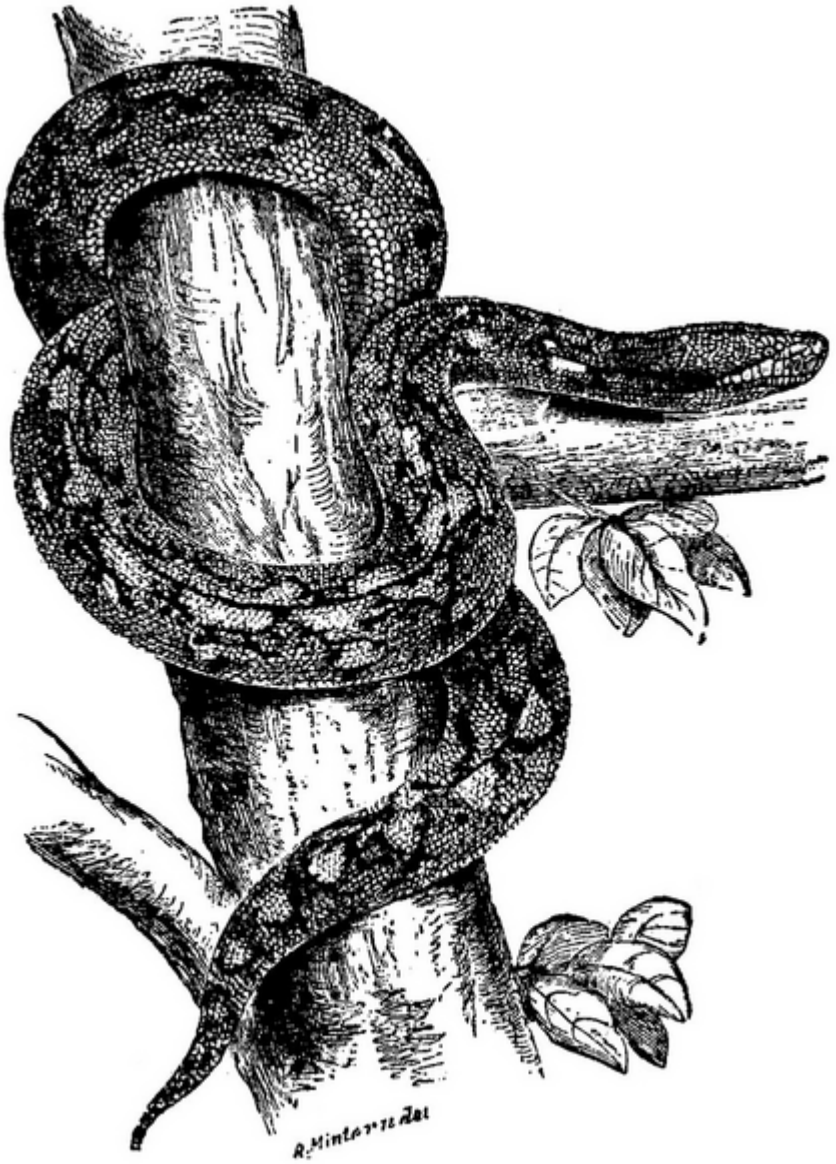


Yet Another Python Book

Release 0.5beta

Tony Jenkins

Apr 09, 2024

CONTENTS

CHAPTER

ONE

GREETINGS!

This is a book about programming. Well, this is a book that is sort of about programming.

Let's start by considering what this book is, what it is not, and why it is like that.

See also:

This book is also available online. The content should be the same. Go here: http://www.tony-jenkins.org.uk/.

1.1 About this Book

This is a book about the Python programming language. Sort of.

This is more a book that introduces the main ideas of programming, and uses Python as the language that illustrates the main ideas.

To be very clear from the outset:

Important: You do not learn to program by reading a book.

Seriously. Programming is a skill, and it is a skill that takes many years to master properly. The purpose of this book is to put you in a position where you understand enough that you can start on that journey.

Suppose you wanted to learn to ride a bicycle, juggle, or repair lawnmowers. You would not expect to be able learn to do any of these things well just by reading a book. (And there are very few books teaching you how to do these things anyway, which is probably significant.) To take just the first - if you wanted to learn to juggle you would first try to get the basics. You might watch some YouTube videos, or get help from someone who already had the skill. Then you would practice, and practice some more. Eventually, you would get to the point where you felt you were a competent juggler.

Programming is like that. Every programmer is constantly learning. Even after 30 years or more experience, programmers are still learning. This book is aimed at getting you to the point that you can start this learning.

This chapter sets the scene for the book, and hopefully convinces you that you are in the right place.

1.2 Design Decisions

There are usually many ways to write a computer program. Some are just as good as others, but sometimes an experienced programmer will have some sort of instinct that one is best. Or it could just come down to experience. Whatever, *design decisions are made*.

Likewise, there are many ways to write a book about computer programming. Let's start by going over the design decisions behind this book.

This book is the result of many years watching (and sometimes even helping) people learn to program. Most of them have got there in the end, but it can be a rocky road, and sometimes a shove is needed.

The key design ideas for this book, in roughly their order of importance, are:

Dynamic
: Paper books (pbooks), in the tech world at least, are dead. Tech moves so fast that it is virtually impossible to produce a traditional pbook that covers current versions of tools and current ideas in methods. This book isn't actually a book in the traditional sense of what that means. It is a set of web pages, generated from a bunch of text files. The content can be changed in minutes, and a new version can be deployed in seconds. Pressing another button can cause a PDF or e-reader version to pop into existence.

Just Enough
: This book does not cover all the small details of a programming language. There is official documentation for that. When a new programmer starts there is no need to worry about all those fiddly little details that only come into play now and again. Why worry about a technique or feature that you'll need very rarely? So, some things are deliberately missed out (although there may be pointers to where the gory details can be found).

Pythonic
: The programming language used here is Python. Python is currently one of the most popular languages around, and is the only really sensible choice for a first language to learn. The aim, though, is to use Python as Python was intended. To be Pythonic, as it were. This means that at various times we will talk about Pythonic things that might not exist (or be as important) in other languages.

One Way
: An important aspect of the original design of Python was that there should always be one, and ideally just one, way to do something. That has maybe slipped in recent versions. Better ways of implementing some features have been added, but the only ones remain for backwards compatibility. But this book will stick to presenting one way to do something, and that will probably be the most recent, and therefore the best, way. If there happen to be more, you'll find them later. There might again be pointers.

Free, as in Beer
: Finally, this book is free, as in beer. You are welcome to use it for anything you want to. The text files can be found on GitHub, and all the tools needed to build the HTML and PDF are free.

This book will also be, in places, somewhat opinionated. No apologies are made for this,

because the opinions are correct.

1.3 Programming

Above all, this book is about *programming*, and the things that programmers do. Before starting to learn how to do something it obviously makes sense to learn what that thing *is*. What is programming all about? What does a programmer actually do? And for that matter, isn't it all AI these days[9]?

A popular image is that programming is a very solitary occupation, with a programmer involved in some sort of late night mortal combat, trying to bend the behaviour of a computer to their will. Well, it does feel like that sometimes, but more often programming is a social occupation, working with others in a creative environment to produce something cool. So this book will occasionally step away from the details of Python to go into **what programmers do**.

Let's be clear about this. Programmers use Google. They use StackOverflow. They wander across the office to ask a colleague to take a look at their code. They use version control tools. They chat on Slack, Teams, and even Discord. They explain programming problems to small plastic ducks[10]. All of these habits, and there are more, make them more productive. While the details are probably beyond the scope of this book, it is so important to know that they exist, and that they are basic parts of the way programmers work. Very often, when a new programmer sees a Senior Developer fix a problem, apparently by magic, all that Senior Dev knew was what to Google. Really.

To put it another way, what experienced programmers have is, ah, *experience*. This means that they have seen most of the common problems before, and know how to solve them. In the trade, this is called *abstraction* - the ability to take a solution to a problem you have seen before, recognise that a "new" problem is actually the old one, and map the solution over. This experience comes with time, and comes from working with other programmers. Another reason why programming is often a social process.

Important: If you know experienced programmers, learn from them. You might have to buy them a coffee, but that will be a good investment in the long run.

---

[9] The short answer to that question is, of course, that no-one knows. Sure, AI could write most of the programs in this book, and it would do a reasonably good job. But would we all be happy relying on software and programs written by an AI? It would seem to be a pretty good idea to have someone look over what the AI is doing, to say the least!

[10] Seriously. It's called Rubber Duck Debugging[11] and is a very useful technique. It works with penguins, elephants, and bears too.

[11] https://en.wikipedia.org/wiki/Rubber_duck_debugging

1.4 Assumptions

So, how to get this experience? To make sure that we are starting from a good place, this book will make some assumptions. Specifically, we want to concentrate on writing programs here. We don't want to be fighting the computer. We therefore need to be able to carry out some basic tasks that any PC user should be able to do.

Important: This aspect is often skipped over in introductory programming courses. It's here for a reason - don't be tempted to skip it here! If you are going to make the computer do something useful, with a program, there's a whole bunch of things you need to be able to do first. Check out the list below!

Note: We are *operating system agnostic* here. Python works just fine on any modern operating system, so we are not going to tie ourselves to anything. More on this later.

This book assumes that you have a PC or laptop available. It doesn't matter what operating system it uses (and we will not worry about OS issues much), and it does not have to be especially powerful. But you need to be able to use it. Specifically, we will assume that:

- You understand how files are organised.
- You can create a sensible structure of folders (directories) to store your files, and know why this is important.
- You can carry out basic file operations, such as renaming, deleting, and so on.
- You can find files if you have forgotten where they are stored.
- You are comfortable installing software, and have the permissions to do so.
- You have an Internet connection, so that you can download the software you need.
- You understand that backups are important, and have access to some solution that will keep your files safe!

Warning: Read the last one above again. Always make sure you have backups. You can never have too many backups. And, most important of all, make sure you can get files back from backups! Losing work through not having a working backup solution can be costly, and painful.

It would also be good to assume that you have some experience of the *command line*. This is likely if you are using Linux, possible if you have a Mac, but unlikely if you have Windows. Some of the details will be covered later, just in case.

This needs to be set out because the ways in which we use PCs and laptops have changed hugely in the last few years. The arrival of PCs in the home has meant that to many people a computer is just an appliance. It's like a fridge, and you can use a fridge without any idea of how it actually works. This is fine as long as all you want to do on the PC is write a letter, read the news, stream the latest Justin Bieber video, or play a game. If

you want to be able to program that computer to do something new, you need to understand something about how it works. Or, at this point, you need to be willing to learn something about how it works.

YouTube is full of videos explaining these things if you need a refresher. We'll talk more about them later.

1.5 Programming Languages

To write a program, we need a programming language. There have been many programming languages over the years. Some have had their time and fallen into obscurity, others are just beginning to gain traction and users. Deep down, though, they are all basically the same. A programmer who learned, say ALGOL[1] in the 1970s could easily be working happily with Java[2] today, and also looking to upskill to Golang[3] in the next few months. Some languages have a habit of clinging on to life even when past their prime (we're looking at you, COBOL[4]), with programmers always needed to support business-critical systems. Some languages, sadly, never really find their niche and just fade away.

This is not to say that all programming languages are equal. There are some fundamentally different designs out there. But the underlying concepts *are* basically the same, and those are the concepts that concern us here. Armed with a good knowledge of the basic ideas it should be possible to pick up any programming language, even the ones that haven't been designed yet.

Note: If you like analogies, we could say that all cars are basically the same. But a small Kia is different to a mid-range Audi is different to a Bugatti. They all have their uses. Some are more popular than others. Some have fallen into misuse. New ones are always interesting. Get the idea?

Taking the analogy a bit further, many languages have a "niche", or an application where they are most suitable. Python is really good for rapid development when the requirements of a system are changing daily. You wouldn't use Java in that sort of environment (unless you hated yourself), but you might decide to engineer the eventual solution with Java, once requirements are well understood. You wouldn't write an operating system in Python, you would use C. But you wouldn't use C to quickly put together an app for your web pages. To use a cliche, it's "horses for courses".

In addition, many programming languages do have a sense of style and idiom. This relates to how the language is used (or how programs are expressed using it). There are also conventions that determine how programmers structure their program code, and how they use the language in other ways. It is important to understand these, and to try to work within each languages's conventions. This is similar to learning any foreign language - it would be possible to translate, say, French into English word-for-word, and the result would be understandable, but would probably seem very strange. A much

[1] https://en.wikipedia.org/wiki/ALGOL
[2] https://en.wikipedia.org/wiki/Java_(programming_language)
[3] https://en.wikipedia.org/wiki/Go_(programming_language)
[4] https://en.wikipedia.org/wiki/COBOL

better translation could be achieved by understanding English, its idioms, and its use. That is why we bother to learn foreign languages!

Important: This book will follow the standard conventions for Python, which are set out in a document called PEP-8[5]. Very different conventions would apply if we were using Java. And they would be different again if PHP were the language of choice.

There are many surveys of the current popularity of programming languages[12]. This is all a bit artificial, because, as noted above, some languages are more suited to certain applications, and some applications are more widespread than others. The top five languages in these surveys, though, are usually fairly consistent, although the order changes. **Alphabetically**, they are:

- C++
- C#
- Java
- JavaScript
- Python

All these languages are available free, and there are extensive free tools, tutorials, and other docs. But where should a new programmer start? When picking a first language to use or learn, we can reason as follows.

1. JavaScript is tightly tied to the Web, and requires knowledge of web languages like HTML and CSS. It is also usually used with higher level frameworks like React and Vue, which change rapidly. For both these reasons it is not a good choice for a first language.
2. C# and Java are basically the same language, and share much with C++. All are *object-oriented*, and are good all-rounders. There are a lot of object-oriented concepts that need to be understood before they can be used effectively, and this hugely increases the amount that must be learned. For that reason alone, they are not a good choice.
3. Python is also object-oriented but, unlike Java, can be used sensibly without objects. It is a scripting language, suitable for rapid development. It is possible to write useful, even interesting, programs using a small amount of code. It is therefore the best choice.

There has been much debate over the years over the first language to learn. Wars have probably been fought over less. But at the moment, Python is the best choice.

[5] https://peps.python.org/pep-0008/

[12] Amusingly (or depressingly, depending on your point of view), these lists often include things that aren't programming languages, such as HTML, CSS, or SQL.

1.5.1 Python

The language used in this book is Python. Python is a well-established language, having been around for over 30 years now. It is very widely used in a wide range of applications. A solid all-rounder. As noted above, it is currently one of the most popular programming languages, and therefore one of the most in-demands skills.

Python has many features that make it the best choice for our first language.

It is multi-paradigm.
Which means that it can be used in a bunch of different ways. This might not seem important, but contrast this with other languages that support only one way of working. In essence, it means we can start simple, and work up.

It is scripting language.
Which means that programs are just plain text files containing a sequence of instructions. A tool called the *Python Interpreter* takes these instructions, and executes them. Simple.

It can be interactive.
Which means that the Python Interpreter can be used as an interactive tool to try things out, check out ideas, and test programming snippets before using them for real.

It is relatively small.
Which means that Python has a relatively small core, so we can hope to cover most of it. But it also has an architecture that allows it to be extended with external modules. Modules exist to do all sorts of cool stuff. It is massively extensible.

It has a simple straightforward syntax.
Which means that it is usually obvious what a program does. Quite often simply reading a Python program out loud can explain what is going on.

Of course, it is not all good news. Python programs can be inefficient, and Python is not the best language if you want to develop something that will run lightning fast in an embedded system. But that's not the point, and it's not what Python is for.

Python is also intended to be fun. Its name is a nod to Monty Python's Flying Circus[6]. Many examples and tutorials draw from the Python canon. PyPi[7], the standard repository of Python packages is sometimes affectionately called The Cheese Shop[8]. You might notice the name of the GitHub repository where this book resides.

Python is completely free. And is also kind of cool.

[6] https://en.wikipedia.org/wiki/Monty_Python
[7] https://pypi.org/
[8] https://www.youtube.com/watch?v=Hz1JWzyvv8A

1.6 Takeaways

Every chapter of this book will end with a sort summary of where you should be now. After this section:

1. You should understand what this book is, and why it is like that.
2. You should have got hold of a suitable PC or laptop.
3. You should have the basic PC skills to manage files and folders.
4. You should understand why there are different programming languages.
5. You should know why the language we will use from now on is Python.

Right. Now to get this setup. We are starting slowly here. The plan is to head off any problems that might get in the way once we start the serious programming work.

CHAPTER

TWO

BEFORE WE START

At this point most books on programming leap right on in and start on the code. This is fine, for some folks. But it means that sometimes people who are new around here can get lost, because they don't really understand the basic ideas. Often there are a many assumptions being made about what you know before the coding start. And, guess what? If you don't know those things you very quickly get lost in a whole load of detail. You off to a bad start from the get-go.

So in this chapter we'll look a few basic ideas that underpin the whole business of programming. These are mostly familiar, that you've probably met many times before, but we need to think about them in a programming way. And these are all things that need to be kept in mind as we carry on.

Tip: This chapter is short, but rather difficult to arrange in the best order. It might be an idea to read through it quickly once, then again, more slowly!

Let's start by thinking about what a computer program actually is.

2.1 Instructions

A computer program is just a set of instructions. The instructions tell[17] the computer how to carry out some task.

Try It!

After reading this section, try it! Write some instructions that someone should be able to follow to carry out some everyday task. Making a coffee, boiling an egg, starting a laptop and opening a Word file, walking from your place to the nearest corner shop …

We are all used to following instructions. This might be to install some software, walk to a new location in a new town, or make a new and interesting soup. This idea is so common that most of us probably carry on without really thinking about what we're

[17] The word "tell" is not a very good one here, because it suggests that the computer as some awareness, and knows what it is doing. Of course, this is not, yet, true. But as you start out in programming this can be a useful way of understanding what is going on - you have a problem, and you are telling the computer how to solve it.

doing. In fact, in some cases we could follow some instructions without knowing what the end result is intended to be!

Important: But remember that we are intelligent. Computers do not, yet, have intelligence. If a human were given instructions that do not make sense, or even placed them in peril, they would stop following them and seek corrections. Computers will just carry on following their program (instructions), regardless, with no *understanding* of what's going on.

Some sets of instructions are presented in a formalised way according to some conventions. That soup recipe, for example, will start by telling us what the recipe makes, and what quantity. It will then list the ingredients, and then present the steps to take, one at a time, in the correct order. It would be very strange to find a recipe that gave the steps before listing the ingredients; it would be very awkward to use. Following a recipe also assumes that we understand basic cookery terms ("mix", "fry", "stir"), probably some abbreviations ("tsp" means "teaspoon", "g" is "gram"), and that we follow it with some intelligence ("fry until browned", "add enough to thicken the soup"). Also, we would query anything that seemed obviously wrong; we would not add 10 *kilograms* of flour to thicken our soup, because that is clearly wrong (and probably a misprint for *grams*).

In general, any set of instructions contains the same elements. There is a *sequence* of steps. And there is *choice* and there is *repetition*. While we usually follow instructions from the first to the last (or from the top to the bottom), this is very rarely done without one or the other of *choice* or *repetition*. In general, instructions contain:

Statements
: A single instruction to do something: "Add the onions", "Cross the road", "Connect the USB cable".

Choice
: Do one of a number of things (Statements), depending on what is observed. "Add water if required", "Cross the road if it is safe to do so", "Connect the cable if an external monitor is to be used". What is done depends on whether something is observed to be *true* or *false*.

Repetition
: Do (a Statement) several times. "Add the flour until none remains", "Walk uphill until you reach the station", "Click to install each of the updates that will download".

These three are the three most basic building blocks that make up any set of instructions. They are also the basic building blocks of a computer program.

When writing a program, a programmer supplies the computer with an ordered list of instructions, along with the choices and repetitions that are needed to achieve a successful result. The instructions are expressed in a programming language, like Python.

What do these instructions do? They store and manipulate *values*. If you think about it, it's amazing that these simple ideas can get us to complex software programs such as modern games and office tools.

2.2 Values and Types

In many daily tasks we are involved with using and manipulating *values*.

Some of these values are numbers, or *numeric*. We might have to pay a bus fare, buy a needed amount of something, or walk a certain distance. We are good at recognising these values and carrying out tasks that involve them.

Generally to do this we think in units of tens, or fractions of tens, or multiples of 10. Our currency is based on 10s, and we are used to working with 10s. While in the UK we hold on to measuring longer distances in miles, we are increasingly using the metric system, which is based on 10s.

Multiples of 10 (powers) are handy for bigger numbers: 10^2 is 100, 10^3 is 1000 and so on. These numbers are all integers, or whole numbers. This idea is at the heart of the metric system.

Fractions of 10 are used for more exact numbers, and numbers that represent part of a whole. These are *floating-point* numbers. They can also be represented as powers: 10^{-1} is 0.1, or a tenth; 10^{-2} is 0.01, or a hundredth.

It's often stated that our obsession with working in 10s like this comes from the usual number of fingers we observe on our hands. This could be true, or it could just be that this is something we are so used to doing, and something we are taught from an early age, that it's impossible to think of any other way.

Of course, we often use values that are not numbers. An example of another *type* of value we use every day is *characters*. These could be letters, digits, punctuation marks, or even emojis. A sequence of characters might represent a name or an email address. They could also represent a phone number - in this case the characters are also digits, but they are characters unless we plan to add up phone numbers, which is unlikely. A single character can have meaning - a grade on a test, for example. A collection of characters can also have a meaning, sometimes only if they are read in a particular order. We might call such a collection a *string*; the order of characters in a string is usually important.

So, we use *values*, and values have *types*. We carry out operations on values, and the operations we can do are determined by the types. For example, we often add up numeric values to work out how much to pay. We don't add up character values, but we often use a string of them to, say, send an email. We might also compare values, to see if they are the same, or if one is bigger than the other. We might also test to see if a value is in a particular range, or if it is a particular value. All these things, obviously, also go on inside a computer program.

2.3 True and False

There is another type of value that is very important in Computing. It gets a separate section here because maybe it is a little less obvious, even if we do deal with it in everyday life. We deal with the ideas of truth, falsehood, and fakery. Take any statement, and we might say that it is *true* or it is *false*.

Note: Arguably there is a third state, where we know that a statement is true or false

but we do not know at present which.

Any statement can be tested, and from the test its "truth value" can be determined. That said, some statements are always true, and this can never change:

```
Python is named after Monty Python's Flying Circus.
```

Some statements, on the other hand, are always false, and this will never change[18]:

```
Johnny Depp created the Python programming language.
```

Often, statements are either True or False, depending on something that can be tested. So this statement is true as I type this:

```
It is Tuesday today.
```

It could be true as you read this, or it could be false. I have no way of knowing right now. I have just read it on a Monday, so now it is false. In order to determine whether it is currently true or false, you would need to test it, maybe by checking your phone.

Programming revolves around these two values, for reasons we will see in a moment. When it comes to making sure that a program works correctly, they are probably the most important values! A statement is true, or it is false. Perhaps it is true that a program's user has clicked a button in the interface, and so the program better respond in some useful way. Maybe it is false that the user has permission to access that part of the application. Maybe it is true that Mario just drove into a banana skin, and so the program better make him skid.

True and *False* are called **Boolean** values, named after George Boole[13], who in 1847 first applied mathematical ideas to logic[20]. The word Boolean is usually written with a capital B for this reason.

Boole also showed how True and False can be combined using what are now known as Boolean (or logic) operators. For example, if there are two statements, **and** both are True, we can agree that a combined statement is True:

```
John Cleese wrote the Parrot Sketch.
The Parrot Sketch was in Monty Python's Flying Circus.

John Cleese wrote a sketch that was in Monty Python's Flying Circus.
```

There are a whole bunch of *logic operators*, but most of them are only really useful when working with electronics or hardware. For programming purposes, three are usually enough. `AND` and `OR` combine two logic values (let's call them `A` and `B`, like this:

[18] He didn't. See Guido van Rossum[Page 12, 19].
[19] https://en.wikipedia.org/wiki/Guido_van_Rossum
[13] https://en.wikipedia.org/wiki/George_Boole
[20] This is a rare case in computing of an idea being named after a person (eponymy). Bonus credit if you can find more.

A	B	A and B	A or B
False	False	False	False
True	False	False	True
False	True	False	True
True	True	True	True

If you read it, the result is very much as you would expect if you just read it out loud:

```
A is True.
B is True.

Therefore A and B is True.

A is True.
But B is False.

Therefore A and B is False.
```

The third useful operator, `NOT` just flips the value. So a True becomes False, and vice versa:

A	not A
False	True
True	False

Why is this important? Let's look at how computers (for the want of a better word) "count".

2.4 Binary

So, how does a computer store the data it needs? Computers do not have 10 fingers, but they do have electrical switches[22]. A switch has two possible values; it can be "on", or it can be "off", just like a light-switch at home.

So computers count in 2s, which is called *binary*.

Remember that humans count in 10s. We find 10s easy, probably because we are taught to use 10s from an early age. The origins of this are probably that we have 10 fingers, and we can use these to count. Children are still taught to count in 10s, and to use their fingers to help them.

Powers are important here. This is when a number is multiplied by itself. To handle larger numbers we give certain powers of 10 special names, so:

- 10 x 10 (or 10^2) is a hundred.
- 10 x 10 x 10 ((or 10^3) is a thousand.

[22] In early computers, "on" and "off" would have corresponded to two positions of an actual switch or button, of course.

and so on.

Note: Counting in 10s like this is called *base 10* or sometimes *denary* or (less accurately) *decimal*. In Computing we also sometimes meet *Octal* (base 8) and *Hexadecimal* (base 16). See that those last two are powers of two. That's important. It's all to do with how computers store data, with the memory arranged into chunks of 8 bits (a *byte*). More on this later.

Computers do not have fingers! A computer is an electronic device, based around *switches*, where a current is either flowing, or not. So a switch is something that is either "on" or "off". So if a sentient computer could count, it would count in 2s, in much the same way as humans use 10s. This is called *base 2*, or *binary*.

This means that every data value stored inside a computer, either in memory or on a disk, is *encoded* in binary. The details are not important here, but an overview is. Basically:

- An integer can just be stored as its binary equivalent.
- Various cunning ways exist to store floating-point numbers with fractional parts[21]. Again, this usesd binary.
- Character data can be stored by using a table to convert between integer values and the characters. The most common one is Unicode[14]. You may also see references to Unicode's predecessor, ASCII[15], which offers a smaller set of characters.

So if a computer could somehow write out an integer it would have just two symbols to work with, *1* and *0*. It would also work in powers of 2: 2^2 is denary 4, 2^3 is denary 8, and so on.

Hint: To avoid confusion it is usual to add a subscript to a number when different number bases are involved. So 8_{10} means the number 8, in denary (base 10). Likewise, 1000_2 is a binary value. (The two happen to represent the same number).

