ADRIAN F. CLARK

CE705: PROGRAMMING IN PYTHON

COMPUTER SCIENCE AND ELECTRONIC ENGINEERING
UNIVERSITY OF ESSEX

Contents

Preface

These notes accompany a module in which postgraduate students learn how to program using the Python language. It concentrates on Python version 3, as there were some syntax changes between versions 2 and 3 of the language.

The text is not a complete introduction to the language, it is a place from which learning starts: students are expected to use the extensive online resources on the language to fill in the gaps in what is written here. The introduction to the basics of the language is pretty rapid (but the audience is *postgraduate* students) and glosses over the minutiae of the Python syntax as these should be clear from the example code. Comments on these notes, and feedback in general, are welcomed.

The material in these notes is not lectured; the principles and examples are intended to be demonstrated and discussed in interactive sessions in a software laboratory, so that students can try things out for themselves as they are explained. The intention is that students gain practical programming skills that can be applied in their project work. For this reason, there is a concentration on presenting complete, working programs throughout the notes. Moreover, these programs are well-presented and commented according to good programming practice, so that it is clear how programs should be presented and documented when students come to do assignments.

Most introductions to programming use a variety of examples, so they are suitable for readers with a wide range of backgrounds. The majority of students taking this module are following MSc schemes that involve numerical work, so the emphasis here is on processing data and performing numerical computations. (In any case, processing real-world data is more interesting than learning how to sort an array of numbers in 100 different ways, or how to interface to a database of products on yet another shopping website.) In particular, the numerical and scientific Python modules `numpy` and `scipy` are introduced, as are graph-plotting facilities.

1
Getting Started

1.1 Introduction

The module aims to teach you two things:

How to program. The aim is to give you the ability to program medium-sized programs without too much trouble, and give you insights into how to write larger ones. Most of the examples we look at will involve processing numerical data.

The Python language. Python is an interpreted language with an easy-to-learn syntax; it is fairly high-level and has many modern facilities. It is easily portable between operating systems, and has become astonishingly widely used in all areas of science and engineering. There are also many add-ons that provide additional functionality; we shall look at a couple of these later in the module. I do the vast majority of my research in Python, resorting to compiled languages (usually C) only when the code needs to run really quickly.

You should be aware that **there are two versions of Python** in widespread use, v2.7 and v3.6, and they have slightly different syntax — you can easily distinguish them by looking at the `print` statements. I'm going to teach you v3.6 because that is what will be used from now on; but I use v2.7 in my research and may often slip into using that syntax — the interpreter will keep me right, and I'm sure you will too.

The module is **presented as a series of three-hour sessions**. I'll give demonstrations and speak for some of the time; but for the remainder of the sessions, you'll be writing programs — it's very much a hands-on module. During the early sessions in particular, you will be writing only very short programs. The aim is for everyone to get them working before the group as a whole moves on. I will help you out when you're struggling and try to make sure you understand what your program is doing.

If you have some programming experience already, you may find that you are finished well before the end of the allotted time — in which case, you're welcome to leave early. However, note that I keep a record of who has shown what to me, so **don't leave without showing me what you have done**. People who don't finish in the allotted time should try to finish their work in their own time and show me what they have done in the next session.

As you work through the various exercises during the course, you can probably find solutions to them on the Web and simply paste them into your editor — but I strongly discourage you from doing this as **entering the code with your own fingers** and learning how to overcome syntax and run-time errors is **a vital programming skill**, one that you can only learn by practice. In the long run, you're only cheating yourself by using other people's code.

I *don't* recommend that you buy a textbook for this module — there are plenty of websites that purport to teach Python (though, to be honest, most of them are not all that good). There is also good formal documentation for the language and, more importantly, its library online. If you think you need a book to help you on your way, I suggest you look at some in the Library to find one that you understand, and then buy that — just make sure it describes Python 3.

You'll do all your **program development under Linux**. There are two reasons for this. Firstly, I want you to gain a feel for what the computer is doing, and using Linux from the command line is the best way I know for achieving this. Sophisticated development tools such as Eclipse and `PyCharm` are excellent; but you will be all the more effective as a programmer if you understand what they are doing for you and, should it be necessary, can do them yourself. Secondly, it will be helpful for your career in the long term if you have some familiarity of the Unix environment in general and Linux in particular.

All the programs you write should yield *identical* results under Linux, Windows and MacOS. You can also install the same program development environment on your own hardware, but note that you *must* be able to develop, run and demonstrate your software under Linux because **this module will involve you delivering programs that work in the laboratory under Linux**.

1.2 Getting to grips with Linux

If you have worked only on Windows in the past, you will be used to starting an application and working within it for doing everything associated with a task; for example, Word for preparing documents, Visual Studio for software development, and so on. Although it is possible to work in this way under Linux, that approach is not typical — the original Unix philisophy was for each program to **do one thing but do it well**. It is normal under Unix (*e.g.,* Linux or MacOS X) to have **several programs running on different parts of your computer's display** at one. A fairly typical screen layout for program development is shown in Figure 1.1, featuring one editor window, a separate terminal window for running programs, and a web browser for looking at reference material. Moving from task to task just involves clicking in the relevant window. Indeed, under Linux you can configure the window manager to switch automatically to whichever window the mouse cursor moves to ("focus follows mouse") and automatically bring it to the front ("auto-raise").

The first thing you need to do is **bring up a terminal window**. You can do this by selecting from menus *etc* but the easiest way is to type the

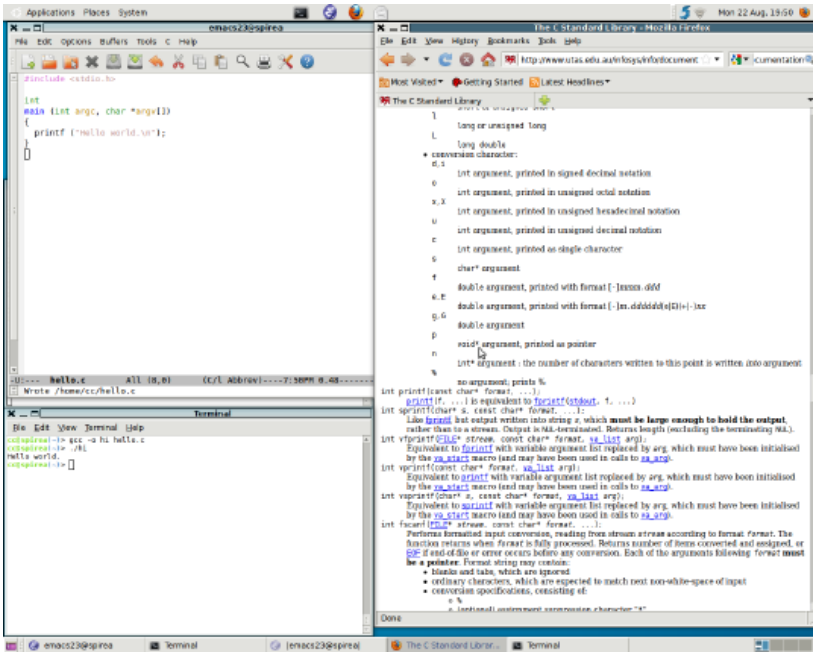

Figure 1.1: Typical screen layout under Linux

keystroke `Ctrl-Alt-t`. You will then **enter commands in the terminal window** to do the things you need to do. As with Windows, you *can* do most things through the window manager but you will become more proficient more quickly by learning the commands that the window manager is executing on your behalf.

If you're familiar with Windows, you'll know that files can be stored in folders, and that the components of a filename are separated with the backslash character. Hence, a full filename might be something like `C:\alien\hello.py`. Filenames on Linux (and MacOS) are similar in concept though the syntax is different: a full filename might be something like `/home/alien/hello.py`. You will see that it uses *forward* slashes rather than backward ones and that there is no disk name (there is a more elegant way of integrating disks into the filesystem in Linux).

When you login on Linux, you can think of yourself as being 'in' your **home directory** (*directory* is the Unix jargon for folder). Typing the command

```
pwd
```

(for 'print working directory') into your terminal window will make your **shell** (the program that interprets your commands, usually `bash` these days) print it out.

To create a directory, use the `mkdir` command. You can then 'move' into it:

```
mkdir ce705
cd ce705
pwd
```

The command to list the contents of a directory is `ls`. For an empty directory such as `ce705` in the above example, `ls` returns nothing. This is

a common characteristic of many Unix commands, which tend to avoid extraneous output. Let us create a file and then use `ls` again:

```
touch TEMP
ls
```

You will see a single line of output containing the filename `TEMP`. (The `touch` command simply opens and closes any files provided on the command line, which is a simple way of updating its modification time, or creating an empty file if one is needed.) You might like to see more detail than simply the filename, and providing the **command qualifier** `-l` (for *long*) does this:

```
ls -l
```

You should see a line of output like

```
-rw-r--r--  1 alien  staff  0 10 Jul 09:13 TEMP
```

The first "word" summarizes the access permissions of the file: the creator has read and write access to it, other members of the creator's group has read access to it, and the system manager has read access to it. (You can create files that the system manager cannot read, but then they won't be backed up. The system manager can 'become' you to read them anyway, so there is no hiding place.) The user who owns the file is `alien`, and he is a member of the group `staff`. The file contains `0` bytes of text, and it was created on 10$^{\underline{\text{th}}}$ July this year at 9:13 a.m.

Typing the full names of commands and filenames quickly becomes tedious, so your shell will probably be set up to provide **tab completion**. Whenever you type a command or filename, you can type the first letter or two of it and then hit the TAB key: the shell will complete as much of the filename as it can and output a list of the possible completions.

To **delete a file**, use the `rm` (remove) command

```
rm TEMP
```

If you're a newbie, things might be set up so that you are asked if you really do want to delete the file. Experienced users turn this feature off (and figuring out how to turn it off is one way in which you become an experienced user).

There are a number of **shorthands for directories**. Firstly, your current directory can always be referred to as `.` (a single dot). The directory immediately above the current one in the directory hierarchy is always `..` (two dots). Your login or home directory is always `~` (pronounced *tilde* or *twiddles*), and the login directory of the user `joe` is `~joe`. So when you've been working in some weird corner of your files on some topic, typing a command like

```
cd ~/ce705
```

will put you in the directory you created earlier.

As an aside, you will see when I give demonstrations that I have set things up so that my working directory is part of the prompt issued by `bash`. This is because I do different parts of my work in different directory trees; I imagine you organise your files in a similar way.

The final feature of the Unix shells that it is good to know about is **re-direction and pipes**. Let us create a few files in our directory

```
touch f1 f2 f3 f4 f5 f6 f7
```

How could we count how many files are in it? For this small a number, we could do it by hand but it would be good if we could get the computer to do it for us. Unix provides commands for lots of useful tasks, and one such program is `wc` (word count), which can actually count lines, words and characters. So if we can pass the output from `ls` to `wc`, we can have the machine do the work for us.

The first step is to save the output of `ls` in a file

```
ls > LISTING
```

This runs the `ls` command as usual but tells the shell to **save the output** in the file called `LISTING` (which should not exist). Note carefully the direction in which the angle-bracket is pointing: it is *to* the filename. In Unix jargon, we have **re-directed the output** to the file. You can look at the contents of the file by typing the command

```
cat LISTING
```

The `cat` (con**cat**enate) program copies the contents of the file to your screen. You will see that there is a line in the file for each file in your directory, so counting the number of files is the same as counting the number of lines in `LISTING`. To count the number of lines in `LISTING`, we can pass it to the `wc` command:

```
wc -l < LISTING
```

The `-l` qualifier tells `wc` to output only the number of **l**ines. Note that the angle bracket points **from the file to the program**, the opposite way to when we created `LISTING`.

