 20)
plt.show()
```

Matplotlib will produce many other styles of plot like bar charts, scatter plots, bubble plots and even pie charts (though you probably shouldn't). Sometimes, you need to do more than just plot the data however, and this is where the next two packages come in.

Seaborn

Seaborn is another visualisation library, built on top of matplotlib. It can produce a wider range of plot styles, plot directly from a Pandas data frame, and perform some simple statistical estimation like calculating regression lines and distribution functions. It also comes with some example datasets, and we will use one of them in the following examples. The data describes the size of tips left in a restaurant at different times by different people. Here we import Seaborn, fetch the data into a Pandas dataframe and print its first 10 rows.

```
import seaborn as sns
import pandas as pd

tips = sns.load_dataset("tips")
display(tips.head())
```

It is then a simple one line command to produce a scatter plot with total bill and tip on the two axes, using point colour to show day of the week and point size to show the size of the dining party.

```
sns.relplot(x='total_bill', y='tip', data=tips, hue='day', size='size')
```

Just like matplotlib, Seaborn can make the calculations necessary for building a histogram. It can also perform simple statistical estimation to plot probability distribution curves. The following code plots a histogram from a single numeric variable and adds a rug plot, which shows a small line for each data point along the X axis. It also uses kernel density estimation to produce an estimate of the probability distribution of the variable.

```
sns.distplot(tips['tip'], kde=True, rug=True)
```

Plotly

Matplotlib and seaborn are both very good for producing static plots, which means the plots are not interactive. That is fine for a paper or a report, but for exploring data or presenting web based visualisations, it can be useful to allow a user to interact with the graph. By zooming in and out or rotating the graph, it can be easier to pick out the important relationships in the data. Plotly is a good package for producing interactive graphs. There were some significant changes in version 4 of Plotly, and that is the version we are using here. The key aspects of Plotly that might make it the right choice for your project are that the plots are interactive, they can be hosted online for you by Plotly if you have an account with them, and the full description of a plot is represented in JSON so you can easily share your plots or use them in other languages, like JavaScript. We will just use a Jupyter notebook and the offline version of the package, which means we can display the charts in the web browser that is running our notebook and we do not need a Plotly account. You will need to install Plotly, however, and you can find instructions on how to do that for your computer at the Plotly website at `https://plot.ly/python/`.

A simple Plotly graph allows you to zoom in and out, select an area of the graph, take a snapshot of the graph to the clipboard, and hover over points to see the data behind them. Here is a simple example, where we import the required package and plot a scatter plot. Plotly will plot from Pandas in the same way that Seaborn does, so the `tips` data frame in the following code is the same as that used in the Seaborn example above.

```
import plotly.express as px
fig = px.scatter(tips, x="total_bill", y="tip")
fig.show()
```

Three dimensional graphs are much easier to read if you can rotate them and Plotly allows you to do this. Try the following code. The toolbar presented with the graph now has a tool for rotating the image.

```
fig = px.scatter_3d(tips, x="total_bill", y="tip", z="size", color="sex")
fig.show()
```

In the two examples above, we create a variable called `fig` and call the `fig.show()` function to create the graph. If you type `print(fig)` you will see that it is a Plotly Figure object, but it contains a JSON object. You can save this JSON, share it with others who want to plot the chart, and use it from any of the other languages that Plotly supports, such as Matlab, R and JavaScript. You can also define your chart directly in JSON and ask Plotly to display it, rather than calling Plotly functions to create the graph. To display a graph from its JSON description, use `show` from the `plotly.io` module:

```
fig = {
    "data": [{"type": "bar",
              "x": ['A', 'B', 'C'],
              "y": [1, 2, 3]}],
    "layout": {"title": {"text": "A Bar Chart"}}
}

import plotly.io as pio
pio.show(fig)
```

That was a very quick look at three plotting packages. There is not space here to present a full tutorial on each of them, but you can find those easily online. You have seen some examples of usage, and you can choose among them as appropriate to your needs.

1.3.16 Geographic Data

When plotting data about physical locations, it can be instructive to plot them on a map. Matplotlib and Plotly offer functionality for drawing maps and plotting various types of data on them. Plotting data on a map requires choices about both the way the data are presented and the way the map is used.

Finding Your Location

To plot data on a map, each datum to be plotted must be associated with a location. The simplest way of representing global location is to use longitude and latitude. From any point on the globe, travelling north or south changes your latitude and travelling east or west changes your longitude. The equator is at zero latitude and the Greenwich Meridian (which runs through Greenwich, in London) is at zero longitude. Longitude and latitude are angles from the earth's centre, measured in degrees and further divided in to minutes and seconds. However, you will often find them represented as decimals, which makes them slightly easier to manipulate in code.

Longitude starts at zero at Greenwich and increases in a positive direction moving east (to the right on a map) and in a negative direction going west. It increases in each direction until half way around the globe, at 180, where it switches sign and starts moving back towards zero, so it ranges from zero to 180 east and to -180 west. Longitude, on the other hand, starts at zero at the equator and runs up to the poles, being 90 at the North Pole and -90 at the South Pole. Latitude measures distance from the equator, so it only ranges between -90 and 90. Table 1.4 shows the locations, using a decimal representation, of a few important places.

Place	Latitude	Longitude
London	51.507351	-0.127758
New York	40.712776	-74.005974
New Delhi	28.613939	77.209023
Buenos Aires	-34.603683	-58.381557
Sydney	-33.868820	151.209290
Stirling	56.116524	-3.936903

Table 1.4: The latitude and longitude of some places you might know.

Choropleth Plots and Heat Maps

One typical way to organise data for plotting on a map is to calculate an aggregate measure of some variable for each of a set of regions. That might be average income by state, total population by county, or annual rainfall by square kilometer, for example. When the regions are small uniform areas (like a square kilometer) the plots are called heat maps. When the regions vary in size and shape because of some natural or man made division of the land (counties or lake systems, for example) then the plot is called a choropleth.

A heat map can be made with the latitude and longitude of the points to be plotted, but a choropleth needs the boundaries of the regions to be defined too. Map plotting packages like Plotly allow you to name the regions in the data you provide. In its simplest form, the data for heat maps and choropleths consists of a list of locations (either by name or coordinates) and an associated value. The values are shown by a colour chosen from a continuum. Many different colour maps are available (a colour map provides a look up from numbers of a scale to colours). The most human readable are those that grade from light to dark on a single colour or across two colours, say from blue to red. Here is an example of how to plot a Choropleth map using Plotly. The results of running the code are shown too.