Important: Knowing and recognising the powers of 2 is a hugely important skill in computer science:

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
```

If you have ever bought a laptop, you will recognise those numbers from the system specs! Currently, laptops tend to have either 8 or 16 GB of memory, and offer either 256 GB or 512 GB of storage. These are all powers of 2, and it all comes back to how computers store data.

So how would a computer represent, say, 3_{10}?

[21] This means that there are some decimal numbers that it is impossible to represent precisely inside a computer. Different ways of representing numbers with decimal parts exist, and have different levels of accuracy, but this is not something you need to worry about in normal programming.

[14] https://en.wikipedia.org/wiki/Unicode

[15] https://en.wikipedia.org/wiki/ASCII

Easy. Look at the powers (it helps to see them in reverse order:

```
2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1
```

3 is just 2 + 1. So, in binary 3 is 11_2.

How about a bigger number, like 42? Calculate it like this. First find the powers of 2 that are needed (it's just like giving change using the smallest number of coins possible):

```
42 = 32 + 8 + 2
```

Add in the missing ones:

```
42 = (1 x 32) + (0 x 16) + (1 x 8) + (0 x 4) + (1 x 2) + (0 x 1)
```

And read off the 1s and 0s. In this case 42_{10} is 101010_2.

Most of this will be hidden as we write programs, but it helps to understand that this is happening "behind the scenes". Let's now think about how a computer stores and processes these values.

2.5 How Computers Work

This is not the time or place to go deeply into the inner workings of a modern computer, but it *really* does help to understand programming if you have some idea of what's going on inside the box. After all, that's what a program is for; it's to make the computer do something useful.

Note: What follows is very imprecise, but is mostly accurate, at least from a programmer's point of view. This is not a book about hardware!

If this was a hardware book, you would learn that the main components of a computer are a *CPU* (Central Processing Unit), and some *memory*. The CPU is the part that carries out the instructions, and the memory is where the data is stored. There also needs to be some way to get data in, and get results out. Data being processed by the computer can either be volatile, or non-volatile. Volatile data is lost when the computer is powered off, and non-volatile data is not.

So, nside a computer is some memory. The memory stores all the programs that are running, along with the data they are using. It's usually called RAM. The memory is volatile (everything in it is lost when the computer is powered off), so there is usually also some less volatile storage, which these days still means a disk drive, or it could be "Cloud" storage. There is usually a lot more non-volatile storage available, simply because it's a lot cheaper.

In either case, data is stored in binary, as 1s and 0s, and binary is used to represent all the different kinds of data that a program might use. The computer spends a lot of time shuffling data between the volatile and the non-volatile storage, which can have a significant impact on the performance of a system.

The heart of a computer is the CPU. This is the chip that can carry out operations on data. Usually it only has a very few operations it can do, like adding two numbers, or

comparing two numbers, but by combining them we can write complex programs. The CPU can only work with programs and data that are in the volatile memory. To allow for this the CPU has a small amount of memory internally, and any data needed is copied into there so it can be processed. (That's another performance bottleneck).

So, when a program runs, it is first loaded into the memory. If the program requires some data (say a user has to type in a value, or some file is needed off a disk), that data is also stored in memory. When the CPU needs it, it is copied into the CPU's memory, where it can be processed. Once done, the result is copied back into the main RAM, and the program carries on. These days this all happens very quickly, but it's still happening. It is still necessary to write programs that are efficient, and that don't waste time copying data around. That's why we need to understand what's going on

It is, obviously, much more complicated than that, with a modern CPU having multiple cores to allow it to process many things at the same time. But hold on to this idea of data being stored in memory, copied to the CPU, and written back. It's important.

We finish with a look at how data is stored in that non-volatile memory (usually a hard-drive of some sort).

2.6 Text Files

Important: This section is very important. Modern operating systems, especially Windows, condition us to associate files with the applications that use them. We double-click a file and the appropriate application opens, as if by magic. This is fine (and undoubtedly convenient) for the user who sees their laptop as an appliance, but it gets in the way when we want to do serious work.

"Stuff" on a computer is organised into files (which are also stored in a binary format). A file might represent a document, an image, or anything else that might be useful. Often a particular application is needed in order to use a file, so we sometimes talk about "Word Files" or "Photoshop Files". Files for applications like these are usually stored in some format that makes them useful only with that application; you can't open a Word file with Photoshop, or vice versa. This is OK, but remember that the files are only useful for as long as the appropriate application is available. If Word is suddenly unavailable (or, more likely, is not installed on a particular computer) all those fine Word files are useless.

The simplest file is just a *plain text* file. It contains characters, encoded in binary, probably in turn using Unicode. The characters could represent anything - a shopping list, a Python program, a set of system specs. This format has been around for as long as modern computers have been. Should we find a plain text file from the 1960s or 1970s we would have a very good chance of accessing its contents in the 2020s.

The tight coupling of applications and files is becoming an issue in general Computing. Files created with applications that have become obsolete are themselves obsolete, with the owners unable to get at the data within. This is a big problem for businesses that rely on this data, and often means that they have to spend a lot of money maintaining obsolete software just so they can get at their historic data. The format in which we store our data is important - we can access documents written on paper hundreds of

years ago, but getting at a document written in Wordwise of a home microcomputer in 1985 is basically impossible[23].

One format that will always be used and will always be decipherable is that good old *plain text*. In Windows, such files are often opened and modified with the Notepad editor, but they can be opened and modified with many, many tools. Programs are written in plain text files. This means that programs written decades ago can still be read and understood, even if the computers that could run them are long gone. It also means that *every* computer has a tool that can be used to edit programs in plain text files (assuming the computer has some sort of keyboard!).

A side effect of this is that there is a lot of choice when it comes to creating Python programs (or programs in any other languages). Some tools are sophisticated, and offer features specific to Python. Others are more general purpose. Some are very basic, but at least allow you to get the job done. More on these later.

Hint: If you have some valuable data, consider keeping it in a plain text file. So if you lose that beautiful Word CV, at least you have the data so you can rebuild it. And if you really want to store some data so it will be around for 50 years, print it out and put the paper somewhere safe.

Tip: This book applies this principle! The files that make up this book are plain text. A simple mark-up language called reStructuredText[16] is used to mark sections, fonts and so on. Even if that language was no longer supported anyone could take the text files, and reasonably quickly extract all the content. The HTML or PDF that you are looking at is created from the plain text files by a bunch of Python programs (which are themselves plain text files, of course).

A side issue is that the files that make up this book can be edited on basically any computer.

The practical upshot of all this is:

Important: There is no such thing as a "Python File". A Python program is a plain text file that happens to contain the instructions that make up a Python program. It can be created or changed with any tool that can work with plain text files. As we will see, that tool could just be good old Notepad, or it could be something more sophisticated.

A second upshot is that this book can be neutral as regards the operating system, and overall toolset you choose to use for your programming. Any OS can handle plain text files, and there are many, many, great tools out there. This book will not be tied to any particular toolset, and will not assume that you have any particular tools, apart from Python, installed.

[23] We're talking about the format of the data on the disk here, but the same applies to the physical format. Not so long ago, for example, every PC had a CD drive. Now, very few do. So what shall we do with all that data we archived to CD in the 1990s and 2000s? Let's hope none of it was important, eh?

[16] https://docutils.sourceforge.io/rst.html

2.7 Takeaways

The takeaways from this chapter are very simple. You need an understanding of each topic above. For example:

1. You need to understand (in everyday terms) instructions, sequence, choice, and repetition.
2. You should know that values have different types, and have some idea of how these are stored in a computer.
3. You should have a basic idea of how a computer stores and processes data.
4. You should understand that files have different formats, and why plain text is the one format to rule them all.

CHAPTER

THREE

GETTING STUFF TOGETHER

Promised we would go slowly. But it's very important that we get everything in place before starting to program. That way we will be able to concentrate on the actual programming, rather than getting distracted tweaking the environment.

So, now to get everything together so that we can start programming. You may already have access to the required software provided by someone else, but there really is no substitute for setting it all up yourself. For one thing, you learn how to do it. For another, you can set things up just as you find you like them. And you can change them as the way you like to work changes. No two programmers work in the same way. If you want to get good at this you are going to have to think about how you work best.

Important: Bearing in mind that every programmer works best in a different way, this book will not assume that you are using any particular tool. It's up to you. By all means ask for recommendations, but don't feel you have to follow them!

First, we need a reasonably powerful PC or laptop[37]; anything from the last five years or so will be fine. The programs we will be working on here are not going to need huge amounts of system resources, so that old laptop on the shelf will do fine if you blow the dust off. There are no important requirements for disk space or memory for Python itself, but when we come on to think about development environments you might want to check out their requirements.

So, having dusted off some kit, and that it has a web browser and Internet connection, there are two main things to get installed.

1. The Python language.
2. An IDE or editor that will allow us to edit (and ideally run) programs.

Important: Any section like this is very difficult to keep up to date, as versions are always changing, and web resources are moving. Be prepared to seek out information if things don't look exactly as described here!

But first let's consider the operating system itself. The choices start there.

[37] A laptop is best, because it can come with you. The ideal setup for most programmers is a laptop along with an external monitor on any desk where they are likely to roost.

3.1 A Note on Operating Systems

The operating system is the software that makes the computer "go". It controls access to files, takes care of video and audio, and does all the things that makes the computer useful. Strictly speaking, on top of the operating system there is a more software, such as a window manager, which deals with all the tricky bits of the interface, but in some modern operating systems the distinction is rather blurry.

It is likely that the operating system you have used most up to now is some flavour of Microsoft Windows. Before we start programming round here, this is a good point to pause and to realise that there are alternatives to Windows. Remember that Windows is just the operating system that happens to be running on your computer. In most cases it is possible to replace it with something different.

Of course, Apple Mac macOS systems provide one such alternative, all in a very attractive Apple package with curvy corners, and closely tied to their very neat hardware. This is a very popular setup among programmers who prefer an interface with a rich attractive look-and-feel, and (if we're honest) also think that the hardware itself looks cool. This system becomes even more attractive for those who favour an iPhone, have an Apple Watch, and so on. The Apple ecosystem is very well integrated, and it is a very good choice for those who like to have everything "just work".

The Open Source operating system Linux is also popular with programmers, who value its free-ness, power, and customisability. Linux systems come packaged as *Distributions* (usually "distros"), and generally "just work" when installed on standard hardware. The only problems are usually with "bleeding edge" hardware, or where various big corporations have done deals to lock users in to using their products.

Popular Linux distros at the moment are Ubuntu[24] and Linux Mint[25]. Both these allow for a wide choice of window managers, which in turn allow users to customise their systems to their precise liking. Since Linux is really aimed at users who are going to be writing programs, and using the associated tools, it is often the case that such things in Linux "just work" rather more than they do in Windows.

So there is a choice. The most recent StackOverflow Developer Survey[26] showed that macOS and Linux are only a little behind Windows in the current popularity stakes.

Windows obviously has a huge advantage in terms of number of users as it comes ready installed on most laptops, even if the user has no intention of using it. (And, as we will see, if you decide to use a modern IDE, it will work just the same on any of the three, so the operating system itself becomes less relevant.)

Using macOS obviously requires buying a Mac, so this is a choice not to be made lightly!

Linux, however, is free, and is easy to install. It even lurks within recent releases of Windows in the form of the Windows Subsystem for Linux[27]. Most Linux systems will boot from a USB drive, allowing you to try them out without installing anything. A flavour of Ubuntu[28] or Mint[29] are good places to start. And if you know a Linux user they are very likely to be very keen to show you how it works.

[24] https://ubuntu.com
[25] https://linuxmint.com
[26] https://survey.stackoverflow.co/2022#section-most-popular-technologies-operating-system
[27] https://learn.microsoft.com/en-us/windows/wsl/install
[28] https://ubuntu.com/#download
[29] https://linuxmint.com/download.php

Linux is also a lot less resource-hungry than Windows, so it can be a fine choice for older hardware that struggles to run current versions of Windows.

This book is not going to preach (any more) about which operating system is best to use, although it was created on Linux, specifically Linux Mint with the Cinnamon window manager. The message is that you should be using the operating system that makes you the most productive. You should know that there is a choice, and you shouldn't be using something just because you always have done! And whatever system you choose, make sure you *really* understand how to use it.

Ok. Sermon over. Let's get Python flying.

3.2 Getting Python

Python is free, and can be downloaded from The Python Home Page[30]. At the time of writing the current version is 3.12.2. The Downloads section lists the currently available versions, and shows for how long each is supported (releases are usually supported for five years). If you have a choice, just pick the most recent.

Note: The Python Home Page is a good place to start for all things Python. There are tutorials, documentation, and links to other resources. It is worth spending some time looking around.

There are Download links for Windows and Mac. Windows users will also find it in the Windows Marketplace, but it is usually easier to install from the main Python downloads page, and this also means that the version will be the latest one.

Linux users most likely already have Python installed[38], so have nothing to do. Opening up a terminal and typing:

```
$ python3 --version
```

will reveal whether Python is installed, and will show tghe version number. If it is not installed, Linux will probably show the commands to install it.

New versions of Python are released regularly (the schedule is on the Downloads page). It is unlikely that you will need to update while you are learning the basics but if you do, head back to the Downloads page. Linux users will find that Python will just update automatically, Windows and Mac require fresh downloads.

> It is usually only important to update if the second part of the version number changes; so updating from 3.11 to 3.12 is important, but from 3.12.1 to 3.12.2 is not.

Important: Whatever version you install, be very sure that it is some flavour of Python 3. Downloads for the old Python 2 are still available for various good historical reasons,

[30] https://www.python.org

[38] The way that Linux distros are updated means that this might not be the latest version. That's fine. The version will update from time to time.

but Python 2 cannot run most programs written in Python 3.

3.2.1 The Python Interpreter

As mentioned before, a useful feature of Python is the *Python Interpreter*, a handy tool that has many uses, but which comes in useful most often for testing out fragments of code. It will serve here to check that we have Python installed.

Hint: The default prompt in the Python Interpreter is three > characters, viz >>>. And by convention whenever you see something like this:

```
>>> print('Spam and Eggs')
```

this shows code being typed at the interpreter prompt. Be sure not to copy the prompt if you copy this code. If you are reading this online, the magic copy button that appears when you hover the pointer over the code will not copy the prompt. Good, eh?

Firing up the interpreter will show the current version of Python running. It can be found by hunting through menus on Windows (so consider making a shortcut on the Desktop or in the Taskbar), or on Linux by opening up a command-line and typing:

```
$ python3
```

The response will be something along the lines of:

```
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

So here we have Python version 3.10.6, which happens to be running on Linux. Things will obviously look slightly different on Windows, but all the information will be there.

To exit the interpreter, simply issue the *exit()* command:

```
>>> exit()
```

Assuming you see the interpreter, you do have Python, and can carry on to install an IDE.

3.3 Choosing and Getting an IDE

An IDE (that's *Interactive Development Environment*) is a tool that allows a programmer to create, edit, and run programs, all in one handy interface. They are not strictly required for programming, and using one does introduce something else to learn, but in the long run a good IDE makes the whole programming task easier and more enjoyable.

There are many choices here, and as with operating system, this is a personal choice. The important thing again is not just to use a tool because you have been told to. There are many, many choices. And the choice you make now may well stick with you through

your whole career. Evaluate some of the options, and pick the one that fits best with your needs.

Tip: Most of the modern IDEs look basically the same, and many share similar menu structures. Keyboard commands are similar, too. It is usually the case that *<CTRL>-Z* will undo the previous edit, for example. So, if you pick one IDE for now, it wouldn't be too much effort to change to another later.

This section will outline some of the main choices, but there are more. Remember that most of the popular IDEs will run on any operating system, so the choice of one does not impact on the choice of the other.

A second glance at the same StackOverflow Developer Survey[31] shows the current state of play. *Visual Studio Code* is the top choice. This is Microsoft's free code editor offering, which started out life as part of Visual Studio. If you look into these, be sure to realise they are different things!

This score is a little artificial because VS Code has an ecosystem of plugins that means it can be used to work with any language, making it a very general-purpose tool, popular with developers using different languages. The third choice, JetBrains IntelliJ is for Java only. Its Python cousin, PyCharm, is also high on the list, as are their close relatives WebStorm (for JavaScript), PhpStorm (PHP), and others from the same stable. So VS Code is top, but not by as much as it might seem.

An obvious reason for its popularity is also that VS Code is free. JetBrains IDEs are commercial software, and require a paid-for licence. to any business employing a significant number of developers, this is important! But JetBrains do offer free versions of their most popular IDEs - IntelliJ and PyCharm - so that does keep the competition going.

Let's look at the two obvious contenders. The screenshots here are taken on Linux, but the IDEs look the same on other operating systems.

3.3.1 Visual Studio Code

VS Code is a relatively new tool, but one that has gained a lot of traction very quickly. It is available for Windows, Mac, and Linux, and is extremely configurable and tweakable. Certainly it's a tool that is not going to go away any time soon. Pedantically, VS Code is a *text editor* rather than an IDE, because out of the box it has limited support for specific languages. That said, its power increases greatly with the addition of plug-ins, and there are many available for Python.

The image above shows VS Code being used to create a short Python program. The plug-ins for Python have been installed, and a rather cool dark blue colour scheme (*Winter is Coming*) is in use. The small arrow to the top right would run the program, with the output appearing just below the program code. All very neat.

VS Code also scores in the popularity stakes because it is free. It can be grabbed for free[32]. The download page detects your operating system, and offers helpful instruc-

[31] https://survey.stackoverflow.co/2022#section-most-popular-technologies-integrated-development-environment
[32] https://code.visualstudio.com

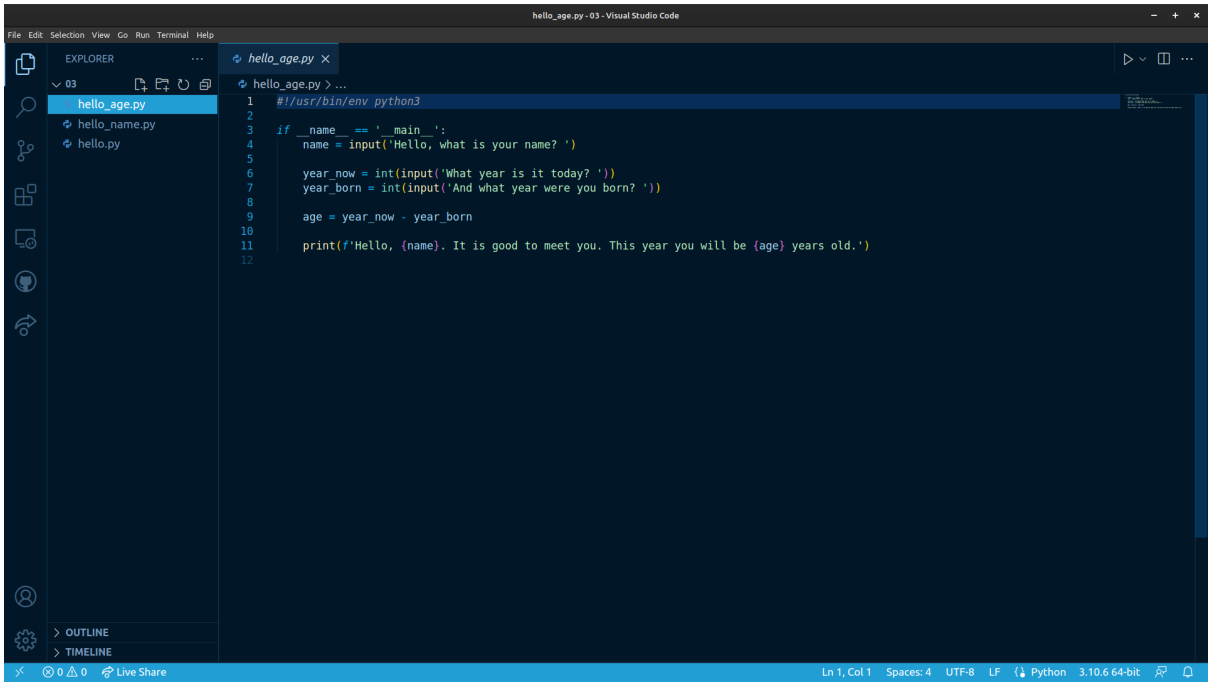


Fig. 1: VS Code with Python Program

tions. It is then a case of installing plug-ins (called *extensions*) for the required languages. This is just a case of opening the settings, and searching for "Python".

Hint: Rather than searching for Python plug-ins, you can just enter a short Python program. VS Code will detect that it is a Python program, and offer to install the plug-ins for you. The same trick applies to other languages.

VS Code is being actively developed, and new features are added regularly. It is a fine choice of IDE, especially if you plan to use different languages for other projects.

3.3.2 PyCharm

PyCharm is a commercial product[39], developed and marketed by JetBrains[33]. Happily there is a free "Community" version; this lacks many of the features of the "Professional" product, but those are not likely to be of much, if any, interest to us here. The Community Edition will be fine. It is a straightforward download[40], and as usual the download page will detect your operating system and offer the correct version.

The image above is actually the "Professional" version of PyCharm; the "Community" version would have fewer menu options across the top. It is using a third-party plugin for a theme, again a dark one. Clicking the small green arrow to the left of the program would run it, and the output would appear below.

[39] JetBrains offer educational discounts for students and staff. At present this consists of free access to the full versions of all their tools. All that is usually required is to create an account with a University or College email address.

[33] https://www.jetbrains.com

[40] JetBrains also offer a tool called the JetBrains Toolbox, which is a one-stop shop for all their tools. This is a good way to keep all the tools up to date, and to install new ones. It is also free. It may be that this will become the official way to install and manage JetBrains tools in the future.

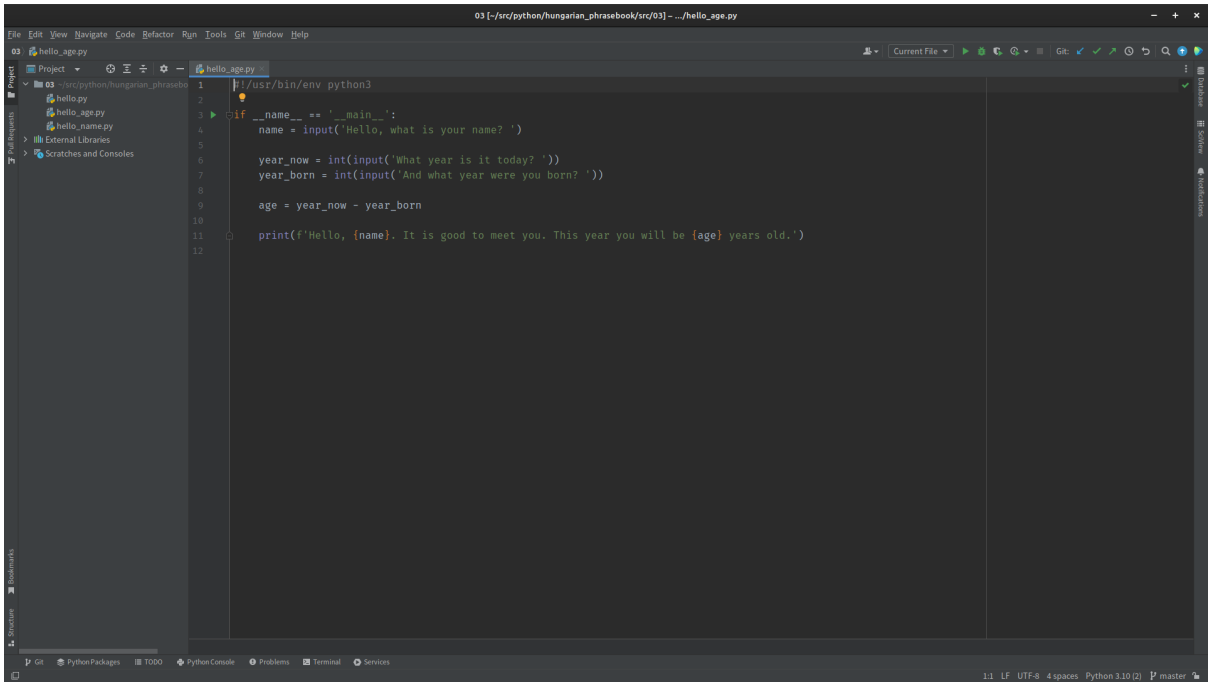


Fig. 2: PyCharm's Previous UI with Python Program

As a full-featured IDE, PyCharm will do a lot more out of the box than VS Code. This is a good thing in that there is less to install and PyCharm is the more powerful of the two. But it can be a bad thing in that many of the options available are irrelevant to the current project, and can get in the way. This is probably why in early 2023, JetBrains started development of a new interface for PyCharm.

This new interface is less cluttered (all the menus are actually still there, hidden under the "burger" top left), allowing the programmer to concentrate on the task at hand. At the time of writing this, the new UI has become the default, but the older version is still available under the Settings panel.

The new UI reduces the number of options that are available by default, with the result that it makes PyCharm look rather like VS Code, which may not be entirely a coincidence. This new interface is *highly recommended*.

3.3.3 Picking and Choosing

The choice of IDE is a personal one, but also one that can stay with you for a long time. Neither PyCharm or VS Code is going to go away any time soon, so time invested in learning how to use them is time well spent. Both are highly customisable - colour schemes and themes are just where it starts. It is worth spending time seeking out tutorials and other hints and tips.

Think of this process as being similar to buying a new car. It is usual to test drive a few new cars so as to get a feel for them. And also to investigate what options are available, and how they can be customised. There is no one car that is universally acceptable, and likewise there is no one IDE. Everyone has preferences, and favourites. And these can change over time.

It is also worth looking beyond the features that the IDE provides. For example, VS Code

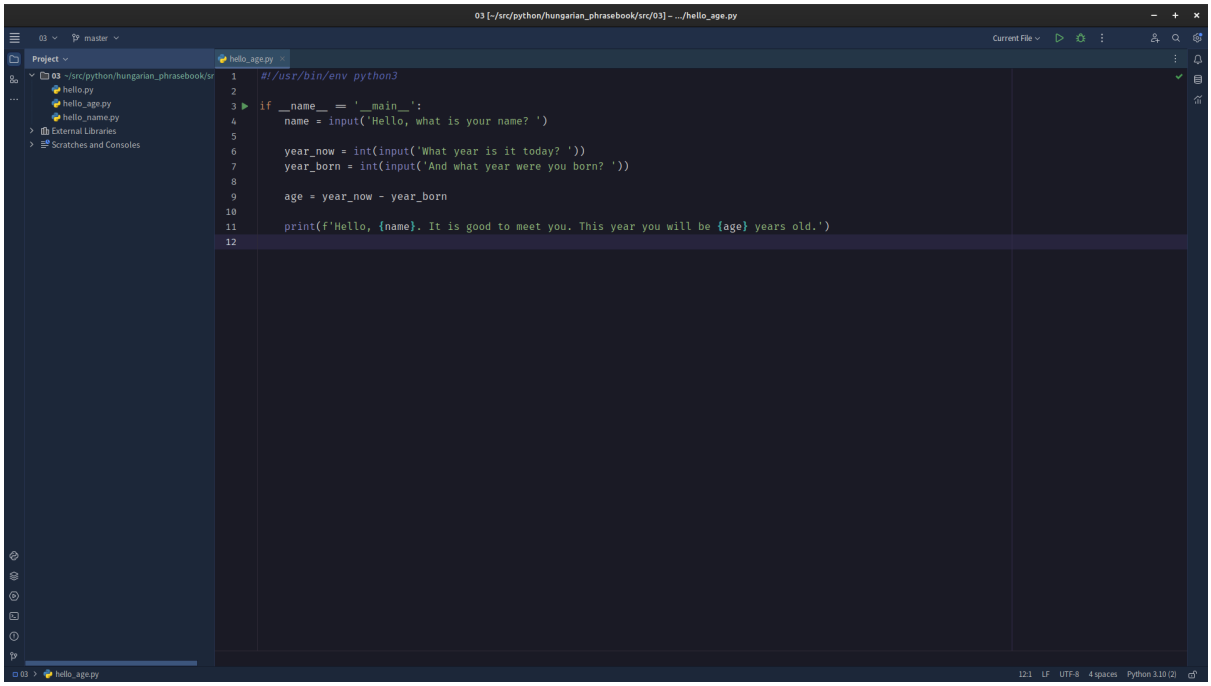


Fig. 3: PyCharm's New UI with Python Program

is rather more lightweight than PyCharm, so tends to start faster. This would make it a better choice on older hardware. And, of course, PyCharm is a commercial product that usually requires a paid-for licence; this is not a consideration as long as the Community Edition meets your needs, but could become an issue in the future.

Your IDE is going to be the main tool you use when programming. Tools are very personal things. It is worth getting them right. You are going to be spending a lot of time with your tools, so make sure they are the right ones for you.

Note: When using an IDE, much of the operating system underneath is hidden. PyCharm and VS Code work much the same on Windows, Linux, or Mac. This means that while the operating system choice might seem to be the most important, it really isn't. This book was mostly written with PyCharm, usually running on Linux, but occasionally on Windows. Needs must.

There are obviously other options for creating programs. After all, programs are just good old text files, so good old *Notepad* would do the job. Use whatever tools make you most productive. But make them yours.

3.4 Other Tools

It's also worth considering spending some time looking at other tools that will be of use when programming. Obviously some sort of backup solution will be needed, for example. This could be some simple Cloud-based storage, such as OneDrive that comes bundled with Windows.

Important: A USB stick is not a backup solution.

It is also worth looking at *version control* tools. These are tools that keep track of programs as they are developed, changed, and otherwise maintained. These will become essential later on, but there is no good reason not to get started with them now. The standard is Git[34] which is free and available for all operating systems. There are plenty of tutorials to give you the basic idea, and Git is actually built in to both VS Code and PyCharm.

A site like GitHub[35] combines version control and cloud storage[41]. As well as keeping work safe it is also a fine place to build a portfolio of work, such as a Book[36]. As we work in more mobile ways, it is very useful to keep program code in the Cloud, so that it can be downloaded and work on using whatever PC or laptop happens to be available at the time.

3.5 Takeaways

The most important messages here are:

1. Choose your toolset for programming wisely. Listen to others, but make the choice yourself.
2. Once chosen, customise it. Devote time to this. Get it *just right*.
3. Time invested in learning and customising a tool might not produce any programs, but it is not wasted time. Quite the opposite, it is time that will pay you back over and over again in the future.

This book includes Python programs, so the whole thing was developed using PyCharm (new interface, tweaked colour scheme). The files themselves are on GitHub. The reason for all this is that is what the author prefers to use. You are encouraged to be different!

Now, let's make some code.

[34] https://git-scm.com
[35] https://github.com
[41] See also GitLab[Page 27, 42], BitBucket[43].
[42] https://about.gitlab.com
[43] https://bitbucket.org/product
[36] https://github.com/TonyJenkins/hungarian_phrasebook

CHAPTER

FOUR

GETTING STARTED

Now to get serious and get programming.

There are three things to master here:

1. How to enter a program.
2. How to get it to run.
3. How to find the results.

It has been said[44] that once a new programmer can do these three things, the rest is comparatively easy!

It is important not to skip over this part so as to get to the more useful and interesting programs. We are going slowly for a reason! It is vitally important to be able to work effectively with your IDE. You need to be able to efficiently enter programs, run them, and check the output. Most likely you will make mistakes entering the programs (everyone does), so you will also need to be able to find and fix these errors.

We will think more about errors later on, but for the moment let's agree that there are two types of error:

Syntax Errors.
: The program written does not conform to the rules of Python, so the Interpreter does not understand what it needs to do, and so the program fails. Usually, the IDE will spot most of these and highlight them in some way. This means that syntax errors are usually easy to spot and correct.

Semantic Errors.
: The program is correct Python, and so can be run, but does the wrong thing. This is discovered by comparing the results of running the program to the desired results should the program be running correctly. Since this can be a lengthy and somewhat tedious process, these errors are harder to spot and correct.

Do not panic if you get errors. Everyone does. What you need is experience in spotting them, and then fixing them. Once you've seen an error the first time you'll be able to fix it next.

Tip: As you enter programs you may start looking around for a "Save" button to write the current file to the disk. There is probably no need for this, as IDEs tend to be setup

[44] It is said in *The C Programming Language* by Brian Kernighan and Dennis Richie, a book affectionately known as "K&R".

to save as you type. This is the default for PyCharm, and can be turned on for VS Code. Obviously it can be set to your preference.

One way to learn some programming is simply to enter some programs, and run them[45]. Gradually you come to understand what all the commands do. So let's start with three short programs, and try that. Remember that a program is just a text file. So you need to be able to create that file using your IDE. And you need to store the file *somewhere where you will be able to find it again*. That might seem obvious, but it often gets forgotten. Pay attention to where your IDE stores the file!

Tip: You might find it easier at first to create the folder for your programming project, and even a file, using the usual operating system tools. The file can be created with a simple text editor, like Notepad. The folder can then be opened with the IDE.

First, let's meet some programs.

4.1 Three Programs

Here are three programs. You should be able to see what they do just by reading them. If some detail isn't obvious at the moment, no need to worry. For the moment we are just interested in getting them to run.

Enter them, one at a time, in separate files, into your IDE[46]. If you have syntax errors the IDE will probably spot them and highlight them in some way. How this looks depends on settings and colour schemes, but expect some sort of underlining as the IDE expresses its unhappiness. If they all look correct, you can then run them in whatever's the most convenient way. Remember the little green arrows in the two IDEs introduced before.

> **Caution:** Take care when copying programs from a book. Long lines can get split in awkward places. You should see a symbol at the start of any line where this has happened.

