Introduction to Program Construction 1

Learning to Program in Python

Dr. Seán Russell

September 2, 2021

Contents

Chapter 1

Introduction

This purpose of this book is to teach someone who has never programmed before to solve basic problems using the Python programming Language. There are no assumptions made about what you already know about programming. In fact, it is assumed that you pretty much don't know anything about how computers work.

When learning to program, you will need to write **many** programs. The more practice that you get the better you will be at the smaller details and the more time you can spend thinking about **how** to complete tasks using code. Eventually, you will not have to think about small details at all (or at least a lot less), you will know them so well that you can just complete these parts.

The process of programming can be **difficult**. But when considered as a set of activities that we perform together, we can see that it is made up of some activities that are easier than others. To write a program to do something we need to do the following;

- Understand **what** we are trying to do
- Understand **how** it can be done (without the computer)
- Explain to the computer how to do the task
- Correct any errors we have made (there will be many mistakes)

Learning to program means that we are learning to do all of these activities at the same time. The first 2 activities change with every new program we write, there will always be a different task to understand and a different way to complete it. We will get better at these through **practice** and using some techniques to help break programs into more manageable pieces.

Explaining to the computer is about telling the computer what to do in a language it can understand. This is very difficult at the beginning, but gets much easier the more we practice. We will learn how to tell the computer to remember information for us and perform calculations using that information. We will learn how to tell the computer to make decisions and repeat calculations until our task is finished.

Correcting errors can sometimes be the most frustrating of these activities, when there is a problem with our code the error that we get is often difficult to understand. To get better at this there are two things we need to do. First we must understand what our code is going to do before we execute it (this makes it easier to see when something is wrong). Secondly, **when** we see an error, we must try to understand and remember it. We will see the same error again, but the next time we will be able to solve it more quickly if we remember what was wrong last time.

When you run into a difficult task, completing it will usually involve all of the above activities. When we are programming, a very good habit to form is be to ask yourself the following questions:

1. Do I understand what this program is supposed to do?
2. Do I understand how to complete this task without a computer?
3. Do I know how to tell the computer to do these steps?

4. When something goes wrong, what caused this error the last time it happened?

You should be able to answer yes to each of these questions before moving to the next one. If you do not understand what the program is supposed to do, how could you create the program? If you could not do the steps yourself on paper, how can you explain to the computer what steps it should do? If you do not know how to explain these steps to the computer, how can it your write the program?

The last question is all about when something goes wrong (and this will happen often). The process of fixing errors in our programs is called debugging, and it is hard. Brian Kernighan talks about the difficulty of debugging in this quote: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Throughout the book, we will focus on one of the most important aspects of debugging, understanding what exactly the program should be doing **before** we execute it. We will be using a technique called code tracing to help us understand concepts as we study them. The better our understanding of each of these concepts the more likely that an error we face will be a mistake like misspelling or a problem with syntax (which are generally easier to fix) rather than a problem with the logic of our program.

1.1 What is a Program?

If we are going to be programmers, we should probably first know what a program is. A program or application is a sequence of instructions. These instructions tell the computer **how** to complete a task. When a computer follows the instructions in a program, we say it executes the program. The instructions in the program are executed one by one.

As a simple example program for a person, imagine giving directions from your dormitory to the classroom. The person following your directions has to follow them in order. As they execute this "program" they must follow the directions by completing each one before starting the next. If they follow these directions exactly, they will arrive at the classroom.

The difference between giving a person directions and writing a computer program is that **computers are stupid**. If we want a computer to do something, we must tell it **exactly** how it is to be done. The computer will follow these **instructions** precisely and do nothing else. If we forget a step, it will not work correctly. If the instructions are not detailed enough, it will not work correctly. If we do not put the instructions in the correct order, it will not work correctly. Humans are usually smart enough to fill in gaps in a set of instructions or realise if the instructions are not in the correct order.

To a computer, we would need to explain everything! A human can understand the instruction exit the building, but to a computer we would need to explain all of the small details. Consider the following set of instructions:

- Find the door in the room
- Move to the door and exit the room
- Find the stairs
- Move to the stairs
- Travel down the stairs
- Exit the main door

This might seem like a reasonably thorough set of instructions, but lets look at the first one as an example. "Find the door in the room". Does the person following the instructions know what a door is? How should they find the door? Should they start rotating on the spot until they see it? What if the door is not visible from their current position?

This is perhaps a silly example, but it gives us an idea of the challenge that we face when programming. Every program we write must be explained in terms of the **instructions** that the computer can understand. This means that we must have a good understanding of the instructions that we can use in a given programming language.

Programming can be done in many different programming languages. These languages can usually be categorized based on how they work. Python is a procedural programming language (there are many other procedural programming languages) and we will learn the techniques of writing programs for this type of language. Other procedural languages will have similar techniques, but the code will look a little different. What is important it that we learn the techniques, once we know these we can very easily learn a new procedural language (we just need to learn its grammar).

Abstraction

A programming language is just a way of describing our instructions to the computer in a way that the computer can understand. That leads to the question, what does a computer actually understand? All that a computer really understands is electricity! An instruction is just some voltages on a group of wires. This is a really hard way to think about programming, so what we usually do is use abstraction to hide the complicated detail and only see the bits that are important to us. Abstraction is done at many different levels, here is an example at how we can look at some of the abstractions used in modern programming languages:

1. We forget voltages and just consider if each wire is on (high voltage) or off (low voltage)

2. We forget wires and just think of an instruction as a group of 1's and 0's (1 for on, 0 for off), it is just a number

3. We forget numbers by giving a name to each of the instructions that is easier for humans to understand

4. It takes several of these instructions to do very simple tasks, so we represent them using more familiar concepts

5. Group these instructions together into functions that can be reused

6. Group functions and data that are related together into objects that can be used

The lowest level we will ever discuss is number 2 (and only in this example). These ones and zeros that are used to represent our instructions and data are known as machine code. Machine code is usually represented in the binary or hexadecimal number systems. Instructions in this form are extremely difficult for humans to understand and write.

```
8b 4d fc
03 4d f8
89 4d f4
```

Listing 1.1: Adding 2 numbers in Machine Code

This is an example of machine code (number 2), it shows 3 instructions used to add 2 numbers together and remember the result. Each line shows the hexadecimal numbers that are a shorter way to write a binary number. Each line represents the state of 24 wires on or off. This is not a very nice way to look at a computer program.

```
1 movl    -0x4(%rbp), %ecx
2 addl    -0x8(%rbp), %ecx
3 movl    %ecx, -0xc(%rbp)
```

Listing 1.2: Adding 2 numbers in Assembly Language

This is the same example, but abstracted to a higher level (number 3) so we have names for the pieces in the program. The first line loads a number to a location on the computer, the second adds another number to that location and the third saves that result to another location on the computer. This is a higher level of abstraction, but still not a very nice way to look at or write computer programs.

```
1 z = x + y
```

Listing 1.3: Adding 2 numbers in a Programming Language

Finally, this example takes us closer to what we really see and think about when programming (number 4). The code takes 2 values (here `x` and `y` can be any values) and adds them together. The answer is then remembered using the name `z`. As you learn programming, you will use many abstractions to make extremely difficult tasks much easier than they otherwise would have been (the can still be hard though).

Python allow us to group instructions together into functions that can be reused. This is hiding a lot of the low level details, but there are still a significant amount of detail. Languages like C (procedural) and Java (object-oriented) will hide differing levels of detail. Comparing the two, Java would use more abstractions (hide more detail) than C, however Python is hides more detail than both.

Some programming languages (like Python) allow us to use more abstractions to build bigger and more complex programs quickly. These are known as high-level programming languages. Other programming languages (like C) use less abstractions and allow us to create programs that are faster and more efficient. These are known as low-level programming languages.

What all of these programming languages have in common is that we must have some way of converting the instructions we write (like in example 2) into machinecode that the computer can understand. This is done using special programs that have been written for us called compilers or interpreters.

In this textbook we will be learning Python, which is a interpreted language. Instructions that we write in Python are translated into machine code while the program is being executed. The instructions that we write are written in **text** files. These files containing instructions are usually called code or source code. We give the files names that end with a special name so that we know they are code. For example, code written for the Python programming language would be saved in a file with a name ending `".py"`.

Editing Source Code

We will spend a lot of time working on code in a text editor. All computers will come with a text editing program that can be used to work on these files (TextEdit on Apple computers and Notepad on Windows computers). However, while these programs will work, there are many better applications that are designed to be more useful for programmers. Text editors for programmers usually provide features that make the process of programming a little easier.

The features you should expect to have are syntax highlighting and preferably some form of formatting assistance. Syntax highlighting is a simple feature that changes the colour of parts of the text. For example, keywords in a programming language will be one colour, text will be a

different colour and numbers will be a different colour. This is really helpful for spotting when something has not been typed correctly.

When I am programming on a Windows computer, I usually use the program Notepad++. When I am programming on an Apple computer, I usually use the program Textmate. There are many other options available, but it is a strong recommendation that you download and install at least one programming text editor on your own computer. Additionally, the Python interpreter IDLE has a text editor for Python.

1.2 Programming with Python

Python is an interpreted programming language. In order for us to be able to program in Python, first we must have the interpreter installed on our computer.

We can use the interpreter in two ways, we can type our code into the shell where it is executed when we are finished typing, or we can save our code in a separate file and have the interpreter execute the whole file.

Versions

There was a significant change between versions 2 and 3 of the Python language. Code written fro version 2 may not work correctly when executed using version 3 and code written for version 3 may not work correctly when executed using version 2.

We will be learning version 3. When reading about Python on the internet make sure the examples you look at are for python 3 and not python 2.

Installing Python

Python can be installed in a number of different ways. The recommended method in this book is to download the installer from the official Python website `https://python.org`. Multiple versions of Python are available, you should download version 3.8 or higher.

Once downloaded, the installer should be executed and all of the steps followed until the process is complete.

We can verify that the interpreter has been installed correctly but typing IDLE into spotlight on an apple computer or the task bar on a windows computer. If the interpreter installed successfully the IDLE application will be in the list. Open the application and you should see the window shown in Figure 1.1

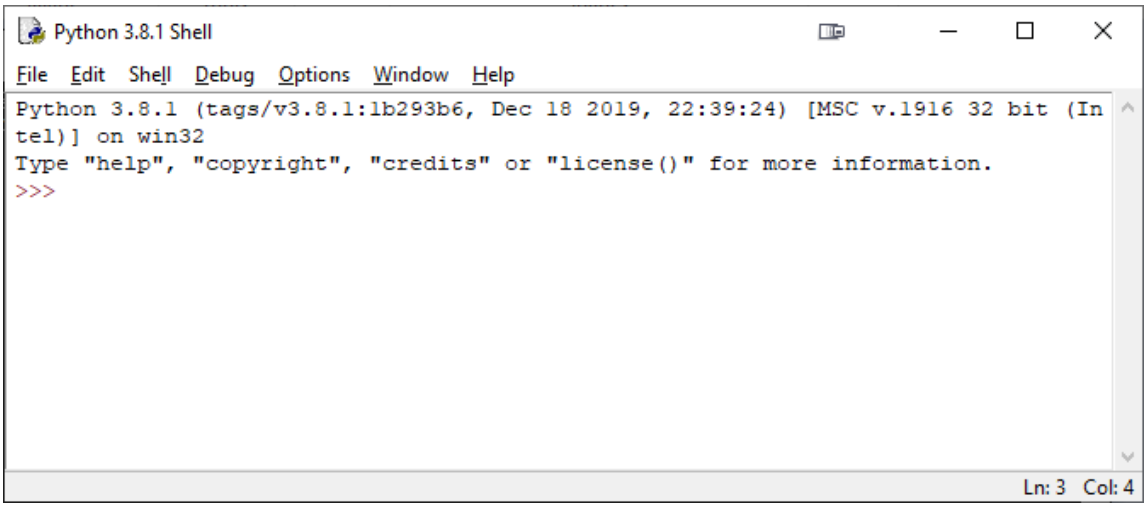


Figure 1.1: The IDLE application containing the Python Shell

The Python Shell

IDLE is a development environment for programming in Python. We can use IDLE to edit and execute Python files we are working on, or we can use the shell to type instructions one at a time.

The python shell allows us to interact directly with the python interpreter. When the interpreter is started, it will print a prompt (three greater than symbols) `>>>`. The prompt is where we can type statements and expressions. When the code we have typed is finished, the result will be displayed and another prompt will appear ready for the next statement.

Programming in this way is very useful when learning or testing something. However, when writing larger programs it is not very convenient to use the Python shell. Instead we would save the code of our program in a text file and interpret the file in one step.

Executing Python from a File

When we are writing our code, instead of typing it in the shell we should save it in a file. This can be done by selecting `File > New File` from the IDLE menu. This should open a new window that we can use to type code. This window is a basic text editor for python a does not automatically execute each line of code as it is typed.

Instead, we can type many lines of code in the window and then save the file by selecting `File > Save` ad giving the file a name ending in `".py"`. Saving the file is necessary before the code can be executed. At this point we can execute the code, this can be done by selecting `Run > Run Module` from the IDLE menu. The code will be executed and any output will be shown in the IDLE shell.

This is useful when we are learning. When we have made a mistake we only need to edit the code in the file, rather than have to type it all out again. Then to test if we have fixed the problem we can execute it again and again until we have solved the problem.

Difference Between Shell and Executing Files

When typing code into the shell, the result of every calculation is immediately printed to the screen. This is very useful for development as we can check that the code is working as expected. However, when we are developing a program for others to use, we will not be expecting the user to type the commands one by one. Instead, we can give them a Python file containing all of the code to be executed.

When executing a python program from a file, we only see the output that we want to see. We choose what information is shown on the screen, rather than the result of each calculation being shown automatically.

First Program

When learning a new programming language, the first program written by a programmer is usually hello world. This is a very basic program with no input or complex calculations, all it does is display the message `"Hello world"` on the screen for the user to see.

We are going to look at how this is achieved in Python.

```
1 print("Hello world!")
```

Listing 1.4: Hello world in Python

In python, hello world is a single statement. This code uses (calls) a function named `print` to show a message on the screen.

Chapter 2

Fundamentals of Programming

Before we begin specifically to learn about programming in Python, we will first look at the idea of what programming is. We will look at the what programs are made up of and the kind of things we can do in our programs.

2.1 Statements

A program is a sequence of instructions or statements. Each statement results in some action (or actions) being performed. These statements allow us to control the computer. Using different statements, we can perform a number of actions, such as:

- Display information on the screen for the user to see.
- Read information the user types using the keyboard into our program.
- Remember information in the program and use it later.
- Process information and perform calculations.

Statements can usually be classified into three types:

- Input and output statements (IO)
- Variable declaration and manipulation statements
- Conditional and control statements

Input and output statements are the statements that allow us to display information on the screen (Output) or read information from some other source into the program (Input). These types of statements allow us to write programs that can interact with the user.

Variable declaration and manipulation statements are used in remembering and changing information in our programs. They allow us to remember a piece of information such as a number or a piece of text and use it again later in the program.

Conditional and control statements are used to give programs a way to choose what should happen next. These types of statements are generally a little more complicated than the others. Using these we can make simple choices such as what to do next or more complicated choices like how often to repeat an operation.

Additionally, we will also study how to define sub-programs (functions) which allow us to break programs into smaller pieces that are easier to solve individually.

2.2 Data Types

Data types are one of the most fundamental concepts in computer programming. Every piece of information in a computer program is categorised based on its type. It is very important for us to understand what types of information we are working with as this will determine what we can do in our statements. Some programming languages will hide much of this detail (more abstraction, like Python) and other (like C) will require us to be very specific about the type of a piece of data.

There are only two types of data represented in a computer:

- Whole numbers, called integers, such as 1, 2, 3, -4533...
- Real numbers, called floating point numbers, such as 123.456, -14.1...

Some programming languages (like C and Java) often have different sizes of these data types, however this is not the case in Python. We will learn about the data types we can use in Python in the next chapter. Additionally, it is important to know that there are other types of data (often more complicated) that we can use. However these are built using either integers or real numbers.

Text

Text is an example of one of the more complicated data types. In programming it is called the string type (which represents text in our programs such as `"Hello"` or `"goodbye"`). Strings represented using whole numbers, but we will learn about how this works later.

2.3 Expressions

Code in programming is full of expressions. All statements will contain some expressions. An expression is a piece of code that can be calculated to some **value**. The calculation may be very complex or very simple, what is important is that an expression will have a result. The simplest type of expression are values that we write directly into our code, such as `12` or `"Hello"`.

Literals

Literal values (sometimes incorrectly called constants) are when we put values directly into the code of our program. Simple examples of literals would be values in an expression like `1 + 2`. When we want to include numbers in our code we can just type them, however if we want to include text in our code we must tell the compiler that it is text and not code. We do this by surrounding the text we want to include with quotes `"..."`.

String Literals

When the compiler finds text surrounded by quotes, it knows that it does not have to try and understand the text. When we forget to add these quotes, the compiler tries to understand the meaning, but usually fails because the words do not match the grammar of the programming language. For example, the text `hello world!` does not make any sense to the compiler, so it cannot understand.

Whenever we need to have some text in a program, we must always surround it with quotes. This is called a string literal, or more commonly just called a string.

Escape Characters

Some of the characters we want to put in our strings cannot be easily typed on a keyboard. Such as the newline character (makes the output move to the next line of the screen). These special characters can be typed using the **escape** character (backslash). Combining the escape character with another character can have a special meaning. Table 2.1 shows two of the most commonly used special characters.

Table 2.1: Examples of Escape Characters

Character	Escape Code	Meaning
Newline	`\n`	Moves following output to the next line
Tab	`\t`	Moves following output one tab to the right

For example the code `"Hello\nworld"`, would be output as the word hello on one line and world on the next line.

Operators

We can create larger expressions using operators. A simple example of an expression is the code `12 + 34`. Here we have two values and an operator, the value that this expression has is the result of completing the addition operation (46). Because this expression has a value, we can use this expression in other places, such as in a larger expression or as the parameter to a function.

Data Types

As every value in our programs will have a type, this means that the result of any expression will also have a type. We will learn later how to calculate what type each of our expressions will be. This will be an good skill to have.

2.4 Library Functions

Programming languages provide certain functionality that we can use. This functionality is provided by library functions. These functions are already written for us to use, created by the designers of the programming language. A function is a sequence of instructions that are grouped together and given a **name**.

In Chapter 1, we saw an example of the built-in function `print` being used in the example. This function is how we can display some text (or numbers) on the screen for the user to see. Showing information on the screen is a task that will be in most of the programs that we write, but without this function it would be very difficult.

There are generally **many** different built-in functions available in different languages. Usually, a programmer will become very familiar with the functions for input and output available in the library. However, learning all of these functions is not necessary (or even a good idea). Programmers couldn't possibly remember all of the functions available and how to use them, instead we usually learn **how to read about them** in the language documentation (or on the internet). This way, we can use any library function that we need.

Parameters

When we use one of these built in functions (or any other functions), we sometimes need to give some information to the function. For example, the `print` function cannot know what to show on the screen unless we tell it. The information we give to the function is known as parameters.