```
import plotly.graph_objects as go
import pandas as pd

agg_data = pd.read_csv('https://raw.githubusercontent.com/
           plotly/datasets/master/2011_us_ag_exports.csv')

fig = go.Figure(data=go.Choropleth(
    locations=agg_data['code'], # Spatial coordinates
    z = agg_data['total exports'].astype(float), # Data to be color-coded
```

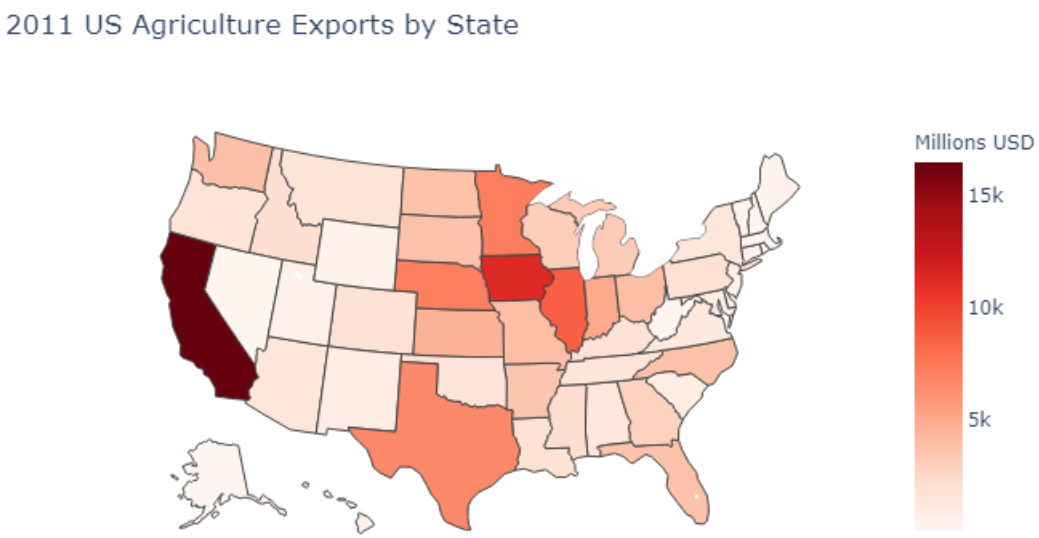

Figure 1.3: The results of using Plotly to plot US export data on a choropleth map.

```
    locationmode = 'USA-states', # Set of places match entries in `locations`
    colorscale = 'Reds',
    colorbar_title = "Millions USD",
))

fig.update_layout(
    title_text = '2011 US Agriculture Exports by State',
    geo_scope='usa', # limit map scope to USA
)

fig.show()
```

The example above downloads some data on US exports by state from 2011 and plots the value of total exports on a map, shown in figure 1.3. If you run the code yourself, you can explore the data, which is stored in the Pandas dataframe called `agg_data`. You can see what Plotly is doing for you by looking at the data and the code. The locations used to divide the choropleth into regions are state codes (AK, NY, etc.) and the code `locationmode = 'USA-states'` tells Plotly to use the boundaries it has defined for these codes.

Scatter and Bubble Plots

A heat map usually covers a fair portion of a map—it is not generally used to plot data for a list of specific coordinates. When the number of points to plot is smaller, a scatter plot is used. Points may be small markers (squares or circles, for example) or larger icons such as pins to show locations (such as all the hotels in a city). They can be coloured to reflect a value along a colour map in the same way that heat maps are. They can even change in size to create a bubble plot, though this can become confusing on a map if the circles are larger than the area the data refers to. Here is an example of a scatter plot on a map, made with Matplotlib and a related project called Cartopy.

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
```

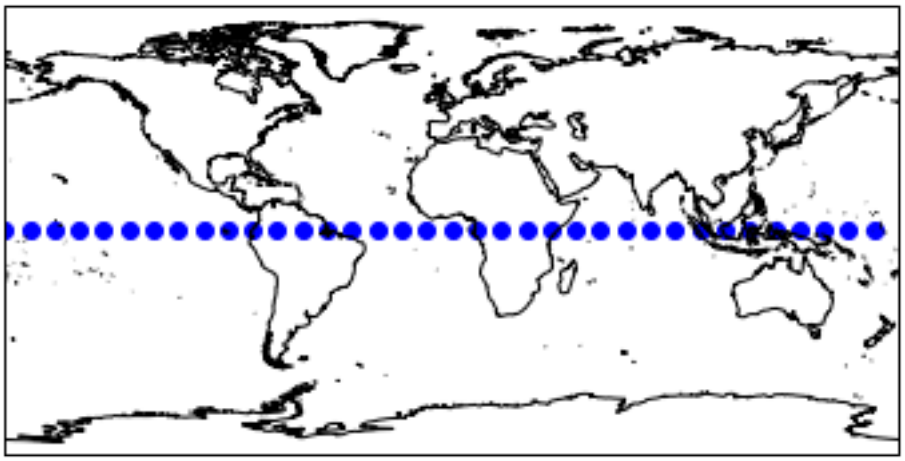

Figure 1.4: A Plate Carrée projection of the globe with points plotted at lattitude 0, and longitudes ranging from -180 to 180 in steps of 10.

```
def plotmap(lon, lat):
    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.set_extent([ext_x1, ext_x2, ext_y1, ext_y2])
    ax.coastlines(resolution='50m')
    # Resolution can only be one of '110m' '50', and '10m'.
    ax.scatter(lon,lat,marker='o',color='b',zorder=1)
    plt.show()

x = range(-180, 180 ,10)
y = [0]*36
plotmap(x, y)
```

In this example, we import cartopy, having installed it already with `conda install -c conda-forge cartopy`. We then define a function that plots a scatter graph on a map of the world with the coordinates given in two lists: longitude and latitude. The function defines the extent of the map with four numbers: the west and east extremes of the longitude and the north and south extremes of the latitude to be plotted. The coastlines are drawn at a resolution of 50m and the plot is drawn. The plot uses the `PlateCarree` projection, which projects the lines of longitude and latitude onto straight evenly spaced lines.