Listing 1: `hello.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
    print('Hello, World')
```

[45] Most programmers between the ages of, say, 50 and 60 learned programming this way, hacking away at programs on the home microcomputers of the 1980s. ZX Spectrum all the way.

[46] If you are looking at this as a web page, you will see that a handy button will appear when you hover the mouse over the program code. This allows you to copy the code, *but* you will learn more if you type it yourself.

Listing 2: `hello_name.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
    name = input('Hello, what is your name? ')
    print(f'Hello, {name}. It is good to meet you.')
```

Listing 3: `hello_age.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
    name = input('Hello, what is your name? ')

    year_now = int(input('What year is it today? '))
    year_born = int(input('And what year were you born? '))

    age = year_now - year_born

    print(f'Hello, {name}. It is good to meet you. This year you will
↪be {age} years old.')
```

Important: Indentation is important in Python. This refers to the amount of space at the start of each line. In these simple programs, there is only a little indentation, but it must be correct. So gthe first two lines **must** start in the left-most column, and all the following lines **must** be indented. If you don't copy this, it will be a syntax error.

If the programs fail to run and produce some useful output, take a close look to see that you have them exactly right. Notice how your IDE will add colours to the code; as you get used to this it will help you spot mistakes as you type. Some IDEs will also highlight errors in the code as you type, or make possibly helpful suggestions.

Remember that a *syntax* error is where there is a problem with the way the program is expressed, meaning that it cannot run. A *semantic* error, on the other hand, is where the program runs, but does not do what you want. The first is usually easy to spot, the second is not.

Tip: You will see that some of the programs contain quotation marks, pairs of which have to match up. Some IDEs will insert a closing quote as soon as you enter an opening quote. This does save typing, but can be confusing at first. Of course, you can root around in the settings to turn the behaviour off.

Make sure you save the code somewhere, and that you know where this is! Ideally store your programs somewhere where you will always be able to find them.

These three programs are probably the first three that any new programmer creates. So what do these programs do?

hello.py
: … is the traditional first program that anyone writes. It just prints a cheery greeting on the screen.

hello_name.py
: … is the traditional second program. This time the program displays a prompt, the user enters their name, and a personalised cheery greeting is displayed. Behind the scenes, the user's name is encoded as binary, stored in the computer's memory, and then retrieved, but all this is invisible to the programmer.

hello_age.py
: … displays a cheery greeting that includes the user's age, which is calculated. Again, values are entered, and will be encoded as binary. Python always takes input as a string of letters, so there is a small fix to show that the values entered are going to be integers, and will be used for a calculation. Again, a lot happens behind the scenes even in this small programmer, but Python is a high-level language, and the programmer doesn't need to worry about all this.

Before leaving these programs, make copies, and change a few things. The worst that can happen is that you change something and the program stops working. Spend time getting used to your development tools; this will save a lot of pain later on.

4.2 Programming in a Good Place

Before carrying on to learn more, it really is worth taking some time to think about how you work best, and about which tools suit you best. You have started to work with an IDE. Maybe there were some things that were a bit annoying, or maybe you really don't like the colours it used for the programs. Or did you keep getting distracted? Or did your arms get tired with the typing?

There are two aspects to making sure you are programming in a good place. First, there are the software tools you use to create your programs, and second there is the question of the physical environment.

4.2.1 Tools of the Trade

You shouldn't be using tools just because you've been told to, or your friends do, or because you always have. You need to have the best tools for the job, and that's different for different people. If you have a choice of tools to use, get them all installed, and try each one. Maybe use the programs above. Enter and run those programs in different IDEs. See which IDE you prefer. There is no single choice that works for everyone.

A modern IDE is immensely powerful. And as a programmer you will spend most of your day looking at your IDE and working with it. So you should take time to learn how it works, and how to make it work for you. You probably only need a small fraction of what your IDE can do, but make sure you know what this small fraction is, and how to use it efficiently.

Modern IDEs are also extremely customisable. No-one uses an IDE with all its default settings. Take a look at the various colour schemes available, and pick one you like[47]. Make the font bigger if you need to. Hide menus and tools you are not going to need,

[47] As long as it's a dark colour scheme. No-one uses light schemes. Seriously.

make those you always need available all the time. You will probably find that the IDE will allow you to create an online account to store settings so that you can sync them between your different computers. Seriously, take time to use the IDE, make it yours. You are going to be using it **a lot** over the next several years!

..index::
single: Programming; craft

Programming is a *craft*. Craftspeople use tools, and they get the tools right.

4.2.2 The Physical Side

And look around.

Over time, a programmer will learn about what makes the best environment for them when they work. They will change things so that they are working in the best place. Good employers will recognise this, and should encourage it.

There are probably two aspects to this, but they are connected. First there is the hardware - the PC, laptop, screen, and so on - and second there is the overall environment - heat, light, desks, ambience. Hardware even includes the keyboard and mouse. Environment covers furniture, noise, and more.

There is again no "one size fits all" here, but most programmers probably prefer to work from a laptop, which they carry with them everywhere. (This is even more common in these days of hybrid working). The laptop can be used wherever it is needed "on the road", or can be used in the office at a conventional desk. Usually, on a desk it will be hooked up to a second display for comfort and productivity. Docking stations can add ports and other connectivity.

Having multiple displays is very important. If you have never tried this, you should! This small change may not immediately occur to people, but having two (or even three) screens available can seriously improve productivity, and is healthier too as it keeps your eyes busy. And why are monitors always landscape, with the longer edges top and bottom? Have you tried a portrait monitor? It's a game-changer, especially for reaing long programs or documents.

Hint: Getting the right setup is important. Think out of the box too. If you're going to work with long programs, how about having a second monitor in portrait orientation? That's another potential game-changer. Or if you're often on the road, how about a portable second monitor for the laptop? They can be no bigger than a tablet, and can usually be powered by the laptop.

Desktop PCs still have their place but, again, consider having more than one display. And that keyboard? Is it one you can happily type on all day? Are you good with where all the keys are? Is the mouse comfortable to use? Do you keep randomly hitting some key that has some annoying effect? If so, you can probably remap it.

And finally, think about the overall environment. A tidy desk is best. And a chair you can sit on for a while. Do you want somewhere to open books, or are you happy to prop up a Kindle, or read on one of your monitors? Natural light is ideal but, if not available,

there should be plenty of light sources. Think about whether you work best with music playing, or without? Above all, make sure there is a ready supply of coffee to hand[48]!

Important: Having said all that, it is also important to appreciate the need to take breaks away from a screen. This can sound a bit "Health and Safety Gone Mad", but there is a serious point to it.

4.3 Takeaways

After this chapter you should:

- Be able to enter programs, run them, and see the output.
- Have familiarised yourself with your programming tools, and at least started to customise them to suit your own preferences.
- Have thought about the best physical environment for your programming work, and ideally arranged this.

Now we can move on to see how a program processes and stores values. After all, that's what computers are good at, and also if you think about it, what every program does!

[48] Other beverage choices are possible, but most programmers go with coffee. And not decaff.

CHAPTER

FIVE

SOMEWHERE TO START

The time has come to create a program, and also to understand how it works. So in this chapter we will look at the three building blocks we need:

1. Data.
2. Input.
3. Output.

We have actually seen all these in the programs in the previous chapter. so shopefully you have some idea of what is to come.

Tip: Remember that a good way to learn to program is just to read programs, and understand them. This is easier than writing new programs, but an excellent place to start. It's the same with any language; people who learn a foreign language can usually read and understand it well before they are able to write an essay in it.

Right. At the most basic level every program does the same thing.

1. It takes some data.
2. It transforms that data in some way.
3. It does something with the new data.

There are obviously many variations here. The data could be read from a file, or from a keyboard. It could be processed in many different ways. It could be written back to the same place, a different place, or just displayed on a screen or printed out. But all programs do basically the same thing.

- A word processor reads a file containing a document, allows the user to edit it, and then saves the new version.
- A game reads a file containing the user's profile and a level description, allows the user to play the game, and then stores the new saved state.
- A utility to store WiFi settings on a smartphone reads the connection information as the user types, uses it to test and make a connection, and then saves it to a file so that the phone can connect automatically next time.

It really is that simple. If you edit a document using Word, you are just using a program to change the file that represents the document. If you play a race in Mario Kart you are really just using a program to change the file that represents your saved state in the game.

Any program needs to store data, so this seems a good place to start. We already know that data is stored in a binary format, and that it can be on disk or in memory. It's time to see how Python does this. Happily, as we will see, Python's high-level approach means that a lot of the details are hidden from us.

5.1 Creating Values

If a program is going to store data, a programmer needs to be able to refer to that data. Every data item processed in a program is going to need a unique name, and the computer will then keep track of what value is currently associated with what name. So our simple model for how a program works becomes:

1. Read the data, giving each item a unique name.
2. Process or transform the data.
3. Output the data, using the names to make sure that everything ends up in the right place.

So, every data item needs a unique name so that it can be referenced. In Python, the simplesy way new data item is created is simply by giving a value to a name. So, let's create a data item called `eggs` and give it the value 3.

Try It!

You can copy the commands in this section and run them at your own Python interpreter. On the web, hover the pointer over the code, and a little button will appear to allow you to copy the code.

Remember that `>>>` represents the prompt of the interpreter, and should not be copied. In fact, on the web you shouldn't be abe to!

```
>>> eggs = 3
```

This looks like a very simple line of code, which indeed it is. But, behind the scenes, several things have happened:

- Python is aware that there is a "thing" called `eggs`, and has added this to its list of known names (its "namespace").
- By looking at the value assigned to it, Python has determined that `eggs` is an integer.
- The value 3 has been stored at a particular location in memory, using the correct amount of memory for an integer.
- An entry has been made in a lookup table to allow Python to find the current value of `eggs`. The table stores the name, and the memory location where the value can be found.

To give things their proper names, we have now created a *variable* with the *identifier* `eggs` that has been *assigned* the value 3. From now on we'll try to use the proper terms.

So, what happens the next time we access the value stored in the variable with identifier `eggs`? Maybe we just want to `print` it:

```
>>> print(eggs)
```

Python checks the namespace. It sees something called `eggs` there, so goes to the lookup table to see where this is stored in memory. It then accesses the memory location to (in this case) print the value out.

So, every time a new value is created, an entry is made in the lookup table. And every time that value is needed, the process above is used to extract it from memory. Obviously all this happens "behind the scenes". In the olden days programmers had to do much of this work themselves, but high-level languages like Python remove all that pain. But it still pays to understand that this is happening.

Note: This description is another that is deliberately not precise, but will hopefully give you an idea of what is going on.

You need the ideas of creating a variable (value), and Python then maintaining a table of where each variable is currently held in memory.

5.2 Values and Types

The value stored in `eggs` above is an integer, a whole number. The type of a variable is important, as it determines what operations can be carried out on. We can use an integer in artihmetic, for example. It might not make sense to do arithmetic on other data types. Let's see.

Python determines the type of the variable by examining the value assigned to it. As well as the operations that are valid on it, the type of the value determines a number of things, not least how much memory is needed to store it. A floating-point value (a number where the decimal part is important) is created like so:

```
>>> swallow_speed = 12.5
```

This value requires more memory to store the decimal part, but the rest works the same as before. And all the details of how a floating-point number is stored inside the computer memory are, thankfully, hidden.

Boolean values work like this:

```
>>> brave = False
>>> run_away = True
```

Warning: Boolean values are basically integers in disguise. Be careful using them, and never ask a user to enter one.

Finally, strings are sequences of characters. Usually they have some meaning, like a name, but they could contain anything, including encrypted text:

```
>>> name = 'Sir Robin'
```

Hint: Strings have quotation marks around them. You can use single or double quotes, but it is best to stay consistent[49]. Boolean values do *not* have quotes.

Python is a *dynamically typed* language, which means that the types of variables are determined when they are first used, inferred from the value given. It also means that the type associated with a name can change, but this is usually very confusing, and is best avoided!

In summary, Python provides these four built in types[50]. They are called the *primitive* types.

int
: An integer, a whole number. Positive or negative with, effectively, no upper limit.

float
: A number with a decimal part.

str
: A string of characters. Effectively any length.

bool
: A True or False value.

A built-in command called `type` can be used to find out what type is currently stored in a value. It will get some use in the next sections as we work at the interpreter, but it is rarely used in programs. As we will see in the next chapter, Python's philosophy is to run a program, and deal with any errors caused by types as they arise. So very rarely does a program need to check a variable's type.

5.2.1 Investigating Integers

Integers are probably the most common data type, so we'll start with those. Some programming languages offer many different types for whole number values, the choice depending on the range needed, whether they can be negative, and the like. Python keeps things simple and offers just the one[51]:

```
>>> eggs = 3
>>> type(eggs)
<class 'int'>
```

Let's see what we can do with some `int` s.

[49] The style in this book is to use single quotes unless double-quotes are essential, for example if the string itself contains single quotes. You could copy that, or go your own way. Just be consistent.

[50] This is not true. Sorry. There is another type, `NoneType`. This means that a variable exists, but has no value, and therefore no type. We'll need it later, so just hold the thought for now.

[51] As we know, Python is intended to have one, and just one, way to do anything. So why have a whole bunch of different types for whole numbers, when one will do? Looking at you, Java.

Doing the Maths

Integers are numbers, and the most common uses fo them obviously involve all the things we do with numbers. All the usual arithmetic operators are available. It is quite possible to use the Python interpreter as a handy calculator, which will also show the four mathematical operations. Here we have addition, subtraction, multiplication (the symbol is `*`) and divison (`/`).

```
>>> 2 + 2
4
>>> 8 - 6
2
>>> 3 * 4
12
>>> 8 / 2
4.0
```

Important: Take a close look at the last operation above. `4.0` is a floating-point value. So if we divide an integer by another integer, the result is a floating-point number, even if the decimal part is zero. Why so? Because *in general* the result of dividing two integers will have a floating-point part, so it makes sense to always return a `float` as the result.

This all looks straightforwarde, but there is one detail to cover. There are three different kinds of division. Above is what we might call "normal" division, where the result is a floating-point number. This is usually what is needed, so it is what happens by default. However, sometimes *integer division* is needed. So, it is possible to require that the result is an integer, effectively ignoring any decimal part:

```
>>> 8 // 2
4
>>> 7 // 2
3
```

Obviously this sometimes "loses" something, but this is sometimes the result wanted. As something is lost, it is also possible to find the number that are "left over" after a division (called the "modulus"):

```
>>> 8 // 2
0
>>> 7 // 2
1
```

Hint: A very common use case for the modulus operator (`%`) is to deermine whether an integer value is odd or even. An even value "modulus 2" is 0, an odd value is 1.

Use Case

Suppose we were dividing eggs into boxes of six. We need to divide the total number of eggs we have by six, but a floating-pont answer would not be useful. We can't put 6.33 eggs in a box! So here we would require integer division to tell us how many boxes will be full, and we could use the modulus operator to find out how many eggs would be left over.

Finally, there is also an operator to raise a number to a power.

```
>>> 2 ** 4
16
```

There are many other mathematical operators, useful in scientific applications. But these are not included in strandard Python. It is easy to make them available, though, as we will see later.

Precedence

These operators can be chained together to make more complex *expressions*. For example:

```
>>> 2 + 2 - 3
1
```

In expressions like this the question arises of what order the operators are applied. In the above example it makes no difference to the result, but in an expression like:

```
>>> 2 + 2 * 3
8
```

the order matters. You can probbly work out that in this expression the multiplication has been applied before the addition. How so?

The rule is that if there is more than one operator in an expression the usual mathematical rules of precedence reply. You might remember them from maths courses, where they are usually remembered as as *BEDMAS* or *BODMAS*.

B rackets.
E xponents (powers).
D ivision.
M ultiplication.
A ddition.
S ubtraction.

This explains why the multiplicattion happened first above; it has a higher precedence. This can sometimes give unexpected results to the unwary when the operators in an expression are not applied left-to-right, as in:

```
>>> 2 + 8 / 2
6.0
```

Here division happens first, and this also means that the result is a `float`. The trick is to use brackets to change the order. So if left-to-right was needed hwre we could write:

```
>>> (2 + 8) / 2
5.0
```

In general, even when the *BEDMAS* order gives the result required it is a good idea to add brackets to clearly show the intended order. So our first example here is best written like this, even though the brackets actually have no effect.

```
>>> 2 + (8 / 2)
6.0
```

This is a small detail, and Python probably does what you would expect, but it's always worth checking if the result of an expression isn't quite what you expect.

More Operators

As well as the arithmetic operators, there are a few that get commonly used. They are shorthands for common needs. For example, suppose we want to add one to the value of variable. We could program like so:

```
>>> eggs = 3
>>> print(eggs)
3
>>> eggs = eggs + 1
>>> print(eggs)
4
```

This might seem odd at first, but the thing to remember is that the right-hand side is evaluated first, and the result is assigned to the variable on the left.

Tip: A common source of confusion is the use of = in expressions like this, which is different to the way that it is usually used in maths. This operation is called *assignment*, and some languages use a different symbol for it. But Python sticks with one = for assignment.

Assignment is different to *equality*. See later for that.

This operation (called *incrementing*) is so common that there is a shorthand:

```
>>> eggs = 3
>>> print(eggs)
3
>>> eggs += 1
>>> print(eggs)
4
```

This increments the valure by 1, but any value can go there. So we could *decrement* a value by, say, 2:

```
>>> eggs = 3
>>> print(eggs)
```

(continues on next page)

(continued from previous page)

```
3
>>> eggs -= 2
>>> print(eggs)
1
```

Multiplying and dividing also work like this, but are probably less common. Here are two examples. See again that the division produces a `float`.

```
>>> eggs = 3
>>> eggs *= 2
>>> print(eggs)
6
>>> eggs /= 3
>>> print(eggs)
2.0
```

5.2.2 Focus on Floats

In some applications, integers are sufficient for numeric data, but in general we are interested in numbers that have a floating-point (decimal) part. This is especially true in scientific applications, but also true in simpler problems that involve working out, for example, averages.

Floating-point decimal numbers are tricky to represent accurately in binary in much the same way as some fractions (like one third) are impossible to represent as decimal numbers. As before, some programming languages offer many different data types for floating-point numbers, depending on the accuracy needed, but Python offers just the one:

```
>>> speed = 3.0
>>> type(speed)
<class 'float'>
```

Important: See here that `3.0` is a floating-point value, even though the number after the decimal point is zero. `3` is an integer value representing the same amount, but they are different data types.

```
>>> european_speed = 3.0
>>> type(speed)
<class 'float'>
>>> african_speed = 3
>>> type(speed)
<class 'int'>
```

A slightly interesting question is whether these two values are equal. What do you think? We'll check later.

Since they are also numeric values, floating-point numbers behave in a very similar way to integers. Behind the scenes things are more complicated, as floating-point val-

ues are more complex to store accurately in binary, but happily that is mostly hidden. So all the usual mathematical operators work as before:

```
>>> 2.5 + 3.2
5.7
>>> 2.5 - 1.45
1.05
>>> 2.5 * 3.5
8.75
>>> 3.5 ** 2
12.25
>>> 5.6 / 3.2
1.7499999999999998
```

Tip: Check that last result above. This is what you see when the result of an expression can't be represented exactly. The answer is, obviously, 1.75, but that value can't be represented precisely in binary. (If you try the same expression on your calculator, you will probably get 1.75 because the calculator will do some rounding). This can make working with floating-point values tricky!

Floating-point values can also be combined with integers, where this makes sense to do so. They are both numbers, after all. Arithmetic operations work as you'd expect. The type of the result is determined by the types of the values. So an integer added to an integer is another integer, while an integer added to a float is a float:

```
>>> 3 + 3
6
>>> 3 + 3.5
6.5
```

This is what you'd expect as any other result would lose the decimal part.

The usual rules of the order of operators also apply here.

Conversions

Since `int` and `float` values are both numeric it is useful to be able to convert between them. Converting a floating-point value to an integer will lose something, of course, but sometimes that doesn't matter. An integer is easily converted to a floating-point, with all that really changes being the internal representation.

Conversions can be done just using the name of the required types. Like this:

```
>>> speed = 3
>>> speed_float = float(speed)
>>> print(speed_float)
3.0
>>> type(speed_float)
<class 'float'>
>>> speed_float = 3.5
```

(continues on next page)

(continued from previous page)

```
>>> speed = int(speed_float)
>>> print(speed)
3
```

This code creates and integer, and then converts it to a `float` (see the `.0`). This floating-point value is than changed, and the value is then converted back to an `int`. This loses the decimal part, effectively rounding down.

Tip: A quick hack to convert an integer to a float goes like this. You might see it in some example code and wonder what's going on.

```
>>> speed = 3
>>> new_speed = speed * 1.0
type(new_speed)
<class 'float'>
```

Multiplying by `1.0` (a `float`) gives a `float` as the result.

5.2.3 String Theory

A string is a sequence of characters. Usually it represents something interesting like a name or an identity number, or some other data that has been input by the user or read from a file. Python has many useful features that allow strings to be manipulated, and this is often quoted as a strength of the language. Python is very well suited for any application that involves much processing of strings.

Languages

There are many programming languages, and many programmers would say they have a favourite. The thing is that languages have strengths and weaknesses, and some are more suited to different tasks than others. The trick is often to pick the most suitable language for a given task.

Strings are denoted by quotation marks. Single `'` or double quotes `"` are fine, and are equivalent (but pairs must match). The only time the choice becomes important is if the string itself includes a quotation mark. So these are all fine:

```
>>> 'Sir Robin'
'Sir Robin'
>>> "King Arthur"
'King Arthur'
>>> "Galahad's Sword"
"Galahad's Sword"
```

Tip: If you type a value at the Python interpreter it will just print (echo) the value back, like this.

The simplest way (and probably most common) way to process a string is to extract certain characters. Characters in the string are given index numbers, from left to right. So the first character is at index `0`, the second at index `1`, and the last has an index of the length of the string less one[52].

This last is a bit complicated, and it is surprisingly common to want to find the final character of a string, so the last character also has index `-1`. Indexes work from either end of the string, like so:

```
>>> 'Sir Robin'[0]
'S'
>>> 'Sir Robin'[2]
'r'
>>> 'Sir Robin'[-1]
'n'
>>> 'Sir Robin'[-3]
'b'
```

It is also possible to extract ranges of characters from a string, by providing two indexes, a start and an end. If one is missed off, it defaults to the end of the string.

```
>>> 'Sir Robin'[0:3]
'Sir'
>>> 'Sir Robin'[4:]
'Robin'
>>> 'Sir Robin'[:-1]
'Sir Robi'
```

This "slicing" seems a simple idea, but is incredibly powerful and useful in many applications. There are many, many, more built-in operations for string wrangling, which we will meet later on.

Arithmetic can also work for string, where it makes sense. It makes sense to add two strings:

```
>>> 'Eggs ' + 'Spam'
'Eggs Spam'
```

but it makes no sense to subtract one string from another. Similarly, it makes no sense to multiply or divide strings, but it *does* make sense to multiply a string by an integer. See what it does:

```
>>> 'Spam! ' * 4
'Spam! Spam! Spam! Spam! '
```

We will return to strings later. But, just one last time, handling strings like this is one of the main strengths of Python. So it will ge tthe attention it deserves later on.

[52] Computer Scientists start counting at zero.

5.2.4 Boolean News

A Boolean value is one that is either `True` or `False`. These values have been *discussed before* because they are so fundamental. A value of this type is created in the usual way:

```
>>> brave_sir_robin = False
>>> type(brave_sir_robin)
<class 'bool'>
```

Arithmetic operations make no sense with Boolean values but logic operations clearly do. Check *back here* for a reminder of these.

First `not` takes one Boolean and "flips" it, so True becomes `False`, and `False` becomes `True`.

```
>>> brave_sir_robin = False
>>> run_away = not brave_sir_robin
>>> run_away
True
```

This will seem rather abstract at the moment, but we will use this a lot later on! The other two operators, and and `or` combine two values as expected.

```
>>> eggs = True
>>> spam = False
>>> eggs and spam
False
>>> eggs or spam
True
>>> spam = True
>>> eggs and spam
True
```

The use of Booleans is maybe not obvious at the moment, but they will be crucial later when we need to control the order in which statements are executed. Before that, let's see how values can be combined and compared into *Boolean Expressions*.

Boolean Expressions

A Boolean variable holds either the value `True` or `False`. Similarly, a *Boolean Expression* is an expression that is either `True` or `False`. As with the truth of some statements we met before, some expressions are self-evidently `True`:

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

Important: Heads up! We have used the single equals sign before, for value *assignment*. Two equals signs, as above, is used for *comparison*. So this expression is testing, ah, whether 1 equals 1, which obviously it does.

More usefully, we can test whether a variable holds a certain value:

```
>>> eggs = 6
>>> eggs == 6
True
```

Boolean operators allow for comparing values like this. Look again at the overloading of what an equals sign does! Here are the more common ones:

Operator	Meaning	True Example	False Example
==	Is equal to	1 == 1	3 == 2
!=	Not equal to	1 != 0	1 != 1
>	Greater than	3 > 1	1 > 0
<	Less than	1 < 3	0 < 2
>=	Greater or equal to	2 >= 1	2 >= 3
<=	Less than or equal to	1 <= 1	2 <- 0

These examples all use integers. The same operators will work with floating-point numbers, in the obvious way. Integers and floating-point numbers can be compared, but care is needed because of the difficulty of storing floating-point values exactly.

Tip: In practice, avoid using == with floating-point values. The results are not always what you would expect because sometimes storing the value loses some precision.

We did wonder whether `1` (the integer) was equal to `1.0` (the floating-point). Turns out they are:

```
>>> 1 == 1.0
True
```

A similar experiment will also reveal that, just maybe, Booleans are integers in disguise![54]

```
>>> 1 == True
True
>>> 0 == False
True
```

And finally, comparison operators also work with strings. The meaning of "less than" and friends is based on the internal (numeric) way strings are stored, but is effectively alphabetical.

These operators can be combined with the Boolean operators to give complex expressions. Suppose we were interested in checking that a number was between 0 and 100 inclusive. That involves three operators:

```
>>> mark = 65
>>> mark >= 0 and mark <= 100
True
```

(continues on next page)

[54] Which is fine and, if you think about it, the obvious way of doing it!

(continued from previous page)

```
>>> mark = 150
>>> mark >= 0 and mark <= 100
False
```

Building up expressions like this will become very important later on.

A final operator is worth a mention here. The `in` operator gives a Boolean value depending on whether or not one value is contain inside another. This is often used with strings:

```
>>> 's' in 'spam'
True
>>> 's' in 'eggs'
True
>>> 'spam' in 'fritters'
False
```

This is really just a shorthand for a whole bunch of `and` tests together, but it can be useful. It also has the happy side-effect of making code more readable.

Now, let's try and formalise what this chapter has discussed.

5.3 Values and Variables

To call it what it should be called, a value in a program is a *variable*. A variable has a type, and a value. Usually the value changes as the program runs. In Python a variable is created just by giving it a value:

```
>>> foo = 3
```

This creates a variable named `foo` with the value `3`. As we have seen before `foo` is given a type of `int`, which is inferred from the initial value.

Tip: Remember that `3` is an integer but, `3.0` is a floating-point. And if it comes to that `'3'` is the string that represents the digit three.

It is important to pick a name for the variable (correctly, this is called its *identifier*) that describes its purpose. The example above is meaningless, so it is much better to pick an identifier that explains what the value is:

```
>>> number_of_knights = 3
```

So now we can see that this value is presumably going to be used to store the number of knights that have done something, or otherwise become interesting in some way.

Meaningful identifiers are good. But there is a balance to hit between names that are too long and names that are too short and cryptic. These are both bad choices, for reasons that should be obvious:

```
>>> nok = 3
>>> the_number_of_knights_seeking_the_holy_grail = 3
```

A further downside to long identifiers is that errors in spelling them can lead to errors in programs that are very, very hard to find[53].

By convention, variable identifiers in Python are written in `lower_snake_case`. With words separated by underscores, and everything in lower case. Programs in this book will stick with this convention, as should you.

Important: Conventions like this are important. They may seem pointless now, but if your programs don't follow them you will confuse experienced programmers if you ask for help. In a work setting, if you didn't follow them you would just be told to go away and rewrite the code "properly"!

The thing to remember is that most programs are developed and maintained by teams. It makes a lot of sense if all the members of the team use conventions like this to ensure that their code is consistent and understandable.

Another example of a convention is when a program needs to handle a *constant* value. This is a variable that will be used in the program, but the value will always be the same. A variable that will not vary, if you like. By convention, the identifiers of these values are witten in `UPPER_SNAKE_CASE`. This is irrelevant to Python, but very useful to someone reading a program. So when a programmer sees:

```
>>> KNIGHTS_IN_HORSE = 4
```

It is clear that this is a value that is used in the program, but which will never change.

Important: This might seem a bit odd, but the idea is to define the constant value in one place, and then potentially use it in many places. If it needs to be changed, it changes in just the definition, so is just changed the once.

Using constants like this also improves the readability of programs. It is oftem said that programs are read much more often than they are written!

5.4 Input and Output

Armed with variables, there are two more things needed before we can write a useful program. First, we need to be able to take some values as *input*, and then we need to *output* the results. The simplest cases here are to take input that the user types on the keyboard, and to display the results on the screen. More realistic programs read or write files, or use graphical interfaces, but the simple "screen and keyboard" approach will do for now.

For the moment we will also assume that the user behaves as expected, and enters values that make sense in the current program. Obviously in real life users do not behave that way, but assuming they do will simplify the problem for now!

[53] Although obviously your IDE should quickly flag up such spelling issues.

5.4.1 Getting Input

To get some input from a user we need to display a helpful prompt, and then wait while they type. Usually, their input will be ended when they hit "Enter" or "Return". Once we have the input we need to store it away in a suitable variable. There is a lot going on here, but Python provides a single command to do the job.

The `input` command displays a prompt, and waits for the user to type. Once the user hits Enter, the value entered is *returned* and can be stored in a variable. The value is always returned as a string, so sometimes there is a need to convert it to a required type. It is very unlikely that the user would be asked to enter a Boolean value, so the conversion is almost always to an integer or floating-point value.

We briefly met the way to convert values between types earlier in this chapter. In these examples, remember that entering the identifier of a variable at the Python prompt just displays the current value of that variable:

```
>>> name = input('What is your name? ')
What is your name? Sir Robin
>>> name
'Sir Robin'

>>> number_of_knights = int(input('How many knights follow the quest?
↪'))
How many knights follow the quest? 5
>>> number_of_knights
5

>>> speed = float(input('Enter the average speed of an African
↪Swallow: '))
Enter the average speed of an African Swallow: 37.5
>>> speed
37.5
```

Take a close look at the brackets in the second and third examples. There are two *functions* used - `int` and `float` - to convert the values. The brackets need to match up, as shown. Your IDE should show an error if the brackets match incorrectly.

This will be enough, for now, to allow us to write programs that take input. In later episodes we will need to validate the input for its type, and possibly its value, but this will do for now. Specifically, in the next chapter we will see how to cope if the user enters values of the wrong type, or at least data that cannot be converted to the correct type.

5.4.2 Displaying Results

The command to display a value on the screen is `print`. We saw it earlier, but passed over it. It takes either a literal value, like this:

```
>>> print('Hello, World')
Hello, World
```

Or it can take a variable identifier (notice there are no quote marks in the `print` here):

```
>>> message = 'Spam and Eggs'
>>> print(message)
Spam and Eggs
```

The `print` command can also print a collection of values. The easiest way to do this is to make sure they are provided separated by commas, and by default they are printed with spaces between.