This leaves the file `LISTING` in our directory and is generally a bit cumbersome to do, so the shell provides a more elegant way to achieve the same thing:

```
ls | wc -l
```

The vertical bar symbol is known as a **pipe** because it connects the output of one command to the input of another. You can connect as many programs together using pipes as you need. There are command-line programs for doing all sorts of things: selecting files that match a particular pattern, sorting them, removing duplicates, and so on. People used to write complete programs in the shell (so-called *shell scripts*) but they are now increasingly written in languages such as Python.

This introduction **just touches the surface** of the commands and capabilities of Linux, giving you enough to get started writing and running programs. There are a few fuller introductions on the web, such as `http://www.ee.surrey.ac.uk/Teaching/Unix/`, written by Michael Stonebank at the University of Surrey, and `http://linuxcommand.org` — there are many others too. There are also books in the Library that discuss Linux in particular and Unix in general.

1.3 Program development under Linux

Although there are many editors on Linux systems, and a few integrated development environments (IDEs) such as Eclipse and PyCharm, **it is recommended that you use Emacs** for this module. Emacs has been around a long time but provides many features that are programmer-friendly: syntax-awareness, brace-highlighting, incremental search, symbolic debugging, managing compilation, and so on.

You start Emacs by opening up a terminal window and typing

```
emacs &
```

Note the ampersand at the end of the line: it tells the shell to run it in the background so that you can continue typing commands in the window.

When the Emacs window comes up, please resist the temptation to make it fill the screen. Feel free to make the window longer but it's wise not to make it wider: source code is best presented in lines of no longer than 80 columns, the initial width of the Emacs window. In fact, for your assignment **I require lines of your program to be no longer than 80 characters**.

You can use Emacs in the way you're used to when using Windows, by pulling down menus or clicking on buttons. However, as you become more familiar with it, you'll work more quickly if you learn the keyboard shortcuts — I work entirely through shortcuts. Whichever way you work, you will need to type Emacs commands from time to time, and you bring up the command prompt by typing (in Emacs terminology) `M-x` ("meta-X"). On a PC keyboard, this is `Alt-X` (hold down the `Alt` key, type `x` simultaneously).

In utilities such as Word, you'll be used to entering the text you want and then saving it to a file. With Emacs, it's better if you do things in a different order: first edit the file you want — the keystrokes for this are `C-x C-f` (control-X, control-F) — then type the name of the file you want in answer to the prompt. If you **end the filename in `.py`**, Emacs will switch into Python "mode" in which it understands the syntax of the lines you type and that helps you get your code right.

Every time you save a file in Emacs, **it creates a backup version of the file** (by appending the ~ character to the filename) which contains the file's content before you started editing it — and this applies to all files edited by Emacs, not just program source. This is a really useful feature: I can't tell you the number of times I have reverted to a backup file when later edits have gone horribly wrong.

1.4 Your first program

It has become a custom that your first program in any language should simply print out the phrase "Hello, world." Here it is in Python:

```
print ("Hello, world.")
```

The program consists of a single line, an invocation of the `print` function with a string. In Python v2, `print` is a statement rather than a function so the brackets around the string would be missing.

The easiest way to run this program is to start the Python interpreter, which you do by typing the command

```
python3
```

(The `python` command runs v2 of the interpreter.) The interpreter will print out some messages and then prompt you with

```
>>>
```

Type the `print` statement above in response to the prompt and run it by hitting the return key. Using the Python interpreter interactively is a good way of trying things out when you're not sure of a language or library feature. There are even enhanced Python interpreters such as iPython. However, using the interpreter interactively becomes increasingly pesky as the length of your program increases, so it is best to store complete programs in files and run them through the interpreter — this is how we shall work for the entire module.

If you enter the "Hello, world." program in the file `hello.py` using Emacs, you can run it by typing the command

```
python3 hello.py
```

in a terminal window.

Everything is Unicode in Python, so there is nothing special to do if you want to put Unicode characters in strings. If you haven't encountered the term before, Unicode is the modern way of representing characters on computers: it supports not just the western alphabet but also accented characters and characters derived from other alphabets — Chinese, Thai, and so on.

You can quote a string using double quotes, as here, or single quotes — this gives you an easy way of printing out a string containing one of these types of quote but you can also use the syntax

```
"a \" within a quote"
```

I have rarely had to do this. There are some other ways of writing strings which we'll come to shortly.

1.5 Converting temperatures

To get us going on a more interesting problem, let us write a program to convert temperatures from Fahrenheit to Celsius. The Fahrenheit scale has its freezing point at 32 degrees and its boiling point at 212 degrees, and this maps onto the familiar 100-degree Celsius scale. To convert a Fahrenheit temperature f to Celsius c, the equation is therefore

$$c = \frac{f-32}{212-32} \times 100$$

which simplifies to

$$c = \frac{f-32}{1.8}$$

A complete program that performs this conversion is:

```
f = 97.8
c = (f - 32) / 1.8
print (f, c)
```

The first line of this program stores the value 97.8 in a *variable* called `f`. Variables are regions of computer memory in which numbers can be stored. A convenient way to imagine memory is as named pigeon holes (Figure 1.2), with each pigeon hole being able to store a single number. Figure 1.3 shows a diagrammatic representation of a long line of pigeon holes after the first line of the program has been executed, so that the value 98.7 is in the pigeon hole `f` and the pigeon hole `c` has no value.

Figure 1.2: Pigeon holes for holding letters

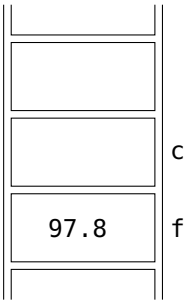

Figure 1.3: You can think of computer memory as a long line of pigeon holes

The second line calculates the Celsius temperature corresponding to the value in `f` and stores it in `c` — this line of code is almost a direct copy of the equation given above, though note the use of brackets to ensure that the subtraction is done before the division.

The last line is a `print` statement, just as in our "Hello, world." program, but this time prints out the values of the variables `f` and `c`.

Variable names in Python can be of any number of characters and may contain letters (including underscore) and digits, though the first character must be a letter. Upper- and lower-case letters are considered as being different. Most programmers use longer variable names than the single-letter ones used here, as in `fahr_temp`, `fahrTemp`, or `fahrenheitTemperature` — though the author personally finds that really long variable names make programs less easy to understand. It is wise to avoid starting variable names with underscore as many Python programmers use such names for internal variables within modules (you will come to understand what this means later in these notes).

Incidentally, if you've programmed before and are used to programming languages in which variables have to be declared as being of a particular type, you'll find Python is a little different. Firstly, **nothing is declared**, though you get a run-time error if you read a variable before writing to it. Observe that `f` contains integer values but `c` ends up with floating-point ones; the Python interpreter takes care of the conversion (and gives a run-time error if you're trying to do something silly). In Python 3, dividing one integer by another can yield a floating-point number, unlike earlier versions of Python and most other programming languages.

1.6 Printing a temperature conversion table

In principle, one could modify the above program to hold whatever Fahrenheit temperature is required and run it to find the corresponding Celsius one, but this would be time-consuming and overly messy. It is probably more useful to have a program that prints out the conversion from Fahrenheit to Celsius for (say) every 10 degrees. As you would expect, this has some similarities to the program we have just considered:

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
```

```
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
for f in range (32, 220, 10):
    c = (f - 32) / 1.8
    print (f, c)
```

Let us start with the second line of the program and work through it — the meaning of the first line is explained a little later.

The second and third lines of the program form a quoted string but use three double-quotes as delimiters rather than one — this is to allow it to span several lines. (You can also use three single-quotes.) Why is it here? It describes the purpose of the program — for more substantial programs, it should be much longer, describing how the program is used, its algorithms, any restrictions, and so on. This type of in-program documentation is a mark of good programming, so it is a good habit to get into. To encourage this, you'l score higher in your assignment if you provide such a comment.

The next line is blank. **Blank lines help split up code into sections** and greatly aid readability. You will see that the code also has **spaces within lines to aid readability** — these are good habits to get into too.

The next couple of lines both start with # characters. These are *comments*: everything from the # to the end of the line is simply ignored by the Python interpreter. You can put comments at the ends of lines of code too. As in this case, you should make your **comments explain the purpose of sections of code** rather than describing what the individual lines do in words. As you might expect, clear comments are also a mark of good programming, and you are expected to comment your own programs well.

The next line, the one that starts with `for`, is a *loop*, and the following two lines are indented relative to this. These two lines are executed several times and are called the *body of the loop*; we see that they are the now familiar lines to convert Fahrenheit temperature to Celsius and print it out. Note that Python *requires* the body of the loop to be indented by two or more spaces relative to the `for`. With Emacs, simply typing the ⟨tab⟩ key at the beginning of each line is enough to indent the code by the right number of spaces — and Emacs understands the Python syntax, so the indentation comes out right if there are loops within loops.

The `for` statement causes the variable `f` to take each of the values returned by the function `range` in turn. The three arguments to `range` are the first value to be returned, the value at which the loop ends, and the amount added each time. Hence, the first time around the loop, `f` is set to 32, the second time around the loop it is set to $32 + 10 = 42$, the next time to $42 + 10 = 52$, and so on. The loop terminates when the value of `f` is greater than or equal to `220`. You will see that the statements within the loop cause `f` to be converted to `c` and printed out for each of these values of `f`. Running the program should yield the following output:

```
32 0.0
42 5.555555555555555
52 11.11111111111111
62 16.666666666666668
72 22.22222222222222
82 27.77777777777778
92 33.333333333333336
```

```
102 38.888888888888886
112 44.44444444444444
122 50.0
132 55.55555555555556
142 61.11111111111111
152 66.66666666666667
162 72.22222222222221
172 77.77777777777777
182 83.33333333333333
192 88.88888888888889
202 94.44444444444444
212 100.0
```

which is a bit ugly but correct. We shall look shortly at making this output prettier.

That leaves us only one line of the program to consider. **The first line of the program is a Unix-ism** which allows you to make the program executable and then run it without having to precede its name with `python` in the shell. If, for example, you enter the temperature conversion code (correctly) into the file `temptable.py`, typing the command

```
chmod +x temptable.py
```

allows you to run it by typing

```
./temptable.py
```

rather than

```
python3 temptable.py
```

The `chmod` command need be typed only once as it changes the permissions of the file, which you can check with `ls -l`. Also, note that under Unix you do not *have* to add the filetype `.py` to files containing Python programs, though you will have to tell Emacs to create them in Python mode by typing

```
M-x python-mode
```

when you first edit it.

Exercise: length conversion. When you have typed in the temperature conversion program and shown that it works, try writing a program that converts feet and inches to metres. You will need to know that 1 inch is 2.54 cm and that there are 12 inches in a foot. Start with one inch and print out the conversion in one-inch increments up to 3 feet.

1.7 Conditionals

Having gained some familiarity with the fundamentals of Python, it is a good time to get to grips with its main flow-control mechanisms. In fact, you have already encountered one of these, `for`. The next step up in complexity is conditionals, and we shall look at these in the context of the earlier temperature conversion table. We know that the freezing and boiling points of water in Fahrenheit are 32°F and 212°F respectively, so let us output these words alongside the appropriate lines:

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
for f in range (32, 220, 10):
    c = (f - 32) / 1.8
    print (f, c)
    # Indicate the freezing and boiling points of water (ugly code).
    if f == 32:
       print ("freezing point")
    if f == 212:
       print ("boiling point")
```

The == operator is read as "is equal to". All the supported comprison operators are shown in Figure 1.4. The `if` statements are executed each time around the loop but each of them succeeds precisely once. Note the indentation of the `print` calls relative to the `if`s. There are two important points to bring out at this juncture. The first is that we are not printing the text out *alongside* the conversion but rather on a line of its own. We can fix this in several different ways; first, we shall look at an easy one but then consider a more elegant one.

operator	meaning
==	is equal to
!=	is not equal to
>=	is greater than or equal to
>	is greater than
<=	is less than or equal to
<	is less than

Figure 1.4: Comparison operators and their meanings

We can re-write the program as:

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
for f in range (32, 220, 10):
    c = (f - 32) / 1.8
    # Indicate the freezing and boiling points of water (quite ugly).
    remark = ""
    if f == 32:  remark = "freezing point"
    if f == 212: remark ="boiling point"
    print (f, c, remark)
```

You will see that the variable `remark` is first set to an empty *string* (series of characters) every time around the loop, and its value changed for certain values of `f`. This lacks elegance and also helps bring out the second point to be made: when the first `if` statement succeeds, the second one cannot possibly. This is a fairly common problem in programming and most programming languages provide a way around it. In Python, a more elegant solution is:

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
```

```
for f in range (32, 220, 10):
    c = (f - 32) / 1.8
    # Indicate the freezing and boiling points of water.
    if f == 32:
       remark = "freezing point"
    elif f == 212:
       remark ="boiling point"
    else:
       remark = ""
    print (f, c, remark)
```

The word `elif` is best read as "else if". Working through this, you will see that `remark` is set only once per iteration, and when one `if`, `elif` or `else` succeeds, the others are not executed. As will be clear from these two examples, the `elif` and `else` clauses are optional.