Parameters are placed between the round brackets after the name of the function. If there is a single parameter, we simply put it between the brackets (E.g. `("Hello")` or `(123)`). If there are more than one parameter, then we must put then all between the brackets but place a comma `(,)` between them. E.g. `("Hello", 5, "Goodbye", 10)`. If a function requires no parameters, we still need to put an empty set of round brackets after its name when we use it e.g. ().

Type

Functions require an exact (or minimum) number of parameters. And more specifically, these parameters must be in the correct order and be the correct data type. For example, the function `abs` requires at least one parameter (a number).

Functions in Expressions

Some functions can also be used in expressions, however these can only be used if the function gives us a value to use (called a return value). There are many functions that work like this, the function `abs` is an example of this. The `abs` function takes a number as a parameter and returns the same value converted so that it is positive. E.g. `abs(-41)` would give the answer 41, the important thing is that it gives an answer.

2.5 Variables

When we are programming, we will not always know all of the values we wish to use in our programs. However, we must have a way to describe the calculations that should be performed when we know the value. Variables are used to give a name to a value instead. Then in all of our calculations we can use the name instead of the value. Additionally, we can change the value of the variable as the program executes which allows us to perform more complicated calculations.

Using variables in programming is similar to how we use variables in maths. There are many examples where we use the name instead of a value to describe a particular calculation. Typical examples are the area of a square (`width*width`) or circle (`pi*r*r`). Defining calculations in this way allows us to write programs that can be used many times for different values.

Variables can also be used in expressions. When we use the name of a variable, the computer automatically finds the correct value and uses it. We can use expressions like `width * 2` and the computer will find the value of width and then multiply it by 2.

Memory Map

A useful tool for working with variables is a memory map. This is a table where we write the name of our variables on one side and their values on the other. When we need to know what the value of a variable is we look it up in the table. When we need to change the value of a variable we change it in the table. This happens automatically in the computer but it can make it easier to understand when learning if we also create and update memory maps for our programs. We will use this technique many times in code tracing exercises.

Name	Value
width	55
radius	10
count	5

Variable Names

One of the difficulties of programming is understanding the programs that we or others have written. We can make our programs easier to understand if we give descriptive names to our variables. For example, if a variable is supposed to remember the width of a square, name it `width`. If a variable is supposed to remember the number of students in a class, name it `numStudents`.

Giving variables names like a, b, c, and d is OK when programming, but when these variables are used in calculations later it can be difficult to remember which is which.

One important note is that variable names cannot contain spaces. This means that a variable name like `numberPeople` is OK, but a name like `number people` will cause an error in your program.

Assignment

When we give a variable a value, this is called assignment. The result of any expression can be assigned to a variable. The symbol for assignment in most programming languages is `=`. A typical

example of an assignment is `age = 45`, however different languages can have different rules about how exactly variables are assigned.

2.6 Pseudocode

When learning to program, it can be useful to be able to communicate and explain our ideas for how a program could be written without having to use code. Pseudocode is a way of expressing these ideas using natural language. There is no specific grammar or syntax, instructions are written in a from that humans can interpret (but could not be understood by a computer). This can be helpful in separating the more difficult task of actually designing a program from the task of translating it into a specific programming language.

```
1 display a message asking the user to enter their weight
2 read number from user into variable weight
3 multiply weight by 2.2 and store in variable weightlb
4 display message to the user telling them their calculated weight in pounds
```

Listing 2.1: Converting the users weight from kg to pounds

This is a basic example that explains the steps involved in a program to ask the user for their weight in kg and then converts it to pounds and outputs the result. Each of the statements are explained in terms of their outcome with no specific requirement for how they are phrased.

Choices

In chapter 4 we will discuss how to have our programs make decisions or choices. To represent the results of our choices in pseudocode, we use indentation. Indentation is the process of aligning the starts of the lines to show instructions that are grouped together or representing different outcomes of choices.

Lets look at an example of a program that calculates the users BMI (Body Mass Index).

```
1 display a message asking the user to enter their weight (kg) and height (cm)
2 read numbers from user into variables weight and height
3 calculate bmi (weight/height/height)*10000 and store in variable bmi
4 if the bmi is less than 18.5
5     display message to the user telling them they are underweight
6 else
7     display a message to the user telling them they are not underweight
8 display a thank you message
```

Listing 2.2: Calculating the users BMI

Here on line 4 we have a part of the program where a choice is made. The program will perform different actions based on the value remembered in the variable bmi. The different outcomes are shown using indentation (spaces are added at the start of the line). The indented line after this shows the action that happens when the condition described on line 4 is true (i.e. bmi is smaller than 18.5). In this case there is a single thing that happens, a message is printed to the screen.

The end of the statements that should happen when the condition is true is shown by a line (6) that returns to the previous indentation level (no indentation). The statement on line 6 is linked to the choice we are making on line 4. In this case it indicates that the next statement or statements should only be executed when the condition described on line 4 is not true (i.e. bmi

is bigger than or equal to 18.5). Again, the statements that should happen in this case are shown using indentation, so in this case only the statement on line 7 is executed in this case.

Finally, line 8 shows a statement that is not indented, this means that this happens all of the time, no matter if the bmi was bigger than or smaller than 18.5.

Repetition

In chapter 5 we will discuss how to have our programs make decisions or choices that result in actions being repeated multiple times. Again, to represent these we will be using pseudocode with some indentation to show which actions are repeated. Repeated actions can also include choices or other repeated actions, so this means that there will be several levels of indentation to show what actions are grouped together.

Lets look at an example of a simple guessing game (simple to play not simple to create).

```
1  remember the secret number in a variable named answer
2  display a message asking the user to enter their guess
3  read number from user into a variable named guess
4  repeat as long as the guess does not match answer
5      if the guess is bigger than the answer
6          display message to the user telling them their guess is too big
7      else
8          display message to the user telling them their guess is too small
9      display a message asking the user to enter their guess again
10     read number from user into the variable named guess
11 display a message to the user congratulating them for guessing correctly
```

Listing 2.3: A number guessing game

We can see here that there are three levels of indentation, no spaces, 4 spaces and 8 spaces. How we group things together is based on changes in the indentation level. An increase in indentation after the contidion of a choice or repetition shows that statements are grouped together, when we return to the previous indention that shows the end to that grouping.

In this example, the group of statements on lines 5 to 10 are repeated until the user guesses the correct number. Inside this, either the statement on line 6 or 8 will be executed depending on the value that the user enter in their last guess.

Understanding

One of the main uses of pseudocode is to help us understand how a program can be completed. This relates directly to the first 2 activities we discussed in Chapter 1, understanding what we are trying to do and understanding how it can be done. If we can represent the program in pseudocode, then we are demonstrating that we understand these two steps and that we are ready to translate our solution into actual code in a programming language.

2.7 Comments

Programming languages allow the developer to add some notes to their code to help explain what is happening. These notes are called comments are are ignored by the compiler when your program is being created. These comments are not for the computer, but for you or another person that will be reading your code later. It might seem silly, but when we are designing complicated programs it can sometimes be difficult to understand code that we wrote previously.

All programming languages have some syntax that allows us to insert these messages into the source code so that they will be ignored and cannot cause any errors. Comments are usually highlighted in a different colour to code and string literals to make it easier to see.

Chapter 3

The Syntax of Python

Python is a flexible high-level programming language. It is designed to be easier to read and program in. In this chapter, we will start the process of learning the syntax (grammar) of Python. Lets first look at the hello world example from the first chapter.

```
1 print("Hello world!")
```

Listing 3.1: Hello world in Python

This is a very simple example that only contains a single statement. `print("Hello World!");` is the only statement in the program. It uses the function `print` and gives a string literal as a parameter to be shown on the screen.

3.1 Syntax

Syntax is the name for the grammar of a programming language. In human languages small mistakes in grammar are not a problem because humans are intelligent enough to understand the meaning. However, computers lack this intelligence and can only understand our meaning when the syntax in our code is correct.

Learning Syntax

Unfortunately, the syntax of most programming languages is complicated. This can make learning to program more difficult. The whole grammar of the Python language is defined (you can see it on the internet[1]), however this is not an easy way to understand the language.

Instead, the approach taken in this textbook (and almost all other books) is to introduce the syntax of each new concept as they appear. The general example of the syntax will be introduced in a blue coloured box and examples of the syntax being used will be shown in red coloured boxes.

3.2 Statements

In this chapter we will look at several different types of statements. Each one will accomplish a different task, but when we are able to write these kinds of statements we will be able to build some simple programs. This is the list of statements we will be learning:

- Using the `print` function to display text on the screen (Section 3.3, 3.3, and 3.5)
- Creating variables to remember a value (Section 3.5)

[1] `https://docs.python.org/3/reference/grammar.html`

- Performing calculations are remembering the result in a variable (Section 3.6)
- Using the `input` function to read information from the keyboard (Section 3.7)

3.3 Output (Printing)

When our program has completed its work or simply needs to display a message to the user, we will need to output this message. In Python output is mostly done using the function `print`. When we just want to display some text to the screen, the `print` function is pretty simple to use. We just place the text we want to be shown on the screen in a string literal as a parameter to the function. E.g. `print("This is my message")`

Lines

By default, everything we output using `print` is all displayed on a single line. Every time it is used, it automatically adds a new line character (\n) to the end of the output. The next time that we use the `print` function, our output is shown on the next line.

Printing Multiple Values

The print function allows us to print multiple values. We place each of the value we want printed as parameters of the `print` function (inside the round brackets separated by commas). The function outputs the values in order with a space between parameters. For example, `print("hello","world")` would output `Hello world` and `print("Hello", 5, "class")` `Hello 5 class`.

Using Functions

A function is some code that does a task for us. Functions have a name that we can use whenever we want that code to be executed. The syntax of using a functions is always the same, first we type the name of the function which is always followed by a pair of round brackets.

```
1 funcname()
```

Listing 3.2: Using a function

This is an example of the syntax, `funcname` represents the name of the function. Any parameters that are required by the function go inside the round brackets. If there is more than one parameter, they must be separated by commas.

```
1 funcname(param1, param2)
```

Listing 3.3: Using a function with parameters

Lets look at an example we have seen before, namely the function named `print`. Lets assume we want to use the function to print the message `"I used a function"` to the screen. In this case, we replace `funcname` with `print` and put the string literal `"I used a function"` between the round brackets.

```
1 print("I used a function")
```

Listing 3.4: Examples of Statement that uses the print function

3.4 Data Types

We have learned that in computers all data is represented as numbers, and all numbers are either whole numbers (integers) or real numbers (floating point numbers). In this section we will learn about how the basic types are represented in Python.

Numbers

Integers

Whole numbers are represented as the type `int` (short for integer). Integers represent values such as 1, 2, 456 and so on. In Python 3, we can represent integer values of any size in our calculations.

Real numbers

Real numbers are represented as the type `float` (short for floating point number). Real numbers represent values such as 1.5, 13.56, 15.0 and so on.

It is important to note that a float value may be the same as an integer value, but they are still different types E.g. 45 is the same as 45.0, but the first is an in and the second is a float.

Approximate Numbers

Floating point numbers are often approximate values. When calculated, they will be the closest value it is possible to represent using a floating point number. This is not a problem in the Python language, but a limitation of trying to represent an infinite set of possible values in computers. For this reason, important values where accuracy is required (such as money) are not generally represented using floating point numbers.

Text

Text is represented as the type `str` (short for string). This type of data is more complex in how it is represented inside the computer, but we can use it in the same way as we use numbers. String values in python are shown surrounded by quotes. We can use single quotes, e.g. `'I am a string'` or double quotes `"I am also a string"` and for representing a string that is multiple lines long, we use triple double quotes:

```
1 """ I am a really long
2   String, spread over many lines. """
3
```

Listing 3.5: A string containing many lines

Forgetting Quotes

Remembering to include quotes for strings in our code is important. The purpose of these quotes is to tell the interpreter that this is a string and that it does not need to interpret the meaning. It is very likely that if the interpreter tries to understand our text, that it will fail.

For example, if we forget the quotes around hello world in our example, the program just gives us an error message. This is because the words hello world do not make sense as Python code.

Boolean Values

Python also makes use of a type called `bool`, this represents boolean values (named after a famous professor of mathematics in Ireland). Boolean values are concerned with logic and have only two

possible values `True` and `False`. Boolean values and expressions will become very important when we are learning about making decisions in Chapter 4 and repeating actions in Chapter 5.

Difference Between Types

It can be easy when first learning to program to confuse the different data types. Thankfully, there is a useful function that we can use to check the type of a value in our code. The function is named `type` and requires a single value as a parameter, it returns a value that we can print to see what the type was.

This table shows a number of values, the code we would use to check what its type is and also the result of that check. The result shows us the class of each of those values, this has a specific meaning in programming that we will learn about later, but for now this simply means type.

Table 3.1: Examples of values and their types

Row	Value	Code	Result
1	`True`	`type(True)`	`<class 'bool'>`
2	`"True"`	`type("True")`	`<class 'str'>`
3	`1`	`type(1)`	`<class 'int'>`
4	`"1"`	`type("1")`	`<class 'str'>`
5	`1.1`	`type(1.1)`	`<class 'float'>`
6	`"Hello"`	`type("Hello")`	`<class 'str'>`
7	`1.0`	`type(1.0)`	`<class 'float'>`
8	`"1.01"`	`type("1.01")`	`<class 'str'>`

Changing Types

There are some interesting points that we should note from the table above, in particular from row 2, 4 and 8. In each of these cases, we might have been confused about the type of our data. In row 2, we might have thought this was a `bool` value while it is in fact a string (`str`). In row 4, we might have thought that this value was an `int`, while it is also in fact a string (`str`). Finally, in row 8, we may have thought that the value was a `float`, while again it was a string (`str`).

This is something we will need to be aware of, because the type of our data changes what operations we can perform and what the results will be. There are several functions that we can use to change the type of our data.

To an Integer

The function named `int` can be used to change the type of a piece of data to an integer. It will work for a string containing and integer value, or for a float value.

For example, the expression `int("123")` would have the result `123` (notice here the quotes are gone, this is now an integer number when before it was a string).

For changing a float value, the expression `int(123.45)` would have the result 123 (changed from float to int).

However, it should be noted that a string containing a real number cannot be converted into an integer this way. The expression, `int("123.45")` will cause your program to fail.

To a Real Number

The function named `float` can be used to change the type of a piece of data to a float. It will work for a string containing and float value, or for an `int` value.

For example, the expression `float("123.45")` would have the result `123.45` (notice here the quotes are gone, this is now a real number when before it was a string).

For changing a int value, the expression `float(123)` would have the result 123.0 (changed from int to float). This is not a particularly useful change, but can be made none the less.

To A String

The function named `str` can be used to change the type of a piece of data to a string. It will work for a `int` or `float` value, or for many other types we will learn about later.

For example, the expression `str(123.45)` would have the result `"123.45"`, and the expression `str(123)` would have the result `"123"`. These are both very useful when building output for the user.

3.5 Variables

A variable is a way of remembering a piece of information with a name so that we can use it later in our program. This piece of information is the result of an expression, so it can be something like a literal value, the result of a calculation or a value that the user has typed on the keyboard.

Assignment

Variables are very flexible in Python and we create a new variable by giving it a value. The process of giving a value to a variable is called assignment. A variable can be used many times to remember different values, every time we assign a new value the old value is forgotten.

We use the assignment operator (=) to remember a value in a variable. The syntax we use is the name of the variable, followed by the operator, followed by an expression (E.g. `height = 178`).

```
1 name = exprssion
```

Listing 3.6: Assigning a Value

The expression that we use can be a literal, a more complex calculation, a value returned by a function or even the name of another variable.

```
1 height = 178
2 weight = 77.5 * 2.2
3 num = abs(123 - 456)
```

Listing 3.7: Examples of Assignment Statements

Line 1 shows an example of assigning a literal value to a variable. This number will now be remembered using the name `height`.

Line 2 shows an example where the result of a calculation is assigned to a variable. The calculation of 77.5 * 2.2 must be completed first (170.5), then this result can be remembered using the name `weight`.

Line 3 shows a more complicated example. Here the result of a function is remembered using the name `num`. The steps involved are the following; first we must calculate the result of `123 - 456` (-333), then this result is passed as parameter to the function `abs`. Finally, the result of the function (333) is remembered using the name `num`.

Variables in Expressions

Once a variable has been defined and a value assigned to it, we can use it in our code. To use a variable is quite simple, we just type the name of the variable. The computer will find the **value of the variable** and use it in that place.

```
1 print("Seans weight is",weightkg,"kg")
2 weightlb = weightkg * 2.2
3 print("Seans weight is",weightkg,"pounds")
4 i = i + 1
5 name = "Sean"
```

Listing 3.8: Examples of Statements Using Variables

Line 1 shows an example of printing the value of a `double` variable named `weightkg`. When this statement is executed, `weightkg` is replaced by the value (lets assume it 77.5) and then this code is executed in the same way as the examples in section 3.3. Line 3 is a very similar example using a different variable.

Line 2 shows an example of using a variable in an expression to assign a new value to another variable. When this statement is executed, `weightkg` is replaced by the value (lets assume it 77.5) and then this value is multiplied by 2.2 (to give us the result 170.5). The result of this expression is then remembered using the name `weightlb`.

Line 4 shows an example of one of the most common statements in programming. Here we are assigning a new value to a variable named `i`. The value we are assigning to `i` however is based on the current value remembered using the name `i`. When this statement is executed, first `i` is replaced by the value (lets assume it is 3) and then we add 1 to this value (to give us the result 4). The result of this expression is then remembered using the name `i`. This code can always be used to increase the value of a variable by 1.

Line 5 shows an example of a variable being used to remember a string value. In practice, there is no difference between assigning different types of values to variables. However, we will be able to perform different operations depending on the type of the variable.

3.6 Operators

Mathematical Operators

Operators can be used to combine expressions together and perform calculations. In Python we can use all of the expected mathematical operators and some others that you may not have seen before.

Table 3.2: Mathematical operators we can use in Python

Name	Code	Explanation	Example
Addition	`+`	Adds two numbers	`x + y`
Subtraction	`-`	Subtracts one number from another	`x - y`
Multiplication	`*`	Multiplies two numbers	`x * y`
Division	`/`	Divides one number by another	`x / y`
Exponentiation	`**`	Calculates one number to the power for another x^y	`x ** y`
Integer Division	`//`	Divides one number by another with an integer answer	`x // y`
Modulus	`%`	Gives the remainder of floor division	`x % y`

Addition, subtraction, multiplication, division, and exponent are all mathematical operations that are well understood. In the following sections we will look at the two operations that are specific to programming, that of integer division and modulus.

Integer Division And Modulus

Programming languages generally provide the functionality of integer division and the related functionality of modulus. In some languages integer division is based on the types of the values

you are using, but in Python it has a separate operator. The integer division operator will work for both integer and float types, but the value returned will always be an integer value.

Table 3.3: Comparing division and integer division

Calculation	Types	Result	Result Type
7 / 2	int / int	3.5	Real number (`float`)
7 // 2	int // int	3	Integer (`int`)
7.0 / 2.0	float / float	3.5	Real number (`float`)
7.0 // 2.0	float // float	3.0	Real number (`float`)

Integer division (and the related operator modulus) are based on the ideas of a quotient and remainder. Where all numbers are integers, x/y has a quotient of `a` and remainder of `b`, we can look at his as $x = a * y + b$. The remainder can be defined in terms of the other numbers $b = x - a * y$.