```
>>> swallow_count = 3
>>> print('There are', swallow_count, 'swallows.')
There are 3 swallows.
```

There are other options, but as with `input`, this will do for now. Keep it simple!

Tip: Those spaces can be annoying, and are not always wanted. The quick fix at this point is to add an optional "separator" to the `print` command, like this:

```
>>> print('Spam', 'Eggs', 'Spam')
Spam Eggs Spam
>>> print('Spam', 'Eggs', 'Spam', sep='')
SpamEggsSpam
```

5.5 Takeaways

There is a lot in this (rather long) chapter. But programming really is all about reading some values, storing them, processing them, and then making the results available. By now you should:

1. Understand that values have different *types*. And know which basic (*primitive*) types Python provides.
2. Be able to create values at the Python Interpreter, and carry out operation on them.
3. Know how to prompt a user to enter some values, and how to convert that to a required type.
4. Know how to display results on the screen.

We can actually write some reasonably useful programs now. The main gap is how to do different things depending on what the user enters. We'll look at that in a while, but first we'll think about what to do if the user doesn't behave as expected. Users are like that ...

CHAPTER

SIX

WHEN THINGS GO WRONG

Many programming books carry on at this point with writing some programs. They tell the new programmer not to worry about what happens if the user does something unexpected, or if something else goes wrong. That is all a bit artificial, because users do often do awkward things. So here we will extend our programming by looking at some basic error situations.

6.1 A Simple Error

Here's a problem:

Buses for Students

A school is running a trip to a local theme park. Buses have been hired. Each bus can seat 46 students. A program is required that reads the number of students who have applied to join the trip, and will determine how many buses are needed, and how many students will be left behind.

We can sketch out a solution along these lines:

1. Set the number of students on each bus as a constant, so we can easily see what happens if we get bigger or smaller buses.
2. Display a prompt.
3. Read the number of students, as an integer.
4. Calculate the number of buses needed using integer division (we cannot have half a bus).
5. Use the modulus (remainder) operator to find how many students will be left.
6. Print the results.

And this matches neatly to a program, which might look something like this:

Listing 1: `school_bus.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    STUDENTS_PER_BUS = 46

    number_of_students = int(input('How many students are there? '))

    buses_needed = number_of_students // STUDENTS_PER_BUS
    students_left = number_of_students % STUDENTS_PER_BUS

    print('Buses Needed: ', buses_needed)
    print('Students Left:', students_left)
```

Running the program, the results would look promising:

```
How many students are there? 62
Buses Needed:  1
Students Left: 16
```

The first line is where the user has entered a value. The results look right.

Now, look closely at this line of the program:

Listing 2: `school_bus.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    STUDENTS_PER_BUS = 46

    number_of_students = int(input('How many students are there? '))

    buses_needed = number_of_students // STUDENTS_PER_BUS
    students_left = number_of_students % STUDENTS_PER_BUS

    print('Buses Needed: ', buses_needed)
    print('Students Left:', students_left)
```

It does quite a lot. It displays a prompt, reads the user's input, converts the string entered to an integer (`input` always reads a string), and assigns the result to a variable. All good, but what would happen if the user entered a value that could not be converted into an integer? The quickest thing to do it to try it and see by entering `lots` instead of a number:

```
How many students are there? lots
Traceback (most recent call last):
  File "school_bus.py", line 7, in <module>
    number_of_students = int(input('How many students are there? '))
ValueError: invalid literal for int() with base 10: 'lots'
```

Eek! This message tells us that the program has failed. It tells us which line this was (line 7), and displays the offending line. There is also a clue to the error ("invalid literal"), and

a name for it (`ValueError`). Obviously we would rather the program ended rather more elegantly, and ideally gave the user more of a clue as to what went wrong. Lets's see how to do that.

6.2 Handling an Exception

This error is correctly called an *Exception*, because it represents an exception to what should have happened. Python is unable to continue with the program but, before all is lost, it is giving us two things:

1. An exception type, that indicates in general terms what has happened. Here it is `ValueError`, so we know it is a problem with a value that Python has tried to process.
2. A message that contains a hint of what Python believes has gone wrong. Here, the literal value entered is invalid.

There are to approaches to modifying the program so that it can handle this error:

1. We could write code that would examine the string entered, determine whether it will convert to an integer, and carry on if it looks fine, printing an error otherwise.
2. We could convert the string entered, whatever it is, and deal with any error that might happen.

The first of these approaches is called **Look Before You Leap** (LBYL) and is a common approach in many older languages. It can lead to complex programs, where all the error-checking can make it difficult to see what is *supposed* to happen.

The second is EAFP, or **Easier to Ask Forgiveness than Permission**. This is a more modern approach, and is common in most newer languages. It tends to keep code that deals with errors all in one place, leaving what should happen alone.

Important: Python prefers EAFP. A lot.

We aim to be *Pythonic* here, so EAFP is what we will use.

Provided we know what exception might happen, it is easy to amend the code. In this example we know that we are concerned about the possibility of a `ValueError`. So all we need to do is tell Python what to do if such an error happens. It looks like this (changes are highlighted):

Listing 3: `school_bus.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    STUDENTS_PER_BUS = 46

    try:
        number_of_students = int(input('How many students are there?
```

(continues on next page)

(continued from previous page)

```
↪'))

        buses_needed = number_of_students // STUDENTS_PER_BUS
        students_left = number_of_students % STUDENTS_PER_BUS

        print('Buses Needed: ', buses_needed)
        print('Students Left:', students_left)

    except ValueError:
        print('Please enter an integer.')
```

So we `try` to do what is expected (see that this is now indented so it is "inside" the `try`). If there is a `ValueError` the program jumps straight down to the matching `except` and does what it says there. So now:

```
How many students are there? lots
Please enter an integer.
```

And after any changes it is always important to check that the program still works as before:

```
How many students are there? 58
Buses Needed:  1
Students Left: 12
```

Looks good! This approach is much easier than trying to examine a string to see if it could represent an integer, and has the extra benefit of leaving the original program logic untouched.

Let's extend this to make the program a little more useful. Maybe we have the option for different sizes of bus.

6.3 Another Exception

To modify the program to deal with different bus sizes, we are going to need to ask the user to enter the number of students who will fit on one bus. So the bus capacity is no longer a constant, but a variable. So the first attempt is to change the constant `STUDENTS_PER_BUS` to a variable that is entered (and to remember to change its identifier to lower-case as it is no longer constant).

Hint: Renaming a variable or constant is common. Don't be tempted to use some sort of search-and-replace to do this, as al sorts of unexpected things could happen if the name exists elsewhere. You will find that your IDE has an intelligent way to do this (look for options called *Refactor*).

So now we have:

Listing 4: school_bus.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    try:
        students_per_bus = int(input('What is the bus size? '))
        number_of_students = int(input('How many students are there?
↪'))

        buses_needed = number_of_students // students_per_bus
        students_left = number_of_students % students_per_bus

        print('Buses Needed: ', buses_needed)
        print('Students Left:', students_left)

    except ValueError:
        print('Please enter an integer.')
```

This is fine, and if we tried it we would find that all still worked. But could anything else now go wrong? If we thought about it we might realise that the user could enter zero for the bus capacity. This is nonsense, but is an integer, so the program would carry on to try to divide the number of students by zero. Dividing by zero is an error, so what would happen? We expect an *exception*, but what would it be called?

The quickest way to find out is to fire up the Python Interpreter and find out.

Hint: You can access the interpreter from inside your IDE. No need to change windows!

```
>>> 2 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The information we want is in the last line. The exception is a `ZeroDivisionError`. Good name for it. To catch this error, all we need to do is add it to the program. The changes are isolated at the bottom, with the other handling of the exceptions - this is why EAFP usually gives neater programs.

Listing 5: school_bus.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    try:
        students_per_bus = int(input('What is the bus size? '))
        number_of_students = int(input('How many students are there?
```

(continues on next page)

(continued from previous page)

```
↪'))

        buses_needed = number_of_students // students_per_bus
        students_left = number_of_students % students_per_bus

        print('Buses Needed: ', buses_needed)
        print('Students Left:', students_left)

    except ValueError:
        print('Please enter an integer.')
    except ZeroDivisionError:
        print('Bus size cannot be zero.')
```

6.4 Exceptions are Good

Suppose we look again for more errors. We might wonder what would happen if the user entered nothing at all. That is obviously an error, but what happens when Python attempts to convert nothing into an integer?

A quick experiment would show that what Python does in this case is generate a `ValueError`! So as it turns out we have already caught that possible error. The worst case is that we will have to rewrite the error message.

EAFP is based around the idea that most of the time the user will do what they are expected to do. So if they do something else that is an *exception* to the normal, and should be handled as such. It is much easier to write programs that assume all will be well, but to include code to show that something has gone wrong. LBYL in contrast often involves a lot of code to handle situations that will very rarely occur.

Note: Sometimes an error situation is considered so unlikely that there is no point in adding code to handle it. The most common example of this is the *Blue Screen of Death* <https://en.wikipedia.org/wiki/Blue_screen_of_death>`_ in Windows.

It's worth noting that many new programmers instinctively go for LBYL approaches, and start writing insanely complex code when a simple bit of LBYL would solve the problem. That is why we have discussed EAFP and exceptions here *before* getting to the ideas needed to write LBYL! But the rule remains, to **always prefer EAFP** if at all possible.

6.5 More Errors

So what errors remain here> There are still things that could go wrong with this program. The most obvious is that the user could enter a negative integer for either of the required values. This is obviously nonsense. The program would display results because it has no reason not to. But there should be some way to stop that - this is coming up next!

And arguably there could be some *sanity checks* on the input. Is a bus with a capacity of 3000 likely? In practice there would only be a small number of capacities available, and we might pick from a list. And a really good program would take the number of students, and work out the best collection of bus sizes to arrange.

Next section we'll look at ways to approach these kinds of problem.

6.6 Takeaways

Exceptions are important. There are a basic part of Python, and handling them is fundamental.

1. You should now understand what an Exception is, and how to spot one.
2. You should be able to trap an Exception in a program, and display an error message.
3. You should be able to use the Python Interpreter to investigate the types of Exception that are generated in certain cases.
4. You should understand LBYL and EAFP approaches, and particularly why EAFP is preferred in Python!

There is surprisingly little left to cover before we can say we've done the basics of programming. Let's get on with it!

CHAPTER

SEVEN

STAYING IN CONTROL

Previously we have looked at *exceptions* and how they can be used to catch some error situations.

These have been errors that mean that Python cannot carry on because it makes no sense to process the values it is given. But what about the case where the values do make sense (in that they are the correct type), but are invalid for some other reason.

Important: Remember that the Python interpreter has no concept of what the data it is processing represents. It just sees a bunch of numbers, strings, and so on. It is the programmer that understands what these mean, and knows how to determine if they are invalid. A programmer might call a variable age, but the interpreter just sees a number. It has no idea that a negative value (or a value over 130 for that matter) is invalid.

Let's see how to deal with these errors. We might say that we are dealing with values that are *legal*, but invalid. We'll start with a simple example, and then look at some more complex ones.

7.1 Values in Range

Remember the example of allocating students to buses. There was an error when the capacity of a bus was entered as `0` (it provoked a `ZeroDivisionError`), but what about if `-10` had been entered? This is also clearly nonsense but the Python interpreter (which has no knowledge of buses) would happily carry on, producing potentially confusing results.

```
>>> students = 100
>>> bus_capacity = -10
>>> buses_needed = students // bus_capacity
>>> buses_needed
-10
```

To stop this we need to apply a small amount of logic to stop the calculation if the values entered are outside the known possible range. Usually, but not always, there will be an upper and a lower range. Let's jump in and fix the program so that the number of students has to be greater then 1. For clarity, we'll leave off the code that handled the exceptions for the moment, and go back to the bus size being set as a constant.

Listing 1: school_bus.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    STUDENTS_PER_BUS = 46

    number_of_students = int(input('How many students are there? '))

    if number_of_students > 1:
        buses_needed = number_of_students // STUDENTS_PER_BUS
        students_left = number_of_students % STUDENTS_PER_BUS

        print('Buses Needed: ', buses_needed)
        print('Students Left:', students_left)
```

The program is now providing a *condition*, and showing that some statements should be executed only if the condition turns out to be `True`. The statements affected by this are *indented across*. So now if an incorrect value was entered the program would produce no output at all.

This is fine, but as it stands the user, looking at a program generating no output at all, might be a little baffled about what's going on. It would be better to provide them with some sort of useful error message. Or we could point them in the direction of how to use the program properly. To do this, we need to say what should happen if the condition is `False`. It looks like this:

Listing 2: school_bus.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    STUDENTS_PER_BUS = 46

    number_of_students = int(input('How many students are there? '))

    if number_of_students > 1:
        buses_needed = number_of_students // STUDENTS_PER_BUS
        students_left = number_of_students % STUDENTS_PER_BUS

        print('Buses Needed: ', buses_needed)
        print('Students Left:', students_left)
    else:
        print('Error. Number of students cannot be less than 1.')
```

The `else` states what happens when the condition at the `if` is `False`. It is indented so that it is directly below its matching `if`, so there is no need for the condition to be repeated.

Important: This works because a Boolean condition is always one of `True` or `False`. The condition in a statement like this can be anything that evaluates to a Boolean value.

The Boolean condition, being `True` or `False` (and having no other possible values) means that one of the `if` or `else` will always be executed, but never, of course, both.

That's enough for simple cases. Before looking at more elaborate ones, let's consider what's going on here.

7.2 Flow of Control

The example above introduced a *conditional statement*. This is a statement that allows us to control the *flow of execution* in a program. Unless we specify otherwise, a program will flow from the top to the bottom, with the interpreter executing each statement in order. This is rarely what is needed, because we want to handle errors, deal with different user input, and so on.

Tip: A *Conditional Statement* in a programming language is often referred to as the **If Statement**, just because of the word that introduces it in most languages. Either term is fine.

A conditional statement allows a programmer to mark that some statements should be executed only if some condition is true. Optionally, there may also be statements that are an alternative, to be executed only if the same condition is false. Since any Boolean statement is either true or false, one or the other sets of statements will always be executed.

Important: There does not need to be an alternative. So, in Python, the `else` part is optional.

Tip: If you think about it, it is always possible to write a conditional statement in two ways, with true and false either way round. It rarely matters, but the best plan is always to pick the one that gives the most natural-looking code.

Python indicates which statements are affected by a control statement using indentation. The statements affected are all indented by a certain number of spaces, directly under the control ststement. When statements are "un-indented" they are no longer affected. Indentation also shows which `else` matches which `if` - the `else` is indented so that it is directly below the corresponding `if`.

So here the first `print` is affected by (or "inside") the `if`, but the second isn't:

```
if number == 1:
    print('Inside the if!')
print('Outside the if!')
```

And here, the `else` is aligned with its `if`:

```
if number == 1:
    print('Inside the if!')
else:
    print('Inside the else!')
```

This alignment is important, because there can be a conditional statement inside another, liks so:

```
if number == 1:
    print('Inside the if!')

    if another_number == 2:
        print('Inside both the ifs!')
else:
    print('Belongs to the first if!')
```

Spaces are usually used for indentation. TAB characters can be used[55], but most IDEs will silently turn them into spaces. Any number of spaces can be used, but the PEP-8 standard calls for 4, and this is what most IDEs will do.

Important: The indentation is not just for show. It is part of the Python language, and is used to show which statements are affected by which control statements. If you get the indentation wrong, the program will not work.

7.2.1 More Choices

The examples so far have allowed for two possibilities, based around a single condition. Sometimes this is all that is needed, like when a user enters a value that needs to be tested for a range, but in others it is not enough. Maybe we need to test a value, and carry out different actions depending on several possibilities. Here's an example:

Exam Grades

A student takes an exam, and gets a mark on a scale of 0 to 100. The pass mark is 40. Any mark of 70 or over is recorded as a *Distinction*. A program needs to read the exam mark and print the corresponding result - *Pass*, *Fail*, *Distinction*.

So here we need to test whether the grade and test it twice. This could be done several ways, but the neatest way is probably to check for a Distinction, and then for a Pass, like this:

[55] But their use is controversial, and best avoided unless you want to end up in arguments. It's like the dark/light theme in the IDE thing.

Listing 3: `exam_result.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    mark = int(input("Enter the student's mark: "))

    if mark >= 70:
        print('Distinction!')
    elif mark >= 40:
        print('Pass!')
    else:
        print('Fail!')
```

Three things to notice about the code here:

- `elif` allows for other choices to be tested. The complete statement will execute the first condition that turns out to be true.
- `else` serves as a "catch all" if none of the tested conditions are true.
- The final condition can be a catch-all because if the mark is not a Distinction or a Pass it *must* be a Fail, so there is no need for an explicit check.

And also:

- The string used for the input prompt includes a single quote (an apostrophe), so the string itself is in double quotes.
- There are be any number of `elif` lines, but the order matters in that the code under the first `True` is executed (all the others are passed over).
- There does not have to be a catch-all `else`[56].

7.2.2 Nesting

As it stands, the exam result program is fine, but if we look back at the earlier programs in this chapter we might reflect that there is no attempt to check that the mark entered is valid! We should surely check that it is between 0 and 100, and provide an error if not. There are two ways to do this. One would be to add the checks to the existing tests, but that might obscure what as going on. And, anyway, we have most of what we need from the earlier program[57] so we might as well reuse it.

The trick will be to put one conditional inside another. This is usually called *nesting*, and is where the indentation will really come in handy to show what's going on. Here it is:

[56] A common misconception among new programmers is, for some reason, that there has to be an `else`. This is not the case. Really.

[57] They were people getting on a bus in the first example, and it's an exam grade now, but the tests are the same. This is *abstraction* - spotting a pattern and reusing the solution.

Listing 4: `exam_result.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    mark = int(input("Enter the student's mark: "))

    if 0 <= mark <= 100:

        if mark >= 70:
            print('Distinction!')
        elif mark >= 40:
            print('Pass!')
        else:
            print('Fail!')

    else:
        print('Mark out of range')
```

As before, the first test could be written several ways. It all comes down to what "feels" best. Here we have used the way that is closest to how the test would be said ("The mark must be greater than or equal to zero and less than or equal to 100.") Notice how the indentation clearly shows which `else` belongs with which `if`.

Note: If you are typing along, your IDE might have taken exception to the first `if` line, and might be prompting you to rewrite it. This is because Python provides a shorthand, so this:

```
if mark >= 0 and mark <= 100:
```

can be shortened to:

```
if 0 <= mark <= 100:
```

Which you use going forward is up to you.

7.2.3 When Not to Test

Finally, there is one remaining source of errors in this program. What happens if the user enters something other than an integer? If you try this you will see that the program crashes. Obviously it would be preferable to inform the user if their mistake. This will not be much work, because *we did just this in the previous chapter*. There we checked that the number of pupils getting on a bus was an integer, so we can reuse that idea here.

Important: Remember that we prefer to use EAFP programming rather than LBYL. So we will *not* examine what the user types in to see if it really is an integer. We will just

carry on trying to use the value input and, if it turns out not to be an integer, we will pick up the pieces when an error is thrown.

Looking back, we see that the *exception* generated when the user enters something invalid will be a `ValueError`, so all we need to do is provide some quick error-handling to deal with this.

Listing 5: `exam_result.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    try:
        mark = int(input("Enter the student's mark: "))

        if 0 <= mark <= 100:

            if mark >= 70:
                print('Distinction!')
            elif mark >= 40:
                print('Pass!')
            else:
                print('Fail!')

        else:
            print('Mark out of range')

    except ValueError:
        print('Please enter an integer!')
```

Check out that indentation, and how it shows the program structure.

Tip: In this example, the programmer has also left a few blank lines. This is a hopeful attempt to make the structure of the program more obvious to someone reading it. Python just ignores these lines, but they can help a reader.

Remember that a program is read by a person more often than it it run!

7.3 Non-Linear Programs

This is all fine, and the program we have here should cover all the possibilities. But it would be preferable if the user was given the chance to re-enter a value if they make a mistake. It might also be handy to be able to run it for a bunch of students.

In summary, we are now able to write programs that run top-to-bottom, and can skip over some lines depending on some conditions. The missing piece[58] is to be able to go

[58] As it happens, that is pretty much all that remains to do. There isn't much more to programming after that.

back up in the program, or to return to some previous place in the code. Let's do that next!

7.4 Repeating Yourself

We often find ourselves repeating tasks. We might look through a drawer of socks *until* we find a matching pair. Or we might look through that box of festive chocolates, avoiding the toffee ones, *until* we find one we like. Or maybe we are asked to boil six eggs for breakfast, one at a time.

These kinds of *repetition* also occur in programming, where they are usually called *loops*. There are two, well, really three, types. To start with the two:

Determinate Loops
: This is where it can be known, before the first time the task repeats, how many times it will be done. It might be different each time the program runs, but it is always known. It's the eggs in the examples above - that will always happen exactly six times, no more, no less.

Indeterminate Loops
: This is where it is now known, and cannot be known, how many times the task will be done before it starts. The number of times depends on something that it unknown. This is the socks in the examples above - you might get lucky and find a pair in the first two, or you might need to look at a lot more.

The "three" comes in because there are two subtly different types of indeterminate loop. Sometimes it can be shown that the loop will always execute at least once, and sometimes it might never execute at all. To use the other example above, if we know that the tub of chocolates is full, we will always check at least one sweet one our way to finding one that is not toffee (so the checking for toffee will always happen at least once). Or, if the tub is open and could be empty, we might not check any sweets at all, so no check for toffee at all.

To implement these in a program there is only really a need for one new programming control statement, to give a loop. But it is convenient to have at least two, one for determinate loops, and one for indeterminate loops. Some languages provide three, with two for the indeterminates. Python goes with two, but a particularly Pythonic way of handling indeterminate loops is usually used. We'll look at these three now.

7.4.1 Determinate Loops

This is the case where it is known, in advance, how many times some statements need to be run. These loops are often referred to as `for` loops, because this is the keyword used to introduce them in many languages. Let's jump in with a simple example. Suppose we want to generate a table converting Fahrenheit temperatures into Celsius. A quick Google and some experimenting at the Python Interpreter would let us work out how to do the conversion:

```
>>> f = 80
>>> c = (f - 32) * 5 / 9
```

(continues on next page)

(continued from previous page)

```
>>> c
26.666666666666668
```

This is easy to put in a program that allows our user to enter the value to convert[59]:

Listing 6: f2c.py (with User Input)

```
#!/usr/bin/env python3

if __name__ == '__main__':

    fahrenheit = float(input("Enter the Fahrenheit Temperature: "))

    celsius = (fahrenheit - 32) * 5 / 9

    print('Celsius Equivalent is ', celsius, 'C', sep='')
```

Note: The sep in the print statement simply stops Python printing a space between each part of the output. It's just there for neatness.

This is fine, but we wanted to generate a table. As it stands the user would have to run the program over and over, recording the results to generate such a table. It would be much neater do just display the table for a *range* of values.

Happily, Python provides a range function that will generate these values. Let's start by displaying the results for Fahrenheit values from 0 to 10. This is not very sensible if this was part of a weather app, but we'll fix that later on. To achieve a table, we need to remove the user input, and just run the conversion on a range of values. We'll start with zero to 10:

Listing 7: f2c.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    for fahrenheit in range(11):
        celsius = (fahrenheit - 32) * 5 / 9
        print('Fahrenheit ', fahrenheit,  'F Celsius Equivalent is ',
↪celsius, 'C', sep='')
```

If you run this program you will see that all the *indented* lines after the for line are repeated. They are executed 10 times, with the value of fahrenheit ranging from 0 to 10. Some notes:

- The count starts at zero. As we have seen before, everything in programming starts at zero.
- This means that the value in the range must be the number of lines to print, not

[59] Note that this program allows the user to enter a floating-point value.

the highest value. We wanted all the values from 0 to 10 here, so that's 11 lines.

- And that `print` statement is getting messy. We'll add a fix for that next chapter.

You might have noticed that 0F is really very cold, and that our table does not yet reach normal temperatures that might be seen in a weather app. We can tweak the program to display temperatures from, say, 25F to 80F, easily:

Listing 8: `f2c.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    for fahrenheit in range(30, 81):
        celsius = (fahrenheit - 32) * 5 / 9
        print('Fahrenheit ', fahrenheit,  'F Celsius Equivalent is ',
↪celsius, 'C', sep='')
```

So now we provide the start point and the end point, remembering that the end must be one more than the final line we require.

Important: Fiddly details like the final value needing to be one more than you might think are hard to remember. Never be afraid to just run the program and see what happens. If it doesn't work quite as planned, just go back and change it.

Finally, we might decide that the table is a bit too detailed, and that it will be enough to give the equivalents in increments of 5. This sounds tricky, but turns out to be simple as `range` allows us to specify the increments.

Listing 9: `f2c.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    for fahrenheit in range(30, 81, 5):
        celsius = (fahrenheit - 32) * 5 / 9
        print('Fahrenheit ', fahrenheit,  'F Celsius Equivalent is ',
↪celsius, 'C', sep='')
```

See that `range` allows us to specify the start, end, and increment. If there is only one value it assumes that the start is zero and the increment is 1, as this is the most common. Two numbers are taken to be a start and an end with an increment of 1. Three numbers, and you're in full control.

The increment can be negative if there is a need to count down; in this case the end must be less than the start! (This is always mentioned in programming books at this point, but practical uses are rare, so you might want to forget it now).

The version of the program above has all the values coded in. So we can look at the program now, and know how many times the loop will execute. That's what makes it determinate. But the values could be entered by a user, like this:

Listing 10: f2c.py (with User Input)

```
#!/usr/bin/env python3

if __name__ == '__main__':

    start_f = int(input('Enter the starting value: '))
    end_f = int(input('Enter the ending value: '))
    increment = int(input('Enter the increment '))

    for fahrenheit in range(start_f, end_f, increment):
        celsius = (fahrenheit - 32) * 5 / 9
        print('Fahrenheit ', fahrenheit,  'F Celsius Equivalent is ',
↪celsius, 'C', sep='')
```

The loop here is still determinate, and we can still use `for` and `range`. We can't tell from the program now how many lines it would print, but just before the loop starts that number would be known (we could print it out if we wanted to). So it's still a determinate loop, even if the number of times it will run will vary.

Note: Before we move on, look again at that program. It really would need some checks that the values entered were integers, and that the start vakue was less than the end. And that the increment would eventually get us from the one to the other. So much of a program is devoted to handling what the user might do!

7.4.2 Indeterminate Loops

These are loops where the number of executions cannot be forecast in advance (that is, before the first execution). This is usually because they are required to run *until* something happens, or *while* something is true. Most languages use the keyword `while` to introduce these, so they are often just called *while loops*.

Note: Remember there are two possibilities: the loop might always execute at least once, or the loop might never execute at all. Some languages offer different ways of writing each one, Python does not. One of Python's design philosophies is that there should be one way of doing something.

A *while loop* causes statements to execute as long as ("while") some condition is true. The statements affected are indented, as usual. Here's an example.

```
sweet = ''

while sweet != 'caramel':
    sweet = input('What sweet did you find? ')
    if sweet != 'caramel':
        print('Ugh! Try again!')
```

(continues on next page)

(continued from previous page)

```
print('Yay! Found one!')
```

This snippet will repeat the prompt until the user enters `caramel`. The final `print` statement is not indented, so only gets executed once the loop has finished.

That is all there is to it. Some things to bear in mind:

- If the loop condition is `False` initially, the statements inside the loop will never execute.
- Something inside the loop must potentially change the value of the condition, else the loop will never exit. This is an *infinite loop*, and is usually a problem!
- The loop can contain other statements, including other loops.
- After the last statement inside the loop (shown by the indentation) the statements start over. There is no need to include anything to make this happen.

Now, an idea that we will return to later is that an important aspect of good programming is not to repeat yourself. Spot the repetition in that example loop? Let's fix it.

More Cunning Loops

Important: What follows in this section is in many ways specific to Python, and the way Python is usually used. If this book was about, say, Java, this section would not be here! An important part of using any language is to use it in the accepted way that has been developed by its community.

In the Python community, we say that we aim to be *Pythonic*.

The loop up above contains duplication. Or, if you prefer, it includes information about our problem twice. Or, to put it another way, if we wanted to change one thing we would have to make two changes to the code. Suppose for some mad reason we wanted `toffee` instead of `caramel` - we would have to make two changes. That's bad.

The Pythonic way around this works as follows. It makes use of an infinite loop, and an explicit statement to exit the loop. It gives neater code, possibly at the cost of making the loop's condition harder to find. Done this way, the example above would look like this:

```
while True:
    sweet = input('What sweet did you find? ')
    if sweet != 'caramel':
        print('Ugh! Try again!')
    else:
        print('Yay! Found one!')
        break
```

The advantages of this version are:

- The condition is in the code just the once. So if we change it we change one thing.

- The initial value of the variable `sweet` is irrelevant.
- The entire logic of this code is not all together (*encapsulated* is the word).

The main downside is that we have to burrow into the code inside the loop to find out how the loop will terminate.

Note: Anything that is always `True` can be used in loops like this. We have seen that Boolean `True` is really just *1* in disguise, so some programmers save a bit of typing by writing `while 1`. Something like `while 1 == 1` is not unknown!

Unless you have been told not to, for some very good reason, this is the way to write while loops in Python.

7.5 Pulling It Together

Let's finish this section by using the new ideas here to create a complete program. This example is not the most interesting, but it will use all the ideas from this chapter:

Times Tables

Write a program that prints a "times table". The program should prompt the user to enter an integer between 0 and 12 (inclusive) and should display the "times table" for that value, from "0 times" to "12 times".

While small, this is going to be the most complex program we have written so far. It is unlikely that we would be able to sit down and type the whole thing in without errors, so the trick is to break it down. We'll define a useful first version, and then add in features[60]. Let's go like this:

1. Write a program to print a single table, the "7 Times Table", say.
2. Change it so the user enters the number, without worrying about invalid entries.
3. Reject numbers outside the range 0 to 12 inclusive.
4. Allow the user to enter again if their number is out of range.
5. Fix the possibility that the user will not enter an integer.

Hold on! There is a lot in that list! Where did it come from? In many ways, that list is the hard part of programming. The hard part is taking a complex problem and breaking it down into smaller problems. If you do this, and repeat as needed, eventually you get to problems that are so small they can be solved. And gluging all the solutions together should solve the bigger problem!

Anyway, let's start with the first stage. This is a *determinate* loop, because we know there will always be 13 lines. The maths is not difficult, and a bit of fiddling will get the layout looking reasonable. A first version:

[60] In practice we would also save working versions of the program we we went. So that if we got into a mess adding a new feature we could always retreat to a known working state.

Listing 11: 7times.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    for table_line in range(13):
        print(table_line, 'x 7 =', table_line * 7)
```

We should test this program, even though it is simple, just to be sure it works. Then we take a look to see if it's a good start. Remembering that we should never repeat anything, we might spot that the 7 is in there twice; it makes sense to fix this, as for one thing it will make the next version easier. So, for the moment, let's turn that 7 into a constant, like so:

Listing 12: 7times.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    TABLE = 7

    for table_line in range(13):
        print(table_line, 'x', TABLE, '=', table_line * TABLE)
```

Now, to print a different table all that needs to be changed if the value of that one constant. Again, we would test.

Once satisfied, we need to change it so that the user enters the number, *but we do not care at this point about validating the input*. This is now easy, because all we have to do is change the way the constant gets its value (and at the same time change its name so that it is a variable).

Tip: No need to change the name of the constant in three places. Your IDE should have a feature to do this for you. Look for somehing called `Refactor` or similar.

The required change is just to the one line. This is the whole point of splitting the task like this! The new version:

Listing 13: any_times.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    table = int(input('Enter the table you require (0-12): '))

    for table_line in range(13):
        print(table_line, 'x', table, '=', table_line * table)
```

See that we give the user a hint of the allowed values, but we are not yet checking them. Also, there is a space between the end of the prompt and where the user will type. There is no harm in keeping our simple dialogue neat.