The function is tested by calling it with a list of x coordinates from -180 to 180 in steps of 10 and a list of constant latitudes of zero. You can see the result of the projection in the resulting plot - note that the line is straight and the dots have constant spacing. Figure 1.4 shows the results of running this code. Explore the code for yourself by changing the coordinates to be plotted, the extent of the plot and the markers plotted.

1.3.17 Scikit-learn

Machine learning involves training an algorithm to perform a certain task using examples in data, rather than programming the rules explicitly. The Python module commonly used to implement machine learning techniques is called **Scikit-learn**. This section quickly introduces ScikitLearn and gives a general framework for using it. Scikit-learn provides functionality for machine learning and data mining with techniques to perform classification, regression, clustering and all the associated data preparation and analysis. It is open source and free to use commercially, making it a popular choice. It is built on top of some Python modules we have already met: NumPy, SciPy and Matplotlib.

You can install Scikit-learn using `pip install scikit-learn` or you can use `conda` in the same way. If you are using Anaconda distribution, it will already be installed. Once it is installed, you import it using `import sklearn`. There are four main types of functionality provided by Scikit-learn. They are access to datasets; data preparation

functions; methods for analysis and machine learning; and model assessment and visualisation tools. Together, these provide everything you need to practice machine learning. We will look at each of them briefly here, so you have an idea of what to use Scikit-learn for.

Datasets Scikit-learn has a number of built in **datasets** that you can use to practice and learn. They are well known data sets from the machine learning literature and text books. The datasets are accessed via the `sklearn.datasets` package. There are small, toy datasets that come ready loaded and there are larger, real world datasets that Scikit-learn can download for you. The Scikit-learn documentation describes all the data sets in detail. Here is a simple example, where we load the famous iris dataset:

```
from sklearn import datasets
iris = sklearn.datasets.load_iris()
```

Data Preparation Before you can use a machine learning technique to model data, you need to carry out certain preparations. The data need to be loaded, the variables to be used in the model need to be extracted and pre-processed, and the data need to be split into different subsets to be used for training, optimising and testing the model. Scikit-learn can do all of these things. The `sklearn.preprocessing` package is used to scale, normalise and discretise data. Many machine learning algorithms work better if all the variables have been standardised to have zero mean and a variance of one, for example.

Supervised learning such as regression and classification generally uses two sets of data to build a model. The first represents a set of input patterns and the second a set of corresponding outputs. These need to be stored in two separate NumPy arrays. The arrays are ordered and each input is associated with an output as they share the same location in their respective arrays. For now, we will produce a simple data set and perform a linear multiple regression to give you the idea. Run the code in the next set of examples to construct some toy data and model it using linear regression. First we import the libraries we will need and build the dataset.

```
import numpy as np
x = np.random.random((10000,2))
coeffs=[2,5]
y=np.dot(x,coeffs)
y=y+np.random.normal(0,0.4,10000)
```

This produces a dataset with 10,000 examples where the inputs, x are vectors of length 2, each with a random number between 0 and 1 and the outputs, y are a linear combination of the inputs: $y = 2x_0 + 5x_1$. We have also added some normally distributed noise to the y values, with a mean of zero and a standard deviation of 0.4.

The data need to be split into two subsets: one for building the model and the other for testing it. The split obviously has to keep the inputs and outputs in their corresponding arrays together. This is most easily done using the Scikit learn function, `train_test_split`, which is found in the `sklearn.model_selection` module. It takes two arrays (the inputs and the outputs for your model) and returns four arrays: the inputs and outputs for the training and for the test procedure. Among other things, you can specify the proportion of the data to be separated for testing. Taking our input array, `x` and the outputs, `y`, the following splits the data into train and test sets:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33)
```

Model Building The arrays `x_train` and `y_train` are used next to build the model. We will try a simple linear regression. For this, we will use the `linear_model` module.

```
from sklearn import linear_model
reg = linear_model.LinearRegression()
reg.fit(x_train,y_train)
```

The code above sets up a linear model to use linear regression and then fits our training data to it. The form of the linear model will be $x = ax_0 + bx_1$ where a and b are the coefficients learned from the data. We can find out what these coefficients are in our model with

```
print(reg.coef_)
```

Run the code yourself and look at the coefficients. You should see an array containing two values: [2. 5.], which are equal to the coefficients we used to create the data originally, now inferred from the data alone.

Model Assessment Finally, we can see how well our model performs on the test data we set aside. Test the mode using this code:

```
import math
y_pred = reg.predict(x_test)
print("RMSE = ",math.sqrt(mean_squared_error(y_test, y_pred)))
```

Using the data in this example, the value you should get for the root mean squared error is about 0.4 (depending on the random values generated in your data). The value tells you the average distance between each prediction on the test data and the true value in the data. Notice that is is very close to the standard deviation of the noise we added to y when we created the data. We skipped some important steps here—most notably model optimisation using cross validation—but these steps will be covered in more detail in a later chapter.

1.3.18 Computer Vision with OpenCV

Images stored on a computer are represented as an array of numbers. These data can be manipulated to change the image or to infer things about the image. When we talk about computer vision (CV) we mean the challenge of allowing a computer to automatically infer facts about an image from the array of pixel values that represent it. Common CV tasks include detecting objects in an image, tracking the movement of an object in a video, segmenting an image into regions or classifying the main subject of a picture. These tasks are performed with a combination of image processing and machine learning. This section describes the use of a Python module called OpenCV, which is a large library containing many useful image processing functions. Install OpenCV for Python by typing