Integer division can be used to find `a` (the quotient) and modulus can be used to find `b` (the remainder). Lets have a loot at a table with some examples.

Table 3.4: Comparing division and modulus

Dividend	Divisor	Integer Division	Modulus
7	2	`7 // 2 = 3`	`7 % 2 = 1`
8	2	`8 // 2 = 4`	`8 % 2 = 0`
15	3	`15 // 3 = 5`	`15 % 3 = 0`
17	3	`17 // 3 = 5`	`17 % 3 = 2`
19	5	`19 // 5 = 3`	`19 % 5 = 4`

Operator Precedence

Calculations in Python are performed typically from left to right, but they respect operator precedence. Of the operators that we have seen, exponentiation has the highest precedence. This is followed by multiplication, division, integer division and modulus which all have the same precedence. Finally, addition and subtraction have the lowest precedence of the operators that we have studied.

So if we have a calculation such as $10 + 5 * 3$, then $5 * 3$ (15) is calculated first and then $10 + 15$ giving us the final result of 25. We can use round brackets to change the order that calculations are performed in. If we change the calculation to $(10 + 5) * 3$, then $10 + 5$ (15) is calculated first and then $15 * 3$ giving us the final result of 45.

String Operators

Python also has operators that can be used on other data types. In this section we will study two operators that can be used on strings (data type `str`).

Table 3.5: String Operators

Name	Code	Explanation	Example
Replication	`*`	Replicates a string a number of times, where x is a string and y is an integer	`x * y`
Concatenation	`+`	Joins two strings together	`x + y`

The first operator, concatenation, can be very useful when building text for output to the user. What it does is copy two strings and create a new string containing the contents of both. For

example, the expression `"Tea " + "Leaf"` would have the result `"Tea Leaf"`. Note the result is an exact combination of both, the first three letters and the space from the left side and the four letters from the right side.

The second operator, replication, is less frequently useful but can also be used building structured output for the user. What it does is copy a string a number of times and concatenates all of them together. For example, the expression `"Tea " * 3` is the same as `"Tea " + "Tea " + "Tea "` and has the result `"Tea Tea Tea "`.

3.7 Input

The statements that we have learned about so far allow us to perform some calculations and the output the result. However, if we write a useful program (lets say for example converting weight from kg to pounds), then when I want to use it again I need to change the source code and execute it. This is obviously not very convenient, so next we will study how to read information from the user in our programs.

In Python we use the function `input` to read information that the user types on the keyboard. This function reads a single line of text from the user and gives us a `str` object as a result. This means that if we want a number from the user, we have to use one of the function `int` or `float` to convert the string after it is entered.

This is important, because different operators may look very similar but have very different results. For example, the expression `a + b` would have the result 5 if `a` is `2` and `b` is `3`, however, it would have the result `"23"` if `a` is `"2"` and `b` is `"3"`.

The `input` function can take at most a single parameter and returns a string (`str`) when it is finished. The parameter is to allow a message to be shown to the user, if this is a string then that string will be shown on the screen. The text that the user types is returned as a string. This string can be long or short, it will be all of the text typed until the user presses the enter or return key.

The result of the function should be assigned to a variable, so that we can use it later in our program. This example shows how we can use the text that the user types to create a customised message to display to them.

```
1 name = input("What is your name? ")
2 message = "Hi " + name + ", nice to meet you."
3 print(message)
```

Listing 3.9: Read a line of text from the user

Blocking

`input` is a blocking function. What this means is that it causes our program to stop and **wait** until the user has typed something. If the user does not type anything, then our program will continue to wait and may appear to be frozen (broken). For this reason, it is always important to output a message to the user to tell them what they are expected to enter.

Reading a Number from the User

When in our programs we need an integer from the user, we need to convert the value after we have read it. If we want an integer value, we use the `int` function and if we want a real number we use the `float` function.

```
1 agestr = input("How old are you? ")
2 age = int(agestr)
3 print("Next year you will be", age + 1, "years old")
```

Listing 3.10: Reading a number from the user

Example 6 shows us how this is done for an integer value. The first line of code calls the `input` function and remembers the result in a variable called `agestr`. If the user typed the text 34 and then pressed enter, the then function would return the value `"34"` and this string would be remembered in the variable. The second line uses the int function to convert the value from a string to an integer and remembers the result in the variable `age`. Again, if `agestr` contained the value `"34"`, then age will contain the value `34`. Finally we print some text containing the value, here the addition of the number 1 to the value could not be done if the variable was a string and not a number.

```
1 weightst = input("How much do you weigh in kg? ")
2 weight = float(weightst)
3 weightlb = weight * 2.2
4 print(weight, "kg is the same as", weightlb, "pounds")
```

Listing 3.11: Reading a real number from the user

Example 7 is similar to example 6. The value is read from the user and remembered as a string, then converted to a real number, used in calculations and then in output for the user. The only real difference is the use of the function `float` instead of `int`.

Shortened Code

We do not need to remember the value of the string that the user has typed after we have converted it to a number. Usually, this code us shortened into a single statement. The result of the `input` function is passed directly to the `int` or `float` function as a parameter. This shows example 6 with the first two lines written as one.

```
1 age = int(input("How old are you? "))
2 print("Next year you will be", age + 1, "years old")
```

Listing 3.12: Reading a number from the user

3.8 Comments

Python allows us to include comments in our source code in two ways. The first is used to include some text that is only on one line, and the second allows for several lines of comments to be inserted.

Single line comments use the syntax `#`, when a hash (pound) is placed on a line then all following text on the line will be ignored by the computer. This can be placed at the start of a line or after some code that we have written.

Multi line comments use a syntax that is the same as a multi line string. To show where a comment starts we use the syntax `"""` and to show where a comment ends we use the syntax `"""`. Any text that is between these two symbols will be ignored by the computer.

```
1 name = input("What is your name? ") # ask for name
2 """ Builds a message to be shown to the user with concatenation """
3 message = "Hi " + name + ", nice to meet you."
4 print(message)
```

Listing 3.13: Example of Comment Syntax

This example only shows how we can insert comments into our code, but it is not a good example of when we should insert comments in our code. Typically, this is done when the code we are writing is complicated. As we start seeing more complicated examples of programs later in the book, we will see some examples of good comments.

3.9 Code Tracing

One of the most important steps in programming is understanding. Understanding the task you are trying to complete, the steps needed to complete it, and what happens in each of those steps. In this section, we will be studying a technique called code tracing that will help us understand what is happening in our code as it executes. This will be essential to understand when we are trying to fix problems in our code.

Tracing can be done with different levels of detail, we can simply look at the result of a line of code, or we can look how each step of the calculation is performed and what the result is. In this section, we will look in detail at exactly how a some statements are executed in the following example. For each step of the process, we will be looking at the following pieces of information:

1. The order that the steps are happening in
2. An explanation of the step being carried out
3. The code that was executed
4. The result (if there is one)
5. The type of the result

Calculations and Output

```
1 a = 12
2 b = a + 1
3 c = a + b * 2
4 print(a, b, c)
```

Listing 3.14: Some basic Statements

The first three lines of code are all assignment. The left side of an assignment is always the name of a variable, and the right side is always an expression. When we are performing an assignment, we must always calculate the result of the expression before we can perform the assignment and remember the value in memory.

The first line of code (line 1) assigns a value to the variable `a`. The expression assigned to the variable is a literal value. Table 3.6 shows what happens in this line of code.

Table 3.6 shows the current memory map of the program (after the execution of line 1). Currently, only a single variable has been created and it has been given the value 12.

Table 3.6: Breakdown for line 1 of Listing 3.14

Order	Explanation	Code Executed	Result	Result Type
1	Assignment	`a = 12`	12	`int`

Table 3.7: Memory map after line 1 of Listing 3.14

Name	Value	Type
`a`	12	`int`

Line 2 introduces a little bit more complexity to an assignment statement. Here the expression must be evaluated before we assign the value. Table 3.8 shows the steps and the order that they are completed in. This order might appear backwards, but it is a consequence of needing to know the answer before completing the next step.

Table 3.8: Breakdown for line 2 of Listing 3.14

Order	Explanation	Code Executed	Result	Result Type
1	Load Variable	`a`	12	`int`
2	Addition	`12 + 1`	13	`int`
3	Assignment	`b = 13`	13	`int`

In this example, the main operation is assignment, however before we can complete the assignment we need to know the value of the expression on the right hand side. The expression on the right hand side is an addition, however before we can calculate the result of this expression, we need to know the value of the two expressions we are adding. For the right hand side of the expression this is easy, the value is a literal in the code and no more calculation needs to be done. For the left hand side, we do not have a value, but the name of a variable. So the name must be replaced by the value that it represents. Table 3.9 shows the current memory map of the program (after the execution of line 2).

Table 3.9: Memory map after line 2 of Listing 3.14

Name	Value	Type
`a`	12	`int`
`b`	13	`int`

Line 3 is a little big more complicated again. Again, the main operation is an assignment and we need to know the result of the expression. The expression on the right side of the assignment is an addition, but before we can carry out the operation, we need to know the results of the left and right sides. The left side is the name of the variable `a`, this must be replaced with the value of the variable. The right side is a multiplication operation, and we must know the value of the left and right sides of the expression. The left side is the name of a variable and it's value must be loaded from memory. The right side is a literal value and does not need any more calculation. The complete breakdown of this single line of code is shown in Table 3.10.

This line of code was a good example of how the computer executes code. While the example might seem simple, it shows us that even a simple line of code can actually be many steps to the computer. It also gives us a good example of the order in which these steps are carried out.

For example, operator precedence tells us that the multiplication must be carried out before the addition. However, to the computer it sees the addition of two expressions and calculates the value of each expression from left to right. This is why the first step the computer performed is to

Table 3.10: Breakdown for line 3 of Listing 3.14

Order	Explanation	Code Executed	Result	Result Type
1	Load Variable	`a`	12	`int`
2	Load Variable	`b`	13	`int`
3	Multiplication	`13 * 2`	26	`int`
4	Addition	`12 + 26`	38	`int`
5	Assignment	`c = 38`	38	`int`

locate the value of the variable `a`, then it tries to find the result of the expression on the right (`b * 2`). And before the result of this expression can be calculated, the computer must find the value of the variable `b`.

When we know the value of `b`, we can multiply it by 2, when we know the result of the multiplication, we can add it to the value of `a`. Finally, when we know the result of the addition operation, we can assign that value to the correct place in memory and our memory map in Table 3.11 shows the updated values.

Table 3.11: Memory map after line 3 of Listing 3.14

Name	Value	Type
`a`	12	`int`
`b`	13	`int`
`c`	38	`int`

The final line of code uses (calls) the function `print`. The `print` function requires parameters to know what it should print on the screen, so before the function is called we must find the value of the parameters. The first parameter is the variable `a`, the second parameter is the variable `b` and the third value is the variable `c`. The value of the variables must be loaded before we can call the function. The breakdown is shown in Table 3.12.

Table 3.12: Breakdown for line 4 of Listing 3.14

Order	Explanation	Code Executed	Result	Result Type
1	Load Variable	`a`	12	`int`
2	Load Variable	`b`	13	`int`
3	Load Variable	`c`	38	`int`
4	Call Function	`print(12, 13, 38)`	None	`None`

The first three line of the table is exactly as expected, the value of each variable is loaded and can be used in the function. In the final line of the table, we see something new. We are using (calling) the function`print`, and the result is shown as None. This is because no value is returned when the print function is used. This might seem strange, but it is important to separate the ideas of a result and of output. When the code is executed, the text `12 13 38` is printed on the screen for the user to see, this is the output. A result is something that we can use in our code later i.e. the result of a + 1 was 13 and we could save that in a variable to use later.

Input and Output

```
1 age = int(input("How old are you? "))
2 print("Next year you will be", age + 1, "years old")
```

Listing 3.15: Reading a number from the user

In this example, let us assume that the user will type the number 18. Then we can see the operations that are carried out in the code. Table 3.13 shows the breakdown of the first line.

Table 3.13: Breakdown for line 1 of Listing 3.15

Order	Explanation	Code Executed	Result	Result Type
1	Call Function	`input("How old are you? ")`	`"18"`	`str`
2	Call Function	`int("18")`	`18`	`int`
3	Assignment	`age = 18`	`18`	`int`

We can see that the parameter of the function `int` is the result of the function `input`. In this way, we do not remember that string containing the text that the user typed, instead we remember the number that they entered. The memory map of the program is shown in Table 3.14.

Table 3.14: Memory map after line 1 of Listing 3.15

Name	Value	Type
`age`	18	`int`

The second line is a call of the function print, however before the function can be called we have to calculate the value of the parameters. The two parameters that are string literals require no work, but the second parameter contains a variable and an addition operation. The breakdown ins shown in Table 3.15.

Table 3.15: Breakdown for line 2 of Listing 3.15

Order	Explanation	Code Executed	Result	Result Type
1	Load Variable	`age`	`18`	`int`
2	Addition	`18 + 1`	`19`	`int`
3	Call Function	`print("Next year you will be", 19, "years old")`	`None`	`None`

The memory map is unchanged and the program outputs its result to the screen. Lets compare this program to the earlier version that achieves the same task.

```
1 agestr = input("How old are you? ")
2 age = int(agestr)
3 print("Next year you will be", age + 1, "years old")
```

Listing 3.16: Reading a number from the user

Again, we will assume that the user has entered the text `18` when prompted by the code.

Table 3.16: Breakdown for line 1 of Listing 3.16

Order	Explanation	Code Executed	Result	Result Type
1	Call Function	`input("How old are you? ")`	`"18"`	`str`
2	Assignment	`agestr = "18"`	`"18"`	`str`

This gives us the text `"18"` remembered as a string in the variable `agestr`. The value is shown in the memory map in Table 3.17.

Table 3.17: Memory map after line 1 of Listing 3.15

Name	Value	Type
`agestr`	`"18"`	`str`

The next line of code performs the conversion from a string to an integer. The breakdown is shown in Table 3.15.

Table 3.18: Breakdown for line 2 of Listing 3.15

Order	Explanation	Code Executed	Result	Result Type
1	Load Variable	`agestr`	`"18"`	`str`
2	Call Function	`int("18")`	`18`	`int`
3	Assignment	`age = 18`	`18`	`int`

After this code has been executed, we can see that there are now two variables in the memory map (shown in Table 3.19). `agestr` which remembers the text that the user entered, and `age` that remembers the integer value that the text represents.

Table 3.19: Memory map after line 2

Name	Value	Type
`agestr`	`"18"`	`str`
`age`	`18`	`int`

Finally, the last line is identical in execution to the previous example.

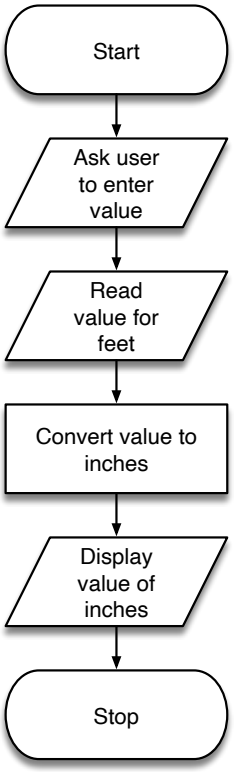
Chapter 4

Making Choices

All of the examples that we have seen so far have been made up of a sequence of statements. These statements are always executed in the same order, even though the values of the variables may be different. The way this happens is known as the **control flow**. This basically means the order in which the statements of a program are executed.

4.1 Flowcharts

Flowcharts are a useful tool for describing the control flow of programs. Flowcharts are commonly used in planning and describing work processes, this is similar to how programs can be described. Flowcharts are diagrams made up of different shaped boxes to represent actions linked by arrows to show the order in which to carry out the actions.




```
1 feet = int(input("Enter a value in feet "))
2 inches = feet * 12
3 print("That is", inches, "inches")
```

Listing 4.1: Converting feet to inches

If we consider the example program above, the control flow is very straight forward. Each statement is executed one after another.
This gives us a very simple flowchart, shown on the left, where we simply follow the arrows from one step to the next.
Each of these shapes has a specific meaning, such as the start or end of the program, input or output operations, or performing some calculations.

Start and Stop

This symbol is used to show the starting and stopping locations in the program. Text is added so we can tell the difference between starting and stopping positions. The starting point usually has the text "Start" or "Begin" and the finishing point usually has "Stop", "End" or "Return".

Input and Output

This symbol is used to show that an input or output operation happens at this point. Text is added to describe what value is being displayed on the screen or read from the keyboard.

Calculation

This symbol is used to show that some calculation is being performed at this point. Text is added to explain what calculation is happening. This text might be very specific and describe only a single instruction, or more vague and describe many operations in not much detail.

4.2 Choices

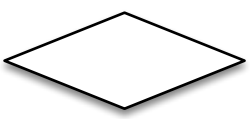
Every day, we make many choices. These choices can usually be described using sentences such as "If I am hungry, I will eat my lunch" or "If it is cold, I will wear my coat". These sentences are called **conditional** sentences.

Conditional sentences have two parts, a **condition** and an **action**. The condition parts of these sentences are "If I am hungry" and "If it is cold". The action parts of these sentences are "I will eat my lunch" and "I will wear my coat".

The action part of these types of sentences are carried out **only** when the condition part is **satisfied** or **true**. This makes sense, because if I am not hungry, I would not eat my lunch. To test if these conditions are true we can simplify them to just the statement. E.g. "I am hungry" or "It is cold". These must be correct or incorrect, true or false. For example, if I state that "It is cold", but the weather is very hot, then the statement is false.

In programming we have similar statements that allow us to make choices in our programs. These are know as **conditional** statements. They allow us to choose what code is executed, based on some **condition**. This gives us the ability to change the control flow of the programs we write based on some condition.

Conditional Symbol



This symbol is used to show that a choice is made at this location. Text is added to explain what the condition of this choice is. There will always be two possible paths to follow after a conditional symbol. The two arrows will usually be labelled with `true` and `false` to show which arrow we should follow when the condition is true and which arrow to follow when the condition is false.

4.3 If Statement

There are several different types of conditional statements. The first we will learn about is called and **if** statement. The `if` statement, and all of the other conditional statements are more complicated than the other statements that we have used before. That is because there are different parts to these statements with different meanings.

There are two parts to an `if` statement, a condition and an action. To use an `if` statement, first we must use the keyword `if`. However, for the computer to know which part is the condition and which part is the action, we must also use a special structure.

`If` statements are usually typed on multiple lines. On the first line we use the keyword, then we type the condition, then this is followed by a colon (:). Actions that should happen when this condition is true are placed on the following lines. However, as we do not know how many actions there will be for an `if` statement, we need some way of knowing which statements are actions that should only happen when the statement is true and which should always be executed.