Next we want to check that the number entered is between 0 and 12. For the moment, we will just display the error, and not worry about anything else. This is obviously a conditional statement after the `input` line. The condition could be written two ways (either to define what is accepted, or what is not accepted), but the most obvious seems to be to say what values are allowed, like so[61]:

Listing 14: `any_times_2.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    table = int(input('Enter the table you require (0-12): '))

    if 0 <= table <= 12:
        for table_line in range(13):
            print(table_line, 'x', table, '=', table_line * table)
    else:
        print('Value out of range!')
```

So now the user knows of their error, and could just run the program again. Or we could offer them a chance to make good their error. This seems a better user experience, so we'll do that.

This is another loop. It's indeterminate, but will happen at least once. Instead of worrying about what the loop condition should be (and worrying that it would be the same as the condition in the conditional!) we'll be Pythonic. We'll put the whole program in an infinite loop, and jump out of the loop after successfully printing the table. There's not much to change; the only tricky bit is getting the indentation right:

Listing 15: `any_times_3.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    while True:
        table = int(input('Enter the table you require (0-12): '))

        if 0 <= table <= 12:
            for table_line in range(13):
                print(table_line, 'x', table, '=', table_line * table)
            break
        else:
            print('Value out of range!')
```

Important: If you needed another argument for being Pythonic when using these

[61] We saw a shorthand for testing if a value is between two others, so we'll use that here.

loops, here it is!

So now we `break` out of the loop only if we printed a table. Otherwise, we ask for the input again. Take a close look at the indentation. For example, the `break` is indented so as to be outside the `for` so that the whole table is printed. If you're unsure, move it across to line up with the `print` and see what changes.

Finally, we want to make sure that the user really does enter an integer. We are in Pythonic mood here, so we will not examine the value that's entered (LBYL - Look Before You Leap), we will just pick up the pieces if there is an error (EAFP - Easier to Ask Forgiveness than Permission). Long ago, we saw that there would be an exception generated, and it would be a `ValueError`, so we need to try that.

Listing 16: `any_times_4.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    while True:
        try:
            table = int(input('Enter the table you require (0-12): '))

            if 0 <= table <= 12:
                for table_line in range(13):
                    print(table_line, 'x', table, '=', table_line *
↪table)
                break
            else:
                print('Value out of range!')
        except ValueError:
            print('Please enter an integer!')
```

See how working like this means that we hardly touch the parts of the program we know work. That's the trick of the thing!

We now have a working, we hope, program. It contains examples of everything we have used so far. Study it!

7.6 Takeaways

There has been a lot in this chapter, but we have at least now completed a working program that does something useful. Here are the main ideas:

1. Boolean conditions allow us to control the order in which statements are executed, and/or which statements are executed. This is called the *flow of control*.
2. Conditional statements are introduced with the keyword `if` and allow for different statements to be executed. More possibilities can be covered using the optional `elif` and `else` keywords.

3. Determinate loops are written with the `for` keyword. The `range` function is a handy way to control the number of times loops execute.
4. Indeterminate loops use the `while` keyword. They can directly use a condition or, pythonically, be used as infinite loops in conjunction with the `break` instruction.
5. Indentation shows which statements are inside which loops and conditions.
6. There can be, and often are, loops within loops, and conditions inside conditions. And loops inside conditions, and so on.
7. The trick of programming is to split the big problem into smaller problems. And to then repeat, until the problems are small enougn to solve.

Our programs are now going to get complex, at least in the sense that they will be longer. So next we will look at ways to keep the amount of code we are working on to a manageable amount. Say 24 lines or so. Onward!

CHAPTER

EIGHT

THE WHEEL. DO NOT REINVENT

Python is quite a small language. We have already looked at most of the more important features, and will spend the next few chapters filling in some more details. For a moment we will pause, though, and look at some of the ways in which we can use useful *modules* of program code that come with Python. And in passing we'll mention some of the many other ways in which Python can be extended.

In many ways, modern programming is case a case of bringing together "stuff" from many sources into a whole that solves some new problem. A modern app might use a database, a framework for web applications, a toolkit for developing front-end websites, and more. Even with just a programming language there are many extensions and handy collections of code that can make our lives easier.

And using them is not cheating! Using them is just what professional programmers do. Not using them is just making more work for yourself.

Moreover, Python exists in the open source world, so many of these extensions are developed, tested, and used by huge communities across the world. Any bugs are quickly found and dealt with, and packages are extended and new versions released regularly. This means that you can be confident that the extensions you are using are tried and tested, and maybe you will be able to contribute to the community yourself one day.

Let's illustrate this with a simple programming task:

Check a String

Validate a user's input to ensure that the first character of a string they enter is an uppercase letter, and the last is a period (full stop).

We could start thinking about this problem along these lines:

```
Hmm. We can get the first character of a string at index ``0``, and we
can somehow write an ``if`` statement. Maybe we can see if it's
between ``A`` and ``Z`` or something. The last character will be index
 ↪``-1``,
and testing that will be easy.
```

This is the wrong way to think! Looking for uppercase letters, and checking how a string ends sound like they might be common things to need to do. They are. And so *there is a built-in way to quickly and easily do either*. This is always the way to think - does what I am trying to do seem like something that has been done many, many times before?

For the record, there is a built-in function in Python called `isupper` that will tell us if a character is an upper case letter. There is another for testing how a string ends, imaginatively named `endswith`. So to validate the string as required it's just a case if gluing these two together:

Listing 1: `string_check.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    stringy = input('Enter string to test: ')

    if stringy[0].isupper() and stringy.endswith('.'):
        print('String passes the test.')
    else:
        print('String fails the test.')
```

Tip: Another advantage of using built-in functions like this is that the resulting code when "read aloud" can often do a very good job of saying what the program is doing.

Of course, we do not know how `isupper` and `endswith` work. Nor do we want to know, or need to know. We just know what they do, and how we use them. We trust that they have been tested thousands of times, and are trusted.

Tip: As you enter a program, your IDE may well pop-up helpful hints about what you might want to type next. Look at these, and you will see possible functions that might come in handy later on.

These handy building blocks are available to us in two ways. Some, the most commonly used, are found in the *Python Standard Library*. Others, needed less often, are available for download as packages; these are found in the *Python Package Index*, which is sometimes affectionately known as *The Cheese Shop* after a famous Monty Python sketch.

8.1 The Standard Library

Many of the things we have mentioned so far - the built-in data types, for example - are part of Python's *Standard Library*. The library provides, in its own words "standardized solutions for many problems that occur in everyday programming". So this is the place to look for something that solves a problem that crops up in many different programming scenarios. It is safe to assume that the Standard Libray is available wherever Python is installed.

The contents of the Standard Library are listed in the docs at https://docs.python.org/3/library/index.html . You can tweak the drop-down at the top left of this page to change to the exact version of Python you have. Reading down the list we see:

More Data Types
: There is a richer collection of data types, mostly to handle collections of primitive

data items. We have those in the plan for the next chapter.

More Exceptions
: We have used a few of the more common exceptions, but there are more, and they cover every likely mishap. (You can actually add your own, but that's beyond our scope at the moment).

Collections of Code
: And then there are many collections (correctly called *modules* of useful code for common programming situations. These are the places to look for solutions to common problems.

The last here is worth pausing over. Take a look down the list and you see a whole bunch of potentially useful stuff for common applications. These are available, effectively built-in to Python, so there really is no good reason not to make use of them. Some of them are quite esoteric, and are probably used rarely. Some are useful much more often. Even experienced programmers should glance down the list now and again to remind themselves of what's available.

Here's an example. Suppose we have a file processing program, and we need to write some temporary data to a separate file. We need to generate a name for that file. Without thinking we might start writing some code to string a few characters together to form the name. Then we might stop and wonder what to do if it turned out somehow that the file already existed … But, look, here is the `tempfile` module. It contains code that will create a temporary file for us, and guarantee the name is unique. That is superneat.

Some of the more commonly used modules are (in the order they appears in the docs):

`string`
: Handy functions for processing strings (in addition to those that are always available).

`textwrap`
: Useful for consistently shortening lines of text, adding line breaks, and so on.

`datetime`
: One of many modules for handling dates and times, getting the current time, doing maths on dates, and more.

`math`
: All the usual maths functions you might remember from A level, and that calculator.

`decimal`
: Used for fixed-point numbers. Very useful when a value always has to have the same number of decimal places, often for amounts of money.

`fractions`
: For maths on, ah, fractions.

`random`
: Functions to generate pseudo-random numbers, make random choices, and the like. Useful in games and statistics.

`statistics`
: What it says! Don't code standard deviations, modes, and medians, they're in here.

os.path
: A handy interface to manipulating files on the system. See also `shutil` for a higher level interface.

zipfile
: For creating and opening Zip files. There are also modules for other common compression and archive formats. This is ver useful if you don't have WinZip or similar to hand.

csv
: For opening and processing files in comma-separated value format. Files exported from Excel usually.

getpass
: Reads a password (without it being displayed on the screen).

There are plenty more, but the later parts of the list get a bit specific. There are modules that provide functions for various Internet protocols, different multimedia, basic GUIs, testing, and many more.

So, if all this is available, how do we get at them? Let's do a simple example.

8.1.1 A Module Example

Here's a task:

Password

Write a program that reads a password from the user, twice, and confirms that the two passwords entered are the same. For debugging, display the password entered, along with the user's login name.

Eek! Where to start with that? Read the password is easy with `input`, but we know that the user's typing will be displayed. And how to find out who is logged in to the computer? All very tricky.

Obviously this is an example, and obviously the last module listed above, `getpass` is going to be our friend here. To find out what it can do, we can reads the docs, Google for an example (often a better call), or check the built-in help. To start using a module it needs to be made available, via an import:

```
>>> import getpass
```

And then there are two ways to remind ourselves of what is available in the module. The first just lists the names of the contents, and the second provides more details.

```
>>> dir(getpass)
>>> help(getpass)
```

Initially, you may find the built-in help, and the official docs, a bit difficult to follow. That's not a problem - a quick Google will surely lead you to an example of how to use the module.

The official docs[62] for the module tell us that there are two functions in this module. One gets a password, *without echoing on the screen*, and the second gets us the current user's id. These will surely simplify writing this program.

First, to get the password. The docs for this function tell us much:

> `getpass.`**`getpass`**`(`*`prompt='Password: ', stream=None`*`)`
>
> Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to `'Password: '`. On Unix, the prompt is written to the file-like object *stream* using the replace error handler if needed. *stream* defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).
>
> If echo free input is unavailable getpass() falls back to printing a warning message to *stream* and reading from `sys.stdin` and issuing a `GetPassWarning`.

We see that:

- We use the function as `getpass.getpass`. This is the usual deal, where a function is referenced with its name and the name of its module. It covers the chance that two modules will contain functions with the same name.
- We provide a prompt. If we do not, then `Password:` will be used (with a space).
- On Unix-like systems, we can read from different streams. This can be ignored.
- The password will not be displayed. If it could be, then a `GetPassWarning` exception will occur. Since this exception is also defined in the module it will be `getpass.GetPassWarning`.

So to make our program we will need to:

#. Import the `getpass` module. # Prompt the user for a password. #. Prompt again to re-enter (a different prompt here would be good). #. Compare the two, and display the user's id if they match. #. Display an error if they don't match. #. Display an error if it is not possible to enter the password without it displaying.

Phew! This looks a bit deep, but using the modules, it's very neat. Here we are:

Listing 2: `password.py`

```
#!/usr/bin/env python3

import getpass

if __name__ == '__main__':

    try:
        password = getpass.getpass()
        re_entry = getpass.getpass('Re-enter: ')

        user = getpass.getuser()

        if password == re_entry:
            print('User:     ', user)
```

(continues on next page)

[62] https://docs.python.org/3/library/getpass.html

(continued from previous page)

```
            print('Password: ', password)
        else:
            print('Passwords did not match.')

    except getpass.GetPassWarning:
        print('Password will show on screen. Exiting!')
```

The `import` goes at the top of the program, before the start of the `main` part of the code. This is another convention that it is important to follow. (If there are multiple `import` statements, another common convention is to include them in alphabetical order). Then the program just uses the module.

If you type this program into your IDE you will probably find that it is aware of the module, and once it has been imported, the IDE will suggest the names of the functions. That saves remembering them.

A final point is that this module we are using here is *cross-platform*. That is, it will work on any operating system. The version you are looking at above was written on Linux, but it would also work on Windows or Mac. Obviously Unix-like systems (Linux and Mac) manage users and passwords in different ways to Windows, so this really is a big thing. Using these modules there is no need to develop and maintain different versions of a program for different operating systems.

Now, some more details about good practice in imports.

8.1.2 Importing

In the password example we wanted to make use of everything defined in the module. There were only three things there, in fact. Other modules contain many more functions (and other stuff), and typically a programmer doesn't need all of them for a particular task. So it is good form to only import the parts of a module that are needed. Example:

Triangles

Write a program that takes the lengths of two sides of right-angled triangle, and displays the length of the third side.

For anyone who may have slept through maths, the requirement here is to square the two numbers given, and display the square root.

Important: For simplicity, we will ignore the need to validate the numbers input here. But we will return to this issue later on in the chapter. So don't panic.

Obviously square roots are a common thing to require in many maths applications, and a function can be found in the `math` module. As before, we do not know how it works, nor do we care. We just need to know we can give it a number, and it will give us back the answer we need. But there are many more things in the `math` module that we don't need, so we use a different `import` that states explicitly what we want.

Listing 3: pythagoras.py

```
#!/usr/bin/env python3

from math import sqrt

if __name__ == '__main__':

    side_a = int(input('Enter the length of side A: '))
    side_b = int(input('Enter the length of side B: '))

    side_c = sqrt((side_a * side_a) + (side_b * side_b))

    print('Side C length:', side_c)
```

Importing in this way have a few advantages. First, it is obvious to someone reading this program that it will make use of square roots, but nothing else "maths-y". Second, it saves invisibly including other things that might have names that could possibly clash with variable names. And, third, there is small saving in that there is now no need to add the name of the module before the name of the function.

Note: If you look closely at the line in that program that calculates the answer, you might realise that the two sets of brackets are redundant because multiplication always happens before addition. This is true, but leaving them in, as here, can make it more obvious what the line is doing (squaring, then adding).

Never forget that programs are written to be read as well as run.

The same approach works with any module, and typically a program will have several `import` statements at the top. Let's finish this section with a quick look at other Python Packages.

8.2 The Python Package Index

The PyPi[63] contains packages contributed by the Python community that have not made it into the standard distribution. This is simply because it makes sense to keep what is distributed small, and nothing to do with the quality of the submissions. Some of them are very highly regarded, especially in fields like Data Science.

This book uses a bunch of packages out of PyPi. One example is Pygments[64] which takes the program source code, and produces coloured and highlighted exampled, as you've seen many times by now.

Managing packages can be tricky, and is beyond out scope here. Packages often depend on others, and this can lead to a web of interconnected dependencies. The standard Python tool is `pip`, which is only a Google away should you want to try it out.

[63] https://pypi.org
[64] https://pypi.org/project/Pygments/

If the standard library packages fail to meet a need, it is always worth searching over PyPi. But you do need to evaluate what you find there. Things to look out for include the number of "Stars", the most recent updates, and the quality of the docs. You should also check that the licence allows you to use the package for your intended purpose.

Among the PyPi packages that might be of interest are:

`cryptography`
: Avoid having to code all those crypto algorithms with this.

`python-dateutil`
: Adds to the existing `datetime` module with things like "next month". Very useful for calendar apps.

`numpy`
: The standard package for scientific computation.

`pandas`
: Powerful package for Data Science. (An example of a dependency is that Pandas makes much use of, and therefore requires, NumPy.

`beautifulsoup`
: Used for processing HTML pages to extract information - "screen scraping".

But in general, if you need to do it, there is probably a package.

Packages on PyPi are all free, but are released under different licences. Sometimes the original author would like a credit in anything that uses the package, for example. A few exclude commercial use. These can be ignored, but it is good form to read and abide by the licensing terms.

8.3 Takeaways

This chapter introduced the idea of a *module* of useful things. These things might be functions (which are handy bits of code), or new exceptions. They can also include constant values (the `math` module include a value for Pi, for example). The key points:

- It is important to look through the standard library and get an idea of what it there.
- PyPi contains many more packages, generally more specialised.
- The `import` statement includes a package. It does above the `main` part of the program.
- It is best practice to import just that which is needed, using `from ... import`.
- Using library functions is helpful in writing programs that will run across different operating systems.
- Licensing is not be ignored.

Using modules also allows us to split a large problem into smaller ones. We have seen before how this is a crucial part of programming. Now we will go on to see how we can develop our own "chunks" of programs, that we can reuse in different applications. And we can organise them into modules, should we want.

CHAPTER

NINE

KEEPING IT SIMPLE

Right at the start of the book we went through some basic ideas that, together, formed what you need to know to start programming. If you look back, you will see that we have now covered everything. Sure, there are some things that will make our programming lives easier, and some features of Python that need a mention, but everything is very much there.

We could use what we have learned to write complex programs. But there would be a problem. These programs would quickly become very long, and the whole process would be unsustainable. How would we cope if a small change in a program at line 11 caused an expected error at line 276? Or even line 102982? Complex systems are made up of tens of thousands of lines of program code, often more. No-one can cope with that complexity, so there needs to be some way to keep the day-to-day programming tasks to something we can comprehend.

So, how many lines of program should a developer be working on? A reasonable rule these days would be an *absolute maximum* of 50, but ideally far fewer. The guideline used to be around 20, going back to the days when programming was done on a single-screen dumb terminal.

The guideline really is that a programmer should be able to see *all* the code they are working on, without scrolling up and down. In the olden days, this was about 20 lines.

Now it's a few more, but not by much.

The ideas in this section are all about keeping the amount of code a programmer works on down to a reasonable limit. But first, some more to convince you that this really is a good idea.

9.1 Code is Crafted

A common saying these days is that programming is a *craft*. This means many things, but essentially it tells us that programming is all about producing good, elegant solutions, that can be maintained and repaired, and that will work for a long time. It is not enough for a program to work - it must work well. Or, if you prefer, it should be *crafted*.

Think about a craftsperson making a chair. A chair can be made by nailing a few pieces of wood together, and screwing some legs on. But that would not produce a good chair. A good chair has to fulfill its basic function, but is also needs to looks good. If it is well made, and we will be happy to have it in our homes. We would expect a good chair to be used for a long time, and we would expect to be able to repair it. We would expect it to have been well made, or *crafted*.

Note: A while back, we used to say that software (programs) was *engineered*. And there was a whole discipline called *Software Engineering* and people called *Software Engineers*. It's still a reasonable reference - good programs are built and structured well in a similar way to complex machines - but it does seem to have fallen out of fashion.

Here are some ideas that help us think about what makes code good, or indeed *crafted*.

9.1.1 Code is Read

Code is read much more than it is written. This implies that good code should be readable, so that a programmer who is not the original author can quickly and easily see what it does, and how it does it. This means that good code:

- Follows all the conventions adopted by those skilled in programming in that language.
- Uses meaningful identifiers for all variables and constants.
- Does not rely on "neat tricks" or anything that obfuscates what is going on. Simplicity is best.
- Breaks the problem down into small chunks that are easy to understand, and which help isolate what might need to be changed.

Remember also that the programmer reading the code might still be the original author, a few years down the line!

9.1.2 Programming is a Team Effort

Anything other than the simplest programs is developed by teams of programmers. So there needs to be a way to split the work. Specifically:

- If a program is held in one single monolithic file, it is impossible for several programmers to work on it together. So splitting the work up is vital.
- If a program is held in a single file, on a single server, it will be difficult for programmers working in different locations to collaborate.
- In a complex program, it is unlikely that any one programmer will know how every single aspect works. So there needs to be a way to isolate parts of the program, for attention from certain programmers.

Important: The issues here introduce the need for *source code control*. More on this at the end of the book.

9.1.3 Multi-tasking is Difficult

Do you find it difficult to do seven things at once? Probably.

A single program that does seven things is difficult to write, for the same reason. It is difficult to keep track of what should be doing what, and how that affects other things.

Splitting the program into smaller chunks can resolve this problem. The key idea is that each "chunk" does exactly one thing. This means:

- That the code to be worked on will be shorter.
- Programmers can work on one chunk each, and later combine them.
- If the code has to do one thing, the chances are it will do that thing properly!
- And as the code does only one thing, it is relatively easy to test.
- If the code turns out to be incorrect, it is obvious where the fix needs to be.

This idea leads up to the most important concept here - DRY Code (and WET Code).

9.1.4 Don't Repeat Yourself

Once you have some program code that solves a problem it makes sense to keep using it wherever it can be used. This is, of course, the basic idea of *abstraction*, or taking a solution to one problem and using it elsewhere. For example, if a program requires a user to enter an integer value ten times, why write the code for that out ten times? Why not write it the once, and then "call" it whenever needed?

This idea exists in many areas of Computing[71]. Do something once, or save a data item once, and that becomes the definitive version. Once you have this version, use it wherever it's needed. The score is that you have recorded this one aspect of the system once, and in one place. If you need to change it for some reason, just change it once.

[71] Especially in databases, where the single most important idea is that a database should store every fact about its world *exactly once*.

This gives us the idea of DRY code[72], which is good code.

Important: DRY is *Don't Repeat Yourself*.

Do it once, and reuse it. DRY is good.

The alternative to DRY code is, obviously WET code.

> **Warning:** WET is Write Everything Twice.
>
> WET is Waste Everyone's Time.
>
> WET is We Enjoy Typing.

You should always aim for DRY Code. DRY Code is good for teams, good for keeping things simple, and good for crafting code.

9.2 Code Reuse

Let's jump in with an example. In the chapter on Modules, we had this program.

Listing 1: `pythagoras.py`

```
#!/usr/bin/env python3

from math import sqrt

if __name__ == '__main__':

    side_a = int(input('Enter the length of side A: '))
    side_b = int(input('Enter the length of side B: '))

    side_c = sqrt((side_a * side_a) + (side_b * side_b))

    print('Side C length:', side_c)
```

And we noted that the two `input` lines really need some sort of validation to make sure that the numbers entered are greater than zero. In fact, if the user's experience is to make sense, we really shouldn't be asking for the second number until the first has been entered correctly. It would also be best to allow the user to reenter their value if an error is detected.

The code to validate one number is straightforward, not least because we have seen it before! We simply stick the `input` inside the usual infinite loop, and `break` out once the number is acceptable. It looks something like this:

```
while True:
    side = int(input('Enter a length: '))
```

(continues on next page)

[72] See *The Pragmatic Programmer* by Dave Thomas and Andrew Hunt (Pragmatic Bookshelf, 2019).

(continued from previous page)

```
    if side > 0:
        break
    else:
        print('Value out of range. Try again.')
```

To fix the program, we could just use this code twice. *But we would be repeating ourselves!*. We need to way to put this "chunk" of program somewhere, so that we can use it more than once. That turns out to be easy, but defining a *function*. To do this, we basically just give this code a name, and define it at the start of the program. And then we can use it twice. The amended program looks like this:

Listing 2: `pythagoras.py`

```
#!/usr/bin/env python3

from math import sqrt


def read_side():
    while True:
        side = int(input('Enter a length: '))

        if side > 0:
            break
        else:
            print('Value out of range. Try again.')

    return side


if __name__ == '__main__':

    side_a = read_side()
    side_b = read_side()

    side_c = sqrt((side_a * side_a) + (side_b * side_b))

    print('Side C length:', side_c)
```

A lot going on here! Things to note:

- The function is defined *below* the `import` and above the main program.
- The function contains the usual code to read an integer, and has a name that explains what it does.
- The function is used (called, or "invoked") (twice) from the the main program.
- When it has finished, the function has a `return` line that sends a value back the main program.

Hint: By convention there are *two* blank lines above and below the function.

Important: There is actually very little new here. Using this function is exactly the same as using any of the built-in functions we have used before. The only real difference is that we wrote the function, and we can see its code.

This is fine, and we are being DRY, but let's make it a little better. At the moment, the prompt displayed when the user enters a value is always the same. In the original program it was different. We can change the way the function works by sending it some values. These values, called *parameters* work like this:

Listing 3: `pythagoras.py`

```
#!/usr/bin/env python3

from math import sqrt


def read_side(prompt):
    while True:
        side = int(input(prompt))

        if side > 0:
            break
        else:
            print('Value out of range. Try again.')

    return side


if __name__ == '__main__':

    side_a = read_side('Enter Side A Length: ')
    side_b = read_side('Enter Side B Length: ')

    side_c = sqrt((side_a * side_a) + (side_b * side_b))

    print('Side C length:', side_c)
```

So now (run it and see!) the program will display two different prompts. The string provided where the function is called passes into the function, and matches up with the `prompt` variable that is used in the `input` line.

Finally, the killer! We have forgotten to check that the value entered is an integer. We know that an exception will be thrown if the value is something else, and we know the code to catch this. Now we are using a function, though, we need to enter this new code just the once. And it will work in both places where it is used. DRY!

Listing 4: pythagoras.py

```
#!/usr/bin/env python3

from math import sqrt


def read_side(prompt):
    while True:
        try:
            side = int(input(prompt))

            if side > 0:
                break
            else:
                print('Value out of range. Try again.')
        except ValueError:
            print('Please enter an integer!')

    return side


if __name__ == '__main__':

    side_a = read_side('Enter Side A Length: ')
    side_b = read_side('Enter Side B Length: ')

    side_c = sqrt((side_a * side_a) + (side_b * side_b))

    print('Side C length:', side_c)
```

In Python, the technique we are using here is called a *function*. Every programming language provides something similar although, as usual, the names may change. "Function" is common, but you may encounter *procedure*. In some languages the correct term is *method*, although these are subtly different. For the moment "function" is what they will be called.

Now, let's step back and look a bit more at what is going on here.

9.3 Functions Explained

We have already seen how to use the functions that come as part of Python's Standard Library, and also those that can be imported from modules. What is new now is that we are going to write our own functions.

Just like the built-in functions, our functions should be tried and tested units of code that:

- Do exactly one thing (and therefore do it well).
- Are less than about, say, 25 lines of code.

- May take input values ("parameters") that affect what they do (like the value passed to `math.sqrt`.
- Usually[65] `return` some value once they have completed.

Ideally, functions also have the potential to be used in other programs, but this may not always be possible. Even then, using functions splits up the program into smaller chunks that are easier to write and manage.

Functions are defined at the top of a program, below any `import` line and before the main program starts. They are checked by Python, but the lines in them are not executed. And by convention functions are separated by *two* blank lines.

Important: It may seem that functions are complicating the issue. After all, you can write a program without them. But this is to miss the point. Functions make writing the program easier, by breaking down the task. Sure, you can write a 100 line program without functions, but try writing a 100,000,000 line system in one file!

Functions are called from the main program just like the built-in functions. Functions can, and often do, call other functions.

Let's look at some examples.

Example Function

Write a function to determine if a number is even.

The maths here is easy - we simply need to divide a number by 2, and see if the result is an integer. Or, and probably easier, we could use the modulus operator (`%`) and see if the result is zero. The function will need to receive the number to be tested as a parameter, and will return `True` if it is an even number, or `False` otherwise. And for a name `is_even` sounds like a good call[66].

Now to write the code for the function. The first line defines the name, and optionally, the names of any parameters. It can be useful if the identifier of the parameter gives a hint of the purpose of the value. So:

```
def is_even(number):
```

Then the rest of the function (called the *body* goes below. With a `return` to send the result back to whatever is using the function. There can be more than one `return` - whichever is reached first will terminate the function. So we can write:

```
def is_even(number):
    if number % 2 == 0:
        return True
```

(continues on next page)

[65] "Usually", because a function could just be a bunch of `print` statements. Even if a function just does some action (like opening a network connection, say) it usually returns a value to indicate whether or not it was successful.

[66] Choosing function names carefully can often mean that lines of program can be read, and that reading them explains what they do. This can eliminate the need for any other tedious way of explaining or documenting the code.

(continued from previous page)

```
    else:
        return False
```

That's it.

Of course, the function should be tested. This is often done just by writing a short program that tests the function with a variety of input values. So we could quickly print all the even numbers in a range:

Listing 5: `even_tester.py`

```
#!/usr/bin/env python3


def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False


if __name__ == '__main__':

    for count in range(-10, 11):
        if is_even(count):
            print(count)
```

And that should convince us that all is well with the function.