```
pip install opencv-python
```

Images are represented by an array of points called pixels. Each pixel has an associated colour, which is represented by a set of numbers called channels. There are different ways to represent colour, known as *colour spaces*. The default in OpenCV is the colour space known as BGR, which means the three channels contain the contribution of Blue, Green and Red in that order. In OpenCV the image is represented by a 3 dimensional NumPy array, `[y,x,c]`, representing the dimensions of row number, column number and colour channel respectively. Each channel value is between 0 and 255, so it can be represented by an unsigned 8 bit integer.

The pixel at coordinate $[0,0]$ is at the top left of the image, so there is a natural mapping from pixel location to array location as we write an array as rows and columns. If `img` is a NumPy array, then `img[y,x]` represents the pixel at row `y` and column `x`. The coordinates are specified row first, which might not be what you would expect. That means `img[0]` is the top row of the image. and if you type `print(shape(img))` you will see the number of rows followed by the number of columns. You can try it for yourself like this:

```
import cv2
import numpy as np

img = [[[x,x,x] for x in range(256)] for i in range(256)]
img=np.array(img).astype(np.uint8)
img[:,:,0]=255-img[:,:,0]
img[:,:,1]=0

cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

A few interesting things are happening in the code above. After importing `cv2`, which is the package name for OpenCV, we create a (255, 255, 3) shaped numpy array of 8 bit unsigned integers (`uint8`). Each row in each channel of the image contains the numbers from 0 to 255 in order. Next we use NumPy slicing to manipulate the colour channels individually. The slice `img` [:,:, `c`] extracts or operates on channel `c`, where `c`=0 is the blue channel, `c`=1 is

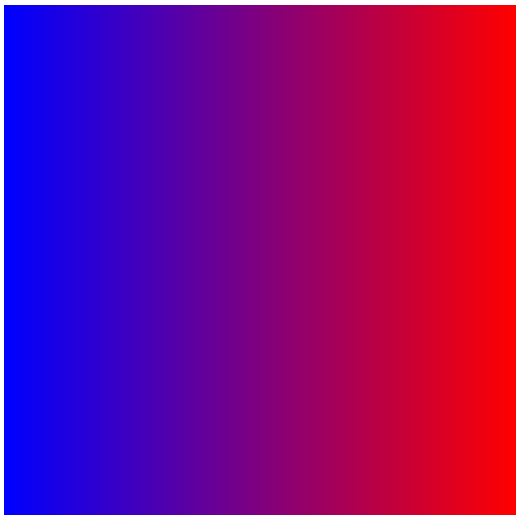

Figure 1.5: Constructing a colour fade by building a NumPy array.

green and c=2 is the red channel (assuming we are using the BGR colour space). In the example above, we set the green channel to all zero and reverse the blue channel, so it runs from 255 down to zero. The resulting image, shown in figure 1.5 fades from blue to red from left to right. The final three lines show the image in a new window, wait for you to press any key, and then close the window that contained the image.

Take a moment to understand this code and play around with it. The colour values are such that [0, 0, 0] is black and [255, 255, 255] is white. Try changing the colour of pixels at a chosen location. Remember, the coordinates are [row,column,channel]. If you are using Jupyter a notebook, you can use Matplotlib to display the image inline in the browser, rather than opening a new window, like this:

```
from matplotlib import pyplot as plt

plt.imshow(img)
plt.show()
```

A Note on Colour Spaces We have been using the default colour space for OpenCV, BGR. If you plot images with matplotlib, which uses RGB, you will need to change the colour space first. You might also need to change the colour space to gray scale for some operations or to Hue, Saturation, Value (HSV) for others. If you are curious, there are plenty of guides to colour spaces online. We will use them as appropriate in some of the examples that follow.

Now that we can manipulate a NumPy array and view it as an image, we can make use of OpenCV's long list of processing functions to manipulate an image. We do not need to write code to manipulate the NumPy array ourselves—we let open CV do it for us. This makes things much easier!

Load an Image

OpenCV will read an image file or take a frame from a webcam. You can then process and view the image. To read an image from a file and display it, use:

```
img = cv2.imread('mypicture.jpg')

cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

From now on, we will not bother including the three lines of code to display an image unless it is essential to the example—put them in yourself wherever you want to see an image. To extract an image from a webcam, use:

```
cap = cv2.VideoCapture(0)
ret,img = cap.read()
# ... (Use the image or display it)
cap.release()
```

To show a live stream from a webcam, just capture image after image and show them in the same window:

```
cap = cv2.VideoCapture(0)

while(cap.isOpened()):
    ret,img = cap.read()
    cv2.imshow('Cam',img)
    k = cv2.waitKey(10)
    if k == 27:
        break
cv2.destroyAllWindows()
cap.release()
```

You only need to call `cv2.VideoCapture(0)` once, to establish a connection with the webcam. The argument (0 in this example) is the index of the camera. It will be zero for a built in webcam on a laptop or a plugged in USB camera on a computer without a built in camera. If you plug a camera into a laptop with a built in camera, the new one will be at index 1. In this example, we wait for the escape key (code 27) to be pressed before stopping. As well as closing the window that shows the webcam image, we also release the camera. If you want more than one window open with different images, give them different names. Now we can access and display images, we can start to process them. We will start with the mundane—transforming and changing an image, and then move on to more interesting tasks like finding objects in an image,

Simple Image Transformations

You can rotate, resize and crop images easily. Here we find the current image shape and use it to double the size of the image.