Indentation is to show which statements are part of the `if` statement and which statements are not. Indentation is the process of preceding some statements with a number of spaces or tabs. Statements with the same number of spaces or tabs in front of them are grouped together. The idea of indentation will be covered in detail in section 4.6.

```
1 if condition:
2   action
3   action
4
5 other statement
```

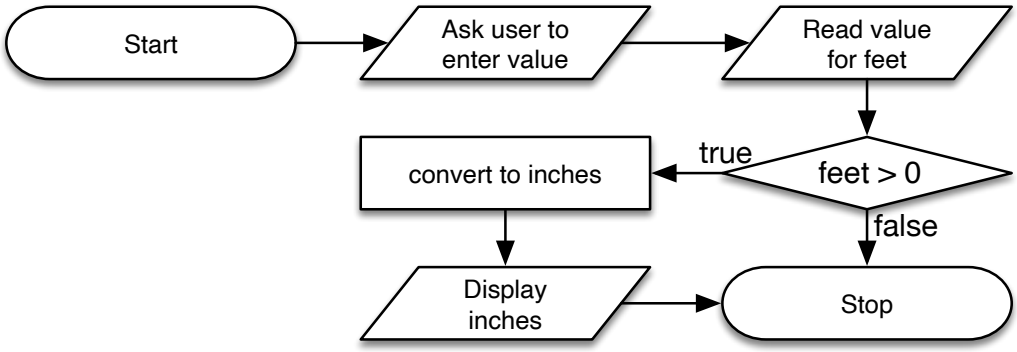
Listing 4.2: Structure of an if statement in Python

In this example, we see where each of the parts of the `if` statement are defined. When this code is executed, the two actions are only executed if the condition is true. The other statements that come after are not a part of the `if` statement and are always executed, even when the condition is false. We can tell this because the other statements are not indented.

```
1 feet = int(input("Enter value in feet: "))
2 if feet > 0:
3     inches = feet * 12
4     print("That is", inches, "inches")
```

Listing 4.3: Example of an if statement in Python

In this example, we see an actual condition and two statements. Only if the value of feet is bigger than 0 will the two statements be executed. Lets look at how we would describe the control flow in this program using a flowchart.



The `if` statement is shown as a single diamond, containing the condition. There are two arrows coming out from this symbol, the first is labelled `true` and points to the calculation and output symbols before the program finishes. The second arrow is labelled `false` and points directly to the stop symbol. This means that if the value for feet is not greater than 0, then the program will not perform the calculation and output steps.

4.4 Else Statement

The `if` statement allows us to choose if a piece of code will be executed or not. This can be very useful, however, we can make it more useful by adding the `else` statement. The `else` statement allows us to add some code that will be executed when the condition is not true. This allows us to make a choice between two pieces of code.

```
1 if condition :
2     actions when true
3 else :
4     actions when false
```

Listing 4.4: Structure of if and else statements in Python

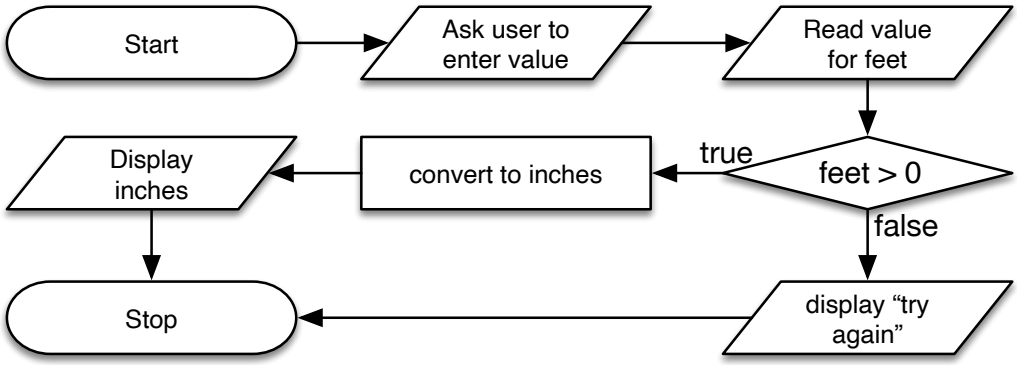
It is impossible to use and `else` statement without first having an if statement. Using an `else` statement, we will always be sure that some piece of code will be chosen.

```
1 feet = int(input("Enter value in feet: "))
2 if feet > 0:
3     inches = feet * 12
4     print("That is", inches, "inches")
5 else:
6     print("You must enter a positive value, try again")
```

Listing 4.5: Example of if and else statements in Python

a

Here we see the same example again, but this time we have added an `else` statement. Now if the user enters a value that is not greater than 0, the program will not just end, instead it will display a message to the user telling them they must enter a positive number. If the user correctly enters a positive value, then they will only see the calculated value output. They will not see the message about entering a positive number. Lets look at how we would describe the control flow of this program using a flowchart.



In this flowchart we can see that after the condition symbol, we will always be pointed to some other action before the program finishes.

4.5 Elif Statement

Using the `if` and `else` statements together allows us to make a choice between two pieces of code. However, sometimes we want to choose between more than two pieces of code. To do this we can use the `elif` statement. As the name suggests, this is a combination of an `if` and `else` statement together. Effectively this allows us to keep searching until we find a condition that is true.

```
if condition1:
  actions when condition1 is true
elif condition2:
  actions when condition2 is true
```

Listing 4.6: Structure of `elif` statement in Python

Just like the else statement, you must have an if statement before you can have an `elif` statement. In this example, when `condition1` is true, the first group of statements are executed, only if `condition1` is false do we check if `condition2` is true. If `condition2` is true, then the second group is statements is executed.

We can add many `elif` statements, one after another. We can also add an else statement to the end. The code within the else part of the statement will only be executed when all other conditions are false.

If we look in more detail at the example, when the user enters a value of 0, they get a message saying to enter a positive number. A more specific message telling them to enter a number greater than 0 would be better. We can use an `elif` and else statement to achieve this.

```
feet = int(input("Enter value in feet: "))
if feet > 0:
  inches = feet * 12
  print("That is", inches, "inches")
elif feet < 0:
  print("You must enter a positive value, try again")
else:
  print("You must enter a value greater than 0, try again")
```

Listing 4.7: Example of an `elif` statement in Python

Here we can see that there are two conditions, `feet > 0` and `feet < 0`. When the first condition is true, the calculations are performed and the result is displayed on the screen then the program finishes. When the first condition is false, the next condition (`feet < 0`) is checked. If this condition is true, then the message telling the user that the number must be positive is printed.

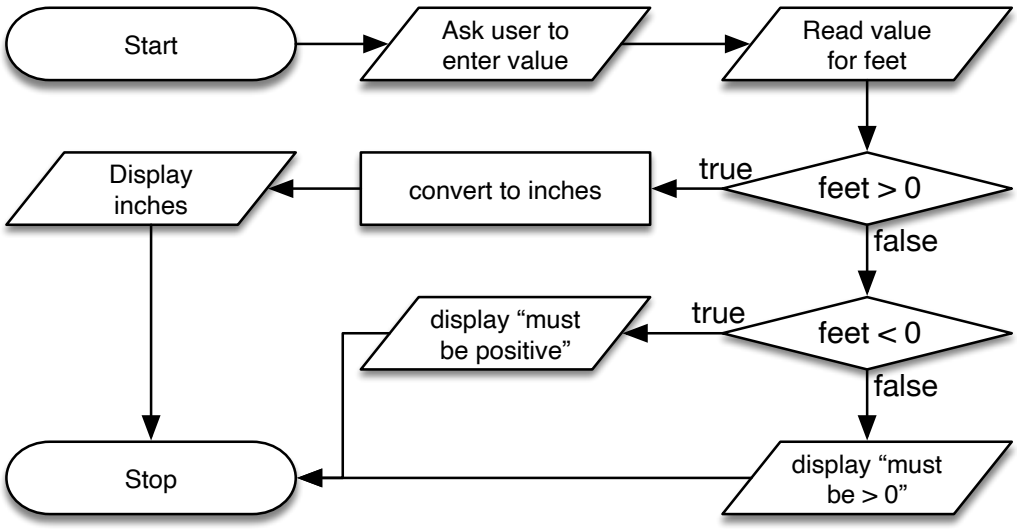
The code in the else section is only executed when all of the other conditions are false. This means that the value for feet is not greater than 0 and it is also not less than 0. This means that the value must be 0, and we print a message to the user telling them the value must be greater than 0.

In important point to note is that the `elif` statement is just a nicer way of showing an if statement placed inside the else statement of another if statement. The following example has the exact same functionality as the previous one. However, without using the `elif` statement, we end up with more lines of code and levels of indentation.

```
1 feet = int(input("Enter value in feet: "))
2 if feet > 0:
3   inches = feet * 12
4   print("That is", inches, "inches")
5 else:
6   if feet < 0:
7     print("You must enter a positive value, try again")
8   else:
9     print("You must enter a value greater than 0, try again")
```

Listing 4.8: Alternate structure of the previous example

If we look at the flowchart for this example, we can see that the decisions in the code are represented by two condition symbols. If `feet` is bigger than 0, we will choose the first set of actions. However, if it is not bigger than 0, we will make another decision based on if it is smaller than 0 or not.



4.6 Indentation

Indentation is very important in Python. If you do not indent your code correctly, then the code will not execute correctly. In other languages, like `C` or `Java`, indentation is only used to make code easier to read and understand.

In each of the examples in this chapter actions inside the if/else/elif statements has been indented to show which part of the code should be executed when the condition is true or false. In these examples, this was done by adding 2 spaces to the beginning of each line.

In the last example, lines 3 and 4 are indented to show that these should only be executed when `feet > 0`. In the same way, line 5 is not indented to show that it is not part of that group of code and should not be executed when the condition is true.

Lets look at an easier example and the output when the program is executed.

```
1 x = int(input("Enter number: "))
2 if x > 5:
3   print("bigger")
4 print("finished")
```

```
1 ------- user types 3-----
2 Enter number: 3
3 finished
4
5 ------- user types 10----
6 Enter number: 10
7 bigger
8 finished
```

Figure 4.1: Example of Indentation and Output

In this example we have an if statement with the condition `x > 5`, after this there are two statements. The first statement is indented meaning it should only be executed when the condition is true. The second statement is not indented, meaning that it should be executed whether the condition is true or false.

The first example of output of the program show that when the user types 3 (and the condition is false) we only see the second statement executed. The second example of output shows that when the user types 10 (and the condition is true) we see both statements are executed.

```
1 x = int(input("Enter number: "))
2 if x > 5:
3   print("bigger")
4   print("finished")
```

```
1 ------- user types 3-----
2 Enter number: 3
3
4 ------- user types 10----
5 Enter number: 10
6 bigger
7 finished
```

Figure 4.2: Example of Indentation and Output

In this second example, the indentation of the second statement is changed. Now as a result, when the user types 3 (and the condition is false) none of the other statements are executed. When the user types 10, both of the statements are executed and we see the same result as before.

Spaces or Tabs

To indent your code you can use spaces or tabs. However, you cannot mix spaces and tabs together in the same file. This can cause confusion because tabs are not the same number of spaces on different computers. This is why spaces are usually preferred. Typically, programmers will use 4 spaces to indicate a single indent.

4.7 Conditional Expressions

The condition part of an `if` statement must always be a conditional expression. A conditional expression is one that results a boolean (`bool`) value i.e. it evaluates to either true or false.

`feet > 0` is an example of a conditional expression, here the answer can only be true (feet is bigger than 0) or false (feet is not bigger than 0). Conditional operators can be used to create

larger and more complicated conditions. In Python, the simplest condition we can have is a either the literal value `True` or the literal value `False`.

Comparison Operators

The conditions that we have seen in the examples so far have all been comparisons. These operators can be used to compare two numbers. There are six comparison operators we can use to compare numbers in Python.

- Are two numbers the same value? `==`
- Are two numbers different values? `!=`
- Is one number bigger than the other? `>`
- Is one number less than the other? `<`
- Is one number bigger than or equal to the other? `>=`
- Is one number less than or equal to the other? `<=`

An example of each of these being used would be:

- `if feet == 5 :`
- `if feet != 5 :`
- `if feet > 5 :`
- `if feet < 5 :`
- `if feet >= 5 :`
- `if feet <= 5 :`

Conditional Operators

These comparison operators are very useful, but sometimes we need to define a condition that is more complicated. For example, if we are writing a program for the user to play a guessing game, we may want them to enter a number between 0 and 100. This requires the combination of multiple conditional expressions. We can use conditional operators to do this.

There are three conditional operators that can be used. These can be separated into binary and unary operators. Binary operators combine two values in some way, similar to mathematical operators that we are familiar with. Unary operators change the result of a single value, this is similar to adding a minus sign to a number.

Binary Conditional Operators

There are two binary operators, `and` and `or`. The `and` operator combines two boolean values, if both of these values are true then the result is true. If either of the values are false, then the result is false. The `or` operator also combines two boolean values, if either of these values is true, the result is true. The result can only be false if both of the values are false.

Table 4.1: Truth table for the and, or and not logical operators

left	right	left and right	left or right	not(left)
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Unary Conditional Operator

Unary operators only change a single value. The `not` operator reverses the value of the expressions. This means that if we the expression is true, then `not(expression)` would be false. Table 4.1 gives the result for expressions `left and right`, `left or right` and `not(left)`.

Now if we go back to our example, a program for the user to play a guessing game, where we want them to enter a number between 0 and 100. This can be done in two ways, using either the and operator or using the or and not operators. Using and, we want the number to be both greater than or equal to 0 **and** less than or equal to 100. This gives us the expression `x >= 0 and x <= 100`. Table 4.2 shows the values for each part of the expression for different values of `x`. We can see that when the value of `x` is in the correct range, we get the expected result.

Table 4.2: Condition for guessing game using and

Value of x	x >= 0	x <= 100	x >= 0 and x <= 100
-14	False	True	False
14	True	True	True
123	True	False	False
44	True	True	True

Using the `or` and `not` operators, we first define the conditions we don't want e.g. the number is less than 0 or the number is greater than 100. This gives us `x < 0 or x > 100`. This will tell us if the number is not what we want so we need to reverse the answer using not. This gives us `not(x < 0 or x > 100)`. Table 4.3 again shows the values for each part of the expression for different values of `x`. Again, the final result matches the previous table and the result we expect to see.

Table 4.3: Condition for guessing game using not and or

Value of x	x < 0	x > 100	x < 0 or x > 100	not(x < 0 or x > 100)
-14	True	False	True	False
14	False	False	False	True
123	False	True	True	False
44	False	False	False	True

4.8 Nested Statements

Within a compound statement like an if statement, we can have any type of statement as a part of the statement. This includes other compound statements like if statements. When one if statement is placed inside another if statement, this is called a nested if statement.

```
1  if x > 0:
2    if x < 10:
3      print("x is a single digit number")
4    else:
5      print("x is larger than a single digit")
6  else:
7    if x > -10:
8      print("x is a single digit negative number")
```

Listing 4.9: Example of Nested if Statements

This code shows an if else statement where each the action of both the if and else parts contain another if (or if else) statement. The if statements inside are the nested statements. In these kinds of if statements, the code inside the nested statements is executed based on the results of the conditions of both the outer and the nested if statements. Lets look at each print statement and determine what conditions must be true for these to be executed.

Table 4.4: Truth table for the `or` logical operator

Line Number	Condition	x = 5	x = 123	x = -4
3	`x > 0 and x < 10`	`True`	`False`	`False`
5	`x > 0 and not(x < 10)`	`False`	`True`	`False`
8	`not(x > 0) and x > -10`	`False`	`False`	`True`

The code on line 3 is only executed when the conditions of both if statements are true because x must be greater than 0 for the nested if statement to be executed and x must be smaller than 10 for the statement to be executed. The code on line 5 also needs the outer condition to be true, but needs the inner condition to be false before it can be executed. Finally, the code on line 9 can only be executed if the condition of the outer loop is false and the condition of the nested if statement is true.

When working with nested if statements, it is possible to rewrite the same logic into a set of if statements that are not nested. The if statements we create need to represent the same logic as above and can be done using the and operator. This is not always possible with other nested statements, like loops (which we will be learning about in the next chapter) but can be accomplished when we have a good understanding of the logic of the code.

```
1  if x > 0 and x < 10:
2    print("x is a single digit number")
3  elif x > 0 and x >= 10:
4    print("x is larger than a single digit")
5  elif x <= 0 and x > -10:
6    print("x is a single digit negative number")
```

Listing 4.10: Same Example with no Nested if Statements

4.9 Code Tracing

In this section we will return to the code tracing technique that we have used before. This will allow us a close look at what happens when we are making choices in our code. We will look at one of our examples from earlier in the chapter.

```
1 feet = int(input("Enter value in feet: "))
2 if feet > 0:
3     inches = feet * 12
4     print("That is", inches, "inches")
5 else:
6     print("You must enter a positive value, try again")
```

Listing 4.11: Example of if and else statements in Python

Tracing True

Line 1

The first line of code is performing several steps, it displays a message to the user, reads text that the user types, converts this text to a number and then assigns the number to a variable. For this example, we will assume that the user has typed the text 6 on the keyboard when prompted for a value.

Table 4.5: Line 1: `feet = int(input("Enter value in feet: "))`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`input("Enter a value in feet: ")`	`"6"`	`str`	
2	Call Function	`int("6")`	`6`	`int`	
3	Assignment	`feet = 6`	`6`	`int`	2

Table 4.5 shows the breakdown of the execution in this line of code and Table 4.6 show the memory map after the code is executed.

Table 4.6: Memory map after line 1

Name	Value	Type
`feet`	`6`	`int`

Line 2

The next line of code is our first example of tracing the condition of an if statement. Here we are checking if the value of `feet` is greater than `0`. So first, before we can do the comparison, we must find the value of `feet` from memory. The comparison then becomes `6 > 0`, which is `True`. Table 4.7 shows the breakdown of this line of code. The final line of the table shows how the decision made by the if statement is displayed, the code is checking only if the condition is `True` or `False`, it does not return any value. What is generally most interesting on this line is the detail of which line will be executed next.

Table 4.7: Line 2: `if feet > 0:`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`feet`	`6`	`int`	
2	Comparison	`6 > 0`	`True`	`bool`	
3	If Condition Test	`True`			3

Line 3

We have seen several examples of this type of code being executed, the variable is loaded, used in an operation and the result is remembered in a variable prompting an update to the value in the memory map. Tables 4.8 and 4.9 show the breakdown of the code and the resulting memory map.

Table 4.8: Line 3: `inches = feet * 12`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`feet`	6	int	
2	Multiplication	`6 * 12`	72	int	
3	Assignment	`inches = 72`	72	int	4

Table 4.9: Memory map after line 3

Name	Value	Type
feet	6	int
inches	72	int

Line 4

Finally, we output the result of the calculations. Table 4.10 show the breakdown of the code executed.

Table 4.10: Line 4: `print("That is", inches , "inches")`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`feet`	6	int	
1	Load Variable	`inches`	72	int	
2	Call Function	`print("That is", 72 , "inches")`	None	None	-

While much of this program was similar to the previous examples we have seen for program tracing, the introduction of the if statement adds a new consideration (we must consider which line of code will be executed next). In this example, the result of the condition was `True` so we executed the code on lines 3 and 4, but the next time the program is run the result could be `False` and line 6 would be executed instead.