1.8 `while`, `break` and `continue` statements

`for` loops are good for controlling loops when the number of iterations required is known, which is often the case for numerical programs such as our temperature conversion table. However, they are not general enough for every eventuality. The most general looping construct in Python is the `while` loop, and the easiest way to understand it is to re-write the loop in the temperature conversion program to use one:

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
f = 32
while f < 220:
    c = (f - 32) / 1.8
    print (f, c)
    f = f + 10
```

In this case, the loop management is spread over three lines, which is why it's less elegant for this task: there are three places where the loop control can go wrong. Nevertheless, you will find that `while` loops are widely used in general programming.

There are two other statements that work with `while`, and again the best way to show them is with an example.

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table.
This program has a deliberate bug in it."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.  We do some funky things for
# f > 100 and when f == 52.
f = 32
while f < 220:
    if f > 100:
```

```
        break
    if f == 52:
        continue
    c = (f - 32) / 1.8
    print (f, c)
    f = f + 10
print ("That's all, folks!")
```

This does *not* print out the complete temperature conversion table that we have seen up to now. The first thing of note is the `break` statement that is executed when `f > 100`: this causes control to jump to the first statement following the loop, which here prints out "That's all, folks!" You will find `break` is especially useful when reading input.

The second statement of interest in this example of code is `continue`, which transfers control to the next iteration of the loop. In this program, *it introduces a bug* because the line `f = f + 10` is not executed and the program loops indefinitely (until you interrupt it by typing `Ctrl-C`). The `continue` statement is useful when you are working through a series of steps on data but, for some reason, there is one value for which the computation is not to be performed. We shall see examples of both `continue` and `break` in programs later in the module.

One final thing that is worth mentioning is that the line

```
f = f + 10
```

is commonly written in the form

```
f += 10
```

which you can read as "add 10 to `f`." You can do the same thing with the other arithmetic operators.

1.9 Some subtleties

There are two subtleties in the versions of the temperature conversion program that have been skipped over until now, and both have the same underlying cause. The first of these is that we have compared the Fahrenheit temperature when checking for the freezing and boiling points, not the Celsius one. This is because `f` holds an *integer* (a whole number) while `c` holds a *floating-point* (real number) value. **The representation of real numbers in a computer is not exact** — indeed, it could never be for irrational numbers such as $\pi, e, \sqrt{2}$ *etc*. Computation using floating-point numbers in Python gives about 15 decimal digits of precision, and the more computation you do, the less accurate your results become. (If you are also taking CE816, I'll explain why in that module.)

With this in mind, statements such as

```
if c == 100: remark = "boiling point"
```

are clearly dangerous. The two ways around this are

```
if round (c) == 100: remark = "boiling point"
```

which converts the real value in `c` to its nearest integer; and

```
if abs (c - 100) < 1.0e-5: remark = "boiling point"
```

which checks that the absolute difference between `c` and 100 is small enough. I tend to use the latter: although it is less readable, it allows me to change the amount of inaccuracy I am prepared to live with.

The second subtlety actually arises from the same cause, and that is that `range` can be invoked with integer values only — you will get an error if you say

```
for f in range (32.0, 220.0, 10.0):
```

If you were to make the second argument 212.0 rather than 220.0, the number of iterations around the loop *could* depend on the accuracy of your computer's arithmetic, and hence *could* give different results on different hardware. The designers of Python do not want this to happen and so they do not allow you to program it.

The last thing to discuss about the `temptable` program is its output: the number of decimal places in the output for `c` varies depending on its value, and the table would be **more readable if the values were all neatly aligned vertically**. This is fairly easy to achieve: all we need to do is tell Python how the values of `f` and `c` are to be **formatted**. We shall print `f` in a region that is three characters wide. We shall restrict `c` to two decimal places; as there can be as many as three characters before the decimal point and the decimal point itself is a character, we need a region that is six characters wide to display the Celsius temperature. The variable `remark` holds a string, so our `print` statement becomes

```
print ("%3d %6.2f %s" % (f, c, remark))
```

You will see that the parameter to `print` has three components: the first is a string which we call the *format string*, the second is `%`, Python's *format operator*, and the third one a series of variables in parentheses (round brackets). There must be exactly as many variables in parentheses as there are percent characters in the format string. Comparing the numbers in the format string with the preceding text, you will see what they mean. The `%d` means that the corresponding variable (`f` here) contains an integer (to be printed out as a **d**ecimal), the `%f` means it contains a **f**loating-point number, and the `%s` means it contains a **s**tring.

If you have programmed in C, C++, or scripting languages such as Perl or Tcl, you will find this construct familiar. It is, however, now deemed old-fashioned and Python 3 provides alternative ways for formatting data for output; you are encouraged to read about them. The reason you are being told about the older way is that it will make it easier for you to read code in other programming languages and to migrate from them to Python.

We end up with the following, our final version of the temperature table program.

```
#!/usr/bin/env python3
"""A program to print out a Fahrenheit to Celsius conversion table,
from freezing point to boiling point in 10-degree steps."""

# Cycle over the Fahrenheit values in 10-degree steps.  For each
# Fahrenheit temperature, calculate the corresponding Celsius one
# and print both out on the same line.
```

```
print ("Fahr  Cels")
for f in range (32, 220, 10):
    c = (f - 32) / 1.8
    # Indicate the freezing and boiling points of water.
    if f == 32:
       remark = "freezing point"
    elif f == 212:
       remark ="boiling point"
    else:
       remark = ""
    print ("%3d %6.2f %s" % (f, c, remark))
```

2
Getting started with numerical computing

As has been mentioned a few times already, the main emphasis in this module is on processing real-world data. In this chapter, we shall gather some experimental data and write code to process it. In doing so, we shall find that a good way to structure a program is to write functions (sometimes called *subroutines* or *modules*, though the latter has a different meaning in Python).

2.1 Timing a pendulum

There are many possible sources of experimental data that we could use but the one we shall focus on here is straightforward: the period of a pendulum, the time it takes for one to make a single swing and return to its starting position. It is said that Galileo used his pulse to time the swings of a pendulum while he was in church and then derived the mathematical formula that describes it; we shall follow his general approach, though we shall use more modern timing equipment than a person's pulse.

A typical pendulum consists of a weight, often called a *bob*, on the end of a string. The end of the string not attached to the bob is fixed. A few minutes' play with this type of pendulum shows that one with a longer string takes longer to swing than one with a shorter string, so our ultimate aim is to find a relationship between pendulum length and period.

Consider an experiment to time the period of a pendulum using everyday components. Constructing the pendulum itself is easy, just a piece of string wrapped around a heavy weight; and it can be fixed into a doorway using a drawing pin. Timing it is also easy as practically every smartphone has a stopwatch app.

Initially, it might seem sensible to time a single swing; but a few moments' thought should lead you to a different conclusion. It is difficult to start or stop a stopwatch (or stopwatch app) *exactly* as the pendulum starts its swing, so the measuring process is not exact. Timing a single swing will mean that the inaccuracy you introduce by measuring is a large proportion of the period you are trying to measure. Conversely, if you measure the time required for several swings, your single measurement introduces the same amount of error but it is amortised over however periods you timed. This makes much more sense. Also, we should take the measurement not once but several times. You will see that we end up with a table of results rather like that in Table 2.1. Each entry in the table shows the time taken

and the number of swings for which the pendulum was timed; note that three timings are recorded for a length of 10 cm but only two for 12 cm. (Also note that these values are purely for illustrative purposes; I would be really surprised if a real pendulum of these lengths had these periods!)

	time taken (s)		
length (cm)	#1	#2	#3
10	60.1/4	60.0/4	61.1/4
12	65.1/4	65.0/4	

Table 2.1: Timings and number of swings of a single pendulum of different lengths

2.2 Representing the data

The first thing to do is find a good way of representing the data in Table 2.1 in a Python program. If we are interested in the relationship between pendulum length and period, we obviously need to have both available within the program. From the previous chapter, the only way that we know is to do something like

```
pendlen1 = 10
pendlen2 = 12
timing1 = 15.10
timing2 = 16.26
```

where the values of `timing1` and `timing2` were calculated by hand. This is pretty ugly as we would have to write different code for working with `pendlen1` and `timing1` from that for `pendlen2` and `timing2`. The designer of the Python language realised this and provided a way to store related pieces of data in a structure known as a **list**:

```
pendlen = [10, 12]
timing = [15.10, 16.26]
```

Note that the values are enclosed in *square* brackets in the assignment statement — this is how the Python interpreter knows a list is involved. The best way to think of a list is as a series of consecutive pigeon holes, using the analogy introduced in Chapter 1 — see Figure 2.1. Note that the ordering of the elements in the lists is the same for both `pendlen` and `timing`: the first element is next to the name and the second one in the memory location above it.

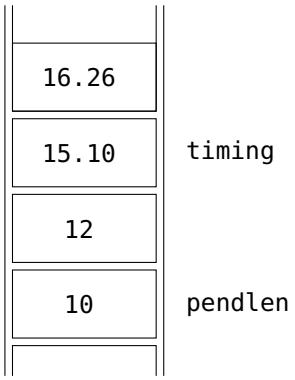

Figure 2.1: Lists can be thought of as consecutive pigeon holes

Note that lists can contain any data type: we are using integers and floating-point numbers here for `pendlen` and `timing` respectively, but lists can also contain any of the other data types we shall encounter in the course, even other lists!

Having stored the values in lists, how do we get at the individual elements of them? Come to that, how can the program find out how many elements are in a list? The follow section of code illustrates both of these:

```
pendlen = [10, 12]
timing = [15.10, 16.26]
n = len (pendlen)
for i in range (0, n):
    print (pendlen[i], timing[i])
```

You will see that the `len` function returns the number of entries in a list. We print out the list one element at a time in the following `for` loop, by appending an *element number* or **index** to the relevant variable name in square brackets, so that running the program gives:

```
10 15.1
12 16.26
```

which is what we expect.

2.3 Functions and subroutines

Considering we are using a computer, it is a pretty poor idea to have to average the various timings in Table 2.1 *by hand*! It would be much better to have the computer do that for us. We need to calculate the average for each row in the table (*i.e.,* for each element of `timing`), and it would be poor programming to have to write that code several times. Instead, Python lets us define a piece of code in one place, known as a **function** or **subroutine** (or sometimes a **method**), and use it several times. This is attractive for a wily programmer, as he or she will need to debug the code only once.