```
print(img.shape)
height,width,_ = img.shape
bigger = cv2.resize(img,(2*width, 2*height), interpolation = cv2.INTER_CUBIC)
print(bigger.shape)
```

To rotate an image, a rotation matrix needs to be defined, which specifies the centre of rotation, the direction of rotation, and the number of degrees to rotate. That matrix is then applied to the image:

```
rotation_matrix = cv2.getRotationMatrix2D((width / 2, height / 2), 90, 1)
# or -1 for other way
rotated_image = cv2.warpAffine(img, rotation_matrix, (width, height))
```

Selecting a region of the image (or cropping to it) can be done using NumPy slicing. The first two indexes in the image array are the row and column indexes, so the slice `img[r1:r2, c1:c2]` selects the region where the rows `r1` to `r2` intersect with the columns `c1` to `c2`. This selection only refers to the region of interest. Simply assigning that region to a new variable like this: `cropped = img[r1:r2, c1:c2]` just makes `cropped` point at the selected pixels in `img`. To extracted a cropped copy, use:

```
cropped = img[r1:r2, c1:c2].copy()
```

Detecting Edges

A common part of analysing an image involves finding the edges of the various objects in the image. This process is known as edge detection and its goal is to produce a set of points that lie around the periphery of an object. It works by looking for steep changes in brightness. A popular method for finding edges in an image is known as the Canny edge detector, named after John F. Canny, its inventor. It works by first removing noise from the image, which

Figure 1.6: An image of a parrot and the result of running a Canny edge detection algorithm on the image.

reduces the risk of finding lots of little spurious edges. It then calculates the brightness gradient across the image in both the vertical and horizontal directions. The algorithm looks for thin edges, so next it applies what is known as non-maximum suppression, which means it looks around each possible edge for any that are stronger and removes all but the largest gradient value. Finally, it applies two thresholds. Any edges that are weaker than the smaller threshold are removed. Any that are above it are kept, and any in between are only kept if they are connected to edges that have been kept.

Here is an example of using the Canny edge detector in Python. You can try it by loading an image into `img` and then showing it on the screen, as in the previous examples. The three arguments are the lower and upper threshold, and the size of the aperture used to look for gradients. You can experiment with these three arguments to try and get a better result.

```
edges = cv2.Canny(img, 100, 200, aperture_size=3)
```

The image that you will see as a result of edge detection will have a black background with white lines marking the object edges, much like the example in figure 1.6. This means the edges are just a series of pixel locations that fall on an edge. You are still left with the challenge of what to do with those locations. The simplest thing is to locate all the straight lines, which is described next.

Detecting Lines

If you just want to detect straight lines in an image (rather than complex shaped edges), you can produce a more usable result using using a **Hough transform**. This process produces the coordinates of all the straight lines found in an image. It works on a binary image, so you need to threshold or use Canny edge detection first. Producing the set of lines is done using the `cv2.HoughLines` function, as shown below. This returns a list of lines defined in polar coordinates by two parameters that by convention are named rho and theta. They do not define lengths of the lines, only their location and direction in polar coordinates. Rho is the distance of the line from the origin and theta is the angle of the line. The code required to find the lines and covert their parameters to lines plotted on the image is given below. Note that we set an arbitrary length for the lines (500 in the code below).

```
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)
lines = cv2.HoughLines(edges,1,np.pi/180,200)

print("Found",len(lines),"lines")

for line in lines:
    for rho,theta in line:
        a = np.cos(theta)
        b = np.sin(theta)
```

```
        x0 = a*rho
        y0 = b*rho
        x1 = int(x0 + 500*(-b))
        y1 = int(y0 + 500*(a))
        x2 = int(x0 - 500*(-b))
        y2 = int(y0 - 500*(a))
        cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

plt.imshow(img)
```

Detecting Faces

Face detection means locating all the faces in an image (as opposed to face recognition, which involves putting names to faces and is more complicated). Face detection is often used in social media to offer users a chance to tag people in images. Face detection can be performed in Python using something known as Haar cascades [2]. This is a machine learning approach to face detection and requires a lot of training data. Fortunately, OpenCV has a set of pre-trained Haar cascade models you can use. The cascades are stored in XML files and there are pre-built models for detecting faces, eyes and smiles. You need to load the cascade file by name:

```
face_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
```

and then apply it to a gray level image. The function `detectMultiScale` returns a list of rectangles that define the locations of the faces it has found. The following code iterates over that list and draws the rectangles onto the image.

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
cv.imshow('Faces',img)
```

Haar cascades can be trained to detect other types of image—they are not just for faces. The default XML Haar models that come with OpenCV include detectors for faces, cats' faces (really), eyes, smiles, bodies and licence plates. If you search online (Github, for example) you may find somebody has already trained a cascade for the thing you want to detect. If not, you can train one yourself using OpenCV's cascade classifier training tools. They are outside the scope of this chapter, however.

1.3.19 Natural Language Processing

A lot of the data produced every day is in the form of natural language, which means language as humans speak it, rather than computer languages like Python. Computers have traditionally struggled to understand natural language with its subtleties, complexities, jokes, sarcasm and slang. There are things that computers can do with natural language, however, and this section describes some of them. We will be using a Python package called Natural Language Tool Kit (NLTK). As with OpenCV for computer vision, NLTK can do both the mundane such as searching or counting words and the more interesting, such as gauging the sentiment of some written text or summarising the main concepts in a document.

Data Sources

We will need some interesting texts to process, and NLTK can help. It has easy access to a number of text sources including books from the Gutenberg project, US presidential addresses, web chat, and news from Reuters. NLTK includes an application you can run to browse and download data from its corpora. After importing NLTK, type `nltk.download()` to run the download application (it runs in a new window, so check it is not open behind your browser if you are using Jupyter!). Some of the corpora are large, so choose carefully what you download. Some of the NLTK functionality also requires you to download a corpus by name. You can download them directly in your Python script. We will highlight these as we use them in the text that follows. Note that once they are downloaded, you do not need to run the line of code that downloads them again. If you try to use a resource that needs to be

downloaded, the error given by NLTK will helpfully tell you what to download. You can pass an additional argument to `nltk.download()` to specify where the resource is to be saved on your computer. You might want to do this to avoid large amounts of data going in a Windows roaming profile, for example. Here is an example that downloads the punctuation tokenization resource to a given folder:

```
nltk.download("punkt",download_dir="C:\\Users\\me\\.jupyter")
```

You can also find lots of useful sources of text online. There are a number of useful packages available for extracting data directly from websites, including Scrapy and Beautiful Soup.