Tracing False

Line 1

That is the outcome that we will be tracing in this example, we are assuming that when prompted the user typed the text `"-6"`. Table 4.11 shows the breakdown of the execution in this line of code and Table 4.12 show the memory map after the code is executed.

Table 4.11: Line 3: `feet = int(input("Enter value in feet: "))`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`input("Enter a value in feet: ")`	"-6"	str	
2	Call Function	`int("-6")`	-6	int	
3	Assignment	`feet = -6`	-6	int	2

As a result of the above code a variable was created and assigned, we will show the created variable and its value and type using a memory map.

Table 4.12: Memory map after line 3

Name	Value	Type
`feet`	`-6`	`int`

Line 2

Next we look again at the condition of an if statement. Here we are checking if the value of `feet` is greater than `0`. So first, before we can do the comparison, we must find the value of `feet` from memory. The comparison then becomes `-6 > 0`, which is `False`. Table 4.13 shows the breakdown of this line of code.

Table 4.13: Line 2: `if feet > 0:`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`feet`	`-6`	`int`	
2	Comparison	`-6 > 0`	`False`	`bool`	
3	If Condition Test	`False`			6

The larger consequence to this line of code is that the result determines which line of code will be executed next. The result False means that the next line of code executed is line 6.

Line 6

Table 4.14 shows the breakdown of this line of code. Again it should be noted that the function `print` does not return a value when it is printing to the screen and this is why the result and type are both listed as None.

Table 4.14: Line 6: `print("You must enter a positive value, try again")`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`print("You must...again")`	`None`	`None`	-

Chapter 5

Loops

All of the programs that we have completed so far have completed only a single task. Often in programming, it is necessary to repeat a task many times. Using the programs we have written so far, we would have to execute these programs many times.

Lets create an example, when a student finishes an exam they get a percentage to show how much they answered correctly. However, at the end of a module the students performance is shown by a single grade (A,B,C...). Lets write a program to calculate this grade based on the percentage the student achieved. This can be achieved using some `if`, `else if` and `else` statements.

```
1  percent = int(input("Please enter the percentage: "))
2  if percent > 70:
3    print("A")
4  elif percent > 60:
5    print("B")
6  elif percent > 50:
7    print("C")
8  elif percent > 40:
9    print("D")
10 else:
11   print("F")
```

Listing 5.1: Program to convert a single percentage to a Grade

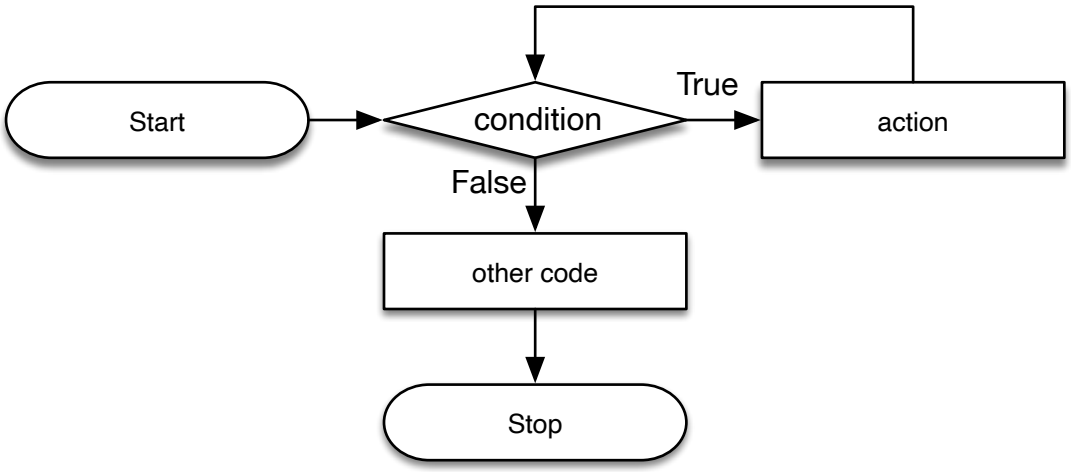
This program will perform the calculations for us easily, however it can only convert a single result. This means that I must execute the program once for every student in the class to get the correct grades. One solution would be to copy the code many times in a single program. This way we would only have to execute the program once, and we would be able to convert all of the results.

The problem is that this could would get longer and more difficult to read. Additionally, if we find a mistake in the code we have to fix this mistake in many places. This might be OK if something only needs to be repeated a very small number of times.

5.1 Repeating Code

Most programming languages provide conditional statements that allow us to repeat a section of code. These statements are known as **loops**. Loops require a condition to tell them when they must repeat.

There are many different types of loops in different programming languages. These all have a different structure, but they all function in approximately the same way. In this course, we will be only looking at a single type, the **while** loop. Before we look at how loops are defined in code, we will look at how they operate using a flowchart.



When we begin to execute a loop, the first thing we do is check the value of the condition. This is done before we execute any action. If the condition of the loop is true, then we perform any actions that are part of the loop and then we go back to check the condition again. We repeat this process until the condition of the loop is false.

If the condition of the loop is false, then we do not execute the code inside the loop and immediately move to the code that comes after the loop.

All loops can typically be broken down to the following parts:

Setup This is where we declare any required variables and give them the correct values

Condition This is where we check if the code inside the loop should be executed

Action This is the code inside the loop that will be repeated

Step This is some change that will eventually stop the loop executing - This is also inside the loop with the action part

The Setup

The setup of a loop is usually very simple, here we will create any required variables and set their values. For example, if we want to count how many times the loop executes, we must have a variable to do this. This must be created before the loop. Typically we use a variable with a simple name (like `i`) and give it the initial value 0.

The Condition

The condition of a loop is very similar to the condition of an if statement. The main difference is that the loop will not stop executing until this condition is false. Every time the action and step code is completed, the condition of the loop is checked again. If the condition is still true, we execute the action and step again.

Choosing the correct condition for a loop is based on what we want to achieve and generally requires coordination between the setup and step parts of the loop. If we want to repeat a piece of code a certain number of time, then the condition will be related to the count of how many times the loop has executed. For example, if we wanted a loop to execute 5 times, in the setup we would need to create a variable to count how many times the loop has executed (`i`) and in the step we would need to increase this by one every time the loop is executed. Finally the condition of the loop would be `x != 5` or `x < 5`.

The Action

The action of the loop is the easiest part, this is simply the code that we want to be repeated. This can be a single statement, or a large group of statements. For example, if we wanted to have a loop that calculates many grades from results, then all of the code of the converter would be the action of the loop.

The Step

The step is one of the most important parts of the loop and is the easiest place to make a mistake. The step is some code that will cause the loop to stop at the correct time. This is generally done by changing some value so that the condition is eventually no longer true.

Depending on the condition that we write, different steps will be required. Just like the action part, this will be executed every time the loop executes.

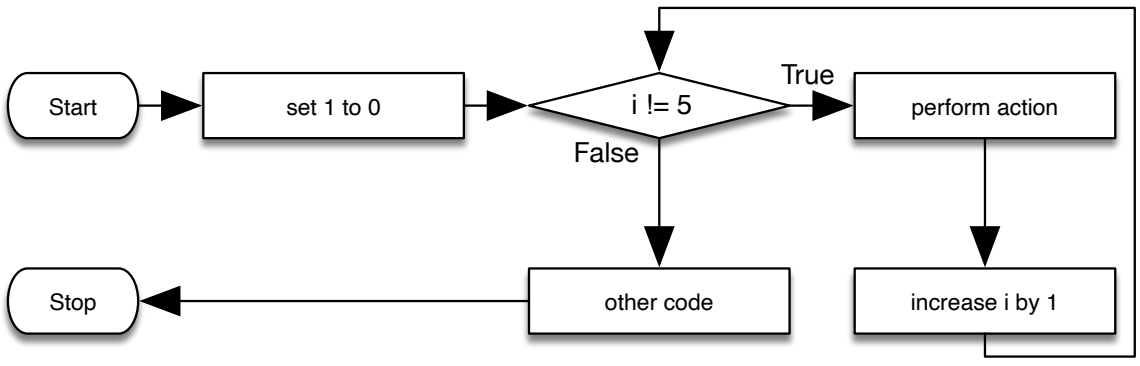
Example Loop Parts

Assuming that we are trying to create a loop that executes 5 times, what do we write for the setup, condition and step? If a loop needs to execute a certain number of times, then we need to count how many times it has executed, this will require a variable. Generally, we will start this variable (`i`) with the value 0, because the loop has not executed yet when the variable is created.

We want the loop to execute 5 times, so we need a condition that is false when the counter reaches 5. This means it will be true 5 times and then false. The condition `i != 5` works for this purpose.

Finally, we need to define the step part. In this we need to make some change that will make the condition eventually become false. Because we are counting, the easiest solution is to increase the value of the variable i by 1. E.g. `i = i + 1`.

Having looked in more detail at how loops work, now we can see the updated flowchart for loops.



This loop will execute 5 times, performing the action each time, before it continues to execute some other code.

5.2 While loop

There are several different conditional statements that can be used to repeat sections of code. While there are some small differences between these statements, we can achieve the same tasks using just one type of loop. We will be studying the while loop. The syntax of the while loop in Python is shown below:

```
setup
while condition:
  action
  step
```

Listing 5.2: While loop in Python

It should be noted that just like the `if` statement in the last chapter we use indentation to highlight which parts of the code will be repeated in the `while` loop.

Lets look at a real example of a simple `while` loop. This loop repeats the code to display the message `"hello"` on the screen 5 times.

```
i = 0
while i < 5:
  print("hello")
  i = i + 1
```

Listing 5.3: Example of a while loop

To fully understand how loops work, we need to look at how the variables change as the loop is executed. This table shows the values of the variables and expressions as the loop executes

Table 5.1: Steps when loop is being executed

Stage	Value of `i`	Value of `i < 5`	Result
Before the loop starts	`0`	`True`	Loop executed
After the loop has executed 1 time	`1`	`True`	Loop executed
After the loop has executed 2 times	`2`	`True`	Loop executed
After the loop has executed 3 times	`3`	`True`	Loop Executed
After the loop has executed 4 times	`4`	`True`	Loop Executed
After the loop has executed 5 times	`5`	`False`	Loop **not** executed

5.3 Standard While Loop

Most of the time, when we are creating a loop, we will know how many time it should execute. This type of loop is very easy to write. Assume that we want to execute some piece of code 100 times (Once for every student in a class).

The setup for this loop would be to create a variable that starts at 0 e.g. `i = 0`. The step of the loop would be to increment this variable e.g. `i = i + 1;`. Finally the condition would be the same variable strictly less than the number of times we want to execute e.g. `i < 100`.

Now lets apply this knowledge to the grade calculation program that we created earlier. All of the conversion code needs to be repeated, we just need to add the loop around it. One point to be careful of is making sure that our indentation is correct. There will be many places in the code where we must indent twice, once for the while loop and once for the `if/elif/else` statements.

```
i = 0
while i < 100:
  percent = int(input("Please enter the percentage: "))
  if percent > 70:
    print("A")
  elif percent > 60:
    print("B")
  elif percent > 50:
    print("C")
  elif percent > 40:
    print("D")
  else:
    print("F")
  i = i + 1
```

Listing 5.4: A program to convert 100 results to grades

This loop will execute exactly 100 times. Each of those times all of the code inside the loop will be executed, so a message will be printed to the user asking them to enter a percentage and the corresponding grade will be printed to the screen.

5.4 Non-Standard While Loops

The previous examples are very useful when we know exactly how many times that a loop should execute. However, sometimes we will not know exactly how many times a loop is expected to execute. The number of times may be based on information the user enters, or on information that is read from a file or even based on how much information there is in a file to read.

To illustrate this example, we will create a guessing game. The idea of the game is that the program will contain a secret number and the user will make multiple attempts to guess this number. When the user guesses, we will tell them if their guess is correct, too high or too low.

For this type of game, it is impossible to know how many times the loop will execute. The user may get the correct answer on the first attempt, or after 13 guesses. In this case, we must write the condition so that the loop stops only when the user guesses the correct number. First, lets look at the program without the loop.

```
number = 23
guess = int(input("Guess a number between 1 and 100: "))
if number > guess:
  print("Too Low")
elif number < guess:
  print("Too High")
else:
  print("Correct")
```

Listing 5.5: Program that gives the user a single guess at a secret number

The program is not too complicated, all we need to do is to add a loop to make the process repeat until the user guesses the correct answer. This means that we need to write a condition for the loop that defines when it should execute. The easiest way to think about this is to start with the condition when the user guesses the correct answer. In this case the value of `number` and `secret` will be the same or `number == guess` will be true.

However, this is the condition for when the loop should stop. In reality, we want a condition that is true when the game is not finished. In this case we just need to reverse the condition. That gives us `number != guess`, or we can use the not operator and use `not(number == guess)`.

In this case, the step that happens every time the loop executes is not incrementing a value. It is supposed to be some code that will eventually cause the loop to stop. But the only way the game can stop is for the user to guess the correct number. That means that the user taking another guess is the step in this loop.

```
number = 23
guess = 0
while guess != number:
  guess = int(input("Guess a number between 1 and 100: "))
  if number > guess:
    print("Too Low")
  elif number < guess:
    print("Too High")
  else:
    print("Correct")
```

Listing 5.6: Guessing game with multiple attempts

5.5 Common Mistakes

There are several common mistakes that can be made while creating loops. It is important that we can recognise them so that we can fix them easily.

Off-by-one Errors

These are a small mistake in your code that causes a loop to execute one time too many or one time too few. These generally relate to a small error in either the condition or the setup part of the loop.

Starting at 1

Most loops will use a variable to count how many times the loop has executed. If we give this variable a value of 1 in the setup, this may cause the loop to execute one time less than it should.

Incorrect Comparison

The comparison operator that you use in the condition defines when the loop will execute. A mistake in the condition can cause a loop that never starts, never stops or executes an incorrect number of times.

A very common mistake is to mix up the use of $<$ and $<=$ or $>$ and $>=$. Using the incorrect comparison will mean the loop executes one time too many or one time too few.

Infinite Loops

A very common error is to create an infinite loop. As the name suggests, this is a loop that can never finish. This can indicate that there is a problem with the condition of the loop. However, it is more commonly caused by forgetting to have a step in your loop.

Stopping an Infinite Loop

When you execute a program containing an infinite loop, the program can never finish. When this happens, we must tell the interpreter or program to stop. This is done by typing the key combination `Ctrl-C`.

5.6 Code Tracing

Loops are the most complex statement that we have learned so far. Loops are a fundamental part of most programming languages and it is important that we fully understand how they work. To help with this we will trace an example program that includes a loop, this will show us that loops might seem complicated, but in fact are simply following a set number of steps in a certain order.

```
1 i = 0
2 total = 0
3 while i < 3:
4   number = int(input("Please enter a number: "))
5   total = total + number
6   i = i + 1
7 print("The sum of those numbers is", total)
```

Listing 5.7: Program to sum 3 numbers

This program is much longer than previous examples that we have used for program tracing. Executing the full program results in the execution of 16 lines of code. We will first look at each line of the program, then review using a less detailed format.

The program interacts with the user by reading three numbers from the keyboard and summing them. For this tracing exercise we will assume that the user has typed the the following numbers `5, 7, and 3` at the requests.

Lines 1 and 2

Lets start with the first two lines of code to be executed, line 1 and 2. The breakdown of these lines are shown in Tables 5.2 and 5.3 and the memory map after these lines of code is shown in Table 5.4. Here we are assigning a variable to remember how many times we have looped (`i`) and the variable `total` to remember the sum of the numbers entered by the user.

Table 5.2: Line 1: `i = 0`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`i = 0`	`0`	`int`	2

Table 5.3: Line 2: `total = 0`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`total = 0`	`0`	`int`	2

Table 5.4: Memory map after executing line 1 and 2

Name	Value	Type
`i`	`0`	`int`
`total`	`0`	`int`

Line 3

It is at line 3 that we start executing the loop. We will see as we progress that we follow the process of checking the condition then executing all of the statements inside the loop, then checking the condition again. This process will be repeated until we have a result of `False` for the condition of the loop.

Table 5.5 shows the breakdown of the code for checking the condition of the loop. In this code, the computer checked the value of the variable `i` was less than 3, as this is true, the next line of code to be executed is line number 4. No variables were modified so we will not show an updated memory map.

Table 5.5: Line 3: `while i < 3 :`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	`0`	`int`	
2	Comparison	`0 < 3`	`True`	`bool`	
3	While Condition Test	`while True:`			4

Line 4

Here, the `input` and `int` functions read the first number that was typed by the user (`5`) and the variable `number` is assigned this value. Table 5.6 shows the steps completed when executing this code, again highlighting the type of the value entered by the user and the type of the value that we remember in the variable `number`. Table 5.7 shows the memory map after this line has been executed.

Table 5.6: Line 4: `number = int(input("Please enter a number: "))`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`input("Please enter a number: ")`	`"5"`	`str`	
2	Call Function	`int("5")`	`5`	`int`	
3	Assignment	`number = 5`	`5`	`int`	5

Table 5.7: Memory map after executing line 4

Name	Value	Type
`i`	`0`	`int`
`total`	`0`	`int`
`number`	`5`	`int`

Line 5

The next line of code takes the current value of the variable `total` (0) and adds it to the number that was just read from the user (5). Table 5.8 shows the order that these steps are completed in.

As a result, the value 5 is now stored in the variable `total` which is shown in the memory map in Table 5.7.

Table 5.8: Line 5: `total = total + number`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`total`	0	int	
2	Load Variable	`number`	5	int	
3	Addition	`0 + 5`	5	int	
4	Assignment	`total = 5`	5	int	6

Table 5.9: Memory map after executing line 5

Name	Value	Type
`i`	0	int
`total`	5	int
`number`	5	int

Line 6

To know when to stop the loop, we need to count how many times the code inside the loop has been executed. We are using the variable `i` for this purpose. This line of code takes the current value of the variable `i` and adds 1 to it and stores the result back in the variable `i`. The breakdown of this line of code is shown in Table 5.10 and the memory map after this line is competed is shown in Table 5.11.

Table 5.10: Line 6: `i = i + 1`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	0	int	
2	Addition	`0 + 1`	1	int	
3	Assignment	`i = 1`	1	int	3

Table 5.11: Memory map after executing line 6

Name	Value	Type
`i`	1	int
`total`	5	int
`number`	5	int

At this point we have reached the end of the code in the while loop. When we reach this point the next step is always to check the condition of the loop again.

Line 3 - Second Time

Table 5.12 shows the breakdown of line 3. As we can see, the condition of the loop is still `True`, (`i` is still less than 3) so we will execute the contents of the loop again. Again as there was no assignment, we do not need to update the memory map.

Table 5.12: Line 3: `while i < 3 :`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	`1`	`int`	
2	Comparison	`1 < 3`	`True`	`bool`	
3	While Condition Test	`while True:`			4

Line 4 - Second Time

This line of code reads the second number that was typed by the user (7). Table 5.13 shows the steps completed when executing this code and Table 5.7 shows the memory map after this line has been executed.

Table 5.13: Line 4: `number = int(input("Please enter a number: "))`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`input("Please enter a number: ")`	`"7"`	`str`	
2	Call Function	`int("7")`	`7`	`int`	
3	Assignment	`number = 7`	`7`	`int`	5

The value is stored in the variable `number`.