All is well, but the function doesn't do any error checking. If the value passed to the function is something other than an integer, there will be an error (an exception). *This is fine!* The exception should go back to whatever called the function, and we should expect that to deal with it.

Note: Functions can also generate their own exceptions. We'll do an example shortly. (In the jargon, we say that a function can *throw* an exception).

This function had just the one parameter, but in general functions can have any number. In this case, they need to be supplied in the expected order. To illustrate this, let's *generalise* our function for reading an integer. We'll create a version that takes three parameters:

1. The lowest allowed value.
2. The highest allowed value.
3. The message to display when the user is prompted to enter the value.

So we have there an integer, another integer, and a string. And there is an additional rule that the first integer must be lower than the second. We have most of the code itself from our previous example.

The first line of the function again defines it, and names the parameters:

```
def read_int_with_limits(lower_limit, upper_limit, prompt):
```

As usual, we try to use identifiers that show what is going on. In the main body of the function we just need the usual loop, with a check that the value is between the limits. We'll take the chance to *refactor* the code slightly to remove the `break`; we can just `return` at this point[67].

```
def read_int_with_limits(lower_limit, upper_limit, prompt):

    while True:
        try:
            number_entered = int(input(prompt))

            if lower_limit <= number_entered <= upper_limit:
                return number_entered
            else:
                print('Value out of range!')
        except ValueError:
            print('Please enter an integer!')
```

Note: Using the `return` and not a `break` is maybe a little controversial. There could be an argument that the `return` is a bit hidden away when written like this. Some programmers would prefer the `return` to be the last statement in a function. Your call. Either is fine.

This function can be called from anywhere else, with the call providing two integers, and a string, in that order:

```
sides = read_int_with_limits(1, 100, 'Enter the number of sides: ')
players = read_int_with_limits(1, 6, 'How many players? ')
```

Note: These calls show that there has to be a space on the end of the prompt if we are going to get a neat dialogue. Seeing this, maybe we should go back to the function and refactor it again to take the prompt without a space, but to add one in as it is displayed?

We have a problem if there is an error when the function is called, and the second number is lower than the first. By checking over the code we can deduce that the function would fall into an infinite loop, because there is no value between the two. What to do?

There is *absolutely no point* printing out an error. It does not help to tell the program's user that the *programmer* has made an error. The correct thing to do is to `raise` an exception to indicate to whatever is calling the function that something dreadful has happened, and the function cannot do its work. The problem here is with the values provided to the function, so we can simply use an existing exception - `ValueError` looks a good one - and throw this back.

[67] Either way to write this is fine. Some programmers prefer to always have the `return` as the last line. Some go further and say there should only ever be one `return`. We take the view here that anything is fine, as long as it is used consistently, and as long as the resulting code is clear.

```
def read_int_with_limits(lower_limit, upper_limit, prompt):

    if lower_limit > upper_limit:
        raise ValueError('Invalid limits')

    while True:
        try:
            number_entered = int(input(prompt))

            if lower_limit <= number_entered <= upper_limit:
                return number_entered
            else:
                print('Value out of range!')
        except ValueError:
            print('Please enter an integer!')
```

Here `raise` terminates the function and passes everything back to the caller. There should be code there that handles the exception, and does something sensible. In general, using an exception like this informs whatever is using the function that it was unable to do its work, and was in a state where there was no point carrying on.

Right. Let's finish this chapter with an example program that makes serious use of functions.

9.4 A Simple Game

The Rules

A game is played on a 20x20 grid. There is some buried treasure at a random location. The player starts at the bottom left, and can move north, south, east, or west. After every move, they are told how far they are from the treasure.

The aim is simply for the player to move to the treasure in as few moves as possible.

There are many ways to implement this simple game. What follows is just one example. It has been chosen so that it covers all the ideas from this chapter. (And it would also be easier if we could use some of the ideas in the next, but we are where we are[68].)

Let's go.

[68] There is only one place where this really becomes a pain. See if you can spot it.

9.4.1 Thinking It Through

Programming works like this. We have a problem, so now we start breaking the problem down into smaller problems. We can immediately see some of the problems we will have to crack:

- We will need to know the player's current position.
- We will need to randomly generate a location for the treasure.
- We will need to work out the distance between the player and the treasure.
- The program will need to ask for the player's move, and terminate once they are at the treasure.

We can see that it will be possible to detect that the player is at the treasure because the distance from it will be zero. Or there might be another way to do this.

Thinking more about the problem we might spot a couple of issues that will complicate things:

1. A player should not be able to move off the 20x20 grid.
2. The treasure location should not be the same as the player's starting position.

We are going to need to fix these, sure, but they are fine examples of the sort of issue we might (and will) decide to ignore for the moment. We will get a basic version working, and come back to these details later. Our basic version can just track the player around the grid, not worrying about limits.

Important: If something in a program looks tricky, it is often a good idea to pretend it isn't there, and to sort it once everything else is working properly.

The grid for the game is 20x20, so we'll follow the usual X and Y axis model. The X-axis goes from 0 to 19 across, and the Y from 0 to 19 up. Remember that we should count from 0! For this first version, we can now think:

- We need two integers for the player's position, one for the X position (across) and one for Y (up).
- The user needs to enter their move (a choice from N/S/E/W will do). That needs to be validated.
- Once we have a valid move, we can change the player's position.
- The whole thing can loop forever. Eventually it will end when the player reaches the treasure.

We have now thought out the problem. Above we have a basic *algorithm*, which is the way the program will work. We also have some promising ideas of how to represent the real world as data.

We *could* write all this in one program, but **it will be easier to write some functions**. This is especially so as the function to get and validate the move does seem rather like the function we already have up above to validate the entry of an integer. Let's start there.

9.4.2 Tracking the Player

We need a function that allows the user to enter a single character, which must be one of `N`, `S`, `E`, or `W`. If the user enters anything else, they should learn of their error, and be asked to reenter. The `in` operator will come in handy here, and `len` will allow its length to be checked. The input is a string, so there is no need for any exceptions. Here we go:

```
def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')
```

Important: Glance back up at the code for reading an integer. It is *almost the same*. That's **abstraction**.

Good stuff. Now we need to handle the player moving. If we had a function that accepted their current position and a direction to move, that would do it. The problem is that we plan to store the player's position as two integers (one for `x`, one for `y`), so the function would need to return two things. But it can! Just separate the values with a comma like this[69]:

```
def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y
```

Should the program check that the direction is valid? In the case of the current program there is no need because we know that the `get_direction` function will only ever give a valid direction. Worst case, if the direction in `move` was invalid the position would just be unchanged. So we leave it in this case.

Note: The program now has two functions, and it will include more. It is a good idea to include them at the top of the program in roughly the order they are used. Remember to separate them with two blank lines.

---

[69] There is something going on behind the scenes here, but there is no need to worry about it. We can just treat it as being able to return two values from our function.

Armed with the two functions, the main program is now easy, and short. Which was the whole point!

Listing 6: treasure_hunt_1.py

```
#!/usr/bin/env python3

from random import randint


def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y


if __name__ == '__main__':

    player_x = 0
    player_y = 0

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        next_direction = get_direction()
        player_x, player_y = move(player_x, player_y, next_direction)
```

The program is growing (about 40 lines now), but we are only ever working on small sections of it.

9.4.3 Placing the Treasure

Now let's add in the secret location of the treasure. This is a random location, so clearly the `random` module will be our friend here. We have decided to ignore (for the moment) the chance that the random location will be where the user starts, so all we need is two random integers, on a scale from 0 to 19, inclusive. A check in the docs reveals a function called `randint` that does that.

Assuming we have the function available via an `import` here are two ways to write that function.

```
def place_treasure():
    x_pos = randint(0, 19)
    y_pos = randint(0, 19)

    return x_pos, y_pos
```

```
def place_treasure():
    return randint(0, 19), randint(0, 19)
```

These are equivalent in that they do exactly the same. But the first version "spells out" what it is doing, and is arguably a little clearer because of that. The choice here is largely personal preference, but we'll use the first, as clarity is important. We'll also tweak the main program to report where the treasure is; this will be useful for testing, but would need to be removed if anyone wanted to play the game seriously!

Note: A common passtime among programmers is to try and write complex things in one line. This is fine as an intellectual exercise, and can while away the long winter evenings, but *clarity in code in important*. So sometimes it is better to use longer code, just to make sure that everything is clear.

Here's the program as it now is, with the new lines marked:

Listing 7: `treasure_hunt_2.py`

```
#!/usr/bin/env python3

from random import randint


def place_treasure():
    x_pos = randint(0, 19)
    y_pos = randint(0, 19)

    return x_pos, y_pos


def get_direction():

    while True:
```

(continues on next page)

(continued from previous page)

```
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y


if __name__ == '__main__':

    player_x = 0
    player_y = 0

    treasure_x, treasure_y = place_treasure()

    print('Treasure is at (', treasure_x, ', ', treasure_y, ').', sep=
↪'')

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        next_direction = get_direction()
        player_x, player_y = move(player_x, player_y, next_direction)
```

9.4.4 Tracking the Distance

It will add to the excitement if we add in the distance the player is from the treasure. Obviously this will be another function, that will take the two positions as parameters and return the distance between them. A Google will tell us that the required maths is a bit of Pythagoras, suspiciously similar to an example we used earlier. This function is also an example of something that is quite common in many applications, and therefore something that we might have around from some other project. It is also something that it undoubtedly in a package in PyPi, but as it's a one-liner it will be quicker to just code it. That said, we'll keep the identifiers general, in case it does have use elsewhere.

Having said that the function is a one-liner, the brackets turn out to be fiddly, so for

ease and clarity it's been spelled out here.

Note: This maths also involves square roots, so the `math` module is needed. Remember that when there are several `import` statements it is good form to include them alphabetically.

Listing 8: `treasure_hunt_3.py`

```
#!/usr/bin/env python3

from math import sqrt
from random import randint


def place_treasure():
    x_pos = randint(0, 19)
    y_pos = randint(0, 19)

    return x_pos, y_pos


def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y


def distance_from(x1, y1, x2, y2):

    dx = x2 - x1
    dy = y2 - y1
```

(continues on next page)

(continued from previous page)

```
    return sqrt((dx ** 2) + (dy ** 2))


if __name__ == '__main__':

    player_x = 0
    player_y = 0

    treasure_x, treasure_y = place_treasure()

    print('Treasure is at (', treasure_x, ', ', treasure_y, ').', sep=
↪'')

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        print('Distance to Treasure: ', distance_from(player_x,
↪player_y, treasure_x, treasure_y))
        next_direction = get_direction()
        player_x, player_y = move(player_x, player_y, next_direction)
```

Now all that is really needed is to determine whether the user has "won".

9.4.5 The Endgame

The player wins when they arrive at the treasure. Two ways exist to spot this:

1. The distance between the two will be zero.
2. The co-ordinates of the two match.

Either would work, but the first relies on floating-point maths. What would happen if the distance was reported as `0.000001`, for example?[70] It is therefore better to just compare the positions. If this is done in a function, as obviously it should be, it would also be quick and easy to drop in a version using the other approach.

The new function just needs to take the two positions, and return a Boolean to indicate whether or not they are the same. There is a common "recipe" here. Many functions have a structure along the lines of *if some condition is True, return True, otherwise return False*. In this case, it is just as easy to return the condition. And it saves typing. Compare the two:

```
if player_x == treasure_x and player_y == treasure_y:
    return True
else:
    return False
```

[70] In practice, a program should never check that a floating-point value is *exactly* zero. It should check that the value is less than, say, `0.0000001` and treat that as zero. For the same reason, never compare two floating-point values for equality.

which is precisely the same as:

```
return player_x == treasure_x and player_y == treasure_y
```

This is a case where spelling things out doesn't really add anything. The single line is fine, and would be understood by anyone reading the program. (You might even find that your IDE would highlight the first version above as an "error" and offer to fix it to the second.)

The latest version of the program uses the second structure, but uses general identifiers in case the function could be useful elsewhere. The function is used in the main program, which exits once the treasure is found.

Listing 9: `treasure_hunt_4.py`

```
#!/usr/bin/env python3

from math import sqrt
from random import randint


def place_treasure():
    x_pos = randint(0, 19)
    y_pos = randint(0, 19)

    return x_pos, y_pos


def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y
```

(continues on next page)

(continued from previous page)

```
def distance_from(x1, y1, x2, y2):

    dx = x2 - x1
    dy = y2 - y1

    return sqrt((dx ** 2) + (dy ** 2))


def same_position(x1, y1, x2, y2):
    return x1 == x2 and y1 == y2


if __name__ == '__main__':

    player_x = 0
    player_y = 0

    treasure_x, treasure_y = place_treasure()

    print('Treasure is at (', treasure_x, ', ', treasure_y, ').', sep=
↪'')

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        print('Distance to Treasure: ', distance_from(player_x,
↪player_y, treasure_x, treasure_y))

        if same_position(player_x, player_y, treasure_x, treasure_y):
            print('Treasure Found!')
            break

        next_direction = get_direction()
        player_x, player_y = move(player_x, player_y, next_direction)
```

9.4.6 Final Tweaks

We noted two special problems right at the start, which were left to the end. Time to fix them.

The easiest to fix is that the treasure should not be generated right next to the player. A better fix would probably be to say that it has to be a reasonable distance away, so this is a very quick fix indeed by just changing the lower limit of where it can be generated. Our functions help us find the correct spot to make the change quickly, and there it limited risk that we will break anything.

This is also a good moment to note that the dimensions of the game area are actually defined in the program twice, so (assuming a square playing area) we are repeating ourselves. Time to introduce some constants, which will make changing the rules easier in future. Constants are defined below the `import` and before the functions. We'll

add two, one for the maximum playing area size, and one for the lowest position the treasure can be at.

Listing 10: `treasure_hunt_5.py`

```
#!/usr/bin/env python3

from math import sqrt
from random import randint

BOARD_SIZE = 20
TREASURE_MIN = 6


def place_treasure():
    x_pos = randint(TREASURE_MIN, BOARD_SIZE - 1)
    y_pos = randint(TREASURE_MIN, BOARD_SIZE - 1)

    return x_pos, y_pos


def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    return x, y


def distance_from(x1, y1, x2, y2):

    dx = x2 - x1
    dy = y2 - y1

    return sqrt((dx ** 2) + (dy ** 2))
```

(continues on next page)

(continued from previous page)

```
def same_position(x1, y1, x2, y2):
    return x1 == x2 and y1 == y2


if __name__ == '__main__':

    player_x = 0
    player_y = 0

    treasure_x, treasure_y = place_treasure()

    print('Treasure is at (', treasure_x, ', ', treasure_y, ').', sep=
↪'')

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        print('Distance to Treasure: ', distance_from(player_x,
↪player_y, treasure_x, treasure_y))

        if same_position(player_x, player_y, treasure_x, treasure_y):
            print('Treasure Found!')
            break

        next_direction = get_direction()
        player_x, player_y = move(player_x, player_y, next_direction)
```

The second problem was that the user should not move off the playing area. The constant just defined will be useful here, and it looks as if changes are needed in the `move` function. The simplest fix is just to check the new position, and only to return it if it is still on the playing area. Otherwise, an exception will be sent to show that the move it not allowed.

Important: An exception is used because that can be processed in the program. There is no point printing a message, because there might not be anyone to read it!

Finally, the main program deals with the exception. The complete program is below.

Listing 11: `treasure_hunt.py`

```
#!/usr/bin/env python3

from math import sqrt
from random import randint

BOARD_SIZE = 20
TREASURE_MIN = 6
```

(continues on next page)

(continued from previous page)

```
def place_treasure():
    x_pos = randint(TREASURE_MIN, BOARD_SIZE - 1)
    y_pos = randint(TREASURE_MIN, BOARD_SIZE - 1)

    return x_pos, y_pos


def get_direction():

    while True:
        direction = input('Enter direction to move (N/S/E/W): ')
        if len(direction) == 1 and direction in 'NSEW':
            return direction
        else:
            print('Error! Enter one of N/S/E/W.')


def move(x, y, direction):

    if direction == 'N':
        y += 1
    elif direction == 'S':
        y -= 1
    elif direction == 'E':
        x += 1
    elif direction == 'W':
        x -= 1

    if 0 <= x <= BOARD_SIZE - 1 and 0 <= y <= BOARD_SIZE - 1:
        return x, y
    else:
        raise ValueError('Attempt to move off the playing area!')


def distance_from(x1, y1, x2, y2):

    dx = x2 - x1
    dy = y2 - y1

    return sqrt((dx ** 2) + (dy ** 2))


def same_position(x1, y1, x2, y2):
    return x1 == x2 and y1 == y2


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    player_x = 0
    player_y = 0

    treasure_x, treasure_y = place_treasure()

    print('Treasure is at (', treasure_x, ', ', treasure_y, ').', sep=
↪'')

    while True:
        print('Player is at (', player_x, ', ', player_y, ').', sep='
↪')
        print('Distance to Treasure: ', distance_from(player_x,
↪player_y, treasure_x, treasure_y))

        if same_position(player_x, player_y, treasure_x, treasure_y):
            print('Treasure Found!')
            break

        next_direction = get_direction()

        try:
            player_x, player_y = move(player_x, player_y, next_
↪direction)
        except ValueError:
            print('Cannot move off the playing area!')
```

Looking very closely at the `move` function, it could be argued that it is now doing two things. It is moving the player, *and* checking whether they are still in the allowed area. Maybe there should be a separate function to do the second part? Decisions like this crop up all the time - for the moment there is no serious reason to change things, but refactoring might be needed in the future.

9.5 Using Functions

The final version of the simple game is about 80 lines long. But because it makes use of functions, none of the chunks of code are difficult to manage. The main program is the longest (just on 20 lines), but most of the functions are very short. It shows that functions (and thinking in terms of functions) make the task of constructing a long program much easier.

There is also the advantage that a couple of the functions in this program have been adapted from functions that had already been created in different contexts. And some of them could be useful if we were asked to write similar games. This leads neatly into a mention of the possibility of building your own *modules*.

9.5.1 Creating Modules

You can create modules of useful, related, functions just by putting them in the same file. You can then `import` this file in the same way you would do those from the standard library. It is usually a good idea to include some sample code at the bottom of the file that runs the functions, and acts as a basic test. Python will search for modules according to a defined list of possible locations; one of these is the same folder as the program with the `import`, which is therefore usually the easiest place to store the file!

Important: A common "gotcha" is to create your own module, and to give it a name that's the same as one from the standard library. Python sees your version first, so you effectvely hide the standard one. A simple workaround is always to prefix module names with `my_` or the name of your project or business.

A common use of this is to have a module of *helper functions* that you find useful in your daily programming tasks. A software business might have its own too. These would be functions that have no specific use, but which just generally come in handy in a variety of applications.

9.6 Takeaways

Using functions is an important part of writing code that is DRY. You should always aim to represent everything - every value, every piece of logic - from a problem **exactly once**. That way there is a good chance the representation is correct and, if not, a fix can be applied in one place.

Programming is a process of breaking down an overall problem into smaller chunks. These chunks eventually become easy to solve and work with, and correspond to functions.

Specifically:

- Breaking down a problem into functions results in programs that are easier to write, and easier to maintain.
- Programs need to be read and understood. Good use of functions, with clear naming, helps with this.
- Python functions are defined at the top of a program. They take parameters to alter the way that they work, or the result they produce.
- Functions can either be specific to a particular problem, or can be more general. If the latter, they can be written using more generic naming.

Finally, never be tempted to write a long, long program with the intention of "turning it into functions" later. It is insanely difficult. The whole point of using functions is to keep the amount of code currently holding your attention to a reasonable amount. Don't take on misguided approaches that will make your life so very miserable.

CHAPTER

TEN

COLLECTING

As we've already noted more than once, we have actually covered all we need to know in order to create programs. Modules and functions are really just a convenience that help keep the job simple, and allow for useful chunks of program to be reused. They're all about making things easy, and saving unnecessary work. The new idea we'll meet in this chapter is almost the same - we don't strictly need to be able to handle *collections* of related data items, but being able to will make certain tasks much easier.

So this chapter introduces the idea of *collections* of data items. All programming languages will provide features for this and, as usual, the names differ but the basic ideas are the same. In the earliest languages collections were called *arrays*, and that name is still found in some modern languages. Other names you'll see include lists, maps, sets, and more.

Python provides a few collections as standard, and the *collections* module in the standard library contains a whole lot more. In general, a collection:

- Stores a number of related data items.
- May or may not require that those data items are of the same time.
- May store the items in a particular order, or may not consider order to be important.
- May or may not allow duplicate items.

We'll look at four collection types here[76]. These cover all the common use cases, and together form a complete toolbox to draw on. *Strictly* you only need one collection type to do anything, but some do allow for more elegant and neater solutions. Anyway, below we will describe:

Lists
: The Swiss Army Knife of collections. You can do anything with a list. A list stores data items, and maintains the order. Usually the items are all of the same time.

Tuple
: Think of this as being like a row in a database table. This is a collection of data that represents something. The data items do not have to be the same type. Order is not important.

Set
: Basically a list, with the handy extra property that items in it have to be unique.

[76] This almost said three, but we'll include Sets so as to be complete, and because they do have some nifty uses now and again.

This type also supports the usual operations (difference, intersection, etc) from maths set theory.

Dictionary
A key-value collection. It is used by looking up a key, and finding the corresponding value. It's just like looking up a definition in a paper dictionary.

A collection is simply a bunch of related values that it is convenient to treat as a single entity. They have a single name (identifier), so they can be passed into functions, and returned from functions. So a list or tuple or set contains a collection of related data items.

The various collections have slightly different properties. In a list, for example, order is maintained - new values are added at the end, or among the existing values, in which case the others "shuffle along". Sets, in contrast, have no concept of order. Lists are said to be *mutable*, which means they can be changed. Tuples are *immutable*, which implies the opposite. Knowing a few of these details helps with picking the best collection for a particular application.

We start with lists, as these are the most general. Remember, you can do anything with a list.

Note: This chapter will not attempt to cover all the details of each of these collections. That is what the docs are for! The Python docs include Tutorials[73] along with the full gory details[74]. Links to the relevant docs are included in each section below.

And, as always, Google will lead you to more tutorial material and examples.

10.1 Looking at Lists

A list is a collection of values, where the order is assumed to be important (although it doesn't have to be). Let's use a list to explore the whole idea of a collection.

See also:

The official docs[75].

A value in a list is often called an *element*. Usually, all the elements in a list are of the same type, which is to say that lists are *homogeneous*[77]. It is assumed that the order of the elements in a list has some importance, if only that the elements added most recently are at the end. This in turn means that a list can usually be *sorted* into ascending or descending order.

Hint: So lists are a good choice if you have data that needs to be sorted, or where you need values like the highest or lowest.

[73] https://docs.python.org/3/tutorial/datastructures.html
[74] https://docs.python.org/3/library/stdtypes.html
[75] https://docs.python.org/3/library/stdtypes.html#lists
[77] Note this says *usually*. Lists can contain elements of different types, but this often breaks the point of having a list, and the concept of "order" becomes difficult. A tuple is often a better call in this case.

It follows that there must be a way to sort the elements, that they have a concept of order, which brings us back to the idea that all the elements should be of the same type.

Enough description. Let's do an example that shows the basics.

10.1.1 A List Example

Test Marks

A school pupil has taken four maths tests. Whether they pass or fail overall depends on the average of the four marks from the five tests.

Write a program that takes the marks are finds the average.

There are four test marks here, all of which are going to be integers. So, without collections, we might think that something like this would work. (In a real program the values would be entered, but here we'll just assign the values here to keep the code short).

```
>>> mark_1 = 65
>>> mark_2 = 55
>>> mark_3 = 45
>>> mark_4 = 60
>>> average = (mark_1 + mark_2 + mark_3 + mark_4) / 4
```

This is fine. But what would happen if there were five tests? Or three? We would need to add or remove variables, and remember to tweak the division to find the average. These multiple changes mean that this not DRY code! Being able to handle *any* number of marks would be DRY and would give a promising chance of creating some reusable code. We can, of course, do just this with a list.

Important: Yes, you say, but the spec said *four* marks, so why write code that can handle any more? The answer is that this is *generalisation*. It is not much extra effort, and we will end up with some code that could be useful in other applications.

Suppose we wanted the average of 10 temperature readings. If we have general code we could reuse it. That's *abstraction*.

So, let's rework this code to use a list. A list is created in the same way as any other variable, by giving it a value. A list is denoted with square brackets, so to create an empty list:

```
>>> marks = []
```

Or to create it with some values already in it:

```
>>> marks = [50, 40, 30,]
```

Note: That comma at the end might look odd, and indeed you'll find that you could

leave it out. Many programmers prefer this style, though, because it makes adding new values easier. Your call[78].

The important new idea here is that the list `marks` is a single variable, that happens to contain multiple values. Being a list, order is maintained, so a new value can be added, and will be at the end:

```
>>> marks.append(20)
>>> marks
[50, 40, 30, 20]
```

We can start to see how this would be useful when we realise there are handy built-in functions to calculate useful values from the values in the list. Like their number (the length of the list) or their sum:

```
>>> len(marks)
4
>>> sum(foo)
140
```

Hint: There are other built-in functions too, like `max` and `min` to find the highest and lowest values. Always remember to check for built-ins before writing some new code.

The key thing is to think "This *must* have been done before!".

Using a list, the program to find the average of the four marks could use code something like this:

```
>>> marks = []
>>> marks.append(65)
>>> marks.append(55)
>>> marks.append(45)
>>> marks.append(60)
```

```
>>> average = sum(marks) / len(marks)
```

There's not much gained so far, but what if there were a different number of marks? Or how about a program that could handle any number of marks. That sounds a like a loop. If we know in advance how many marks there would be, a `for` loop could work in a program something like this:

Listing 1: `marks.py`

```
#!/usr/bin/env python3

NUMBER_OF_MARKS = 5

if __name__ == '__main__':
```

(continues on next page)

[78] As we will see, this style is more important with tuples. So a good argument for doing this with lists is that it's consistent with other collections.

(continued from previous page)

```
    marks = []

    for count in range(NUMBER_OF_MARKS):
        next_mark = int(input('Enter the next mark: '))
        marks.append(next_mark)

    average_mark = sum(marks) / len(marks)

    print('Average Mark:', average_mark)
```

Note: For clarity, the code to verify that the mark entered is a number (and presumably is also in some valid range) has been left out here.

Here the number of marks is defined as a constant, but it could just be entered by the user. So to make this program work for any number of marks, all that would be needed would be to change the value in `NUMBER_OF_MARKS`.

We can improve the readability of this program a little by pausing to think that averages are something that must be worked out a lot. A look in the docs (or a Google) would reveal that there is no built-in way to find an average in the standard library, but that there is a `statistics` module containing all sorts of promising stuff. So we can use this to make things a little neater (and the program a little shorter).

And one final tweak will be to renamne to `count` variable in the `for` loop. There is a convention (yes, another one) that if the variable is not of any use other than to control the loop it is named _. So:

Listing 2: `marks.py`

```
#!/usr/bin/env python3


from statistics import mean

NUMBER_OF_MARKS = 5

if __name__ == '__main__':

    marks = []

    for _ in range(NUMBER_OF_MARKS):
        next_mark = int(input('Enter the next mark: '))
        marks.append(next_mark)

    print('Average Mark:', mean(marks))
```

For completeness, and for the sake of another example, it's a small change to this program to get to a general-purpose version that could handle any number of marks. The change is to the loop, which in a general solution is *indeterminate*, and we need to pro-

vide the user with a way of indicating that have finished. We'll have them enter -1 now[79]. The constant can obviously be removed, and we end up with a DRY solution.

Listing 3: `marks.py`

```
#!/usr/bin/env python3


from statistics import mean

if __name__ == '__main__':

    marks = []

    while True:
        next_mark = int(input('Enter the next mark (-1 to end): '))
        if next_mark == -1:
            break
        else:
            marks.append(next_mark)

    print('Average Mark:', mean(marks))
```

Another advantage of having all the marks data in a list is that we can do many other useful things with it. The `max` built-in function would give us the highest value, for example. And the `statistics` module is full of other possibly interesting values to calculate.

10.1.2 Working with Lists

This is not the place for a full description of all the things you can do with lists. Instead we'll talk about some of the most common uses, along with examples. Lists are very flexible and powerful, and Python provides several useful ways of putting them to work.

Keep in mind the most important features of a list in Python:

- It is heterogeneous (all the elements in it are of the same type).
- Its order is maintained, and is probably important.

As we have seen in the example, a list is denoted by square brackets, and can be created with or without some initial values:

```
>>> speeds = []
>>> speeds = [1, 2, 3,]
```

It can be cleared either by just recreating it, or by emptying it:

```
>>> speeds.empty()
```

[79] Probably not a good example of UX. Next chapter we'll include a way to do this neatly.

Note: Many of the operations carried out on lists use a *dot notation*, where the identifier of the list is joined to the name of the operation by a dot. The `empty` operation did so above, and we have seen it before as in:

```
choices = ['spam', 'eggs',]
choices.append('beans')
```

The append is using this dot notation. These operations behave rather like the *functions* we have met before, but are correctly called *methods* because of this different syntax. So that is the name used here.

Let's start by looking at what the *order* of a list means.

List Order

The order of a list is maintained, whether or not it is important. This compares with a string, which is a sequence of characters, where the order is usually important. New elements are usually added at the end of list using the append method, like so:

```
>>> choice = ['beans',]
>>> choice.append('spam')
>>> choice
['beans', 'spam']
>>> choice.append('spam')
>>> choice
['beans', 'spam', 'spam']
```

Important: This example also neatly illustrates that there can be duplicate elements in a list.

In rare cases[80] new elements can be inserted at a specified position. Positions count from zero (again, just like a string), and existing elements "shuffle along":

```
>>> choice.insert(1, 'egg')
>>> choice
['beans', 'egg', 'spam', 'spam']
```

The most common useful order is to have the elements sorted. The meaning of "sort" depends on the type of the elements, but usually it does the obvious thing. Very confusing things happen if a list containing several data types is sorted. To sort a list, we just use the `sort` method:

```
>>> speeds = [12, 8, 23, 17]
>>> speeds.sort()
>>> speeds
[8, 12, 17, 23]
```

[80] So rare that this example might have been left out.