What we want is something that we can use by typing something like:

```
ave = mean (pendlen)
rowtime = mean ([60.1/4, 60.0/4, 61.1/4])
print (ave, rowtime)
```

Here, `ave` and `rowtime` are just ordinary variables, `pendlen` is the list defined earlier in the chapter, and `mean` is a **function**. In this case, `mean` takes one **argument**, a list of numbers, and **returns** a single result. You will see that the first call is passed a variable that contains a list while the second contains an explicit list — note the square brackets — of the numbers in a row of Table 2.1.

A function to calculate the mean of a list of numbers is pretty straightforward to write:

```
def mean (vals):
  "Calculate the mean of a list of values."
  sum = 0
  nv = len (vals)
  for i in range (0, nv):
      sum += vals[i]
  result = sum / nv
  return result
```

You will see that the function definition starts with the keyword `def`, and that is followed by the function or subroutine name and then its **parameters**, and the line is terminated by a colon. (To get the jargon right, *parameters* appear in the definition of a routine while *arguments* are what you **pass in** when making a **call** or **invocation**.)

The body of the routine is indented relative to the `def`, as you would now expect. The first line of a routine should be a string containing a succinct summary of what it does and how it is used; this can be extracted by Python's automatic documentation tools, which will be discussed in

a later chapter. The remainder of the routine does the actual work: it sets a variable to zero, then accumulates the values of all the entries in the list; when that has been done, it divides the sum by the length of the list to form the mean. Note how critical the indentation is to having the routine work correctly. The crucial line is the `return` statement, which tells Python what value is given back to the part of the program that called it — in this case, to be assigned to `ave` in the first invocation and `rowtime` in the second.

You will see that the above code contains `sum = 0`. If you have programmed in other languages before, you might think that line should read `sum = 0.0` to ensure that `sum` is a floating-point variable — after all, we subsequently divide it by an integer and integer division usually discards any fractional part. This is not the case in Python 3: if the result of dividing one integer by another has a fractional part, the resulting value becomes floating-point. This is a really useful language feature; it was not the case in Python 2, and is not the case in other programming languages.

One thing that you need to ensure is that `mean` has been defined *before* it is used; in this program, that means putting it towards the top of the code. In programs that I write, routines go after the top-of-program documentation and `import` statements.

Exercise. Write a program that implements this `mean` function to calculate the mean of a list of values, and check it gives the right results for `pendlen` and the rows of Table 2.1. Having got the bit between your teeth, extend the program so that you have routines to calculate the median, standard deviation, skewness and kurtosis — definitions for skewness and kurtosis are given below and there is discussion of them on the Web.

Calculating the median involves sorting the values into ascending order and then choosing the middle one. There is a built-in routine for sorting lists, you just need to find out what its name is and how to use it.

To be able to calculate the standard deviation *etc.*, you will need to calculate powers and root susing Python's maths library:

```
>>> import math
>>> print (math.sqrt (2))
>>> 1.4142135623730951
```

This notation of separating a module from functions residing in it using a dot is a common notation in Python and you will soon get used to it. You cannot use a dot in the middle of an ordinary variable though; use underscore instead. Remembering to type `math.sqrt` to calculate a square root instead of just `sqrt` is tiresome, so Python lets one "pull in" all the names in a way that means we don't have to type the module name all the time:

```
>>> from math import *
>>> print (sqrt(2))
>>> 1.4142135623730951
>>> print (math.sqrt(2))
>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

The last expression fails because this style of `import` does not prefix function names with `math`. Also, this form of `import` needs to be used with caution with multiple modules as you can end up overriding a routine from one module with another of the same name from another module; but it certainly saves typing.

The *skewness* measures the degree of asymmetry around the mean and is given by

$$\frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{x_i-\bar{x}}{\sigma}\right)^3 \tag{2.1}$$

where σ is the standard deviation. The kurtosis measures the 'peakiness' relative to a Gaussian and is given by

$$\left\{\frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{x_i-\bar{x}}{\sigma}\right)^4\right\}-3 \tag{2.2}$$

2.4 Making your own Python modules

Having written routines to calculate all the statistical quantities outlined above, you can store them together in a file, `mylib.py` say. It will look something like:

```
def mean (vals):
   ...

def sd (vals):
   ...

def skewness (vals):
   ...

def kurtosis (vals):
   ...
```

In your main program to analyse the timings, you can then change the code from

```
def mean (vals):
   ...
ave = mean (pendlen)
```

to

```
import mylib
...
ave = mylib.mean (pendlen)
```

Alternately, use

```
from mylib import *
...
ave = mean (pendlen)
```

just as with the maths library.

That's all there is to writing a simple Python module!

One of the key skills in becoming a good programmer is **identifying when a chunk of code will be needed in several places** in the program

and then writing routines that can simply be called. In the same way, you will find that **code written for one program can be re-used in others**. As well as being less typing in the first place, this means that if you find a bug in the code, it needs to be fixed in only one place — though, equally, a bug in a shared routine pervades all the programs that use it. It is well worth reflecting on *why* I suggested you write routines to calculate the mean *etc*. as well as *how* you wrote them. That should also help you understand why storing routines in modules is a good thing to do.

2.5 Creating lists dynamically

Bearing in mind that Table 2.1 contains a series of rows that needs to be averaged, it is tempting to write:

```
pendlen = [10, 12]
timing[0] = mean ([60.1/4, 60.0/4, 61.1/4])
timing[1] = mean ([65.1/4, 65.0/4])
```

but *this does not work*. All the divisions of the total timings by the number of swings is fine; the problem is to do with the way that lists work.

When executing the first line, the Python interpreter knows that `[10, 12]` represents a list of two elements and creates the variable `pendlen` as being a list of two elements. When executing the second line, the interpreter understands that the numbers in square brackets represents a list of three entries and passes them to the `mean` function, then is given the single value that it returns. But where is it to be stored? The variable `timing` does not exist, and even if it did, Python would think of it as holding a single value rather than an element of a list.

The simplest (but ugly) way around this is to write:

```
pendlen = [10, 12]
timing = [0, 0]
timing[0] = mean ([60.1/4, 60.0/4, 61.1/4])
timing[1] = mean ([65.1/4, 65.0/4])
```

where the initial values assigned to the elements of `timing` are overwritten by the values calculated in the invocations of `mean`.

A more elegant way is to tell the Python interpreter that `timing` is a list and to grow the length of that list as the program runs:

```
pendlen = [10, 12]
timing = []
timing += [mean ([60.1/4, 60.0/4, 61.1/4])]
timing += [mean ([65.1/4, 65.0/4])]
```

However, there are so many brackets here that this may look confusing! The second line of code tells the interpreter that `timing` is a list which contains no elements (try this yourself and use the `len` function on it). Each of the following two lines uses the operator `+=`, which you should think of as meaning “append to the list,” and in each of them it has to append *a list* — and the way to convert the single value returned by `mean` into a list is to enclose it in square brackets. If you find this confusing, the following code is identical in effect:

```
pendlen = [10, 12]
timing = []
timing.append (mean ([60.1/4, 60.0/4, 61.1/4]))
timing.append (mean ([65.1/4, 65.0/4]))
```

Having the ability to extend lists as a program runs is a really useful feature of Python but the syntax here is definitely confusing.

2.6 Lists of lists

Although the approach mentioned above is fine, to use it we have had to write a fair amount of code. If I were writing the program to work with the data of Table 2.1, I would first store the timings in a *list of lists*:

```
pendlen = [10, 12]
timing = [
  [60.1/4, 60.0/4, 61.1/4],
  [65.1/4, 65.0/4],
]
```

You will see that each element of `timing` contains a *list* of the various timings that were made, as the following interactive session shows:

```
>>> pendlen = [10, 12]
>>> timing = [ [60.1/4, 60.0/4, 61.1/4], [65.1/4, 65.0/4]]
print (timing[1])
>>> [16.275, 16.25]
```

which is the second complete list held in the list variable `timing`.

It is then straightforward to convert all of these into their means:

```
n = len (timings)
for i in range (0, n):
  timing[i] = mean (timing[i])
```

This is well worth studying and getting to grips with as it will let you write more elegant code yourself.

3
Plotting data

This chapter continues working with the pendulum data of Chapter 2. Here, we shall explore plotting the data on a graph using Python. We shall also derive a mathematical model for the period of a pendulum and plot that on the same graph, and then write software to assess how good the fit between the two is.

3.1 Plotting data using Gnuplot

Although this is rather tangential to learning Python, we shall start by looking at a general-purpose graphing program. Imagine we have a file, `pend.txt`, containing the lengths of a pendulum and the corresponding periods, with one pair per line:

```
10 15.10
12 16.26
14 18.31
16 20.05
```

(Again, these values are fictitious.) The easiest way to plot them on a Unix machine is with a program called Gnuplot. If the `$` below represents your shell's prompt, you start Gnuplot simply by typing its name:

```
$ gnuplot

G N U P L O T
Version 5.0 patchlevel 1    last modified 2015-06-07

Copyright (C) 1986-1993, 1998, 2004, 2007-2015
Thomas Williams, Colin Kelley and many others

gnuplot home:     http://www.gnuplot.info
faq, bugs, etc:   type "help FAQ"
immediate help:   type "help"  (plot window: hit 'h')

gnuplot> plot 'pend.txt'
```

The resulting output appears in Figure 3.1; it looks pretty ugly but does show the data, and is quick and easy to use.

We can make the graph much more attractive by customising its output with the following commands:

```
set grid
unset key
```

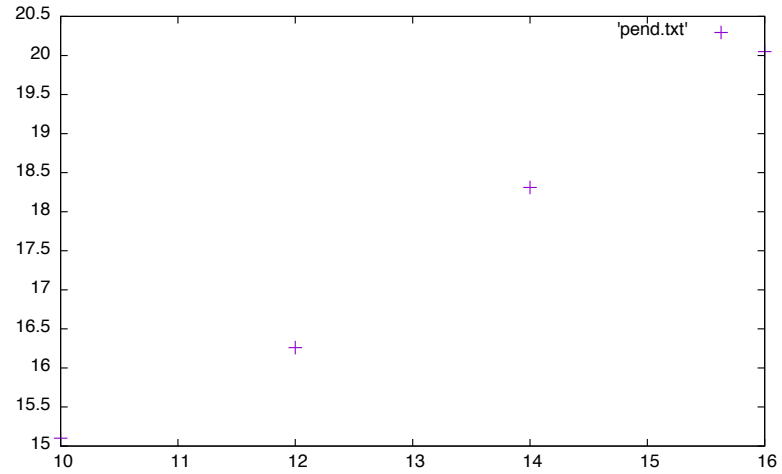

Figure 3.1: Default appearance of a Gnuplot graph