Table 5.14: Memory map after executing line 4 (again)

Name	Value	Type
`i`	`1`	`int`
`total`	`5`	`int`
`number`	`7`	`int`

Line 5 - Second Time

Next, we need to add the number we have just read from the user to the current value of the variable `total`. Table 5.15 shows the breakdown of the execution and Table 5.16 shows the memory map afterwards. As such, the value of the variable `total` is updated to 12.

Table 5.15: Line 5: `total = total + number`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`total`	`5`	`int`	
2	Load Variable	`number`	`7`	`int`	
3	Addition	`5 + 7`	`12`	`int`	
4	Assignment	`total = 12`	`12`	`int`	6

Table 5.16: Memory map after executing line 5 (again)

Name	Value	Type
`i`	`1`	`int`
`total`	`12`	`int`
`number`	`7`	`int`

Line 6 - Second Time

The last line of code in the loop is again to increase our count of the number of times that the loop has been executed. We take the current value of the variable `i`, add 1 to it and store the value in the variable `i` again. Table 5.17 shows the breakdown of the execution and Table 5.18 shows the memory map afterwards. As such, the value of the variable `i` is updated to 2.

Table 5.17: Line 6: `i = i + 1`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	1	int	
2	Addition	`1 + 1`	2	int	
3	Assignment	`i = 2`	2	int	3

Table 5.18: Memory map after executing line 6 (again)

Name	Value	Type
i	2	int
total	12	int
number	7	int

Again, as we have reached the last statement in the body of the loop the next step is to return to line 3 and check the condition of the loop again.

Line 3 - Third Time

Table 5.19 shows the breakdown of line 3. As we can see, the condition of the loop is still `True`, (`i` is still less than 3) so we will execute the contents of the loop again. Again as there was no assignment, we do not need to update the memory map.

Table 5.19: Line 3: `while i < 3 :`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	2	int	
2	Comparison	`2 < 3`	True	bool	
3	While Condition Test	`while True:`			4

Line 4 - Third Time

This line of code reads the second number that was typed by the user (3). Table 5.20 shows the steps completed when executing this code and Table 5.21 shows the memory map after this line has been executed.

Table 5.20: Line 4: `number = int(input("Please enter a number: "))`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Call Function	`input("Please enter a number: ")`	"3"	str	
2	Call Function	`int("3")`	3	int	
3	Assignment	`number = 3`	3	int	5

Table 5.21: Memory map after executing line 4 (again)

Name	Value	Type
`i`	2	`int`
`total`	12	`int`
`number`	3	`int`

Line 5 - Third Time

Next, we need to add the number we have just read from the user to the current value of the variable `total`. Table 5.22 shows the breakdown of the execution and Table 5.23 shows the memory map afterwards. As such, the value of the variable `total` is updated to 15.

Table 5.22: Line 5: `total = total + number`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`total`	12	`int`	
2	Load Variable	`number`	3	`int`	
3	Addition	`12 + 3`	15	`int`	
4	Assignment	`total = 15`	15	`int`	6

Table 5.23: Memory map after executing line 5 (again)

Name	Value	Type
`i`	2	`int`
`total`	15	`int`
`number`	3	`int`

Line 6 - Third Time

Finally, we increase the count of the number of times the loop has executed again. Table 5.24 shows the breakdown of the execution and Table 5.25 shows the memory map afterwards. As such, the value of the variable `i` is updated to 3.

Table 5.24: Line 6: `i = i + 1`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	2	`int`	
2	Addition	`2 + 1`	3	`int`	
3	Assignment	`i = 3`	3	`int`	3

Table 5.25: Memory map after executing line 6 (again)

Name	Value	Type
`i`	3	`int`
`total`	15	`int`
`number`	3	`int`

We have again reached the end of the code that is in the loop, so the next step is to return to the condition of the loop on line 3.

Line 3 - Fourth Time

Table 5.26 shows the breakdown of execution for line 3 again. This time we find that the value of the variable `i` is now `3`. As the condition of the loop is `i < 3`, the condition is now `False` and we are finished executing the loop.

Table 5.26: Line 3: `while i < 3 :`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`i`	`3`	`int`	
2	Comparison	`3 < 3`	`False`	`bool`	
3	While Condition Test	`while False:`			7

The next line of code we will execute is line 7, as it is the first line of code after the loop.

Line 7

Table 5.27 shows the breakdown of this line of code. The result we have calculated is output to the screen with some accompanying text to explain it. It is important to note again that the process of printing to the screen is not the same as returning a value and this is why the result and return type of the function are None.

Table 5.27: Line 7: `print("The sum of those numbers is", total)`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`total`	`15`	`int`	
2	Call Function	`print("The sum of those numbers is", 15)`	`None`	`None`	Finished

As this is the last line of code in the file, we are now finished.

Compact Form

As it took almost 7 pages to fully trace a single small program, we will now look at the same tracing done in a more compact form. Fully expanding and explaining how each line of code is executed is useful when we are first learning about how programs execute. When programmers are a little more experienced it is assumed that you are able to understand these steps in your head and we will instead focus on the overall execution of the program. Having learned about conditional statements, now the focus is on the order that the lines of code are executed in and how the values of variables are changed as this happens.

This compact form will have a single line in the table for each line of code that is executed, so we will not expand the order of execution of each line of code. When the code being executed is shown, any expressions in the code should be fully evaluated. Table 5.28 shows the compact form of tracing for the same program shown in Listing 5.7.

Additionally, the memory map is included in the table, each variable used in the program is shown as a column with the header labelled with its name. When the value of a variable is changed, it is highlighted in green and the new value is shown in that line of code and all following it until the value is changed again. When a value of a variable is used an expression in the current line of code, it is highlighted with a light blue colour. This is not the case when the value is being assigned.

This condensed format allows us to see both the result of the program as well as how the control flow of the program is changed by using a loop. For all following code tracing examples and exercises we will be using this format.

Table 5.28: Compact representation of code tracing a program

Line	Code Executed	i	number	total	Next
1	i = 0	0	?	?	2
2	total = 0	0	?	0	3
3	while True	0	?	0	4
4	number = 5	0	5	0	5
5	total = 5	0	5	5	6
6	i = 1	1	5	5	3
3	while True	1	5	5	4
4	number = 7	1	7	5	5
5	total = 12	1	7	12	6
6	i = 2	2	7	12	3
3	while True	2	7	12	4
4	number = 3	2	3	12	5
5	total = 12 + 3	2	3	15	6
6	i = 3	3	3	15	3
3	while False	3	3	15	7
7	print("...", 15)	3	3	15	-

There is one thing that we can note from this example. No matter how many times a loop executes, the condition will be checked one more time. This is because the condition must be checked before the start of each loop and also once when we determine that the loop has finished.

5.7 Nested Loops

As we learned in the previous chapter, we can put any of the statements that we learned about inside an if statement. As the while loop is also a compound statement, the same is true of it. We can put if statements and other loops inside of a while loop. When we place a loop inside of another loop it is called a nested loop. Nested loops can be very useful when dealing with data that is in a table format or processes like reading data from a file.

When we place one loop inside another, the nested loop is executed repeatedly. If we have an outer loop that executes twice and inside it a loop that executes 3 times, then the whole nested loop is executed twice and any code inside it is executed 6 times. We will look at how an example program containing a nested loop is executed.

The following program prints the counters of both loops each time it is executed. This shows us how the variables change as we execute the loop.

```
1  i = 0
2  while i < 2:
3    j = 0:
4    while j < 3:
5      print(i, j)
6      j = j + 1
7    i = i + 1
```

Listing 5.8: Example of Nested Loops

A lot of steps are executed in this code, each time we are checking the condition of a loop there is an added notation to tell us if it is the outer loop or the inner loop.

Lets look at the important parts of the program flow in that program. Everything inside the outer loop was executed two times, this included a nested loop. If the nested loop was on its own,

Table 5.29: Execution of code containing nested loops

Line	Code Executed		i	j	Next
1	`i = 0`		0	?	2
2	`while True:`	(outer loop)	0	?	3
3	`j = 0`		0	0	4
4	`while True:`	(inner loop)	0	0	5
5	`print(0, 0)`		0	0	6
6	`j = 1`		0	1	4
4	`while True:`	(inner loop)	0	1	5
5	`print(0, 1)`		0	1	6
6	`j = 2`		0	2	4
4	`while True:`	(inner loop)	0	2	5
5	`print(0, 2)`		0	2	6
6	`j = 3`		0	3	4
4	`while False:`	(inner loop)	0	3	7
7	`i = 1`		1	3	2
2	`while True:`	(outer loop)	1	3	3
3	`j = 0`		1	0	4
4	`while True:`	(inner loop)	1	0	5
5	`print( 1, 0)`		1	0	6
6	`j = 1`		1	1	4
4	`while True:`	(inner loop)	1	1	5
5	`print(1, 1)`		1	1	6
6	`j = 2`		1	2	4
4	`while True:`	(inner loop)	1	2	5
5	`print(1, 2)`		1	2	6
6	`j = 3`		1	3	4
4	`while False:`	(inner loop)	1	3	7
7	`i = 2`		2	3	2
2	`while False:`	(outer loop)	2	3	-

then the code inside it would have been executed three times. However, as the loop was nested inside the outer loop, the code inside the loop was actually executed six times (2 * 3). The final output of the code is:

```
0 0
0 1
0 2
1 0
1 1
1 2
```

Listing 5.9: Output of the program

Removing Nesting

It is possible to remove nested loops from your code, but usually this requires much more complicated logic for your condition and step. The example we have just looked at can be modified in this way:

```
i = 0
j = 0
while i < 2:
  print(i, j)
  j = j + 1
  if j == 3:
    i = i + 1
    j = 0
```

Listing 5.10: Example Without Nested Loops

This example uses the same condition as the outer loop in the nested loop example. However, the step that we make in each execution of the loop is much more complicated. Instead of having each loop count independently, we need to increase the values of both and `i` and `j` at the correct times and return the value of `j` to zero at the correct time too.

Chapter 6

Mutability and Immutability

6.1 Mutability

On concept to be aware of when dealing with data is that of mutability. This describes the ability to change something, if it can be changed it is mutable, if it cannot be changed it is immutable.

All of the values that we have seen in our programs so far have been immutable. This includes integers, floating point numbers, boolean values and strings. Once created, none of these values can be changed.

That might seem odd, as we have had variables that change in most of our programs. It made no difference in these programs, but instead of values that change, each value was replaced by a new value instead of changing. From the perspective of how the programs we have written work, this makes no difference.

For example, if we have a variable `x` that contains an integer value, it is not important if code like `x = x + 1` actually changes the current value of `x` or replaces it with a new value. In both cases, when the variable `x` gets used again it will contain the updated value.

Everything is a Reference

Every variable we have used in our programs is what is known as a reference. This means that instead of the name of a variable being associate with a value, it is instead associated with an address where the value can be found. Practically, in the programs we have been writing this makes no difference, instead of changing values we have been changing addresses.

However, this would be different if these values could change. Lets look at how this works for immutable values by tracing the following example.

```
1 x = 153
2 y = x
3 y = y + 1
```

Listing 6.1: Copying Immutable Variables

Line 1

Table 6.1: Line 1: `x = 153`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`x = 153`	`153`	`int`	2

The first statement is a normal assignment, x is given the value 153. Normally we would show this in the memory map with the name `x` beside the value 153. However, this time we will instead show something a little closer to the reality. In the memory map, we see the name `x` beside the address 2. This shows us where in the memory of the computer the actual value of `x` is stored. The table beside it is showing an example of memory, a sequence of numbered places to remember information.

Table 6.2: Memory map and contents of Memory after executing line 1

Name	Address	Type
`x`	2	`int`

Address	0	1	2	3	4	5	6	7	8	9
Value			153							

Line 2

Table 6.3: Line 2: `y = x`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`y = 153`	153	`int`	3

The second statement is again an assignment. However, in this case while we are showing the value 153 being copied to the variable `y`, what is actually being copied is the address of the value remembered by the variable `x`. The result when we use the variables `x` or `y` is still that we see the value 153.

Table 6.4: Memory map and contents of Memory after executing line 2

Name	Address	Type
`x`	2	`int`
`y`	2	`int`

Address	0	1	2	3	4	5	6	7	8	9
Value			153							

Line 3

Table 6.5: Line 3: `y = y + 1`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`y = 154`	154	`int`	-

This final statement is where the immutability of these values is important. Because the value in address 2 cannot be changed, to remember the number 154 it must be stored somewhere else in memory. The address remembered for the name `y` is then updated to match this new value. But no change is made to `x`, it still points to the address 2 (and the value 153).

Table 6.6: Memory map and contents of Memory after executing line 3

Name	Address	Type
`x`	2	`int`
`y`	6	`int`

Address	0	1	2	3	4	5	6	7	8	9
Value			153				154			

Mutable Values

This all worked exactly as we expected, we could very easily forget the fact that our variables hold references instead of values and there would be no issue. However, lets take another look at what might happen if the integer values we are using were instead mutable. For simplicity, we only need to look at the final line of code as this is where the difference will be.

Table 6.7: Line 3: `y = y + 1`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Assignment	`y = 154`	`154`	`int`	-

The execution of the final statement is the same as above, but the differences come from how this is stored in memory. Because the value in address 2 is mutable (can be changed) we update the value to 154. The address remembered for the names `y` and `x` is still 2. This means that if we print the value of x, it will show 154 instead of the expected 153.

Table 6.8: Memory map and contents of Memory after executing line 3

Name	Address	Type
`x`	2	`int`
`y`	2	`int`

Address	0	1	2	3	4	5	6	7	8	9
Value			`154`							

This is one of the reasons that all of the types we have used so far are immutable. It prevents this type of problem in our programs. However, as we are introduced to more data types, some of them will be mutable and we must remember this when we are writing our programs because the variables will act differently than we expect from previous experience.

Ids of Variables

One tool we can use to have a closer look at this is the function `id`. This function takes a variable as a parameter and returns it's memory address. The address returned will be different every time the program is run and will usually be a much larger number than in my example.

To demonstrate this, we will print the address of both the variables `x` and `y` both before and after `y` is increased by 1.

```
1 x = 153
2 y = x
3 print(id(x), id(y))
4 y = y + 1
5 print(id(x), id(y))
```

```
1 9761056 9761056
2 9761056 9761088
```

Figure 6.1: Example of `id` function and Output

We can see that in the first line of output (shown on the right) the numbers are the same. This tells us that both `x` and `y` are referencing the same place in memory. The second line of output shows that when we change `y` it is now referencing a different place in memory (the address has changed).

We will return to a similar example of this when we are working with a mutable data type.

6.2 Sequences

Working with lists of data is very common in programming. We might write a program to calculate the average grade of the students in a class, or to convert a list of results into grades. In program-

ming we use data structures when we want to remember many pieces of data together. There are many many different data structures that can be used such as lists, tuples, dictionaries, and sets. In this chapter we are going to study several of these data structures that work in similar ways. These data structures are known as sequences.

Sequences in Python describes three data structures, lists, tuples and strings (str). These data structures are similar in some ways, but different in others. Lets have a look at these differences described in a table. Then we will discuss the meaning of each row.

Table 6.9: Description of operations for different sequence data structures

Operation	List	Tuple	String
Type of data	Any	Any	Characters only
Same Type	Mixed Types	Mixed Types	Same
Mutable	Mutable	Immutable	Immutable
Size Change	Variable Size	Fixed Size	Fixed Size
Insert items	Yes	No	No
Replace items	Yes	No	No
Remove items	Yes	No	No
Check if Empty	Yes	Yes	Yes
Find Length	Yes	Yes	Yes
Access items	Yes	Yes	Yes
Iterate items	Yes	Yes	Yes

Type of data

This refers to what we can remember in the data structure. For both tuple and list, we can use these to remember any type of information, but a string can only contain characters. This makes sense for the string type as text is only made up of characters.

Same type

It is common in many programming languages for a data structure to only allow a single type, this would mean that all values in a particular data structure would be the same type. This is not the case for lists or tuples we could have strings, integers, floating point numbers and other values all in the same list or tuple. As strings can only contain character, they must all be the same type.

Mutable

The next 5 rows are all related. The list data structure is mutable while the tuple and string data types are immutable. This means that in a list we can add new items, remove items and replace items and as a consequence the size of the list can change. Here size is describing how many things are in the list. As the others are immutable, nothing can be changed and the size will always remain the same.

Functionality

The final 4 rows describe some of the functionality that we can do with each of these different data structures. We can check if the sequence is empty (contains no values), we can find its length (how many values are in it), access the individual values in the sequence and iterate (loop) through all of the items in the sequence.

We will look all of these as we learn about each of these different data structures.

Chapter 7

Lists

The first data structure we will look at is the list. A list is a container for multiple values. We can use a list to store many pieces of information using a single variable name.

A list is a linear data structure. This means that all of the values that are contained in the list are stored in a particular order. We use a number called an **index** to identify which value in the list we want.

Indexes are numbered in increasing order starting at 0. This means the first item in the list as the index 0, the second item has the index 1 and so on.

Table 7.1: Representation of a list containing 12 integer values

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	31	28	31	30	31	30	31	31	30	31	30	31

Table 7.1 shows a represenation of a list containing 12 integer values. The table shows both the index and the value associated with it. These numbers are a representation of the number of days in each month in a (non-leap) year.