And to reverse the sort, just `sort` and the `reverse` the list:

```
>>> speeds.sort()
>>> speeds.reverse()
>>> speeds
[23, 17, 12, 8]
```

If you need to maintain a list in order, the procedure is simple. No need to find the correct place, and insert the new element. Just add it on the end, and `sort` the result:

```
>>> speeds = [12, 8, 23, 17]
>>> speeds.append(10)
>>> speeds.sort()
>>> speeds
[8, 10, 12, 17, 23]
```

That's it!

List Slices

The examples above mentioned that lists behave rather like strings. This is intentional - they are both what Python calls an *iterable*. So it follows that string operations and slices work on lists. Individual elements can be found via their index, counting from `0` on the left or `-1` on the right:

```
>>> speeds
[8, 10, 12, 17, 23]
>>> speeds[0]
8
>>> speeds[-1]
23
>>> speeds[2]
12
>>> speeds[-3]
12
```

The two ways of counting mean that there are always two index values to get any element. Using this index value, elements can be changed:

```
>>> speeds
[8, 10, 12, 17, 23]
>>> speeds[0]
8
>>> speeds[0] = 7
>>> speeds
[7, 12, 17, 23]
```

Slices work too:

```
>>> speeds
[8, 10, 12, 17, 23]
```

(continues on next page)

(continued from previous page)

```
>>> speeds[:-1]
[8, 10, 12, 17]
>>> speeds[2:]
[12, 17, 23]
>>> speeds[::2]
[8, 12, 23]
```

Slices can be assigned too, but that's a bit obscure. Try it and see!

Finding Elements

So you have a list containing some values, and you need to know whether a given value is there. That's a very vague and abstract description, but it's actually quite common. Suppose you are reading car number plates as they pass, but only want to record each one once. Or a user has entered a choice from a menu, and you want to check whether it's a valid choice. Both require that you check *membership* of your list or, if you prefer, whether a given value is there.

There are two ways to do this, but we are not breaking Python's "one way to do something" rule, because it depends on what exactly you want to do. Do you just need to check if a value is in a list, or do you want to do that, *and* find out where it is?

In the first case (you just want to check if a value is in a list) we can use the fact that lists and strings are both iterables, and use the `in` operator. It simply tells us whether a value is, ah, in the list:

```
>>> speeds = [12, 8, 23, 17]
>>> 12 in speeds
True
>>> 14 in speeds
False
```

Alternatively, if we need to know where in the list the value can be found, the `index` method is the one to reach for:

```
>>> speeds = [12, 8, 23, 17]
>>> speeds.index(12)
0
>>> speeds.index(14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 14 is not in list
```

That exception if the value is not to be found means that `index` is often used after `in` has been used to check whether value is there.

Looping Lists

Often a program needs to do something with every element in a list. So it makes sense to allow `for` loops to iterate across a list. In fact, the `range` function that we met when first looking at `for` loops effectively generates a list behind the scenes[81]. So we could have:

```
knights = ['Robin', 'Galahad', 'Bedevere']
for a_knight in knights:
    print('Bold Sir ', a_knight)
```

This is a very common operation when printing results, searching for a value, and so on.

Copying Lists

Copying lists is straightforward, but a bit of a "gotcha". To understand why, we need to think a little about how lists are stored. We can think of it this way:[82] a list is a pointer to a memory location where the first value is stored. That value is stored along with a pointer to the next, and so on. Eventually a value is reached that has no pointer, so this must be the end of the list.

So if we have this:

```
speeds = [1, 2, 3, 4,]
```

there are four memory locations, sort of chained together. If we then have this:

```
speeds_copy = speeds
```

we actually have two versions of the same list. Both point to the same first element, which points to the second. So changing one list will change the other too. See:

```
>>> speeds = [1, 2, 3, 4,]
>>> speeds_copy = speeds
>>> speeds_copy.append(5)
>>> speeds
[1, 2, 3, 4, 5]
```

We added an element to the second list, but is also shows up in the first!

This is sometimes what you want, but admitedly not often. The trick is to use the `copy` method, which actually does create a copy (a "shallow" copy to give its proper name). This now works more intuitively:

```
>>> speeds = [1, 2, 3, 4,]
>>> speeds_copy = speeds.copy()
>>> speeds_copy.append(5)
>>> speeds
```

(continues on next page)

[81] This is in fact pretty much what `range` did in older versions of Python. Now, for efficiency it does something slightly different but we can still think of it as generating a list.

[82] Think of it this way because this is exactly how it works.

(continued from previous page)

```
[1, 2, 3, 4]
>>> speeds_copy
[1, 2, 3, 4, 5]
```

Always use this method if you need a copy.

Note: As you will see in the docs, this is often written as creating a slice of the complete list, like so:

```
>>> speeds_copy = speeds[:]
```

This is so commonly used, it's fine.

10.1.3 Leaving Lists

Lists are a fine general-purpose collection. As we've noted, you only really needs lists. Some older languages do only provide one collection type - usually called an *array*, but most modern languages provide some more. Three of those that Python provides are widely used, and we'll go on to them now.

But keep in mind that you can do anything these types can do with a list (except that for a Dicionary you need two lists). For that reason we'll focus on the differences, and the specific use cases where tuples, sets, and dictionaries come in handy.

10.2 Trying Tuples

At first sight, tuples seem very similar to lists, and you may wonder what they are for. So let's start with the two most important differences:

- Tuples are *immutable*, which means that once created, a tuple does not change.
- Tuples are usually *heterogeneous*, that is a tuple contains data of a range of different data types.

This contrasts with a list which, as we have seen, is *mutable* and usually *homogeneous*. .. hint:

```
If you have studied databases, you can think of a tuple as a row in a
↪database table. And the database table could be represented by a
↪list of tuples.
```

We have, in fact, used tuples before, without knowing it. If you have a function that returns more than one value, it is in fact returning a tuple[83]. That code looked something like this:

[83] So, in effect, the function is returning *one* value, which happens to be a tuple.

```
def find_string_and_number():

    a_string = ...
    a_number = ...

    return a_string, a_number
```

So we returned two values, separated with a comma. That is actually a *tuple*.

A tuple is represented just like that, a number of values separated by commas. So this creates a tuple:

```
>>> details = 'Robin', 12, False
>>> type(details)
<class 'tuple'>
>>> details
('Robin', 12, False)
```

Notice that when the interpreter displays the value of a tuple it adds parentheses. So it is usual to add these in anyway when a tuple is created (and this also allows tuples to contain tuples). As you probably expect, the elements inside the tuple can be accessed by an index number (just like lists) and slices work too:

```
>>> details[2]
False
>>> details[-1]
False
>>> details[:-1]
('Robin', 12)
```

But, as they are immutable, it is not possible to assign values to the elements once a tuple is created:

```
>>> details[2] = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Some final details. To create an empty tuple use empty round parentheses:

```
>>> empty_tuple = ()
```

To create a tuple with a single element, you *must* add a trailing comma, like this[84]. But the brackets are optional:

```
>>> new_tuple = 'Robin',
```

And the same will add an element to the end of a tuple:

[84] This is one reason why adding the trailing comma in lists is good form. Keeps it less confusing.

```
>>> details += ('Brave',)
>>> details
('Robin', 12, False, 'Brave')
```

But if you find yourself adding to the end of a tuple that could well be the problem's way of telling you that you should be using a list!

So, really tuples are best thought of as a handy way to handle several related data items as a single unit. Remember the two key ideas - tuples are *immutable* and usually *heterogeneous*.

10.3 Seeking Sets

Sets are probably the most obscure collection type, but they are worth mentioning because of one special property. The features of a set are:

- It is unordered.
- It can be heterogeneous.
- It does *not permit duplicate elements*.
- It supports all the usual operations associated with a mathematical set, such as intersection and difference.

The most commonly useful properly is the uniqueness. A set is created using curly parentheses:

```
>>> knights = {'Robin', 'Galahad', 'Bedevere',}
>>> type(knights)
<class 'set'>
```

The trailing comma is optional, as with tuples and lists.

Membership testing is possible:

```
>>> 'Robin' in knights
True
>>> 'Arthur' in knights
False
```

Items can be added to a set using the add method, but this has no effect if the item is already there.

```
>>> knights.add('Robin')
>>> knights
{'Galahad', 'Robin', 'Bedevere'}
>>> knights.add('Bors')
>>> knights
{'Galahad', 'Bors', 'Robin', 'Bedevere'}
```

Note: The code above also illustrates that order is not defined for a set.

The set operations can be useful if, say, we have two collections and want to know what items are in the one, but not the other, or are in both:

```
>>> knights = {'Robin', 'Galahad', 'Bedevere',}
>>> brave_knights = {'Galahad', 'Bors', 'Bedevere',}
>>> knights - brave_knights
{'Robin'}
>>> knights & brave_knights
{'Galahad', 'Bors', 'Bedevere'}
```

The two operations here are set difference (–) and intersection (&).

So this can be very useful if this sort of thing is common in your application.

Finally, a common example where sets can be very handy is where you have a list but you want to filter out duplicate items. The simple way to do this is to convert the list to a set, and then back again! Look:

```
>>> knights = ['Galahad', 'Bors', 'Robin', 'Robin', 'Bors', 'Lancelot
↪', 'Bors',]
>>> knights = list(set(knights))
>>> knights
['Galahad', 'Bors', 'Robin', 'Lancelot']
```

Neat.

10.4 Discovering Dictionaries

The final collection that's worthy of a mention is a dictionary. A dictionary is a *key-value* pair, sometimes called a map. As usual, most modern programming languages provide something like a dictionary, and many provide multiple subtle variations.

Note: There are many more collections, including specific types of dictionary, available in the standard library. We are just limiting ourselves to the built-in collection types here.

Assuming you have used a paper dictionary, you already have the idea of what a Python dictionary will do. In a paper dictionary you take a word (that's the *key*) and find the definition (that's the *value*). It's important to realise straight away that this doesn't work the other way around - you don't take a definition and look through the dictionary until you find the right word[85].

So when using a dictionary we have a *key* and some corresponding *values*. For example:

- In a phone book, contact names would be the key (a string), and the phone number would be the value (also a string[86]).
- In a login system, user names would be the key (string), and the password would be the value (an encrypted string).

[85] You can obviously code this in Python, but is is the stuff of nightmares to do.
[86] A string? Yes. Phone numbers usually have spaces in them and are very rarely used in arithmetic!

- In an exam system, a student id would be the key (a string), and their results would be the value (a list of integers).

Specifically, a Python dictionary:

- Is a collection of key-value pairs.
- Has unique keys.
- Does not (reliably) maintain any concept of order. (The order will most likely be the order in which items are added, but it would be a brave programmer who decided to rely on this!)

Given this, a dictionary is clearly going to be useful where the data in a problem neatly fits the key-value idea.

Note: The question of "order" in a dictionary is a tricky one. The dictionaries we are discussing here do not have order, although you can write cunning code to sort them. But if you have a problem that really needs a dictionary with order, you can reach for one of the other available types and `import` them - `OrderedDict` for example.

Let's see a dictionary in use. An empty one is created using curly parentheses:

```
>>> scores = {}
```

Important: Curly brackets like this were also used with sets, above. To create an empty set, the code is:

```
>>> empty_set = set()
```

Presumably they ran out of brackets.

Then it is simply a case of adding values. We specify the key, and the corresponding value:

```
>>> scores = {}
>>> type(scores)
<class 'dict'>
>>> scores['robin'] = 23
>>> scores['bors'] = 76
>>> scores['galahad'] = 40
>>> scores
{'robin': 23, 'bors': 76, 'galahad': 40}
```

And to extract values, just use the key:

```
>>> scores['robin']
23
```

The same works to change a value. Obviously changing a key doesn't really make sense, and has to be done by deleting and inserting a new entry.

```
>>> scores['robin'] = 45
>>> scores['robin']
45
>>> del (scores['bors'])
>>> scores['sir bors'] = 76
>>> scores
{'robin': 45, 'galahad': 40, 'sir bors': 76}
```

Two methods are commonly used to work with dictionaries. They are `keys` and `values`, which provide lists of what they say:

```
>>> scores.keys()
dict_keys(['robin', 'galahad', 'sir bors'])
>>> scores.values()
dict_values([45, 40, 76])
```

And also, `items` gives a list of tuples representing the dictionary:

```
>>> scores.items()
dict_items([('robin', 45), ('galahad', 40), ('sir bors', 76)])
```

To finish with an example of using a dictionary, let's consider the problem of finding the key value in the above dictionary that has the highest value. Here's a shorthand for creating a dictionary:

```
>>> scores = dict(robin = 45, galahad = 40, bors = 76, bedevere = 90)
>>> scores
{'robin': 45, 'galahad': 40, 'bors': 76, 'bedevere': 90}
```

How to find the key with the highest value? We need to use the handy methods that let us get at the insides of the dictionary. We can find the highest value easily:

```
>>> high_score = max(scores.values())
>>> high_score
90
```

We have no way to find a key from just the value, and anyway the value might correspond to more than one key. So the trick is to use `items` and loop across the dictionary elements. This looks a bit strange, but once you've seen it once ...

```
>>> for name, score in scores.items():
...     if score == high_score:
...         print (name)
bedevere
```

The tricky thing about using a dictionary can simply be realising that it is the right tool for your problem!

10.5 Takeaways

This chapter has introduced four of the most common collection data types. Using these is fundamental towriting DRY programs that will work in a range of situations. When picking a collection type it is worth remembering:

- You can do basically anything with lists. Even key-value pairs can be implemented with lists.
- Some collections maintain order, and can be sorted, some do not.
- Some allow duplicate values, some do not.
- Some are well suited for heterogeneous data, others work better with homogeneous data.
- Some operations, like iterating over a sequence, or testing membership with `in` are available for more than one collection. As usual, it is the ones where the operation makes sense!

Many programs involve structures built up of several collections - lists of lists, or dictionaries where the value is a tuple. The trick to arriving at an efficient solution can often be to design the right *data structures*. Applications handling huge amounts of data often require consideration of the best structures to allow for efficient searching too, but that is not likely to be your problem for a while!

As usual, this chapter introduced the basic ideas. Full details are in the docs, and in many online tutorials.

CHAPTER

ELEVEN

FUN WITH FILES

Many pages ago we noted that all programs to basically the same thing. They take some data, they do some processing, and then they write out the changed data. But up to now all our programs have processed data that we have had a user enter from a keyboard. This is sometimes enough, but only works for very small amounts of data, or simple answers to prompts. In general we need to be able to handle data that is read from files.

This will mean that programs can handle a lot more data, but so long as we make sure we are writing DRY programs, the programs themselves will not get much longer or more complex. Some other issues will emergem, though, such as:

- A program may try to access a file that does not exist.
- A program may try to access a file that does exist, but for which there are no permissions (if it is owned by a different user, or requires some admin privilege, say).
- A program may be able to find and open a file, which then turns out to be in an unexpected format.
- A program may try to write a file to a location that requires permissions which are not available. Or which would overwrite an ecisting file.

This list looks a bit daunting, but the simple thing to remember is that any of these error cases will generate an exception. So the only real issue is identifying and trapping these in the same ways as we have done previously. This means that the three new programming ideas needed to work with files are:

1. How to find if a file exists.
2. How to read data from a file into some convenient data structure, like a string or list.
3. How to write data back to a file.

Important: Programs should, as far as is possible, be *platform-independent*. They should run on any operating system. OSs differ in how they name files, and in how folders are added into file names - notably Windows uses \ to separate folders in a hierarchy but Unix-like systems use /.

We will get round these issues here by just looking for files in the same folder (directory) as the program. There are plenty of useful functions in modules like `os` and `sys` for handling file names so as to make sure that code is portable.

The actual processing of the data should use the same ideas as we have seen before. So let's work through the basics of working with a file.

11.1 Finding Files

This is a good opportunity to revise the two different approaches to checking for error conditions. The issue we need to overcome is that we are going to create a program that uses a file; we need to check whether that files exists, and is available for us to read. The two approaches are:

- To examine the files available, generate a list somehow, and see if the file we need is in that list. And then proceed if it looks to be there. *This is Look Before You Leap - LBYL.*
- Process the file regardless, and deal with any errors that arise if the file cannot be found. *This is EAFP: Easier to Ask Forgiveness than Permission.*

Remember that in Python, we prefer the second (EAFP), even though the first - LBYL - might seem to be the obvious way to go about it!

In this simple example, we will create a program that checks whether a file exists and is accessible in the current folder (that is, the same folder as contains the program). The program on its own won't be very useful, but it will illustrate some important points.

Following EAFP, the strategy will be simply to `open` the file we are interested in, and to see whether or not any errors occur when we do! In this simple example we'll have the user enter the file name, and we won't bother with any validation on that. As before, we could look up what error will happen if the file cannot be opened, or we could just try it and see. The command to open a file in a Python program is just:

```
>>> open('spam.txt')
```

Trying that with a file that we know *does not* exist will show what the exception we are looking for will be.

```
>>> open('spam.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'spam.txt'
```

To be safe we should probably check what happens if the file does exist. On Linux (or Mac), the `touch` command creates an empty file, so:

```
$ touch eggs.txt
$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> open('eggs.txt')
<_io.TextIOWrapper name='eggs.txt' mode='r' encoding='UTF-8'>
```

The output is maybe a little mysterious for now, but it is definitely not an error.

So let's craft a program. First, we assume that the file *does* exist (and that the program will crash if we try to open a file that does not exist):

Listing 1: `file_exists.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    file_name = input('Enter the name of the file we seek: ')
    open(file_name)
    print('The file exists!')
```

It is obviously much neater to trap that exception thrown if the file cannot be opened. We know from the experiment above that it is a `FileNotFoundError`, so the code is easy.

This simple example gives us the *boilerplate* code that will be needed every time we come to access a file. Remember they key idea, though: we assume that the file exists, and that all will be well, and pick up the pieces if this turns out not to be the case.

So, there are two, or arguably three things we need to be able to do with a file once we have assured ourselves that it can be opened. The first is that our program will *read* the contents. The second and third are very similar - the program might need to *write* a new version of the file, or it might *append* to the existing content.

Important: In our spirit of keeping things simple, we are going to just deal with *text files* here. You can think of this as meaning files of data that make sense if you display them on the screen. Python can also handle *binary files* - such as MP3s or images - in a similar way to that described below. But remember that if you have a program that needs to open, say, music files, there is probably something in PyPi that will do a lot of the hard work for you.

Many programs obviously read a file, do something with the data, and then write the results. So we start with reading.

11.2 Reading Files

Reading from a file is easy. First, the file is opened (along with the checks above), and then there are methods (functions) that can read the contents. The command to open a file is, ah, `open`, as we have seen, and for reading we also specify the mode. So that the file can be referenced, the result of the `open` command is saved in a variable, which most programmers would call a *file pointer* or maybe *file reference*. So opening a file called `spam.txt` to read from it looks like this:

```
>>> f = open('spam.txt', 'r')
```

Reading (denoted by `r`) is the default, so this can be left out, but it is good form to keep it in. There are then three usual options (remember that we are limiting ourselves to text files here). Assuming the file is opened as above, with a file pointer `f`:

`f.read()`
: Will read the whole file into a string.

`f.readline()`
: Will read the next line of the file into a string.

`f.readlines()`
: Will read the whole file into a list of strings.

Once the file contents are in these variables, they can be processed using all the Python we have seen before.

Let's do a simple example.

Counting Lines

Write a program that takes the name of a text file, and prints the number of lines in the file.

This program is straightforward when we remember that we can read the file into a list, which each element being one line, and that we can easily find the length of a list. A first attempt, where we assume the file exists could be:

Listing 2: `word_count.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    file_name = input('Enter the name of the file: ')

    f = open(file_name)
    lines = len(f.readlines())

    print('Lines in the file:', lines)
```

Once we have finished with a file, it is good practice to explicitly `close` it. This is not strictly needed (it will be closed when the program exits), but it is tidy, and it indicates clearly that the program has finished with the file.

So:

Listing 3: `word_count.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    file_name = input('Enter the name of the file: ')

    f = open(file_name)
    lines = len(f.readlines())

    f.close()

    print('Lines in the file:', lines)
```

And the program can be completed by adding in the usual code to deal with a possibly missing file.

Listing 4: `word_count.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    file_name = input('Enter the name of the file: ')

    try:
        f = open(file_name)
        lines = len(f.readlines())

        f.close()

        print('Lines in the file:', lines)

    except FileNotFoundError:
        print('Error: File cannot be found.')
```

That's done. There is probably one more small detail to mention.

11.2.1 Newlines

If you look at the data that is read from a file, you will see that every string is terminated with `\n`. This symbol represents the "new line" character at the end of each line. Obviously we don't see that character, but it is how the file marks the end of lines, and therefore why there is a line break when we view the file in an IDE, or Notepad.

Note: As well as `\n` you may also see `\t`, which represents a TAB character. There are others (they are called *escape sequences*, but these two are the most common).

So, suppose we want to count the number of characters in a file, as well as the lines. In that case we probably wouldn't count the end of line characters. Such a program is actually a very common "recipe", where we do something with every line of file. Example coming up!

11.2.2 Line-by-line

Processing every line in a file is easy when we remember that the `readlines` method gives a list. We can just loop over that list with a good old `for`. Let's use this technique to extend our line-counting program to count characters.

All we need to do is set a running counter to zero, and then add the length of each line in turn. To get the answer that most would consider correct, we also need to knock one off each line's length for the newline character[87].

[87] Alternatively, and quite cunningly, we could just take the complete length including the new line characters, and then at the end, subtract the total number of lines.

We also need to do a little *refactor* here so that the list containing the file contents is stored in a variable. Deep breath …

Listing 5: `word_count.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    file_name = input('Enter the name of the file: ')

    try:
        f = open(file_name)

        lines = f.readlines()

        f.close()

        characters = 0
        for line in lines:
            characters += len(line) - 1

        print('Lines in the file:     ', len(lines))
        print('Characters in the file:', characters)

    except FileNotFoundError:
        print('Error: File cannot be found.')
```

This structure, where a file is processed line-by-line is very common, so this code is worth studying.

Now we can read files, we can try writing!

11.3 Writing Files

It should be no surprise that writing to a file involves much the same code as reading from a file. First, the file must be opened, then data can be written, and finally it should be closed. Closing a file after it has been written is probably more important that after reading because this should *flush* any data that the operating system might be holding in buffers[88].

Opening the file for writing is the same code as for reading, except the mode is different. Two write to a file:

```
>>> f = open('spam.txt', 'w')
```

or to append to a file (that is, add data to the end):

[88] Output to a file on disk is slow, so typically the system will hold ("buffer") data in some handy memory and then write it to disk when there are system resources available. Closing a file forces anything in a buffer to be written.

```
>>> f = open('spam.txt', 'a')
```

But let's consider what could go wrong here.

11.3.1 Writing Exceptions

In the case of writing a `FileNotFoundError` will only happen if the file name includes a folder, and that folder does not exist.

There is also the possibility of a `PermissionError` if the location *does exist* but the user running the program does not have permission to write a file there.

Trying to write to a file that already exists is tricky, because sometimes this would be an error, but usually it isn't. Writing to a log file, for example, involves adding data to an existing file. So Python will be quite happy to open a file for writing, if that file exists.

Tip: If this is an issue, the `os` module contains handy functions to determine if a file exists.

Or, you could just use the code for opening a file to write a quick function to check. Here's a quick hack:

```
def file_exists(filename):
    try:
        f = open(filename, 'r')
        f.close()
        return True
    except FileNotFoundError:
        return False
```

Although, of course, opening a file to read it does not mean you can write to it. It just means that it exists.

So, when opening a file for writing, we should catch the exceptions that can happen (but these will probably only happen if the filename also includes folder names).

11.3.2 Writing Data

There is only one function to write to a file, and it is imaginatively named `write`. You can almost think of it as the same as `print`, except that output is sent to a file.

There really isn't much to it, but an example will help.

Shopping List

Write a program that prompts the user to enter items they plan to buy, and stores this in a file called `shopping.txt`.

We will extend this program a little in a moment, but first we will assume that it runs just once, so the file does not initially exist. So all we need to do is prompt the user to enter items, and give them some way to indicate that are done.

Listing 6: `shopping.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    shopping = open('shopping.txt', 'w')

    while True:
        new_item = input('What to buy? (END to exit): ')

        if new_item == 'END':
            break

        shopping.write(new_item + '\n')

    shopping.close()
```

The `write` method does not add a newline character so see that we have to do that ourselves. This is sort of the opposite to wanting to ignore the newline when reading a file.

If the file does not exist, it will be created. If it does exist, as things stand, any new data will overwrite what was already there. This might be what is wanted, or might not.

The third mode for writing to files, *append*, takes care of the situation where we want to keep the existing contents. This is useful for a log file or, here, a shopping list:

Listing 7: `shopping.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':

    shopping = open('shopping.txt', 'a')

    while True:
        new_item = input('What to buy? (END to exit): ')

        if new_item == 'END':
            break

        shopping.write(new_item + '\n')

    shopping.close()
```

A tiny change! Now, any new items will be added to the end of the file. If you try to append to a file that does not exist, the file will be created, just as with the `w` mode.

Finally, let's finish the program by trapping the exceptions. The two likely ones were `PermissionError` and `FileNotFoundError`. Since the error in either case would be the

same, this is a good chance to show how to catch two exceptions at once! Here is the code:

Listing 8: shopping.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    try:
        shopping = open('shopping.txt', 'a')

        while True:
            new_item = input('What to buy? (END to exit): ')

            if new_item == 'END':
                break

            shopping.write(new_item + '\n')

        shopping.close()

    except (FileNotFoundError, PermissionError,):
        print('Cannot open file to write!')
```

That comma at the end gives it away - it's a tuple of the exceptions that could be generated.

Since there is space, we'll do one last refactor here. This is nothing to do with files as such, but it does crop up a lot when using them. The problem is that (except in very unlikely conditions[89]) the only place an exception will happen here would be when opening the file. And as we have it the code the handle that has got a long way from the place the error will happen.

So, let's trap the exception in a slightly different way. This is really just for neatness.

Listing 9: shopping.py

```
#!/usr/bin/env python3

if __name__ == '__main__':

    try:
        shopping = open('shopping.txt', 'a')

    except (FileNotFoundError, PermissionError,):
        print('Cannot open file to write!')
```

(continues on next page)

[89] This is called a *race condition*. Basically, our program determines that it can write to a file. But before it comes to do this, some other program on the system does something to the same file, so it can no longer be written. Try running the shopping list program in two windows at the same time. What should happen? What does?

(continued from previous page)

```
    else:
        while True:
            new_item = input('What to buy? (END to exit): ')

            if new_item == 'END':
                break

            shopping.write(new_item + '\n')

        shopping.close()
```

The new idea here is that `else`. It can be thought of as meaning "carry on here if there is no exception". If there is an exception, the `else` is never executed.

Have a look at both examples, and see which you think gives the code that is easir to follow. That's the one to use!

11.4 Takeaways

Most programs use files. Once a file has been opened, there are methods for reading and writing data, and which work depends on the mode in which the file was opened.

Things to remember:

- Opening a file involves specifying the name, and the mode.
- Exceptions will show whether a file opened for read exists, or whether a file opened for write can be written to.
- Reading a file does not affect it. Writing a file can create it, or overwrite it.
- If the contents of a file are to be added to, the mode to use is *append*.

And while we were here, we showed a slightly different way to work with exceptions.

Files are often used as *command line arguments*. How to do that is in the next chapter, along with a few other things that will make our lives easier.

CHAPTER

TWELVE

THOSE LITTLE DETAILS

As we know, one of Python's design features is that should be one, and preferably only one, way of doing something. Right at the start if this book we set out that we would explain and, and preferably only one, way of doing something in Python. That has probably worked, until now.

Programming languages change and evolve, and Python is no exception. It's been around for many years now, and new ideas and feedback from the community lead to changes in the language. This is one of the joys of working with open source languages and projects - everyone can have an input into how somnething develops.

This section includes some features that have been added into Python over the years. Along with these are a few details that we passed over so as to keep things simple. The sections below are in no special order, and will probably be added to over time!

12.1 Ternary

There has been much made of the benefits of producing DRY code in this book. A useful feature in this quest is to use Python's *ternary* version of the `if` statement. As with many things in this chapter, there is no need to use this, and therefore no need to know about it, but using it can produce much neater, and "DRYer" code.

A conditional (`if`) statement chooses between any number of possibilities. A ternary can be used when there are two. It is really just a shorthand, but it can look a little odd at first. It can be read as something like "do this if something is true, otherwise do that". An example:

```
mark = 50
result = 'Pass' if mark >= 40 else 'Fail'
```

That should be obvious from just reading it, which is one of the benefits of using this. Compare with the functionally identical:

```
mark = 50
if mark >= 40:
    result = 'Pass'
else:
    result = 'Fail'
```

That really is all there is to it. A common use case is in a f-string - see below!

12.2 F-Strings

It is more than likely that you have found it difficult (or at least fiddly) to generate neat output from some of your programs. This has not been helped by the way that we have always used the `print` statement, along with a collection of arguments, and optionally the `sep` argument to add or remove spacing. There is, not surprisingly a much neater way to do all this, and to provide neatly formatted strings.

Important: The "best" way to achieve this has changed in Python over the years, so the usual Google and StackOverflow searches may well lead to different solutions. These are fine, but also fiddly.

Currently, the best way for formatting strings is formatted strings, or *f-strings*. The full reasons for their introduction were debated backn in 2016, and are documented in PEP 498[90]. The basic idea is to *interpolate* (that is, include) code inside strings. This is not as tricky as it sounds. A simple example:

```
>>> name = 'Robin'
>>> print(f'Greetings, Sir {name}!)
Greetings, Sir Robin!
```

So the idea is that whatever is inside the curly brackets is executed as Python code, and the result is printed. Here the contents of the curly brackets is just the name of a variable, so the value is printed.

There is nothing here that couldn't be done with conditional statements, string concatenation (adding), and so on, but this is *much neater*.

Here is an example of the brackets containing code. Suppose we have a test mark, the pass is 40, and we want to print the result. Traditionally we would write:

```
if mark >= 40:
    print('Your mark was', mark, '. You have passed!', sep='')
else:
    print('Your mark was', mark, '. You have failed', sep='')
```

This is OK, but is fiddly to get the spacing right. Compare with the f-string version, that makes use of a ternary:

```
print(f'Your mark is {mark}. You have {"passed" if mark >= 40 else
↪"failed"}!')
```

The code is now a one-liner. It is easy to read, and the message is not duplicated, making it a DRY solution.

Hint: In code like this, remember that we need to use double-quotes inside single-quotes, else it will not be obvious where the first string ends. (You could use single inside double, it matters not, as long as you are consistent.)

---

[90] https://peps.python.org/pep-0498/

It is the magic f before the string that is making this happen.

There is more. These f-strings also allow the output to be formatted. The full details are in the docs[91] (but you may be better off Googling for some examples). The sort of formatting available includes padding a string with spaces (before, after, or to centre it), displaying a floating-point value to a number of decimal places, and more.

Some examples of formatting a string for output, using *** to show what's going on:

```
>>> name = 'Robin'
>>> print(f'***{name:<20}***')
***Robin               ***
>>> print(f'***{name:>20}***')
***               Robin***
>>> print(f'***{name:^20}***')
***       Robin        ***
```

This is handy for a neat table of results.

And an example of controlling the number of decimal places in a result:

```
>>> eggs_ratio = 31/7
>>> 31/7
4.428571428571429
>>> print(f'{eggs_ratio:.2f}')
4.43
```

Note that the value is rounded, not just truncated.

There are many more options, but these two are really the most common. Check the docs for more!

Finally, a quick note on a common programming task:

Binary

Write a program that accepts an integer as input and displays the equivalent in binary.

Now, to do this, you might start thinking about loops, calculating powers of two, using remainders and modulus, and so on. And you would come up with a program that was maybe 20 lines long, and you would be very proud.

But here is that, with the conversion done in one line, courtesy of an f-string ...

Listing 1: `binary.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
    decimal_number = int(input('Enter a number: '))
    print(f'In binary, {decimal_number} is {decimal_number:b}.')
```

[91] https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals

Sorry.