```
set xlabel "length (cm)"
set ylabel "time (sec)"
set style data linespoints
set xrange [9.5:16.5]
replot
```

which yields the output shown in Figure 3.2. This has a grid in the background, the legend (key) has been turned off, there are axis labels, and data points do not lie on the surrounding box.

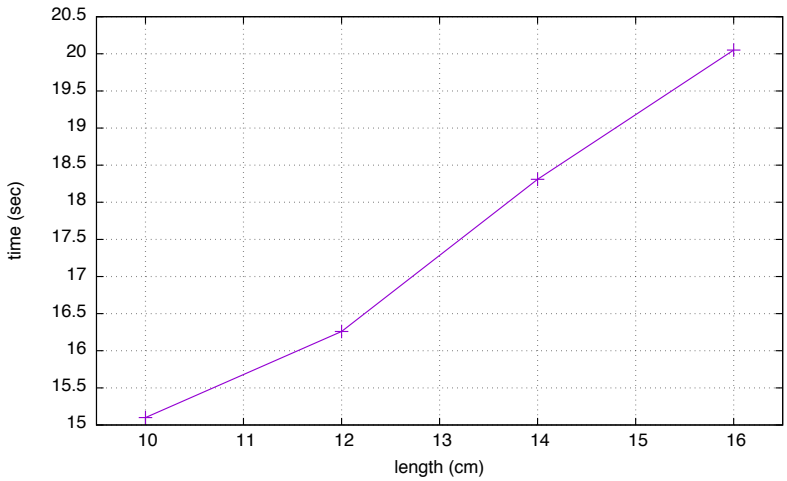

Figure 3.2: Improved appearance of a Gnuplot graph: axis labels, grid, *etc*

You can quit Gnuplot by typing `quit` or `^D`, the Unix end-of-file character. You can also use Gnuplot to plot equations; indeed, the plot we shall produce in Python with theoretical curve and experimental data below could have been drawn purely via Gnuplot — though doing so would defeat one of the objectives of this chapter, namely teaching you how to use Python's most popular graph-plotting package. The best thing about Gnuplot is that it is widely available, straightforward to use, and fairly easy to remember. In fact, it is quite common to use Python to write out commands for programs such as Gnuplot and then invoke it to process the commands; this is one of the reasons that Python is sometimes called

'software glue.'

3.2 Plotting data using Matplotlib

The standard way of producing graphs from Python programs is using Matplotlib. The word 'standard' does not mean that it is distributed as an intrinsic part of Python in the same way as the maths library; rather, it means that the package is widely used and fairly easy to install.

The particular interface to Matplotlib that we shall use is called `pylab`, which aims to be used in a similar way to the graph-plotting functionality of the popular `Matlab` package. Let us start with a pair of lists, one containing pendulum lengths and the other the corresponding mean timings

```
pendlen = [10, 12, 14, 16]
timing = [15.10, 16.26, 18.31, 20.05]
```

The first thing we much do is make the `pylab` module available at the top of the program:

```
import pylab
```

We are then able to call the various plotting routines. The following sequence of calls produces the graph shown in Figure 3.3.

```
# Plot the graph.
fig = pylab.figure ()
pylab.xlim ([9.5, 16.5])
ax = fig.add_subplot (111)
ax.grid (True)
ax.set_xlabel ('length (cm)')
ax.set_ylabel ('period (secs)')
ax.set_title ('Variation of pendulum period with length')
ax.plot (pendlen, timing)

# ...and display what we've drawn.
pylab.show ()
```

You will see that they are broadly similar in meaning to the commands that Gnuplot interprets.

The sequence of calls above seems quite logical: the first one 'turns on' graphics using `pylab.figure` and it returns a way of referring to the figure the graph will appear on. The call to `add_subplot` creates a set of axes on the figure and returns a way of referring to it, and the subsequent calls do things on those axes. (It is well worth reading online about `add_subplot` and its argument as it is not always clear why it is needed.) However, not everything works this way: the call to limit the range of values plotted on the x-axis should logically be called `ax.xlim` but is in fact `pylab.xlim`. These idiosyncrasies will often catch you out — or, rather, they often catch out the author.

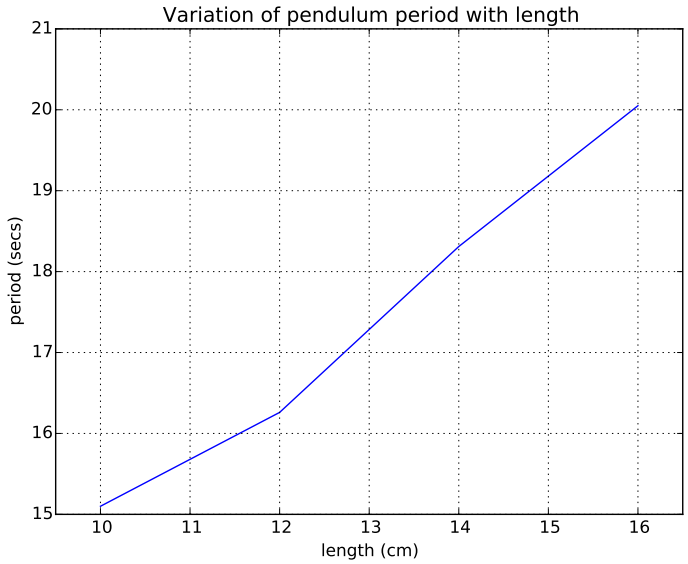

Figure 3.3: Plot produced using `pylab` calls

Exercise: Plotting with Pylab. Extend the program you used to process the pendulum measurements in the previous chapter so that it produces a plot of the mean pendulum period against the pendulum length. When you have it working, use Google to find out how to change the y-axis type to be logarithmic. Does this look help you identify the relationship between pendulum length and period? What about plotting the *square* of the period against the length?

3.3 Period of a simple pendulum

Deriving an equation that relates the period of a pendulum to its length is fairly straightforward, though we shall see that it is only an approximation. Figure 3.4 shows the forces acting on the bob of a simple pendulum. Note that the path of the pendulum sweeps out an arc of a circle and the bob is at an angle θ (measured in radians) from the equilibrium. The blue arrow shows the gravitational force acting on the bob, while the violet arrows are that force resolved into components parallel and perpendicular to the bob's instantaneous motion. The direction of the bob's instantaneous velocity always points along the red axis, which is always tangential to the circle. Consider Newton's second law

$$F = ma \tag{3.1}$$

where F is the sum of forces on the bob, m its mass and a its acceleration. As we are only concerned with changes in speed, and because the bob is forced to stay in a circular path, we need apply Newton's equation to the tangential direction only. Simple trigonometry tells us that

$$F = -mg\sin\theta \tag{3.2}$$

i.e. that $a = -g\sin\theta$, where g is the acceleration due to gravity. The negative sign on the right-hand side of these equations is present because the force is acting to return the bob to the equilibrium position and hence to reduce θ.

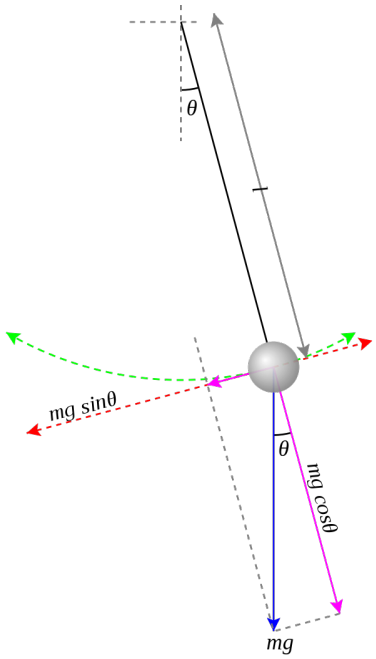

Figure 3.4: Forces acting on a pendulum (diagram stolen from the Wikipedia)

This linear acceleration a along the red axis can be related to the change in angle θ by the arc length $s = \ell\theta$, from which we obtain

$$v = \frac{ds}{dt} = \ell\frac{d\theta}{dt} \tag{3.3}$$

and

$$a = \frac{d^2s}{dt^2} = \ell\frac{d^2\theta}{dt^2} \tag{3.4}$$

which give us

$$\ell\frac{d^2\theta}{dt^2} = -g\sin\theta \tag{3.5}$$

so that

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\sin\theta = 0 \tag{3.6}$$

This equation is difficult to solve. However, if θ is small, $\sin\theta \approx \theta$ and we can re-write the above equation as

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\theta = 0 \tag{3.7}$$

which describes simple harmonic motion.

Given the initial conditions $\theta = 0 \equiv \theta_0$ and $\frac{d\theta}{dt} = 0$ at $t = 0$, the solution becomes

$$\theta_t = \theta_0\cos\left(\sqrt{\frac{g}{\ell}}\,t\right) \tag{3.8}$$

from which we can get an expression for the period τ of a pendulum

$$\tau = 2\pi\sqrt{\frac{\ell}{g}} \tag{3.9}$$

Extend your program to calculate the theoretical period for each measured length, and plot that on the same axes as your experimental values.

3.4 How close are theory and experiment?

We now have a pair of values for each period, the mean of the experimental measurements and the corresponding theoretical value. If we write the experimental mean value as E_i and the theoretical one as T_i, then we can measure how similar they are using

$$\frac{1}{N}\sum_{i=1}^{N}(E_i - T_i)^2 \tag{3.10}$$

This is usually known as the *mean square error* or MSE. Clearly, the smaller the MSE, the better the fit is.

Write a routine to calculate the MSE from a pair of Python lists of the same length. Use it to put the MSE in the title of your graph.

4
Working with text

In this module, the main interface between human and computer is textual. We have already seen the most important aspects of producing text: `print` and the format operator `%`. In this chapter, we shall look in further detail into how text is represented and manipulated in Python. Along the way, we will learn how subscripting works in more detail.

4.1 Chopping up text

Let us start our exploration of text by learning how it is stored and how to access parts of it. Consider the statements

```
text = "Hello, world"
print (len (text))
```

The `len` function returns the *length* of the argument it was called with; you have already seen it used with lists. When given a string, it returns the number of characters in it, 12 in this case. We can select a few characters from it using *subscripts* such as

```
print (text[0])
print (text[0:5])
print (text[7:])
print (text[7:-1])
```

which yield "`H`", "`Hello`", "`world`" and "`worl`" and respectively. You will be familiar with the single subscript in square brackets from our discussion of lists in Chapter 2 but the others look somewhat confusing at first, so some explanation is needed.

The individual characters of a string can be thought of as being stored in the set of pigeon holes introduced in Chapter 1, one letter per pigeon hole in exactly the same way as the list elements in Chapter 2. The letters of the string are stored in the pigeon holes as shown in Figure 4.1.

subscript	0	1	2	3	4	5	6	7	8	9	10	11
value	H	e	l	l	o	,		w	o	r	l	d

Figure 4.1: The layout of characters in a string

`text[0]` makes sense from Chapter 2: it is the first element of the string. As we can see from `text[0:5]` producing "`Hello`", the first number in square brackets is the first character; and the second number in brackets

is *one more than* the last character. This idea that the second number is **one more than the last element selected** is a consistent idea in Python, and you will see it crop up in several other places too.

The third example above, `text[7:]` selects from character location 7 to the end of the string, so this selects "`world`". At first sight, `text[7:-1]` looks odd as it seems as though you are indexing off the beginning of the string. However, Python assumes that **negative numbers in subscripts relate to the end** of the string, so the `-1` here is equivalent to `11`. The part of the string defined by subscripts 7–11 is "`worl`"; as before, the last subscript is one beyond the last character selected.

This notation of subscripts takes a little getting used to, and is different from practically every other programming language in its support for negative indices; however, it *is* consistent and you *can* get used to it. In fact, you can also provide a pair of subscripts to the lists that you used in in Chapter 2 and 3, and it will work in an analogous way.

If you have programmed in other languages, one other unusual thing about Python is that **strings are read-only**. You might think that, with subscripts working as they do, you would be able to program

```
text = "Hello, world"
text[5] = "x"
```

but the second line will cause an error. (I cannot tell you the number of times I have made this mistake!) If you want to do this, you have to write something like:

```
text = "Hello, world"
text = text[0:5] + "x" + text[6:]
```

When used between strings, the + operator concatenates them together. I find this requirement for strings to be read-only very pesky.

With this knowledge in mind, we can think about writing programs that manipulate text. However, before doing that in anger, it is helpful to know how to take information from the command line.

4.2 Handling the command line

Our very first program printed out "`Hello, world`" and (if made executable) could be ran by typing

```
./hello
```

It would be interesting to be able to type

```
./hello Joe
```

and have the program respond "`Hello, Joe`"; and to do this, the program needs to be able to access the words entered on the command line. This is a standard capability of Python and is straightforward to do.