7.1 List Literal

We can define a literal value of a list in our code (just like other values), but we need to use a special syntax. Writing a list in our code is similar to writing a string, we need to place some characters to tell python that this is a list. A list literal starts with an opening square bracket (`[`), this is followed by the values we want in the list (separated by commas) and then finally a matching closing square bracket (`]`). We can create an empty list (containing no items) just using a pair of square brackets (`[]`).

```
1 [ val1, val2, val3 ]
```

Listing 7.1: Syntax for list literal

The list created can be used in an expression, as a parameter to a function or remembered in a variable using assignment. This example shows how we would create the list from the example above in code and remember it using the variable name `months`.

```
1 months = [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Listing 7.2: Assigning literal list to a variable

7.2 Using Lists

Accessing Values

Using this code we can create a single variable containing many values. But how can we access the value that we want from the list? To access a single value in the list, we use the index placed between a set of square brackets. For example, using the example list above we would use the code `months[0]` to access the first item in the list, `months[1]` to access the second item in the list and so on. These can be used in an expression, as parameters to a function or anywhere you can use a value.

Changing Values

If we want to change the value of one of the items in the list, we use the assignment operator. Again, we need to specify which index we want to change. For example, if I wanted to change the value of the second item in the example list to 29 (to match a leap year) we would use the code `months[1] = 29`.

Table 7.2: Representation of a list containing 12 integer values after a value is updated

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	31	29	31	30	31	30	31	31	30	31	30	31

Removing Values

If we want to remove a value from a list we can use the keyword `del`. This should be followed by the name of the list and the index in a set of square brackets. For example, the code `del months[1]` would remove the value 29 from our list. It should be noted that there are a few consequences to this, the size of the list has changed (from 12 to 11) and the index of all values after this value have changed (they are all 1 less).

Table 7.3: Representation of a list containing 12 integer values after a value is removed

Index	0	1	2	3	4	5	6	7	8	9	10
Value	31	31	30	31	30	31	31	30	31	30	31

Adding Values

There are two ways of adding new values to the list. The first allows us to choose somewhere in the middle of the list to insert the value. The second way adds the new value to the end of the list so it is always the last value. The syntax for both of these methods is a little different than we have seen before and we will learn about it in more detail in chapter 11.

Inserting a new value into the middle of a list is done using a special type of function called a method. This method is connected to the list, so the name of the list must also be included in the syntax. The method is named `insert` and requires two parameters. The first parameter must be an integer value and represents the index where we wish to insert the value. The second parameter is the value we want to insert.

The syntax is the name of the list, followed by a dot (.) and the name of the method with the parameters in brackets. e.g. `lst_name.insert(3, 55)`. If we wanted to insert the value 41 at index 4 of the example list we have been using we would use the code `months.insert(4, 41)`

Again we have consequences of this action. The size of the list has changed (from 11 to 12) and the index of all values after this value have changed (they are all 1 more).

Table 7.4: Representation of a list containing 12 integer values after a value is inserted

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	31	31	30	31	41	30	31	31	30	31	30	31

The alternative is to add the new value at the end of the list. Again, this action is done using a method. The method is named `append` and requires one parameter. The parameter is the value we want to add to the list.

The syntax is the name of the list, followed by a dot (.) and the name of the method with the parameters in brackets. e.g. `lst_name.append(55)`. If we wanted to insert the value 45 to the example list we have been using we would use the code `months.append(45)`.

Table 7.5: Representation of a list containing 12 integer values after a value is appended

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value	31	31	30	31	41	30	31	31	30	31	30	31	45

Size of a List

The size of a list (often called length) is the number of items in the list. Because we can add and remove items from the list, the size can change and we may need our code know what size a list is. We can use the function `len` to find the size of a list. We pass the name of the list (or any expression with a list as a result) as a parameter and the function will return an integer value representing the number of items in the list. We will often want to remember this in a variable.

For example, if we wanted to find the size of the example list, we would use the code `len(months)`. If we wanted to remember that in a variable (called `size`), we would use `size = len(months)`.

7.3 Example

To help get an understanding of the use of lists in programming, we are going to study an example of a task that requires a list to complete. The task is to read 10 values from the user, calculate the average and also count how many of the values are below the average.

Reading values and calculating an average can be done without using a list, because we only need to remember the sum of the numbers the user enters. However, to count the number of values that are smaller than the average, we must remember all of the values. This requires a list.

```
1 numbers = []
2 i = 0
3 while i < 10:
4     num = int(input("Enter a number: "))
5     numbers.append(num)
6     i = i + 1
```

Listing 7.3: Reading 10 values into an List

If we were very clever, we would realise that we can change this code so that while we are reading the numbers from the user we can also calculate the average. However, in this example lets look at the code to calculate the average of the numbers in the list.

```
total = 0
i = 0
while i < len(numbers):
  total = total + numbers[i]
  i = i + 1
average = total / i
```

Listing 7.4: Calculating the average of the values in the List

Now that we have calculated the average of all of the numbers, we can complete the process of counting how many of the values are smaller than the average. One technique to note is the use of the function `len` in the condition of the while loop. This code will work correctly for a list containing 1 value or 1000 values. This is because the condition will always compare our counting variable (`i`) to the number of items in the list.

```
small = 0
i = 0
while i < len(numbers):
  if numbers[i] < average:
    small = small + 1
  i = i + 1
print(small, "numbers were smaller than the average of", average);
```

Listing 7.5: Counting how many of the numbers are smaller than the average

Because we have used a list, each part of the program can be written to use a loop instead of having to write each piece of code 10 times. This is very convenient when working with large lists of data. If this example required 100 numbers instead, all we would have to change is the number in the condition of the first loop.

Tracing - Input

First lets look at what happened in the first part by tracing through the program. To prevent this taking two long, we will reduce the example to three numbers and assume that the user enters the number 4, 5, and 23.

After this code is completed, the three numbers typed by the user are stored in the list. As they were added to the the end of the list in the order that they were entered, this is the order that they will be in the list too.

Table 7.6: Trace of the execution of the input of numbers into the list

Line	Code	numbers 0	1	2	i	num	Next
1	`numbers = []`						2
2	`i = 0`				0		3
3	`while True:`				0		4
4	`num = 4`				0	4	5
5	`numbers.append(4)`	4			0	4	6
6	`i = 1`	4			1	4	3
3	`while True:`	4			1	4	4
4	`num = 5`	4			1	5	5
5	`numbers.append(5)`	4	5		1	5	6
6	`i = 2`	4	5		2	5	3
3	`while True:`	4	5		2	5	4
4	`num = 23`	4	5		2	23	5
5	`numbers.append(23)`	4	5	23	2	23	6
6	`i = 3`	4	5	23	3	23	3
3	`while False:`	4	5	23	3	23	-

Tracing - Average

Next we will look at the steps for averaging the numbers in the list.

Table 7.7: Trace of the execution of the averaging of numbers in a list

Line	Code	numbers 0	1	2	i	total	average	Next
1	`total = 0`	4	5	23				2
2	`i = 0`	4	5	23	0	0		3
3	`while True:`	4	5	23	0	0		4
4	`total = 4`	4	5	23	0	4		5
5	`i = 1`	4	5	23	1	4		3
3	`while True:`	4	5	23	1	4		4
4	`total = 9`	4	5	23	1	9		5
5	`i = 2`	4	5	23	2	9		3
3	`while True:`	4	5	23	2	9		4
4	`total = 32`	4	5	23	2	32		5
5	`i = 3`	4	5	23	3	32		3
3	`while False:`	4	5	23	3	32		6
6	`average = 10.667`	4	5	23	3	32	10.667	-

We can see that again the variable that we are using to count the executions of the loop (`i`) has also been used to calculate which index we should be checking. The numbers from the list are added to the variable `total` one at a time, each time the loop executes a number is added. Finally, when the loop is finished we calculate the average of the numbers by dividing by `i` because this number will match the number of values that we have added to the variable `total`.

Tracing - Smaller

Next we will look at the steps for counting the number of values that were smaller than the average.

The structure of the code is again very similar to the previous examples, we use the index to access each value in the list one by by as the loop executes. Every time we find a value in the list

Table 7.8: Trace of the execution of finding the smallest number in a list

Line	Code	numbers 0	1	2	i	small	average	Next
1	`small = 0`	4	5	23	3	0	10.667	2
2	`i = 0`	4	5	23	0	0	10.667	3
3	`while True:`	4	5	23	0	0	10.667	4
4	`if True: # 4 < 10.667`	4	5	23	0	0	10.667	5
5	`small = 1`	4	5	23	0	1	10.667	6
6	`i = 1`	4	5	23	1	1	10.667	3
3	`while True:`	4	5	23	1	1	10.667	4
4	`if True: # 5 < 10.667`	4	5	23	1	1	10.667	5
5	`small = 2`	4	5	23	1	2	10.667	6
6	`i = 2`	4	5	23	2	2	10.667	3
3	`while True:`	4	5	23	2	2	10.667	4
4	`if False: # 23 < 10.667`	4	5	23	2	2	10.667	6
6	`i = 3`	4	5	23	3	2	10.667	3
3	`while False:`	4	5	23	3	2	10.667	7
7	`print(2,"...",10.667)`	4	5	23	3	2	10.667	-

that is smaller than the average we increase the variable `small` by one. When the loop is finished we will have the total of these values stored in this variable.

7.4 List Operators

Python also has operators that can be used on lists. In this section we will study two operators that can be used on lists. These work in the same way as the operators for strings.

Table 7.9: List Operators

Name	Code	Explanation	Example
Replication	`*`	Replicates a list a number of times, where x is a list and y is an integer	`x * y`
Concatenation	`+`	Joins two lists together	`x + y`

The first operator, concatenation, can be very useful when working with multiple lists. What it does is copy two lists and create a new list containing the contents of both. For example, the expression `[ 1, 2 ] + [ 4, 5 ]` would have the result `[ 1, 2, 4, 5 ]`. Note the result is an exact combination of both, with the value from the left list first (in the order they appear) followed by the values from the second list (also in the order they appear).

The second operator, replication, is less frequently useful but can also be used for building lists with an expected number of repeated values. What it does is copy a list a number of times and concatenates all of them together. For example, the expression `[ 1 , 2 ] * 3` is the same as `[ 1 , 2 ] + [ 1 , 2 ] + [ 1 , 2 ]` and has the result `[ 1 , 2, 1 , 2, 1 , 2 ]`.

It should be noted, that both of these operators create a new list as a result, rather than changing the list that we already have.

7.5 Mutability

One of the issues that we have to be aware of when working with lists is that they are mutable (unlike all of the other values we have used in programming so far). This can have some consequences

in our code that we need to be aware of. Lets look at an example similar to Example 1 from Chapter 6. However, this time instead of using immutable integers, we will use lists.

```
1 x = [ 1, 2, 3]
2 y = x
3 y.append(4) #add number to y
4 print(id(x), id(y))
5 print(x, y)
```

```
1 139866393945280 139866393945280
2 [1, 2, 3, 4] [1, 2, 3, 4]
```

Figure 7.1: Example of `id` Function and Output with Mutable Variable

We can see that in this example, the output shows the both of our lists have the same id and when we output the lists they both contain the same values. We might have expected based on the similar operations with integers (or any other immutable data type) that x and y would be different, but as these are mutable we have only copied the reference (address) on line 2. The result of this is that there is only 1 list in our code, but both the variables `x` and `y` allow us to access the same list.

Tracing - Mutable

We are going to have a much closer look at how Python really works with values like this and stores them in memory. This requires a change in how we represent code tracing and the steps that the language usually performs invisibly for us.

Line 1

Table 7.10 shows the breakdown of this line of code. Rather than consider the literal value representing the list as a single value we have instead represented the individual steps that Python must complete to represent these values in memory. The result for all four of these steps is shown as the address that the value is being stored at instead of the value, but the type is still listed as the type of the value. We can see that the numbers in the list are all remembered in different places in memory, and the list remembers these addresses instead of the values.

Table 7.10: Line 1: `x = [1, 2, 3]`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Store Value in Memory	1	7*	int	
2	Store Value in Memory	2	8*	int	
3	Store Value in Memory	3	9*	int	
4	Store Value in Memory	[7,8,9]*	2*	list	
5	Assignment	x = 2*	2*	list	2

*These values are all shown as addresses rather than the value being stored

Table 7.11 shows the memory map and the contents of memory after the code has been executed. Each individual value is remembered and the list remembers the addresses.

Table 7.11: Memory map and contents of Memory after executing line 1

Name	Address	Type
x	2	list

Address	0	1	2	3	4	5	6	7	8	9
Value			[7,8,9]					1	2	3

Representing lists in this way makes sense to computers, but is more difficult to understand for us. We will only represent value this way for this example so that we can have a better

understanding of the memory in system and how it changes when we use mutable objects. You should note that while the specific details of how lists and their contents are stored in memory is important, this in no way changes how we access value in a list and use them. These details and operations (like finding values in memory) are thankfully hidden from us because we do not need to be aware of them. However, when working with mutable values we do need to remember that this arrangement more accurately represents what happens in the computer and explains the results of this code.

Line 2

Table 7.12 shows the breakdown of the code in line two. Instead of loading the value of the variable `x`, the address is loaded and assigned to the variable `y`. The corresponding memory diagrams in Table 7.13 show us how both variables x and y are remembering the same address in memory.

Table 7.12: Line 2: `y = x`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Load Variable	`x`	`2`*	`list`	2
2	Assignment	`y = 2`*	`2`*	`list`	2

*These values are all shown as addresses rather than the value being stored

Table 7.13: Memory map and contents of Memory after executing line 2

Name	Address	Type
x	2	list
y	2	list

Address	0	1	2	3	4	5	6	7	8	9
Value			[7,8,9]					1	2	3

Line 3

This line of code introduces a new type of statement, calling a method. This is similar in principle to calling a function and is something that we will study in a little more detail later, but will primarily be the focus of one of your classes next year. Table 7.14 shows the breakdown of this code, in this case the ordering of the first two steps is not important because this is done automatically and hidden from us.

Table 7.14: Line 3: `y = x`

Order	Explanation	Code Executed	Result	Result Type	Next Line
1	Store Value in Memory	`4`	`1`*	`int`	
2	Load Variable	`y`	`2`*	`list`	
3	Call Method	`2.append(1)`*	`None`	`None`	4

*These values are all shown as addresses rather than the value being stored

The important part of this code is that we tell the list remembered in address 2 that it should also remember the value stored in address 1. This operation does not return any result for us to see, but the changes will have been made and are visible if we examine the memory diagrams in Table 7.15.

This final statement is where the mutability of these values is important. Because both `x` and `y` are connected to the list in address 2, when we make a change by adding a new value to the end of the list `y` the change is also made to the variable `x`. This is because there is in fact only a single list which gets changed by this operation.

Table 7.15: Memory map and contents of Memory after executing line 3

Name	Address	Type
x	2	list
y	2	list

Address	0	1	2	3	4	5	6	7	8	9
Value		4	[7,8,9,1]					1	2	3

Copying Lists

There are several solutions that we can use to copy a list so that we do not have problems with mutability. We can use a function called `list` or we can use the list operators.

The first solution is to use the function `list`. This takes a list as a parameter and copies all of the values into a new list. For example, in our code we could replace line 2 with `y = list(x)` and there would be two lists instead of one. Any changes we made to `y` would not change `x`.

The second solution would be to use the concatenation operator. All we need to do is join our list with another empty list. For example, in our code we could replace line 2 with `y = x + []` and there would be two lists instead of one.

The third solution would be to use the replication operator. All we need to do is replicate our list 1 time. For example, in our code we could replace line 2 with `y = x * 1` and there would be two lists instead of one.

In all of these examples, we would have two lists which contain the same values but are stored in different places in memory. These lists can then be changed independently. Lets assume that the fourth line of code uses one of these techniques to copy the list instead. Table 7.16 shows how the memory diagrams could be shown. Here we can see that there are now two lists in memory and that they contain the same contents. If we add or remove a value from one of the lists, the other list is not changed.

Table 7.16: Memory map and contents of Memory after executing `y = list(x)`

Name	Address	Type
x	2	list
y	6	list

Address	0	1	2	3	4	5	6	7	8	9
Value		4	[7,8,9,1]				[7,8,9,1]	1	2	3

Mutable Contents

The above solutions will all work as long as the contents of the list you are copying are immutable. This is because the copies that we create remember the same addresses for the contents of the list. If the value in address 7 was mutable and we changed it to 45, then this value would be changed in both lists even though we have made a copy. Understanding these problems is the reason that we have looked at these examples, and it is something we should consider whenever we are remembering mutable values.

The solution to this problem is again to copy the mutable values when necessary, however you may need to do this for individual values in the lists rather than for the whole list.

7.6 Nested Lists

We can remember any data type within a list, this includes other lists. When we store a list inside another list, this is called a nested list. This is very useful when data that we have is more like a table, where some pieces of the data are related to each other. Consider Table 7.17, I might have the grades of a number of students for a number of different assignments.

Representing these values would be be possible using a single list, but it would require some complicated logic similar to representing a nested loop as a non nested loop. Instead, we can represent the data using a nested lists. Nested lists an array that uses two indexes instead of one.

Table 7.17: Student grade data shown as a table

Student	Assignment 1	Assignment 2	Assignment 3
182372101	80	0	60
182372102	95	100	95
182372103	60	45	65
182372104	80	83	95
182372105	88	100	90

One index is used to represent the row number and one index is used to represent the column number.

```
grades = [ ["182372101", 80, 0, 60], ["182372102", 95, 100, 95], ["182372103"
↪  , 60, 45, 65], ["182372104", 80, 83, 95], ["182372105", 88, 100, 90] ]
```

Listing 7.6: Nested list created with single assignment

This example shows how to declare a nested list in a single assignment. This is slightly more complicated that initialising a list because we must group the data based on the row it is in. The process is similar to a list, all of the values are placed within a set of square brackets. However, within this the values from each row are surrounded by another set of square brackets. Values in a row must be separated by commas and the rows must also be separated by commas.

```
grades = []
grades.append(["182372101", 80, 0, 60])
grades.append(["182372102", 95, 100, 95])
grades.append(["182372103", 60, 45, 65])
grades.append(["182372104", 80, 83, 95])
grades.append(["182372105", 88, 100, 90])
```

Listing 7.7: Nested list created in separate parts

This example shows an alternative way to create the same list. Instead of creating the whole list as a single statement, we create each inner list individually and append it to the outer list. The first example is more useful when we know all of the values in advance and the second example is more useful when we are creating the list based on user input. The second example can also be more easily created using a loop.

When we create this type of nested list, we are actually creating multiple lists. In this example 6 lists are created, one list to remember all of the other lists and five lists to remember the individual rows of data. Again, we do not need to consider how exactly this data is stored in memory, but we should remember that all of these lists are mutable and be careful if we are creating copies.

The example data we want to represent above contains 5 rows and 4 columns. So we could create a nested list to represent these values. The following table gives an example of how the data could be stored in nested lists. The darker green parts show the values and the pink numbers on the left show the row index and the blue numbers on the top show the column index.

Accessing Values in a Nested List

When we want to access a value stored in a nested list, we need to use two indexes. The first index to choose which row we want and the second index to choose which column we want. Given the array above, the following examples show how we could access different values;

Table 7.18: Representation of data in 2D lists and their indexes

	0	1	2	3
0	"18372101"	80	0	60
1	"18372102"	95	100	95
2	"18372103"	60	45	65
3	"18372104"	80	83	95
4	"18372105"	88	100	90

- `grades[0][3]` is the value 60
- `grades[2][2]` is the value 45
- `grades[3][0]` is the value `"18372104"`

Typically, when we are working with nested lists, we would use nested loops to interact with the lists. The outer loop executes once for every row in the list. The inner loop would then execute once for each column in the list. The variables that we use for counting the number of times that each loop has executed can be used to choose the correct indexes. For example if we are using the variable `i` for the outer loop and the variable `j` for the inner loop, then we would use the code grades[i][j] to access the correct value.

As the loops execute the values will change such that when both loops are finished, we have visited every location in the list exactly once. To make this work, we must start both values at 0 and have the condition of the outer loop be `(i < number_of_rows)` and the condition of the inner loop be `(j < number_of_columns)`.

The following example shows how this technique could be used to read the values of the students grades from the user and store them in the list.

```
1  grades = []
2  i = 0
3  while i < 5:
4    sn = input("Enter the students number: ")
5    j = 1 # we have already read the first value
6    nestl = [sn]
7    while j < 4:
8      x = int(input("enter a grade: "))
9      nestl.append(x)
10     j = j + 1
11   i = i + 1
12   grades.append(nest1)
```

Listing 7.8: Using nested loops to a nested list

The code is a little different than the explanation as we wanted to read the grades as numbers so we read the student number of the student first and then started the loop at 1. When this code is completed, the contents of the nested lists will match the contents of the table on the previous page.

More Dimensions

It is also possible to create nested lists containing more nested lists. For each nested level added, we use another set of square brackets to indicate the index in the next level `grades[0][0][0]`). It is generally uncommon to see nested lists with more than two levels of nesting.

7.7 Code Tracing

In order to make sure that we fully understand the usage of nested lists, we will trace an example program. In this example we will perform a basic operation, searching for the smallest number.

Searching in a list is a good example to use because we will need to visit each index of the list at least once to check if it is the smallest. Visiting each index of of an list is a very common operation, though we may be preforming different operations when we do. The overall structure that we use for our code can be applied to many different operations.