12.3 Command-Line Arguments

Early on in this book, we introduced the *command line*. While many applications use graphical interfaces these days, there are still plenty of times where we need to just type a command into a terminal. For one thing, this is probably how many of the programs that we write will be run. You can run the program from inside your IDE, but you can't assume that your use will have the same IDE, or even any Pyton tool at all. So we fall back on the good old command-line.

Tip: You may well find on Windows that double-clicking a Python program will run it, a terminal window will appear, and then vanish before you can see the results. Google will lead you to ways to preserve the output, but firing up a terminal and running it there really is the best way.

Running a program at the command line is a simple case of starting the Python interpreter with the name of the file containing the program (programs are just plain text files, remember). So on a Linux or Mac system:

```
$ python3 my_prog.py
```

does the job.

Windows is, as usual, a little more complicated, as typing `python` has a habit of opening the Windows Store. The simplest fix is to use:

```
C:\> py my_prog.py
```

making sure you are running in the same folder as contains the program. Better fixes are a Google away.

When running a program like this, it is often useful to have some input on the command line, along with the name of the program file. As an example, here is a simple program that counts the lines in a file (and uses an f-string!):

Listing 2: `wc.py`

```
#!/usr/bin/env python3

if __name__ == '__main__':
    file_name = input('Enter the file name: ')

    f = open(file_name, 'r')
    lines = len(f.readlines())

    print(f'Line Count: {lines}.')
```

This is fine, and it would work, but wouldn't it be neater if the user just typed the name of the file they want to use as input *after* the program name? So instead of:

```
$ python3 wc.py
Enter the file name: text_file.txt
```

the user could just:

```
$ python3 wc.py text_file.txt
```

This is obviously possible, and is a case where we have the program capture *command line arguments*. It works like this:

1. Import the `sys` module.
2. Then you have a list called `sys.argv` which contains everything from the command line. The first element is the name of the program, and then the remainder is anything that was typed after it.

So for this program is we `import` the `sys` module, and then run the program as so:

```
$ python3 wc.py text_file.txt
```

The list `sys.argv` will contain `wc.py` at index 0, and `text_file.txt` at index 1. With this knowledge, we can change the program:

Listing 3: `wc.py`

```
#!/usr/bin/env python3


import sys


if __name__ == '__main__':
    file_name = sys.argv[1]

    f = open(file_name, 'r')
    lines = len(f.readlines())

    print(f'Line Count: {lines}.')
```

and it will work as expected.

Using the command line like this usually introduces the possibility of errors, often when the user misses off a required argument, or when the argument is invalid. So there is some common code that often gets added in. In this case, missing off the argument would give an `IndexError` when the program tries to access the argument. There could also be a `FileNotFoundError` if, ah, the file cannot be found. So a complete version of the program, with error-checking, would be:

Listing 4: `wc.py`

```
#!/usr/bin/env python3


```

(continues on next page)

(continued from previous page)

```
import sys


if __name__ == '__main__':

    try:
        file_name = sys.argv[1]

        f = open(file_name, 'r')
        lines = len(f.readlines())

        print(f'Line Count: {lines}.')

    except IndexError:
        print(f'{sys.argv[0]}: Missing required argument.')
    except FileNotFoundError:
        print(f'{sys.argv[0]}: Cannot open "{sys.argv[1]}"')
```

Remember that `sys.argv[0]` contains the name of the program, so here we are making sure that the user knows what is generating the error.

12.4 None

Any book on programming will at some point provide a list of the built-in *primitive* data types available. As we know, the list varies between languages but usually includes:

- Whole numbers, called integers.
- Numbers with a fractional part, called floating-point numbers.
- Strings, with a single character string possibly being a special case.

Modern languages usually also include a Boolean type, while older languages might just use integers for that. Some languages offer more specific types, for example integers that cannot be negative, or integers that occupy a specific amount of memory.

Python keeps it simple, so way back, we said that these were the four types in Python:

- `int`, an integer.
- `float`, a number with a fractional part.
- `str`, a string, which can have any number of characters,
- `bool`, a Boolean.

This was not strictly true. There is a fifth type. It's called `None` or more accurately `NoneType`[93].

The need for this arises from a particular problem. Python determines a variable's type from the value it is given when it is created, but *what happens if we want a variable*

[93] This is *still* not strictly true. `None` isn't really a type, it's an object, and there is only one of them. See the docs if you really need to know!

that has no initial value? Such a variable has no value, so no type, so it can be given `NoneType`.

It can be assigned deliberately, like this:

```
>>> spam = None
>>> type(spam)
<class 'NoneType'>
```

And we can test whether the variable currently has an interesting value:

```
>>> not spam
True
```

So at the moment `spam` has no useful value. Let's give it one:

```
>>> spam = 1
>>> not(spam)
False
```

This all seems a bit abstract, so let's have an example where this might be useful. Suppose we have a function that finds a value in a list. It takes two parameters, the list and a number to search for, and returns where the number is in the list. There is a built-in function called `index` that will do most of the heavy lifting, so we get something like:

```
def find_number(list_of_numbers, number):
    return list_of_numbers.index(number)
```

This is fine and we could use it like this to look for a number, say 12:

```
position = find_number(all_numbers, 12)
```

But what happens if the number cannot be found? The `index` function will throw an exception. This could be handled in the program using the function, but it *can be neater* to return `None` to say the value was not found. The function becomes:

```
def find_number(list_of_numbers, number):
    try:
        return list_of_numbers.index(number)
    except ValueError:
        return None
```

This is neater because this function now *always returns a value*, so there is no need to worry about exceptions when using it. So the code using the function can simply be:

```
position = find_number(all_numbers, 12)
if position:
    print('Value Found')
else:
    print('Value not found')
```

This is a neatness, but it does often improve the readability of code.

Note: Somewhat related to this is a common structure in Python where we need to check if, say, a list is empty, or a string variable contains no characters. The Boolean `not` comes in handy:

```
>>> s = ''
>>> not s
True
>>> l = []
>>> not l
True
```

This comes in useful when we want a user to enter some values, while giving them a way to indicate that they are finished. A while back we had a program where a user entered some marks, and we calculated the average. It looked like this:

Listing 5: `marks.py`

```
#!/usr/bin/env python3


from statistics import mean

if __name__ == '__main__':

    marks = []

    while True:
        next_mark = int(input('Enter the next mark (-1 to end): '))
        if next_mark == -1:
            break
        else:
            marks.append(next_mark)

    print('Average Mark:', mean(marks))
```

And we noted at the time that having them enter `-1` to show they were done was not the greated piece of user experience ever. A couple of small changes will allow the user to just press `Enter` to show they are done. We use the fact that this gives an empty string, and that `not` applied to an empty string gives `True`. The only other change is that we need to move the `int` conversion so that we can test the possibly-emty string. The improved version is:

Listing 6: `marks.py`

```
#!/usr/bin/env python3


from statistics import mean

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    marks = []

    while True:
        next_mark = input('Enter the next mark ("Enter" to end): ')
        if not next_mark:
            break
        else:
            marks.append(int(next_mark))

    print('Average Mark:', mean(marks))
```

Much better UX!

12.5 Passing

This might seem a little odd, but there is a statement in Python that does nothing. Ever. Nothing, nada, zilch. It's needed because of Python's reliance on indentation to show what statements are in which block. Look at this line of code:

```
if number_entered == 1:
```

The syntax of Python *requires* that there is a statement on the following line. If there isn't, that is an error, and the program will fail to run. In some other languages you could just use an empty pair of brackets or some such to show that there's nothing there, but the indentation in Python means that this won't work.

So there is a need for a statement that does nothing! This might be because there is nothing to do aside from declaring something (see *Custom Exceptions*), oe because the programmer needs a placeholder, or because explicitly saying nothing needs to be done improves the readability of the code.

All of these are the job of the `pass` statement. So in the code above, we could have this, which is valid Python:

```
if number_entered == 1:
    pass
```

Another common use is when writing functions. Quite often you need to write the function header, and want to work on the code that uses it. You will write the function body later. So you use `pass` as a placeholder:

```
def useful_function():
    pass
```

This satisfies the syntax, and stops your IDE generating errors. The code will also run, although obviously it will do nothing.

Finally, a less common use is when you explicitly want to say that nothing should happen. This is obviously irrelevant to Python, but could help someone reading the code.

For example, suppose some code wanted to ignore every value in a certain range. We could write this, reasoning that just to ignore the range would look odd:

```
if number_entered == 1:
    print('One')
elif number_entered == 6:
    print('Six')
else:
    pass
```

The score here is that we are *explicitly* ignoring other values.

Hint: If you use your IDE to create template code for functions, you may well find that it adds a `pass` statement in to make the code valid.

12.6 Custom Exceptions

Exceptions, and programming with them, are very important in Python. This is especially true if we adopt the preferred *easier to ask forgiveness than permission (EAFP)* approach to dealing with errors. We have written programs that have caught and dealt with exceptions, as well as programs that have generated their own.

Up to know we have been content to use the built-in exceptions. Usually it has been possible to find one where the name meets the facts of the case of what is going wrong. But these are by their very nature quite generic; `ValueError` tells us nothing except that a value is wrong, for example. It is often useful to be able to create our own exceptions, and to use those. So if there is a problem with a password, we could generate a `PasswordError`, for example.

Hint: There is a full list of the standard exceptions, as well as plenty of details on how to use them, in the Python Docs[92].

Defining a new exception uses *Classes*, which is a feature of Python we have been using all along, since everything is a class. Look:

```
>>> type(1)
<class 'int'>
```

Integers are a class of *objects*. So when we define a new exception we are going to add a new object to the class of exceptions. The code to do this is simple, and makes use the `pass` statement! To create an exception to indicate a password problem:

```
class PasswordError(Exception):
    pass
```

That's it! Assuming this is defined at the top of a program (or, better, in an imported module) we could have some code that generated a meaningful error. Something like:

[92] https://docs.python.org/3/library/exceptions.html#concrete-exceptions

```
if password != confirmation_password:
    raise PasswordError('Password mismatch')
```

That's all there is to it. Remember that the name of the exception provides a general idea of where the problem is, and the message includes more details.

12.7 List Comprehensions

Lists are a very powerful collection data type. In fact, to be honest, lists are the *only* collection type you really need to know. They tend to be used in similar ways in many programs, and often appear in similarly structured code. Typically there is a `for` loop, that does something to each item in a list, or adds values to a list depending on some condition. As an example, suppose we have a list of marks, and we want to build another list containing just the fails (less than 40, say). The code might be something along the lines of:

```
fail_marks = []

for mark in all_marks:
    if mark < 40:
        fail_marks.append(mark)
```

Or suppose we have a name like `Arthur James Wensleydale`, and want to extract the capital letters. These would represent the initials, and could be useful. We would code:

```
initials = ''

for letter in full_name:
    if letter.isupper():
        initials += letter
```

This looks quite different, but is the same structure. Both these code samples initialise a variable, and then add to it as they examine each element of something else in turn. There are no lists in the second example, but there is what Python calls an *iterable*, and that means they are basically the same thing.

These are cases where *list comprehensions* come in useful. As with some of the other topics in this chapter there is never a case where you **must** use these, but they can lead to neater code.

Important: Remember that good code values clarity over neat tricks. List comprehensions are close to being neat tricks, and can lead to temptation to try to create nifty one-liners.

Always look at your code with an eye on readability and clarity!

A list comprehension takes one list, and produces another, based on some condition. Rather than describe the syntax, here is an example that would create a list of all even integers less than 20:

```
>>> evens = [x for x in range(20) if x % 2 == 0]
>>> evens
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

We can read this as `evens` is a list including all values in `range(20)` where that value `% 2` is 0 (so it is even).

Compare this with that same code written out "long hand", and you'll see the point:

```
evens = []
for x in range(20):
    if x % 2 == 0:
        evens.append(x)
```

In fact, you can see the code for the comprehension in that sample.

That's all there is to it. We can rewrite our first example using a comprehension, so this:

```
fail_marks = []

for mark in all_marks:
    if mark < 40:
        fail_marks.append(mark)
```

becomes the substantially neater:

```
fail_marks= [mark for marks in all_marks if mark < 40]
```

But what of that string example, where we were looking through a name? The trick here is to build a list of the initial letters, and then to convert that list back to a string. Seems over complicated, but the code is rather neat. First build the list:

```
initials = [letter for letter in full_name if letter.isupper()]
```

And then we can use the `join` function, that takes a list and joins the elements together with the given separator. Here the separator is nothing:

```
initials = ''.join([letter for letter in full_name if letter.
↪isupper()])
```

Again, this is very neat.

Using list comprehensions like this is very common, and is seen as Pythonic. So although it might look like a neat trick it is safe to assume that any experienced programmer will understand. Try them in your next project!

12.8 Takeaways

More than any other chapter in this book, this one is probably not complete!

There is really only one takeaway here, and that is that a programmer never stops learning. It is doubtful that there is anyone, anywhere, who knows *all* of Python, and can use it well. Modern programming languages are constantly developing and improving, and any developer needs to set time aside to keep up to date. Languages that are driven forward by a community of users, like Python, have a process where anyone can contribute, and the end results are achieved by consensus.

Much the same applies to any language you may come to use in the future. Languages like Java and PHP have communities and continue to develop too. Even languages with stronger ties to particular companies, like C# as an example, develop, even if they are guided in a somewhat different way.

So, take away that there is plenty more to learn. Look at code written by others, check the many tutorials available online, and, if you want to, keep up to date with new developments in Python.

CHAPTER

THIRTEEN

THE END OF THE BOOK

This is the end.

Well, it could also be a beginning. There remains a massive skills shortage in the IT, especially in programming and development, so maybe for you it could be a beginning?

Remember that programming is a skill. And like all skills it is something that you develop over time. The first step is to master the basics, and then get to a place where you can learn more. Then you learn, improve, and learn some more. Every programmer, even those who been around for decades, is still learning.

Programming languages develop, and the ways in which we program develop. As I type this, the current version of Python is 3.11. One day there will be a Python 4. It will probably look rather like Python 3, but it will surely have new features that will improve the lives of Python programmers in the future.

Tools develop too. A modern IDE is there all the time, by your side, making suggestions and spotting errors. Even something that we now take for granted, like colour highlighting of code is quite new. Newer tools will embrace AI techniques, and who knows where that might lead?

This is one of those chapters here that mightn grow. But, at least at the start, let's keep it short with a few key ideas.

13.1 *Programming*, not Python

This book has been about Python. But it is really about programming. If you feel you can now "get by" in Python you can probably say the same about PHP, Java, and a bunch of other languages. You would need to do some study, but now you should know what you are looking for. Remember that every language has its idioms and ways of doing things - as when we have been *Pythonic* here - which are the things you have to learn.

When approaching a new language you should be looking to see what the structure is for a `while` and `for` loop. You should find out how `if` statements work, and how the language shows what is inside the loop. You should be checking the docs for a list of the provided data types. You will never have to learn programming from scratch again.

Never be afraid of Googling, and remember that StackOverflow really is your friend. *Every* developer uses both daily, and any that tells you otherwise is not telling the truth. Remember that the reasons experienced programmers seem to fix things easily is that they have made all the mistakes before, and they have seen the errors, and know the fixes. Learn from them, and be like them.

Possibly the most important thing to develop is your sense of *there must be a way to do this*. Before writing some new code, you need to have a sense of what the language you are using probably already provides. Or what you will be able to find in the equivalent of PyPi. A good deal of modern programming is putting together something new from stuff that already exists.

Python is currently one of the most popular languages out there. There is *a lot* of Python, and it is not going to go away any time soon. But what we have done here was *programming*.

13.2 Keep Up To Date

Closely related to the comments above, is the need for a developer to keep up to date with programming, and with tech in general. New programming languages come and go. As I type this, some languages - looking at you, Go[94] and Kotlin[95] - are rapidly gaining traction and have a lot of fans. It will be interesting to see how they fare. Other languages (we'll miss you, Ruby[96] seem to be fading away. Developers need to know what is going on, if for no other reason than this is where the jobs will be in the future!

Other techs also impact on how programmers work. Programmers need a working knowledge of Cloud, Containers, Virtualisation, Continuous Integration and Continuous Deployment, DevOps, and all the other nice things currently trendy in the IT world. They at least need to be able to nod sensibly when these are mentioned, even if they do then need to go check WikiPedia.

Docker[97], Kubernetes[98], and even Jenkins[99] are all things to know about.

13.3 Keep Sharp

Practice, practice, then practice some more.

Much programming practice comes on the job. Or you can get involved with Open Source projects, and develop skills that way.

One popular way to develop skills is through practice on short (but often fiendish) exercises called Code Kata. See, for example, the original CodeKate site[100]. These are a way of "limbering up" before taking on a new task, or can be a way of getting some friendly competition going! There are plenty of other options - CodeWars[101] has built a whole community and awards coloured (virtual) belts, and Coding Dojos[102] exist both online and in real life.

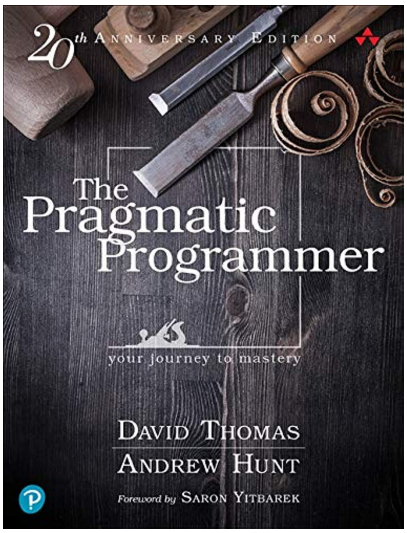
These all promote ways to learn and meet other developers. Improve your skills and enhance your career.

[94] https://go.dev
[95] https://kotlinlang.org
[96] https://www.ruby-lang.org/en/
[97] https://www.docker.com
[98] https://kubernetes.io
[99] https://www.jenkins.io
[100] http://codekata.com
[101] https://www.codewars.com
[102] https://codingdojo.org/dojo/

13.4 Important Reading

There are many books about programming. But most of them are books about programming languages. There are far fewer books about programming itself.

Warning: What follows is an opinion.



The best book to read about programming is *The Pragmatic Programmer* by Dave Thomas and Andy Hunt. It's been around for 20 years, and is a classic. It covers everything from career advice through to how to set up an IDE. The first three or four chapters of this book owe a lot to what's written here.

Like of lot of tech books, it's not cheap, so get it on the Birthday list.

We respectfully nod to this book.

And we should also nod to *Clean Code* by "Uncle" Bob Martin. Like the above, this book emphasises that programming is a *craft*. And it shows how it matters so much that code works *well*. It's just the same idea as a chair made of rough wood is something you can sit on, but a crafted piece of furniture is so much better.

13.5 AI and Programming

Finally, AI, and specifically *Generative* AI is going to change the whole programming business. AI can now write most programs if given the spec, so where does that leave programmers?

In a way we have been using AI to help us program for a long time. Without getting into a definition of AI, many things that our IDEs do have an "intelligent" feel to them. Your IDE "understands" your code, and sometimes makes suggestions about how to improve it. Is that AI? Or is it just following some rules it's been given?

AI will come to be used to do some of the "grunt work" of programming. That's the stuff that exists in every problem, but needs to be made specific. Tools to do this are already being introduced in both PyCharm and VS Code, so there is no point in ignoring them.

But, for the time being at least, programmers still have plenty of skills that AI does not. They could be summarised as *intuition* and *experience*.

AI cannot talk to users. It has never met users, and does not understand how they feel about the redesign of the interface in the system they use every day.

So, sure, AI will change the way programmer work, but it's not replacing us. Yet.

13.6 Takeaways

There is simply one message from the end of the book.

You never stop learning programming. You have now started. Well done! But don't think you will ever, ever, finish.

So, farewell!

GLOSSARY

Artificial Intelligence
: The theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages.

  And, to be honest, writing glossary entries.

Boolean
: A data type that has one of two possible values, usually denoted as true and false, but always analagous to "on" and "off". In Python, the two values are represented by the built-in constants `True` and `False`.

Camel Case
: A naming convention where words are written together without spaces, and each word starts with a capital letter. For example, `camelCase`. In Python, camel case is used for class names. Camel case should not be used for variable names or function names.

Cheese Shop
: A sketch from Monty Python's Flying Circus that features a customer trying to buy cheese from a cheese shop that has no cheese. The sketch is a running joke about the absurdity of the situation. The cheese shop has since become a metaphor for any situation where something is missing or unavailable. The Python Package Index (PyPi) is sometimes referred to as the Cheese Shop.

Constant
: A value that does not change. In Python, constants are usually defined before the main program and are written in all capital letters with underscores separating words. For example, `MAX_SIZE = 100`.

Compiled Language
: A programming language that requires its source code is converted into an executable form, using a compiler, before it can be run. For example, C.

DRY
: Stands for Don't Repeat Yourself. A software development principle that suggests you should not repeat the same code over and over. If you find yourself copying and pasting code, you should probably refactor it into a function or class. See also WET.

Duck Debugging
: A method of debugging code by explaining it to a rubber duck. The name comes from the book *The Pragmatic Programmer* by Andrew Hunt and Dave Thomas. The idea is that explaining the code, line by line, to the duck will help you find bugs. Other forms of rubber wildlife are also acceptable.

EAFP
: Stands for Easier to Ask Forgiveness than Permission. A programming approach that suggests you should just try to do something and catch an exception if it fails. For example, you should try to open a file and catch the resulting exception if the file does not exist. The opposite of LBYL.

Git
: A distributed version control system. It is used to track changes in source code during software development. It was created by Linus Torvalds in 2005 to manage the development of the Linux kernel.

GitHub
: A web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management functionality of Git as well as adding its own features.

Guido van Rossum
: The creator of Python. He started working on Python in the late 1980s, and it has been in continuous development ever since. Guido was made the BDFL (Benevolent Dictator For Life) of the Python Community, a title he held until he stepped down in 2018.

Indentation
: The spaces at the beginning of a line of code that indicate the block to which the line belongs. In Python, indentation is used to define the structure of the code. For example, all the lines of code that are part of a function should be indented by the same amount. Identation is usually four spaces, or multiples thereof.

Interactive Development Environment (IDE)
: A software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. A Python IDE will normally have features specific to Python, such as easy access to the Python interpreter.

Interpreted Language
: A programming language where statements are interpreted one at a time and executed as the program runs. For example, Python. Also Ruby, or Perl.

Interpreter
: A program that reads and executes code. Python is an interpreted language, so the Python interpreter reads and executes Python code.

LBYL
: Stands for Look Before You Leap. A programming approach that suggests you should check for required conditions before executing some code. For example, you should always check that a file exists before trying to open it. The opposite of EAFP.

Monty Python's Flying Circus
: A British sketch comedy television series featuring the comedy troupe Monty Python that originally aired on the BBC from 1969 to 1974. Later more famous for feature films including *Monty Python and the Holy Grail*, *Life of Brian*, and *The Meaning of Life*.

Ni
: A word used by the Knights Who Say Ni in Monty Python and the Holy Grail. The Knights Who Say Ni are a group of knights who demand a shrubbery from King Arthur. They are known for their frequent use of the word "Ni". The word "Ni" has since become a running joke in the Monty Python community.

PEP 8
: Stands for Python Enhancement Proposal 8. PEP 8 is a style guide for Python code.

It was written by Guido van Rossum, Barry Warsaw, and Nick Coghlan in 2001. PEP 8 covers topics such as indentation, line length, and function naming.

Pragmatic Programmer
A book by Andrew Hunt and Dave Thomas that was published in 1999. It is a guide to computer programming and software development that includes tips and tricks for programmers. Every programmer should read it.

Pythonic
A term used to describe code that follows the conventions of the Python language. Pythonic code is clean, readable, and concise. It is idiomatic Python code that takes advantage of the language's features and libraries.

PyCharm
A Python IDE developed by JetBrains. It is one of the most popular Python IDEs and is used by many professional Python developers. PyCharm has many features that make it easy to write, test, and debug Python code.

PyPi
The Python Package Index. It is a repository of software packages for the Python programming language. There are thousands of packages available on PyPi that can be installed using the `pip` package manager.

REPL
Stands for Read-Eval-Print Loop. A REPL is a simple interactive computer programming environment that takes single user inputs (single expressions), evaluates them, and displays the result to the user. The Python interpreter is a REPL.

Semantic Error
An error in a program that makes it do something other than what the programmer intended. Semantic errors are difficult to find because they do not cause the program to crash or produce an error message. Instead, they cause the program to produce incorrect results.

Shrubbery
A small to medium-sized woody plant. In Monty Python and the Holy Grail, the Knights Who Say Ni demand a shrubbery from King Arthur as a condition for passing through the forest. The knights are very particular about the type of shrubbery they want and are not satisfied with the first shrubbery King Arthur brings them.

Snake Case
A naming convention where words are written in lowercase and separated by underscores. For example, `snake_case`. In Python, snake case is used for variable names and function names.

Source Code
The human-readable version of a computer program. Source code is written in a programming language and must be translated into machine code before it can be executed. The translation is done by a compiler or interpreter. Source code is usually stored in plain text files.

Spam
A canned meat product made mainly from ham. It is also a running joke in the Monty Python sketch "Spam". In the sketch, a group of Vikings sing a chorus of

"Spam, Spam, Spam, Spam, lovely Spam! Wonderful Spam!" to drown out other conversation. The term "spam" has since been used to refer to unwanted email.

Syntax Error
An error in a program that occurs when the code does not follow the rules of the programming language. Syntax errors are usually easy to find because they cause the program to crash and produce an error message. Common syntax errors include missing parentheses, missing colons, and misspelled keywords.

VS Code
Visual Studio Code. A free source-code editor made by Microsoft for Windows, Linux, and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring.

WET
Stands for Write Everything Twice. A play on DRY, WET is a sarcastic way of saying that you should not try to reuse code. It is not a good idea to write everything twice, so you should always try to refactor code into reusable chunks such as functions. Also "We Enjoy Typing", or "Waste Everyone's Time". See also DRY.

COLOPHON

This book was produced using the Sphinx Documentation Generator[103]. This in turn uses a bunch of other Python packages that have all been made available by their developers. The theme is Sphinx {book theme}[104].

The main font in the PDF version is `Bitter`. Some text is in `Lato`. The code samples are in `Cascadia Code`.

The whole thing was cobbled together with PyCharm, with Git and GitHub keeping track of things. GitHub Copilot[105] has chipped in now and again.

The cover image shows a Reticulated Python and is taken from the 1911 Edition of the *Encyclopedia Britannica*, which observes as follows. Who knew?

> **PYTHON**, a genus of very large snakes of the family Boidae (see Snakes) inhabiting the tropical parts of Africa, Asia and Australia. They differ from the true boas (q.v.) with which they are often confounded by carrying a few teeth in the premaxilla, by the double row of subcaudal shields and by the possession of a pair of supraorbital bones. Most of them have pits in some of the upper and lower labial shields.
>
> Python reticulatus is the commonest species in Indo-China and the Malay Islands; four upper labial shields on either side are pitted. It is, next to the Anaconda, one of the largest of all snakes, some specimens being known which measured about 30 ft. in length. P. molurus, scarcely smaller, is the python or rock-snake of India and Ceylon. The African species are much smaller, up to 15 ft. in length, e.g. P. sebae of tropical and southern Africa and the beautiful P. regius of West Africa. P. spilotes is the "carpet-snake" of Australia and New Guinea. A small relative of pythons is Loxocemus bicolor of South Mexico, the only New World example.
>
> The giant pythons could no doubt overpower and kill by constriction almost any large mammal, since such snakes weigh hundredweights and possess terrific strength, but the width of their mouth—although marvellously distensible—has, of course, a limit, and this is probably drawn at the size of a goat. Before a python swallows such large prey, its bones are crushed and the body is mangled, into the shape of a sausage. The snake begins with the head, and a great quantity of saliva is discharged over the body of the victim as it is hooked into the throat by the alternately right and left forward motions of the distended well-toothed jaws. If for any reason a snake should disgorge its prey, this will be found smothered with slime. Hence the fable that they cover it with saliva before deglutition.
>
> Most pythons are rather ill-tempered, differing in this respect from the boas. They are chiefly arboreal, and prefer localities in the vicinity of water to which mammals and birds, their usual prey, resort. They move, climb and swim with equal facility. The female collects her eggs, sometimes as many as one hundred, into a heap, round which she coils herself, covering them so

[103] https://www.sphinx-doc.org/
[104] https://sphinx-book-theme.readthedocs.io/
[105] https://github.com/features/copilot

that her head rests in the centre on the top. In this position the snake remains without food throughout the whole period of incubation, or rather keeping guard, for about two months.

IMAGE CREDIT

The image of the VT100 terminal is taken from WikiMedia Commons[106] and is made available under a Creative Commons Licence[107].

[106] https://commons.wikimedia.org/wiki/File:DEC_VT100_terminal.jpg
[107] https://creativecommons.org/licenses/by/2.0/

CREDITS

Thanks are due to the following …

INDEX

T

U

V

W