```
#!/usr/bin/env python3
"""A revised "Hello World" program that responds to any name
passed in on the command line."""
import sys
```

```
if len (sys.argv) < 2:
  name = "world"
else:
  name = sys.argv[1]
print ("Hello, " + name)
```

The main body of the program is a conditional, which is easy to understand from Chapter 1, and the test is comparing the length of something called `sys.argv` with 2. **The variable `sys.argv` is a list of strings**, and `sys.argv[1]` refers to the second element. Why should the first argument provided on the command line end up in `sys.argv[1]` rather than `sys.argv[0]`? After all, Python is supposed to start its subscripting at zero. The answer is that `sys.argv[0]` contains the name of the program.

The line

```
import sys
```

will be familiar, as it is similar to the way in which we accessed Python's maths library in Chapter 2: it tells the Python interpreter to **read in the file of definitions** called `sys`, which is another of Python's standard libraries. There are literally *hundreds* of modules available for Python, doing everything from networking to 3D graphics. Most of them are not standard parts of Python but can be installed from the Internet.

One thing that often trips people up is that the arguments you type on the command line are delivered to your program *as strings* — so if you enter the command

```
./myprog.py 1.6
```

and `myprog.py` contains a line

```
value = sys.argv[1] + 2
```

your program will fail because `sys.argv[1]` is a string and `2` is an integer. To do this, you have to convert the string to be a number first:

```
value = float (sys.argv[1]) + 2
```

There is also an `int` function to convert a string to an integer.

4.3 Temperature conversion revisited

Your knowledge of Python has progressed to the point where you should be able to *read*, if not write, longer programs. The following program re-visits the temperature conversion topic of Chapter 1 (for the last time, I promise!) but, rather than printing out a table, it is intended to be used to convert temperatures, so that

```
./tempconv 32 f c
```

would convert 0°F to Celsius, printing out

```
0.00C
```

To make the program a little more interesting, it supports not only Fahrenheit and Celsius but also Kelvin, the SI unit of temperature, and Rankine,

absolute temperature with a Fahrenheit-sized degree (I have never actually seen the latter used by any scientist).

Much of the program should be familiar. Try to identify what is and is not familiar and work out what the unfamiliar parts do before progressing. The new concepts are discussed following the listing.

```
#!/usr/bin/env python3
"""This program converts temperatures provided on the
command line.

Usage: %s <value> <from_unit> <to_unit>"""
import sys

# Check we have enough arguments on the command line.
if len (sys.argv) < 4:
    print (__doc__ % sys.argv[0], file=sys.stderr)
    exit (1)

# Pull the important values from the command line.
value = float (sys.argv[1])
from_unit = sys.argv[2].lower ()
to_unit = sys.argv[3].lower ()

# Other important values.
abs_zero = 273.15

# First, convert the temperature from whatever unit it
# is in to Kelvin.
if from_unit == "c":
    value += abs_zero
elif from_unit == "f":
    value = (value - 32) / 1.8 + abs_zero
elif from_unit == "k":
    pass
elif from_unit == "r":
    value /= 1.8
else:
    print ("I don't know how to convert from '%s'!" % \
        from_unit, file=sys.stderr)
    exit (1)

# Now, convert from Kelvin to the output unit.
if to_unit == "c":
    value -= abs_zero
elif to_unit == "f":
    value = (value - abs_zero) * 1.8 + 32
elif to_unit == "k":
    pass
elif to_unit == "r":
    value *= 1.8
else:
    print ("I don't know how to convert to '%s'!" % \
           to_unit, file=sys.stderr)
    exit (1)

# Finally, output the result
print ("%.2f%s" % (value, to_unit.upper ()))
```

There are several things of interest here. Firstly, whenever an error message is generated, the `print` call includes the code `file=sys.stderr`. In terms of what happens, this tells `print` to generate its output on a *stream* called `sys.stderr`. This is similar to ordinary terminal output, which is written on a stream called `sys.output`, but cannot be re-directed using the shell's ">" notation — so if you happen to re-direct the output of `tempconv` to a file or pipe, you still see the error message in your terminal window. This is good programming practice.

So what does the `file=` part of the call do? Python allows parameters to be passed to functions by name, and this simply says that it is the `file` parameter that we want to set. You didn't use this in the functions you wrote in Chapter 2 but this is valuable when writing more sophisticated routines — for example, I use it in programs for defining 3D graphics shapes such as spheres and cubes which default to a particular colour but allow the user to specify a different one. A further example is given in Chapter 6.

The call to `exit` is hopefully clear: it stops your program from running. The number passed into the call is returned to the shell as a *status* value, with non-zero meaning error.

Finally, you will see lines such as

```
from_unit = sys.argv[2].lower ()
```

It is clear that this sets the variable `from_unit` to whatever is in `sys.argv[2]`. The invocation of `lower` converts the text in `sys.argv[2]` to lower case so that, for example, "F" becomes "f", simplifying the tests for the units. From what you know, you would expect it to be

```
from_unit = lower (sys.argv[2])
```

In fact, this almost works: `lower` is a function applied to strings, so the code

```
import string
from_unit = string.lower (sys.argv[2])
```

is valid Python. However, it is *deprecated*, meaning that it is obsolete and likely to disappear at some point in the future. The syntax with the call attached to the variable using a dot is a feature of what is known as *object-oriented* programming, which is what most computer scientists regard as the Right Way To Do It. I am less convinced, and it is certainly the case that most novice programmers struggle with object-oriented programming.

Incidentally, there are many useful functions in the `string` module. You can find documentation on them online or by typing the command

```
pydoc string
```

in a terminal window.

4.4 Pig Latin

Pig Latin is not a language spoken by pigs, it's a language game in which ordinary English words are altered as they are spoken. There are several variants of the game but the one we shall consider has the rules:

- if a word begins with a vowel, append “way” to it
- if there are more than two characters in the word, move the first letter to the end and append “ay”
- otherwise, the word is short, so just append “yay” to it

Applying these rules, the line of text

```
now is the winter of our discontent made glorious
summer by this son of York
```

from Shakespeare’s play *Richard III* becomes

```
ownay isway hetay interway ofway ourway iscontentday
ademay loriousgay ummersay byyay histay onsay ofway
yorkway
```

which is, admittedly, quite tricky to say (but I think that’s the point).

Writing a program to perform this conversion is a step up in complexity from our earlier examples but definitely achievable using Python’s text manipulation capabilities. The task is actually made much easier because the shell separates any text written on the command line into individual words. Of course, there is functionality in Python to do the splitting too: it’s the `split` method for strings, so you can simply write

```
words = "now is the winter".split ()
```

to have the individual words appear in the list `words`. You also need to know that the loop

```
for w in words:
    print (w, end=" ")
print ()
```

prints each element in the list `words` separated by a single space. The `print` call with no arguments following the loop ends the line of output that the loop itself has created.

Exercise: Pig Latin. Have a go at writing a program that ‘translates’ and writes out words of English provided on the command line as *igpay atinlay*. My own solution to this is 18 lines of code, excluding comments.

5
Files, Exceptions and Dictionaries

This chapter covers three topics: first, how to read and write files; then how to handle exceptions; and finally *dictionaries*, which are most easily thought of as lists for which the subscript is text rather than an integer. Although these are distinct, unrelated language features, we shall see how their combination contributes to useful programs.

5.1 Reading and writing files

To manipulate a file, there are four fundamental operations:

- open the file
- write to an open file
- read from an open file
- close an open file

We shall look at each of these in turn, and then see an example of how they may be used in practice.

Opening files

A file is opened, not surprisingly, using the `open` function

```
f = open ("myfile.txt", "r")
```

The first argument in the call is the name of the file to be opened, and the second is the **access mode**, which describes how the file is to be accessed. The commonly-used access modes for reading and writing text files are listed in Table 5.1; if omitted, the file is opened read-only. When a file is opened for writing (`w`), any existing content in the file is immediately discarded, while when opening a file for updating (`r+`) it is not; confusing these is a common problem. Another common mistake is to use shell shorthands such as `..` and ~ in filenames, as they are not automatically expanded as they would be by a shell (there are routines in the module `os.path` that do this kind of thing). Hence common invocations of `open` might be

mode	meaning
`r`	reading only
`w`	writing
`a`	appending
`r+`	updating

Table 5.1: Commonly-used access mode for Python's `open` function

```
fin = open (sys.argv[1], "r")
fout = open ("results.txt", "w")
```

The `open` function returns a value that you can use subsequently to indicate the file that is to be written or read. Some languages call this a *file*, a *file handle*, *file pointer*, or a *channel*.

Writing to files

The easiest way to write output to an opened file is via an argument to print:

```
print ("Hello, world.", file=fout)
```

Clearly, `fout` needs to have been opened for writing. You have already seen the use of `file=sys.stderr` when producing error messages, so this shows us that `sys.stderr` is simply a file handle that has been opened for writing.

As in most programming languages, **output to files is normally buffered**, which means it is held in the computer's memory until there is a fair amount of text to be written out, and then the whole chunk is written out in a single lump. This makes output much more efficient than it would otherwise be (but still slows a program down dramatically). There is an optional third argument in the `open` call that specifies the buffering mode but a more common approach is simply to call the `flush` function to empty the buffer if you want to be sure that the user sees the output:

```
print ("Warning: about to overwrite your file!", file=fout)
fout.flush ()
```

Text written to `sys.stderr` is normally unbuffered for obvious reasons, and all other output is normally buffered.

Reading from files

In the Unix world, you will find that you read from files much more often than you explicitly write to them because the shell's re-direction facility is so convenient. When you write programs to run under Windows (unless you use one of the Unix-like shells), that will probably not be the case. There are several different strategies for reading a file: you can read all the lines at once and then chop them up, read one line at a time, and so on. Here, we shall concentrate on the one-line-at-a-time approach as it is moderately efficient (anyway, input is buffered in a similar way to output) and scales better to huge files.

The key to reading a file line by line is identifying when the end of the file has been reached. The best way of programming this has changed a little as Python has evolved and the recommended way is now

```
with open ("myfile.txt", "r") as f:
    for line in f:
       process line
```

where the last line above would be replaced by code to process the line of text in the variable `line`. This approach is, to the author's eye, concise, clear and elegant. Note that **the file is closed automatically** at the end of the `with` clause (normally when all the lines in the file have been read), avoiding having to do so explicitly.

Closing files

When you need to close file handle `f` explicitly, simply

```
f.close ()
```

An example

How these functions work together is best illustrated by an example. Consider a program, `numlines`, that reads in a file and prints it back out with each line being prefixed with a line number. We want to invoke it by typing:

```
> numlines myfile.txt
```

where > represents the shell's prompt. The underlying algorithm is easy:

```
open the file
    while not at end-of-file
        read a line
        increment the line number
        print the line number and the line
    close the file
```

Apart from checking and initialisation, there is almost a line-for-line match to the resulting Python program:

```
#!/usr/bin/env python3
"""
Read a file given on the command line and output it with line numbers.
"""
import sys

# Ensure we were given a filename on the command line.
if len (sys.argv) < 2:
    print ("Usage: %s <file>" % sys.argv[0], file=sys.stderr)
    exit (1)

# Open the file, then read it line by line.  For each line, strip
# off the end-of-line delimiter and any trailing whitespace, then
# output it with its line number.
with open (sys.argv[1]) as f:
    n = 0
    for line in f:
        n += 1
        print ("%5d %s" % (n, line))
```

If you enter this program and run it on a file, you will see that it works, except that there is a blank line after every line of output. To understand why this happens, you need to understand how text files are stored. Basically, there is a special character stored in a file to identify the end of every line in a file, though the character used depends on the operating system: Unix uses ⟨linefeed⟩ (`^J`), MacOS (before OSX) ⟨return⟩ (`^M`) and DOS (and hence Windows) ⟨return⟩⟨linefeed⟩ (`^M^J`). Python is clever enough to convert this character or characters into a generic "start a new line" character, which is represented in a Python string as "`\n`". However, returning to the reason that the output contains blank lines, lines read in

from files retain the training `\n` character from the file. Hence, when you `print` the variable `line`, there are *two* `\n` characters, one read from the file and the second added by `print`.

There are several ways around the problem. One is