Tokenization

In it's simplest form, a piece of text is represented as a string. In that form, there is very little you can do to analyse it. The first step is to split it into shorter parts—usually words or sentences. This process is known as **tokenization** and the parts are known as tokens. You could just split text into words by splitting on spaces, but that would leave some words with punctuation attached. A better approach is to use the NLTK tokenization methods. Here is a simple example, where we define a string to contain two sentences and then tokenize it. The first time we do this, we also need to download the punctuation corpus from NLTK with `nltk.download('punkt')`.

```
import nltk
from nltk import tokenize
nltk.download('punkt') # Only needed if you haven't downloaded it already

s = "This is the first, short sentence. Now this is the
     second one, which isn't so short."
word_list = tokenize.word_tokenize(s)
print(word_list)
```

Counting Words, Bigrams and Trigrams

Having split a document into words, the frequency of occurrence of each word can be counted. For this, we use a method called `counter` from the `collections` package, which takes a list of strings and produces a dictionary where the keys are strings and the values are counts.

```
from collections import Counter

counts = Counter(word_list)
print(counts)
```

Run this code on the sentences defined above and look at the output. The list of counts has a few problems. It contains punctuation and the capitalisation means that the word 'this' is repeated. We can make the string lower case before we start like this `s = s.lower()`. To remove the punctuation, first define a list of what to remove, for example, `remove_list = ['.', ',', ':', ';']` and then extract everything else from the tokenized list:

```
new_list = [w in word_list where w not in remove_list]
```

You could also add words to the removal list to avoid counting words like *a* and *to*. These are known as **stop words** and NLTK provides a list of them so you don't have to think of them all yourself. You need to download the corpus called `stopwords` before you can use it. Here is the code required to remove stop words from a list called `word_list`:

```
from nltk.corpus import stopwords
nltk.download('stopwords') # Only needed once

stop_words = stopwords.words('english')
new_list = [word for word in word_list if word not in stop_words]
```

The variable `stop_words` in the previous code is a Python list, so you can append new words or punctuation to it if you want.

In many text analysis situations, pairs and triplets of words are interesting too. These are known as **bigrams** and **trigrams** respectively (and above 3, they are known as n-grams). You can extract all the bigrams from a sequence using `nltk.bigrams` and the trigrams using (you guessed it) `nltk.trigrams`. Both return generators so you should iterate over them or convert them to a list. As an exercise, produce all the bigrams from our sentences defined in the code above as `s` and then count their occurrences.

Parts of Speech Identification

Another step in text analysis involves identifying which words in a list are the subjects, objects, verbs, nouns and so on. This is known as parts of speech (POS) labelling. Done simply, it requires the following code:

```
tags = nltk.pos_tag(word_list)
print(tags)
```

This produces a list of tuples, one for each word in `word_list`, containing the word and a coded POS tag. The tags are abbreviations such as `NN` for singular noun and `VB` for verb.

Stemming and Lemmatization

When counting or classifying words, it is often desirable to count words that have a common root but have a different spelling due to what is known as inflection. This might be due to different tenses: walk, walking or walked; or a different person: I eat, he eats, for example. Reducing inflected words to a common root reduces the number of unique words a program needs to deal with and improves many types of analysis. For example, if you are searching a document for mentions of running, you should also search for run and ran.

There are two common methods for reducing words to a common root: stemming and Lemmatization (sometimes referred to as stemming and lemming, of course). Stemming is the faster and more crude of the two approaches and involves removing suffixes like *ing* and *ed* to produce a root. This works for *walking* and *walked* but not for *ran* or *awning*. Stemming can produce strings that are not valid words, but in some cases that doesn't matter as long as there is consistency. In other cases, you need to do it properly and find the true root word. A lemma is the dictionary definition version of a word, and lemmatization involves looking up each word to find that form. This works a lot better than stemming, but needs a large lookup table and is slower.

NLTK supports both stemming and lemmatization. It offers a choice of the Porter stemmer or the Lancaster stemmer (the Lancaster tends to produce a smaller final set of words as it is more aggressive in its stemming). Here is an example of using it to stem the word *running*.

```
from nltk.stem import PorterStemmer

ps = PorterStemmer()
print(ps.stem("running"))
```

Try the code above with *ran*—it remains unchanged. Now try the code below. It requires the NLTK WordNet corpus, so download that if you haven't done so already.

```
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()
nltk.download('wordnet')

wordnet_lemmatizer.lemmatize("ran")
```

Using `%%timeit` (see section 1.3.22) I found that the stemming example took an average of 7 milliseconds and the lemmatization example took on average 60 milliseconds on my laptop, As an exercise, write code to tokenize, then lemmatize a full sentence string and produce the word counts that result.

WordNet in NLTK

The lemmatization of the previous section made use of something called WordNet, which is a database of words, their meanings, and the relationships among them. The relationships form a network (hence the name) of words connected by meaning. WordNet can be used like a thesaurus for looking up synonyms, or like a dictionary for finding definitions, or (as above) for finding lemmas. Words are grouped into sets of synonyms, called **synsets**. Each synset has a set of lemmas (synonyms), antonyms, definitions and examples of the words in use. Any single word will have one or more synsets associated with it. NLTK has methods for accessing these WordNet synsets and their contents. Look up the synset for a word like this:

```
from nltk.corpus import wordnet
world_syns = wordnet.synsets('world')
```

We now have the synsets for the word *world* as a Python list of `Synset` objects and can access various parts of it as the following examples show:

```
print("All the definitions:")
for defn in (r.definition() for r in world_syns):
    print(defn)

planet_earth = world_syn[3]
print("An example of use of the 4th lemma:",planet_earth.lemmas()[0].name())
print (planet_earth.examples()[0])

# An example with a different word: good
ls = wordnet.synsets('good')
print(ls[1].definition())
ls[1].lemmas()[0].antonyms()
```

By following links to the various lemmas or antonyms associated with a word, you can traverse the WordNet network finding connected words with different meanings.

Sentiment Analysis

Every day, millions of people offer opinions on social media. They comment on products they have bought, films they have seen, the politics of the day, and what they had for lunch. Some of that content is of use to the companies who make the products or the political parties who want the votes. The problem is that there are too many posts for a human reader to process. The social media platforms themselves are under increasing pressure to ensure the safety of their users by removing things like hate speech, but the same problem exists—how do you gauge the sentiment of a piece of text automatically? This process is known as sentiment analysis, and you can do it in Python with NLTK.

NLTK has a relatively simple built-in sentiment analysis tool called VADER (Valence Aware Dictionary and sEntiment Reasoner). In linguistics, the valence of a word is its goodness or badness. VADER uses a lexicon that defines the valence of a list of words and a set of rules that were derived by asking human opinions on social media posts using Amazon Mechanical Turk. There is a paper describing the approach [1]. VADER is fast and does not require any training, which means it is easy to use but cannot be tuned to work on data from a particular source. It was designed to work best on social media data.

To use VADER in NLTK, import the module and create an instance of a `SentimentIntensityAnalyzer` and use it to call the `polarity_scores` method. This returns a dictionary with scores for negative, neutral, and positive sentiment plus an overall compound sentiment score from -1 to 1, with higher numbers indicating more positive sentiment.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

analyser = SentimentIntensityAnalyzer()
sentence = "I love Data Science"
score = analyser.polarity_scores(sentence)
print(score)
```

Try some different sentences for yourself. Some of the things that VADER uses to judge the sentiment include punctuation (and exclamation mark increases the magnitude of a score!), capitalisation (CAPITALS also increase the score magnitude), words like *extremely*, and negations within three words (this is NOT good!).

1.3.20 Object Oriented Python

At the start of this chapter we discuss how different data structures could be represented. Object oriented programming (OOP) provides a way of defining arbitrary data structures along with methods for manipulating the data they contain. Its allows you to represent things in the real world and the relationships among them. The key concept in OOP is the **class**, which defines the structure and methods of the objects that belong to that class.

There is an important distinction between a class and the objects that belong to that class. A class defines the attributes (variables) and methods that objects belonging to that class possess. Each class is defined once but there can be many instances of objects that belong to the class. A class is what we called an *entity type* at the start of the chapter, and an object is what we called an *entity*.

We might define a class to store data about a company's customers, for example. The class might be called `Customer` and we could then instantiate variables that are of the `Customer` type: `cust1` and `cust2`, for example. Notice how we have defined the class name with a capital letter at the start, but the instances have lower case names. Convention is to use CamelCase with a capital at the start for class names.

A class has variables, which are referred to as **attributes** when they are in a class and functions, which are referred to as **methods**. A customer class could have attributes such as name and email address and methods to do things like calculate total spend. When you create an instance of a class, you provide the values for the attributes. These are passed to a method called `__init__` (there are two underscores before and after the word init), which assigns them to the attributes in the instance.

```
class Customer:
    def __init__(self,name,email,purchases):
        self.name = name
        self.email = email
        self.purchases = purchases

    def total_purchases(self):
        return sum(self.purchases)
```

To create a new instance of a customer, we can use:

```
cust1 = Customer('Kevin', 'kms@cs.stir.ac.uk', [45, 70])
```

We can now operate on this customer instance using the attributes and methods that belong to its class. We can call the method to calculate the total purchases like this:

```
total = cust1.total_purchases()
```

and access an attribute like this:

```
print("This customer is called", cust1.name)
```

The methods and attributes are accessed using dot notation: `obj.attribute_name` or `obj.method`. Inside the method, the attributes are referred to using `self.attribute_name`. This word `self` specifies that the variable in question is a member of the object. That is why we can use the same name as an argument, for example, and write the line `self.name = name`. Note also that `self` is the first argument in the method definition, but that it is not passed to the method when it is called. The identity of `self` is taken from the name of the object before the dot (`cust1` in the examples above).

Now look at the `__init__()` method. You do not call this method directly yourself. It is called when you create a new instance of this class. It is an example of a so called **magic** method in Python—that is a method that is called automatically in the right circumstance rather than by name. This means the method must be called `__init__`. You cannot choose a different name for it. The `__init__` method can also set default values for any attributes that are not passed as arguments by simply assigning their value in the method. You can also put default values in the argument list of `__init__`. Here is an example that does both. It allows a product list to be sent or, if it is missing, sets it to an empty list, but it sets the attribute `points` to zero and does not allow an argument to change that.

```
class Cust:
    def __init__(self, name,email, purchases=[]):
        self.name = name
        self.email = email
        self.purchases = purchases
        self.points = 0
```

When the method `total_purchases()` is called, no arguments are needed (in this case, at least). The function finds the values it needs in the member attribute, `purchases`. Of course, you can define methods that accept additional arguments if you like.

We can define as many customers as we like and their values will be kept separate. We could define a list of customers too, for example. When referring to an instance by its variable name (like `cust1`) we are using a reference. This means that assigning one object to be equal to another only copies the reference, not the whole object. It also means that comparisons with double equals (`a == b`) are true if `a` and `b` refer to the same object rather than being true if `a` and `b` contain the same values in all their attributes. Try the following:

```
cust1 = Customer('Kevin', 'kms@cs.stir.ac.uk', [45, 70])
cust2 = cust1
print(cust2)           # Will print details we set for cust1
print(cust1 == cust2)  # True
cust1.name = "Sally"
print(cust2)           # The value for cust2 changed too
```

In the code above, we saw that as `cust2` is just a reference to `cust1`, the comparison returned `True` and a change in one was reflected in the values of the other. Now try this:

```
cust1 = Customer('Kevin', 'kms@cs.stir.ac.uk', [45, 70])
cust2 = Customer('Kevin', 'kms@cs.stir.ac.uk', [45, 70])
print(cust1 == cust2)  # False
```

Now we see that `cust1` does not equal `cust2`, even though they are identical, because they refer to different instances. That might be what you want, but you might want the comparison to actually look at the values in the instances. You can do that, but you must decide and implement what the comparison means. For the sake of argument, let us say we want define a comparison on total purchase values alone. We can define that to be the case using some other examples of magic methods. This time they are called `__eq__` and `__ne__`. In the following example, the code given should be inserted into the class definition