```
print ("%5d %s" % (n, line[:-1]))
```

and another is

```
print ("%5d %s" % (n, line), end="")
```

but a more elegant alternative is

```
line = line.rstrip ()  # remove the end-of-line delimiter
print ("%5d %s" % (n, line))
```

The `rstrip` function removes the newline and any other trailing whitespace (spaces, tabs) from the right end of `line`.

5.2 Exceptions

When you run the line-numbering program above with the name of a file that does not exist, it crashes:

```
> numlines.py nosuchfile
Traceback (most recent call last):
  File "./numlines.py", line 16, in <module>
    with open (sys.argv[1]) as f:
FileNotFoundError: [Errno 2] No such file or directory: 'nosuchfile'
```

This is pretty ugly — it would be better if your program produced a simple, useful error message and exited. More generally, there may be unusual events that happen as a program runs that you would like to handle and allow the program to continue; but if it crashes like this, you cannot. Python provides a way around this dilemma through the use of a `try` clause. For the `numlines` program above, one solution is

```
try:
    with open (sys.argv[1]) as f:
        n = 0
        for line in f:
            n += 1
            print ("%5d %s" % (n, line))
except:
    print ("Problem reading %s!" % sys.argv[1], file=sys.stderr)
    exit (1)
```

The whole section of code that may cause (the jargon is "throw" or "raise") an exception has been enclosed in a `try` clause, and the accompanying `except` clause is what happens if an exception happens.

The example above catches and produces an error message **for all exceptions** irrespective of their cause. You can also choose to catch just some exceptions; *e.g.,* the code below handles only `ValueError` exceptions:

```
while True:
    try:
        x = int (input ("Please enter a number: "))
```

```
        break
    except ValueError:
        print ("Oops!  That was not a valid number.  Try again...")
    else:
        print ("You entered %d." % x)
        if x == 0: break
```

This example also shows the use of an optional `else` clause for `except`, which is executed if no exception happens. There is also a `finally` clause which is executed irrespective of whether or not an exception has happened; I am not too convinced about its usefulness.

Just as you can handle exceptions when they occur, you can create exceptions. For example, in the program that processed pendulum timings in Chapter 2, a sensible thing to do is to ensure that the lists `pendlen` and `period` have the same lengths. This could be programmed as

```
if len (pendlen) != len (period):
    raise ValueError ("List length mismatch")
```

and causes a `ValueError` exception to occur, producing the message in its string argument. We shall see a better way of doing this kind of checking using `assert` in Chapter 7.

There are many exceptions other than `ValueError` and the documentation of modules and libraries will normally explain which exceptions can occur where and under what circumstances. It is also possible to create new types of exception but that involves object-oriented programming, which is not covered in this course.

You might ask yourself when to raise exceptions and when to produce error messages and exit. The rule of thumb that the author uses is that routines he writes that go into a library generate exceptions, so that a person using the library can catch and handle them if he or she so wishes; but my main program and routines written that support only it produce error messages and exit.

5.3 Dictionaries

The easiest way to think of a dictionary is as an array which is indexed by a text string rather than a number, as a list is indexed. There is a natural order to list indices: they start at zero and increment by one for each element; however, there is no such order to the elements of a dictionary. When you gather together the **keys** ("subscripts") of a dictionary, you will see that they are in an arbitrary order. You might also like to know that Python's name for this kind of data structure is unusual; a more common name is a *hash*. Although a really useful data structure, you should be aware that accessing an element of a dictionary is about an order of magnitude slower than accessing a list, so you should use them only when necessary and not as a more convenient alternative to lists.

The easiest way to see how dictionaries work is in an interactive session with the Python interpreter:

```
>>> sound = {}
>>> sound["cat"] = "meow"
```

```
>>> sound["dog"] = "woof"
>>> sound["hen"] = "cluck"
>>> s = "dog"
>>> print ("The", s, "says", sound[s])
>>> The dog says woof
>>> sound["owl"] = 2820
```

The first line here creates an empty dictionary and the following three lines set entries in it. At first, it seems a little strange that a dictionary is created using braces `{..}` but accessed using brackets `[..]`, though you quickly get used to it. The following pair of lines illustrate how an entry in a dictionary may be used.

Dictionaries **may contain any data type**, and the last line above sets one element of `sound` to an integer. In real-world programs, a fairly common data structure is a dictionary of lists, where each member of a dictionary is a list of information — a list, of course, can have a different datatype in each element too.

A common requirement is to determine whether or not a particular string is a valid dictionary entry (*i.e.,* that the dictionary has an entry with that name). In Python 3, the way you do this is to write something like

```
if s in sound:
    print (sound[s])
```

To give an example, imagine you need to increment a counter every time a word is encountered but to create and initialise a counter when the word is met for the first time. The code would look something like

```
if word in count:
    count[word] += 1
else:
    count[word] = 1
```

Note that this is a fairly new feature of the language; in Python 2, you would have written

```
if sound.has_key (s):
    print sound[s]
```

which is much less clear. You will see this when reading old Python programs.

Another common requirement is to obtain all the valid keys of a dictionary. This is straightforward:

```
kk = sound.keys ()
```

The value returned by `keys` is a list, so it is common to write something like:

```
for k in sorted (sound.keys()):
  print (k)
```

so that the keys are printed out in alphabetic order.

To illustrate the use of a dictionary, let us return to the "Hello, world" program that started our exploration of Python. In Chapter 4, we saw a variant of this in which a name could optionally be given on the command line, so that (where > represents the command prompt) we obtained:

```
> hello
Hello, world!
> hello Joe
Hello, Joe!
```

We shall now extend that program to support an optional argument that specifies the language to use, so that we can also get:

```
> hello -french Joe
Bonjour, Joe!
```

If the language specified on the command line is not supported, boring old English is used.

```
#!/usr/bin/env python3
"""
A multilingual 'hello world' program.

  Usage: hello [-lang] [name]

where <lang> is a supported language.  Unsupported languages are ignored.
"""
import sys

# Set up a dictionary that has the relevant text for all the languages
# that we shall support.
HELLO = {
    "english": "Hello, %s.",
    "french": "Bonjour, %s.",
    "geordie": "Wot fettle the day, %s.",
}

# Initialization.
greeting = HELLO["english"]
who = "world"

# Walk along the command line.
for arg in sys.argv[1:]:
    if arg[0] == "-":  # it's a qualifier
        word = arg[1:]
        if word in HELLO:
            greeting = HELLO[word]
    else:
        who = arg

# Now output our greeting.
message = greeting % who
print (message)
```

Word-counting. A program that counts the number of times each word is used in a document is sometimes useful — for example, researchers have used it to explore the authenticity of some plays attributed to William Shakespeare, Britain's most famous playwright. Write a program to do this, taking the name of the file to be read from the command line.

The output from your program should look something like:

```
    3 aardvark
  297 anaconda
```

```
    ...
42 zorilla
```

The solution to this will use the features discussed in this chapter. You may find it helpful to read through your solution to the 'pig latin' exercise at the end of Chapter 4 before starting.

6
Python as Software Glue

Up to now, we have concentrated on using the Python language itself for programming up complete applications; this is mostly so that you can master the language itself. In this chapter, we shall look briefly at using Python for doing things beyond simply programming analysis applications. The language can be used for developing an amazingly wide variety of applications, ranging from cryptography to networking, and constructing graphical user interfaces. We shall do this in three stages, firstly examining modules that are part of all Python distributions (Section 6.1), then looking at exploiting features built into the host operating system (Section 6.2) and finally some of the hundreds of extensions available (Section 6.3).

6.1 Capabilities built into Python

These notes and the accompanying lectures have looked at the core data types of Python: `int`, `float`, strings, lists, dictionaries, and so on. There are actually quite a few more data types available, generally built on top of these more fundamental ones using object-oriented programming (which we do not cover in this course). For example, there are Python data types for infinite-precision numbers, rational numbers, complex numbers, quaternions, and so on.

You have used a small number of "methods" on strings: `split` to split up a string into words, *etc*. The `string` module provides a great many more useful things for manipulating strings; whenever you need to do something with a piece of text, do look in the `string` module before thinking about writing it yourself.

Still on the subject of processing strings, there is a powerful module for parsing command lines, `argparse`, with capabilities well beyond the simple command lines we have considered to date. As we shall see in Section 6.3, using `argparse` makes it possible to construct graphical user interfaces automatically.

There is a module for *regular expressions* ("regexes"), which are a way of processing text based on the patterns of characters in them. To give a simple example, the code

```
import re
line = "Dogs are cleverer than cats";
searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)
```

extracts the words "`Dogs`" and "`cleverer`". As a more significant example, the author has written Python code for a website that receives an uploaded PDF document, converts it to plain text (using the `os.system` function described in the next section), then uses search with regexes for UK map references: two uppercase letters followed by an even number of digits, with possible spaces between the letters and in the middle of the sequence of digits. These are converted to WGS84 form (as used by GPS systems) and the result plotted on a Google map, so that people using the website can see where documents relate to geographically.

Python includes an extensive module for working with dates, times and calendars. It is possible to print calendars, work out the number of days between specific dates, parse strings to extract dates (in any of several formats) and so on.

Python has built-in support for multi-threaded programs, allowing different threads of execution through a program to be created and managed. One way in which this is used is in `idle`, the integrated development environment included in Python distributions, and in an interactive Python debugger. The latter is potentially useful but most people debug their programs purely by inserting `print` calls into their scripts — though we shall look at a further feature for helping avoid bugs in the next chapter.

There is support for several cryptography techniques and message digest algorithms in the standard Python library. The library also includes the ability to create and manage network connections, send and receive network traffic, *etc*. Built on top of this is a web server and all the capabilities needed to built web crawlers — even web browsers, as there are modules for parsing and creating HTML and XML.

6.2 Using the operating system

If the standard Python modules do not provide everything you need for a task but there is the required functionality in the operating system, then there is a way in Python to make use of it. Of course, the resulting program will not be portable between operating systems, and perhaps even between machines if the required software is not available, but it will let you do what you needed to do.

The easiest interface to the host operating system is the `os.system` function. This takes a complete command, just (with some provisos) as it would be typed into a shell window. To illustrate this with an example, the Linux machines in CSEE's software laboratories have installed on them `flite`, a simple speech synthesiser. This program is invoked with the text to be spoken on the command line, and it can be used in a Python script as follows:

```
import os
text = "Go ahead, make my day."
os.system ("flite " + text)
```

The author uses this approach at the end of long-running jobs to let him know that they have finished.

If you have a Mac, you can adapt this example to use the `say` command instead of `flite`. Windows users don't have such a command but there is

a module called pyttsx which can be used instead.

6.3 Extensions

There are literally *hundreds* of extension modules available for Python. Most of these provide access to capabilities that are not part of the standard Python distribution. A small number of them are considered in the following paragraph but a web search should show you the kinds of things that are available.

If algebra or calculus is something you find painful, there is sympy, a symbolic algebra package. The following example shows how it may be used for differentiation:

```
import sympy
a = sympy.Symbol('a')
b = sympy.Symbol('b')
e = (a + 2*b)**5
print (e)
print (e.diff(a))
print (e.diff(b))
```

which yields

```
(2*b+a)^5
5*(2*b+a)^4
10*(2*b+a)^4
```

One of the most widely-used and important extensions to Python is numpy or *numerical Python*. This provides support for multi-dimensional arrays, somewhat analogous to the popular Matlab software. A simple example is:

```
import numpy
b = numpy.array([(1.5,2,3), (4,5,6)])
print (b)
c = b * 2
print (c)
```

which yields the output

```
[[ 1.5  2.   3. ]
 [ 4.   5.   6. ]]

[[  3.   4.   6.]
 [  8.  10.  12.]]
```

You will see that the **entire array** b has been multiplied by 2 in a single line of code. More importantly, the operation takes place very quickly, much more quickly than it would if all the loops had been programmed in Python — and the speed advantage becomes more marked as the size of the array increases. All serious numerical computation in Python makes use of numpy, and several other capabilities are built on top of it. Perhaps the most important of these is scipy or *scientific Python* which provides useful stuff for computing in the scientific and engineering world such as statistics, solving sets of equations, optimisation, root-finding, and so on.

One of the most common ways of distributing numerical data is via spreadsheets (or, almost equivalently, CSV files). Although the latter can be parsed using little more than `split`, a more sophisticated interface is available through the `pandas` modules — and it also provides support for the data structures used by more sophisticated data analysis tools.

Images can be read and written in a variety of formats using the `PIL` module or its sibling `Pillow`. As well as input and output, these allow the programmer to perform some image processing operations. If the capabilities of these packages are not enough, there is a Python interface to the extensive OpenCV software, which is used in the robotics and media industries for analysing the content of images and videos. (The driverless cars you read about use OpenCV.)

One recent extension to Python, which is still under fairly rapid development, is `Gooey`, a module that allows any Python program that uses `argparse` to process its command line to have an automatically-constructed graphical user interface (GUI) in literally a couple of lines of Python. GUI construction is usually fairly painful in any programming language, so this is an amazingly useful extension!

Figure 6.1: Interacting with a virtual reality system written entirely in Python

Finally in this whistle-stop tour of Python extensions is PyOGL, a module that allows interactive 3D graphics to be programmed in Python; the underlying graphics library is OpenGL. The author has used this extensively to build and interact with 3D models in a virtual reality installation in one of CSEE's research laboratories. This employs a high-resolution, stereoscopic back-projection system so that the user gains a good impression of depth, and allows him or her to control interaction with the model in a variety of ways. Figure 6.1 shows a person interacting with the model using a gestural interface. The important point is that **all the software components of this installation are written in Python** and **all run in real time**: 3D model, networking, control, and interfaces to devices such as mice, keyboard, cameras, Kinects — even a bicycle!

7
Large-Scale Programming with Python

The programs you have seen and written to date as part of this module are all small. The program you will develop for your assignment just about makes it into the medium category. This short chapter explores writing even larger programs, the kind you may need to produce as part of your project or in a commercial environment. It will introduce only two new language features, instead concentrating on making best use of features of Python that you already know about.

7.1 The approach to developing large programs

Modules. By far the best way to produce a large program is to write it in small chunks! We have already seen this with the statistical routines in Chapter 2: each one took the data in the same form and returned a result that the program was able to use. Each individual routine was small enough to be understood on its own and, just as important, could be tested easily. (There will be a longer discussion of testing later in this chapter.)

Documentation. It may seem incongruous to discuss documentation when considering how to write large programs, yet documentation is surprisingly important. One aspect of documentation is the summary of what a routine does in the quoted string immediately following its `def`inition line, known as its *docstring*. In the examples you have seen, this has been kept brief so that you can concentrate on the code but 'real programs' tend to provide much more information. For example, here is an example taken from the author's own code:

```
def annular_mean (im, y0=None, x0=None, rlo=0.0, rhi=None, alo=-math.pi,
                  ahi=math.pi):
    """
    Return the mean of an annular region of an image.

    Arguments:
     im  the image to be examined
     y0  the y-value of the centre of the rotation (default: centre pixel)
     x0  the x-value of the centre of the rotation (default: centre pixel)
    rlo  the inner radius of the annular region
    rhi  the outer radius of the annular region
    alo  the lower angle of the annular region (default: -pi)
    ahi  the higher angle of the annular region (default: pi)
```

```
    """
```

You will not have seen the apparent assignment to formal parameters in the `def` before: these are **default values** that the variables will take if the argument was not supplied; an invocation might be

```
annular_mean (im, rhi=20, rlo=10)
```

You can see that the documentation describes briefly what each parameter is for as well as the purpose of the routine. Default arguments are discussed in more detail in Chapter 8.

The second aspect of documentation is incorporating comments into the code that describe what sections of it do — you will see examples of this in all the programs in this document. These comments do not explain the individual lines of code but instead concentrate on explaining algorithms or the overall way in which the program works.

Sanity-checking. Good programs — and especially good library modules — should sanity-check inside routines that the data passed in makes sense. For example, if a routine (say, to calculate the RMS difference between two sets of data) requires two lists of the same length, it should check for this. The most concise and elegant way to do this is:

```
def rms_diff (x, y):
   assert len (x) == len (y)
   ...
```

The `assert` statement executes the line of code associated with it; if the result is `False`, execution of the program is aborted with a suitable error message.

`assert` statements are especially useful as code is being developed, so they need to go in as routines are being written, not as an after-thought. In a production program, they are ignored if the Python interpreter is invoked with the "`-O`" qualifier — though I don't believe anyone ever bothers to run Python this way.

7.2 Testing code

As you will be painfully aware by now from the programs you have written yourself, making the syntax of a program right is easy compared to making sure it runs and gives sensible outputs under all circumstances — debugging code is probably more than 90% of the effort involved in programming.

Experienced programmers are wily people and have come up with tools to help make sure that a person does not introduce new bugs when correcting other ones. This is done by developing a set of test cases for which the answer is known and making sure the program gives the expected answer for each test case. When a bug is encountered, a test is added to the set of test cases that trips it, then the bug is corrected, and the *entire suite* of test cases is run to make sure the code is as bug-free as possible. This approach is known as **regression testing**.

There are several ways in which regression testing can be done with Python. The more sophisticated ways involve some object-oriented programming, which is not being covered in this module; but a more straightforward approach can be used, which is appropriate for writing libraries or modules (described in Section 2.4).

Adding the tests to a module is done in two stages. Firstly, the tests have to be inserted into the "docstrings", the documentation text at the beginning of each individual routine — you will have seen these at the top of the routines in these notes, though they have not contained tests. For example, consider a simple routine that simply calculates the square of a number. We could add tests to it as follows:

```
def square(x):
    """Returns the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x
```

Inside the docstring, the triple angle bracket (>>>) identifies the Python code that needs to be executed for the test, and the next line or lines give the output that should be obtained.

The second stage is to add, right at the end of the file, the following code:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

The last two lines clearly `import` a module and run it; but the first line will be somewhat unfamiliar. Whenever a Python program is executed, the Python run-time system sets the name of the main program to be the string "`__main__`", and this is stored in a system variable called `__name__`. You can see that the line is effectively saying *if this is the main program, invoke the routine* `doctest.testmod` — and what that routine does is extract and execute all the tests in the docstrings. For example, if a module called `mymodule.py` has docstrings containing tests and the three lines listed above at its end, the command

```
python3 mymodule.py
```

would cause all the tests in it to be executed.

Adding test cases to routines is considered as good programming practice, and you are encouraged to do this kind of thing with your own code. It will prove especially useful as you start writing longer pieces of software.

7.3 When does Python run out of steam?

Python is a really good programming language for rapid development but, although popular, it has not completely taken over the world. Are there

places in which Python is inappropriate to use? The answer is that there are.

Although `numpy` and `scipy` make it possible for operations involving entire arrays or arrays slices to be performed almost as quickly as in compiled code (say, C or C++), if the algorithm cannot be decomposed into such operations, it has to be written using loops in Python. This will make it execute much more slowly. To give an example from the author's own experience, a particular image operation (region labelling) when written in pure Python takes about 30 seconds; but when written using the array-slicing code in `scipy` runs in under a second. If performance is important, you should consider programming some or all of it in a compiled language.

To run a Python program, one needs a Python interpreter and all the support libraries. For an embedded system, where the processor is typically slow and the memory footprint an important consideration, Python is likely to be a poor choice. Similarly, Python is a poor choice for hard real-time systems because there is no way of guaranteeing the response time of Python code.

Finally, Python is a poor choice for any system in which you would be unwilling for people to see the source code. There are ways of getting around this restriction but they are not pretty.

For all other types of program — about 99% of the programs out there — Python is a good choice.

8
Epilogue

The aim of this module has not been to turn you into an expert programmer; rather, it has tried to give you enough knowledge to be able to write programs for other modules and for your project. As with everything else, the more you practise programming, the more adept you will become at it.

What has been missed out. You need to be aware that a number of features of Python have been omitted here; the focus has been on giving you enough knowledge to be able to write useful programs rather to cover the entire language. Some of the things omitted are fairly straightforward to understand, so you can easily read up about them on your own:

Global variables. Within a routine, you can explicitly declare that a variable you are using is `global` — this provides an extra bit of sanity-checking in programs. Conventional wisdom is that using global variables is bad, so you should use them only when you really have to.

Functions can return more than one value. In the programs you have seen in these notes, a function returns a single value; however, the `return` statement can take **a list** of things to be returned:

```
def mean_sd (data):
    ...
    mean = ...
    sd = ...
    return mean, sd

ave, variation = mean_sd (mydata)
```

I personally find this to be a really useful feature of Python.

The following two features were mentioned fleetingly in Chapter 6 without examples of how they are used. They are particularly valuable when writing programs of a decent length, so it is well worth getting to grips with them:

Default values in calls. When defining a routine, you can assign default values to variables, which are used if no value was passed in. For example, the simple script

```
def printx (x=10):
    print (x)
```

```
printx (15)
printx ()
```

yields

```
15
10
```

as its output.

Passing arguments by keyword. In a similar way to default values, you can pass arguments into routines by keyword:

```
def printxy (x=10, y=20):
    print (x, y)
printx (15)
printx (y=10, x=20)
```

yields

```
15 20
20 10
```

Beyond these features, the most substantial thing omitted is **object-oriented programming**. This omission was deliberate because the author believes the advantages of using (or not) an object-oriented approach become apparent only when you have some experience of 'ordinary' programming.

Going further. For some people, programming is as enjoyable as having their teeth extracted; for others, it is a necessity that has to be endured; but for a fortunate few, including the author, programming is a joy. If you have been bitten by the programming bug, you might be interested to learn how to enhance your knowledge.

The best advice I can give is to read other people's code — anybody's code, novice or experienced, working or broken. See how easy it is to understand and, when you find parts of it that are particularly clear, think about how it has been presented to make it so. Looking at well-known examples of good code is also valuable. Although written in C rather than Python, one of the best examples of well-written code that the author has come across is the source of the Tcl interpreter, written originally by John Ousterhout and now maintained by the user community. This is an open source piece of software, so you are able to download the source code and browse through it.

Clearly from this chapter, you will see that there is more about the Python language to be learned; but what about programming in general? A few books that were influential to the author are summarised below; all of them are in the Library.

Software Tools by Brian Kernighan and Phillip Plauger (Addison-Wesley, 1976). This was the first decent book ever written on structured programming and strongly influenced at least two generations of software developers. Although pretty old now, skimming through it should impart the essential ideas of what they are explaining: keep programs simple, write things in short routines, and so on. I often re-read this myself before embarking on a major piece of software.

Programming Pearls by Jon Bentley (Addison-Wesley, 1986). This book describes how a programmer can make their programs simultaneously easier to understand, more elegant and more efficient, mostly through a series of case studies. There is also *More Programming Pearls* by the same author (Addison-Wesley, 1990).

The Pragmatic Programmer by Andrew Hunt and David Thomas (Addison-Wesley, 1999). This book discusses how to become more effective at programming, including 'programming in the large' issues such as testing and version tracking. It is a good read but you will need some experience before the things it talks about make really good sense.

Now, go forth and write beautiful programs!