```
def __eq__(self, other):
    return self.total_purchases() == other.total_purchases()

def __ne__(self, other):
    return self.total_purchases() != other.total_purchases()
```

Now a comparison with `if cust1==cust2` will return true if the total purchases have the same value. In the methods, `self` is the object on the left of the == and `other` is on the right. You can also write magic methods for other comparison types: `__gt__` is >, `__lt__` is < `__ge__` is >= and `__le__` is <=.

You could, of course, write your own function called something like `compare_purchases`, which would be clearer to others reading your code. The difference between the magic methods and one like `compare_purchases` is that the magic ones have a specific name and are used in a specific way (when == is used, for example) and normal methods are just called by name, for example `if cust1.compare_purchases(cust2)` ....

References

Objects can be related to other objects in a number of ways. Relationships between objects can be defined by allowing one object to refer to another. Let us expand our customer example by defining a simple product object:

```
class Product:
    def __init__(self, name, price):
```

```
        self.name = name
        self.price = price
```

The customer purchases array will now be an array of products, rather than prices. We instantiate some products and a customer who has bought them:

```
table = Product("Table", 450)
vase = Product("Vase", 80)
chair = Product("Chair", 170)
cust1 = Customer("Kevin", "kms@cs.stir.ac.uk", [table, vase, chair])
```

The list of products is, of course, a list of references, so if the product objects change, that change is reflected in the array of products linked to a customer. Objects are mutable, so the ID of an object stays the same when its attribute values change. One final change is needed—the `total_purchases` method needs to be fixed so that it sums the price attributes in the list. I'll leave that as an exercise for you.

1.3.21 Inheritance

Another way in which one class can relate to another is by inheriting and expanding on its attributes and methods. Consider the `product` class again. Every product has some attributes in common: price, code, and quantity in stock. There are different types of product too: tables, chairs, lamps etc. You could add an attribute called **`type`** to record this, but each product type has different attributes, so a single object for all products would get messy (you do not want to store the bulb fitting type for a table). The solution is for each product to inherit attributes from the parent class and then add its own specific attributes. For example, a lamp would have bulb type and wattage and a sofa would record the number of seats. The code below achieves this:

```
class Product:
    def __init__(self, price, code, quantity):
        self.price = price
        self.code = code
        self.quantity = quantity

class Lamp(Product):
    def __init__(self, price, code, quantity, bulb, watts):
        Product.__init__(self, price, code, quantity)
        self.bulb = bulb
        self.watts = watts

my_lamp = Lamp(79, 'SK4344', 100, 'screw', 60)
print(my_lamp.__dict__)
```

Take a look at the code above. The fact that the `Lamp` class inherits from the `Product` class is specified by naming the `Product` class in brackets in the `Lamp` definition like this: **`class`** `Lamp(Product)`. The `__init__` method in `Lamp` accepts arguments for both the parent class (`Product`) and itself. It calls the `Product` `__init__` method (note that it has to include `self` as the first argument) with the appropriate arguments.

Finally, we use the `__dict__` member of `Lamp`, which is another magic attribute, to see the object represented as a Python dictionary. Python even allows an object to inherit from more than one parent class at the same time.

When should you use an object oriented approach, and why? Keeping all the attributes and methods associated with an object together can make the code easier to read and reuse. There are well established methods for designing classes and hierarchies from descriptions of real world objects (like our furniture example). These help you share code with others in a well organised way. You may find that small scripts that are used once or twice do not require an object oriented approach, but that larger projects with a longer lifespan can benefit from them greatly.

1.3.22 Timing and Efficiency

How fast are Python programs? It depends on how you write them. For example, NumPy arrays are optimised to perform certain operations, and the code is actually written in C (which is very fast). The trick to writing fast Python

code is to use methods from classes like NumPy that can make use of that C implementation, rather than doing things like looping in Python. So, when you are manipulating large quantities of data with Python, make sure you use the correct package (like NumPy) and try to use built in methods. The same is true for Pandas, but to a lesser extent and NumPy will almost always be faster than Pandas.

As a general rule, you want the iteration over arrays to be done by the C code behind the scenes, not by interpreted Python. Ideally, you want to be operating on whole arrays at once. This is what both NumPy and Pandas are optimised to do. Here are a couple of illustrative examples that use a nice feature of Jupyter notebooks, the %%`timeit` command, which runs the entire cell several times and reports the mean and standard deviation of the time taken to execute the code.

First of all, we define a NumPy array with a million entries and a function to standardise its values (scale them so they have a mean of zero and a standard deviation of 1).

```
mill = np.random.random(1000000)
m = np.mean(mill)
s = np.std(mill)

def standardise(x, m, s):
    return (x-m) / s
```

Now we want to apply our function to the whole array. First we try with list comprehension

```
%%timeit

st = [standardise(x, m, s) for x in mill]
```

which (on my laptop) reported an average of 1.13 seconds for the million calls to `standardise`. Now we do it properly and call `standardise` with the whole array at once:

```
%%timeit

st = standardise(mill, m, s)
```

This time I get an average of 26 ms, which is a massive improvement. You can time sections of code more generally using the `time` package. Here is a simple example, which will report the elapsed time in seconds between the line where `t` is defined and where `elapsed_time` is calculated.

```
import time

t = time.process_time()
for i in range(5):
    st = [standardise(x, m, s) for x in mill]
elapsed_time = time.process_time() - t
print(elapsed_time)
```

As you learn to code, get into the habit of trying different ways of solving the same problem and timing the resulting code. This will start to give you a good idea of which approaches are the fastest.

1.4 Conclusion

There is a lot more to Python than can be covered in one short chapter, but I hope I've got you started with this fascinating and flexible language. If so, there are plenty of online resources that you can turn to for more details. After that, all you need is practice. Set yourself some challenges and try to code the solutions in the most efficient way. Compare different approaches, look online for ways to do things—places like Stack Overflow and Quora are great resources for finding the best way to solve a problem. Most of all, have fun. Coding a good solution is rewarding and creative, even if the journey towards it is sometimes frustrating.

Bibliography

[1] Clayton J Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth international AAAI conference on weblogs and social media*, 2014.

[2] Paul Viola, Michael Jones, et al. Robust real-time object detection. *International journal of computer vision*, 4(34-47):4, 2001.