```
1  grades = [ [80, 95] , [60, 80] ]
2  i = 0 # initial value is same as first index
3  small = grades[0][0] # assume first index is smallest
4  while i < 2 : # condition checks less than size of array
5    j = 0
6    while j < 2 :
7      if small > grades[i][j] :
8        small = grades[i][j]
9      j = j + 1;
10   i = i + 1;
11 print(small, "is the smallest")
```

Listing 7.9: Searching a Nested List for the smallest value

The important points to note in the structure of the code here is that we start our outer loop counter (`i`) with the initial value of 0 (as this is the first index of a list) and the loop will execute as long as this counter is less than 2 (the number if rows in the list). Similarly, inside the outer loop the counter for the inner loop (`j`) is given the initial value of 0, but this happens every time the outer loop executes.

This is very common when working with arrays. Note that the program trace shows the index of each values in the array underneath the name.

The outer loop will be used to change the row that we are looking at. The first time the loop executes the value of `i` will be 0 and we will use this as the first index of the list. As such we will always access a value that is in row 0. Then the next time, the index 1 and so on until we have visited each row in the list.

The inner loop will be used to change the column that we are looking at. The first time the loop executes the value of `j` will be 0 and we will use this as the second index of the list. As such we will always access a value that is in column 0. Then the next time, the index 1 and so on until we have visited each column in the list. When this loop is completed, we will return to the outer loop and these steps will be repeated for the next row.

Because it is a program with a lot of instructions executed, we will break the program trace into multiple parts. This first part shows the initialisation of the variables in the program. Note that the row and column indexes are shown for each value in the array underneath the name of the array.

The basic idea of the code in this case is that we only every remember the smallest value that we have seen so far. This is remembered in the variable `small`. As we are visiting each value in the list if we find a smaller value, then we update the value of the variable `small`. The initial value for `small` is chosen as the first value in the list. There is a reason why this is a good idea, if we choose a bad initial value (one bigger than the values in the list) then we will not find the smallest value. For example, if we choose the value 0, but all of the numbers in the list are negative, our code will think that 0 was the smallest value in the list even though it is not in the list.

The outer loop will be used to change the row that we are looking at. The first time the loop executes the value of `i` will be 0 and we will use this as the first index of the list. As such we will

always access a value that is in row 0. Then the next time, the index 1 and so on until we have visited each row in the list.

The inner loop will be used to change the column that we are looking at. The first time the loop executes the value of `j` will be 0 and we will use this as the second index of the list. As such we will always access a value that is in column 0. Then the next time, the index 1 and so on until we have visited each column in the list. When this loop is completed, we will return to the outer loop and these steps will be repeated for the next row. Note that the row and column indexes are shown for each value in the array underneath the name of the array.

When the program has finished, we have visited each of the indexes in the nested list and compared them against the smallest value we have seen. First we visited the indexes in row 0, then the indexes in row 1.

Table 7.19: Nested Loops and Nested Lists Tracing

Line	Code Executed	grades [0][0]	grades [0][1]	grades [1][0]	grades [1][1]	i	j	small	Next
1	`grades = [ ... ]`	80	95	60	80	?	?	?	2
2	`i = 0`	80	95	60	80	0	?	?	3
3	`small = 80`	80	95	60	80	0	?	80	4
Loop once to check row 0 (outer loop)									
4	`while True : # 0 < 2`	80	95	60	80	0	?	80	5
5	`j = 0`	80	95	60	80	0	0	80	6
Loop once to check column 0 (inner loop)									
6	`while True : # 0 < 2:`	80	95	60	80	0	0	80	7
7	`if False : # 80 > 80`	80	95	60	80	0	0	80	9
9	`j = 1`	80	95	60	80	0	1	80	6
Loop once to check column 1 (inner loop)									
6	`while True : # 1 < 2`	80	95	60	80	0	1	80	7
7	`if False : # 80 > 95`	80	95	60	80	0	1	80	9
9	`j = 2`	80	95	60	80	0	2	80	6
Loop once to check column 2 (inner loop)									
6	`while False : # 2 < 2`	80	95	60	80	0	2	80	10
10	`i = 1`	80	95	60	80	1	2	80	4
Loop once to check row 1 (outer loop)									
4	`while True : # 1 < 2`	80	95	60	80	1	2	80	5
5	`j = 0`	80	95	60	80	1	0	80	6
Loop once to check column 0 (inner loop)									
6	`while True : # 0 < 2`	80	95	60	80	1	0	80	7
7	`if True : # 80 > 60`	80	95	60	80	1	0	80	8
8	`small = 60`	80	95	60	80	1	0	60	9
9	`j = 1`	80	95	60	80	1	1	60	6
Loop once to check column 1 (inner loop)									
6	`while True : # 1 < 2`	80	95	60	80	1	1	60	7
7	`if False : # 60 > 80`	80	95	60	80	1	1	60	9
9	`j = 2`	80	95	60	80	1	2	60	6
Loop once to check column 2 (inner loop)									
6	`while False : # 2 < 2`	80	95	60	80	1	2	60	10
10	`i = 2`	80	95	60	80	2	2	60	4
Loop once to check row 2 (outer loop)									
4	`while False : # 2 < 2`	80	95	60	80	2	2	60	11
11	`print(60, "...")`	80	95	60	80	2	2	60	-

Chapter 8

Sequences

In this chapter, we will continue the study of sequences. First we will study the tuple data type, which can be used in similar way to the list data structure we learned in Chapter 7. The primary difference between lists and tuples is that the tuple data type is immutable.

Following after this, we will study the operations that can be performed on all sequences. These will apply equally to the data types list, tuple and string.

8.1 Tuples

A tuple is a linear data structure like a list. It can contain a number of values that are ordered and accessed using an index. Indexing is the same as in lists, the first index is 0 and all following indexes increase by 1.

Table 8.1: Representation of a Tuple containing number of days in each month

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	31	28	31	30	31	30	31	31	30	31	30	31

This is an example of a tuple named `months`. It looks very similar to the list example we studied in the last chapter. It contains 12 values and the index of each value is shown on the first row of the table.

Tuple Literal

We can define a literal value of a tuple in our code (just like other values), but we need to use a special syntax. Writing a tuple in our code is similar to writing a list literal, we need to place some characters to tell python that this is a tuple. A tuple literal starts with an opening round bracket `(`, this is followed by the values we want in the tuple (separated by commas) and then finally a matching closing round bracket `)`.

```
1 ( val1, val2, val3 )
```

Listing 8.1: Syntax for tuple literal

If we want to create a tuple containing only a single value, we must write this in a special way. We must add a comma after the value, e.g. `( 12, )`. This is to stop Python thinking that this is a value inside a set of brackets that it should evaluate. We can create an empty tuple (containing no items) just using a pair of round brackets ().

The tuple created can be used in an expression, as a parameter to a function or remembered in a variable using assignment. This example shows how we would create the tuple from the example above in code and remember it using the variable name `months`.

```
1 months = ( 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 )
```

Listing 8.2: Assigning literal tuple to a variable

Tuple Function

We can use the tuple function to create new tuples in our code. The tuple function can be used without any parameters and it will created an empty tuple. As tuples are immutable, having an empty tuple is generally of limited use.

Where this function is particularly useful is in creating new tuples based off the contents of other data structures. If we pass the data structure as a parameter, we can create a tuple containing elements from a list, string, set or another tuple as well as the keys from a dictionary. The function works by creating a new tuple containing all of the values from the data structure we passed as a parameter.

```
1 tpl1 = tuple()
2 tpl2 = tuple([1, 2, 3, 4])
3 tpl3 = tuple("Hello")
4 print(tpl1, tpl2, tpl3)
```

Listing 8.3: Examples of creating tuples using the tuple function.

Using Tuples

Accessing Values

Using this code we can create a single variable containing many values. But how can we access the value that we want from the tuple? To access a single value in the tuple, we use the index placed between a set of square brackets. For example, using the example tuple above we would use the code `months[0]` to access the first item in the tuple, `months[1]` to access the second item in the tuple and so on. These can be used in an expression, as parameters to a function or anywhere you can use a value.

Adding, Changing and Removing Values

Because the tuple data type is immutable, we cannot add remove or change any values in a tuple. If we want to perform some of these types of operations using a tuple, we can create a copy with the value added, removed or changed and assign it to the same variable. This will give the appearance that we have changed the tuple, while actually having replaced it with a new tuple.

Size of a Tuple

The size of a tuple (often called length) is the number of items in the tuple. This can be found using the function `len`. We pass the name of the tuple (or any expression with a tuple as a result) as a parameter and the function will return an integer value representing the number of items in the tuple. We will often want to remember this in a variable.

For example, if we wanted to find the size of the example tuple, we would use the code `len(months)`. If we wanted to remember that in a variable (called `size`), we would use `size = len(months)`.

Example

To help get an understanding of the use of tuples in programming, we are going to study an example a similar example to the previous chapter. The task is to calculate the average and also count how many of the values are below the average of a tuple containing 10 values.

In this example, we will assume that we have already created the tuple containing the numbers in some other part of the program and it is stored in a tuple called numbers. In this example we will first calculate the average of these numbers.

```
1 total = 0
2 i = 0
3 while i < len(numbers):
4   total = total + numbers[i]
5   i = i + 1
6 average = total / i
```

Listing 8.4: Calculating the average of the values in the Tuple

This code might seem familiar, that is because it is exactly the same as the code we used to calculate the average value of a list. This is to be expected, lists and tuples are very similar when they are only being used to access data.

```
1 small = 0
2 i = 0
3 while i < len(numbers):
4   if numbers[i] < average:
5     small = small + 1
6   i = i + 1
7 print(small, "numbers were smaller than the average of", average);
```

Listing 8.5: Counting how many of the numbers are smaller than the average

Again, we see that the code is exactly the same. As such we will not go through the process of tracing this code as the results would be identical to the previous chapter.

Tuple Operators

Python also has operators that can be used on tuples. In this section we will study two operators that can be used on tuples. These work in the same way as the operators for lists and strings.

Table 8.2: Tuple Operators

Name	Code	Explanation	Example
Replication	`*`	Replicates a tuple a number of times, where x is a tuple and y is an integer	`x * y`
Concatenation	`+`	Joins two tuples together	`x + y`

The first operator, concatenation, can be very useful when working with multiple tuples. What it does is copy two tuples and create a new tuple containing the contents of both. For example, the expression `( 1, 2 ) + ( 4, 5 )` would have the result `( 1, 2, 4, 5 )`. Note the result is an exact combination of both, with the value from the left tuple first (in the order they appear) followed by the values from the second tuple (also in the order they appear).

The second operator, replication, is less frequently useful but can also be used for building tuples with an expected number of repeated values. What it does is copy a tuple a number of times and concatenates all of them together. For example, the expression `( 1 , 2 ) * 3` is the same as `( 1 , 2 ) + ( 1 , 2 ) + ( 1 , 2 )` and has the result `( 1 , 2, 1 , 2, 1 , 2 )`.

It should be noted, that both of these operators create a new tuple as a result, because the original tuple is immutable and cannot be changed.

Mutability

Tuples are immutable, so we may not consider mutability a problem like it was with lists. This is mostly correct, however we do have to consider situations where the contents of our tuple are mutable. If some of the values inside the tuple are mutable data types, then we may still have a problem as when we change one of these items, it will change in any copies that we might have made of this tuple. This is something that you need to be aware of when working with mutable data types.

The solution to this problem is to copy the values when necessary, however you will need to do this for individual values in the tuple.

Nested Tuples

We can remember any data type within a Tuple, this includes other Tuples. When we store a tuple inside another tuple, this is called a nested tuple. This is very useful when data that we have is more like a table, where some pieces of the data are related to each other. Lets study the students and grades example from the previous chapter.

Nested tuples uses two indexes instead of one (just like nested lists). One index is used to represent the row number and one index is used to represent the column number.

```
1 grades = ( ("182372101", 80, 0, 60), ("182372102", 95, 100, 95), ("182372103"
  ↪  , 60, 45, 65), ("182372104", 80, 83, 95), ("182372105", 88, 100, 90) )
```

Listing 8.6: Nested tuple created with single assignment

This example shows how to declare a nested tuple in a single assignment. This is slightly more complicated that initialising a tuple because we must group the data based on the row it is in. The process is similar to a tuple, all of the values are placed within a set of round brackets. However, within this the values from each row are surrounded by another set of round brackets. Values in a row must be separated by commas and the rows must also be separated by commas.

```
1 a = ("182372101", 80, 0, 60)
2 b = ("182372102", 95, 100, 95)
3 c = ("182372103", 60, 45, 65)
4 d = ("182372104", 80, 83, 95)
5 e = ("182372105", 88, 100, 90)
6 grades = ( a, b, c, d, e)
```

Listing 8.7: Nested tuple created in separate parts

This example shows an alternative way to create the same tuple. Instead of creating the whole tuple as a single statement, we create each inner tuple individually and assign it to a variable. The outer tuple is then created containing all of the individual tuples.

The example data we want to represent above contains 5 rows and 4 columns. So we could create a nested tuple to represent these values. The following table gives an example of how the

data could be stored in nested tuples. The darker green parts show the values and the pink numbers on the left show the row index and the blue numbers on the top show the column index.

Table 8.3: Representation of data in 2D lists and their indexes

	0	1	2	3
0	"18372101"	80	0	60
1	"18372102"	95	100	95
2	"18372103"	60	45	65
3	"18372104"	80	83	95
4	"18372105"	88	100	90

Accessing Values in a Nested Tuple

When we want to access a value stored in a nested tuple, we need to use two indexes. The first index to choose which row we want and the second index to choose which column we want. Given the nested tuple above (named `grades`), the following examples show how we could access different values;

- `grades[0][3]` is the value 60
- `grades[2][2]` is the value 45
- `grades[3][0]` is the value `"18372104"`

Typically, when we are working with nested tuples, we would use nested loops to interact with the tuples. The outer loop executes once for every row in the tuple. The inner loop would then execute once for each column in the tuple. The variables that we use for counting the number of times that each loop has executed can be used to choose the correct indexes. For example if we are using the variable `i` for the outer loop and the variable `j` for the inner loop, then we would use the code grades[i][j] to access the correct value.

As the loops execute the values will change such that when both loops are finished, we have visited every location in the tuple exactly once. To make this work, we must start both values at 0 and have the condition of the outer loop be `(i < number_of_rows)` and the condition of the inner loop be `(j < number_of_columns)`.

Mixed Nesting

This process works the same way if we mix the types that we are nesting. For example, tuples nested inside a list works the same as lists nested inside a tuple or any mixture of different levels of nesting of the two data types.

Example

To apply all of the knowledge from this tuples and lists we will study an example of their use.

Given a list of student ids and results (for quizzes, exam and programming exam) as a list of tuples named `students`, calculate the average result of each component and the final grade of each student. Each tuple in the list will contain the information in the following order student number (str), quizzes result (int), exam result (int), and programming exam result (int). For example one of the nested tuples would look like this: `("18372101", 80, 0, 60)`.

We are also given the weight of each of these components as a tuple named `weight` containing 3 real numbers, quizzes 0.2, exam 0.4, and programming exam 0.4. e.g. `weight = (0.2, 0.4, 0.4)`

This task has many parts, but we should be able to complete them all in one execution of a loop though the list.

```
create variables to remember sums of components (sum_quiz, sum_exam,
↪ sum_prog)
create variable to count number of times loop is executed (i)
Loop through lists one row at a time
    calculate the combined result for the student
    output the student number and final result
    add value of quiz component to sum_quiz
    add value of exam component to sum_exam
    add value of programming exam component to sum_prog
calculate averages of components
output the averages for each component
```

Listing 8.8: Example algorithm

This is the steps that we need to go through to complete this task. Most of these are pretty straight forward, this is how it would be implemented in Python:

```
students = [("182372101", 80, 0, 60), ("182372102", 95, 100, 95), (
↪ "182372103", 60, 45, 65), ("182372104", 80, 83, 95), ("182372105", 88,
↪ 100, 90)]
weights = (0.2,0.4,0.4)
sum_quiz = 0
sum_exam = 0
sum_prog = 0
i = 0
while i < len(students):
  final_result = students[i][1] * weights[0] + students[i][2] * weights[1] +
  ↪ students[i][3] * weights[2]
  print(students[i][0], "got final result", final_result)
  sum_quiz = sum_quiz + students[i][1]
  sum_exam = sum_exam + students[i][2]
  sum_prog = sum_prog + students[i][3]
  i = i + 1
print("Average of quizzes was", sum_quiz/len(students))
print("Average of exam was", sum_exam/len(students))
print("Average of programming exam was", sum_prog/len(students))
```

Listing 8.9: Calculating final Results

8.2 Sequence Operations

There are several operators, functions and statements that can be used on any sequence. This means that they will have the same effect on a list, a tuple or a string. We have already seen that the operators concatenation and replication work in the same way for all three data types. In this section we will study other functionality that is common to sequences.

Example Data

For each function or statement we introduce, we will need to demonstrate what is happening. For this purpose, the following example shows the data we will use, we have a list named `l`, a tuple named `t` and a string named `s`.

```
l = [ 1, 2, 3, 4, 5, 5 ]
t = (6, 7, 8, 8, 10, 11)
s = "Hello world!"
```

Listing 8.10: Example data for sequences

When each new concept is introduced, there will be an example for each data type and the result based on the data in this example.

Accessing Data

There are two ways to access a value from a sequence, we can select a single value from the sequence or a new sequence of values. Selecting a single value is often called indexing and selecting multiple values is called slicing.

Indexing

We have already seen learned how to access a single value in the previous sections, but lets look at an example of selecting a single value from each of the different types of sequence.

- `l[0]` will give us the first item in the list named `l` e.g. `1`
- `t[1]` will give us the second item in the tuple named `t` e.g. `7`
- `s[2]` will give us the third character in the string named `s` e.g. `"l"`

Reverse Indexing

Often we will want to access values that are stored at the end of a sequence. The index of this value will always be based on the number of items in the list e.g. `len(l) - 1` would be the index of the last value in the list `l`. This is the index 5 and coincidentally the value in that index is also 5.

In Python, we can use negative numbers to index a sequence backwards. Instead of starting at the first value and counting up from 0, the indexes start at the last value and count down from -1. E.g. -1 is the index of last value int he sequence, -2 is the index of the second last item and so on. Consider the following examples of reverse indexing.

- `l[-1]` will give us the last item in the list named `l` e.g. `5`
- `t[-2]` will give us the second last item in the tuple named `t` e.g. `10`
- `s[-3]` will give us the third last character in the string named `s` e.g. `"l"`

Slicing

Slicing a sequence of values is more complicated that selecting a single value. Additionally, instead of a single value this gives us a new sequence, which is the same type as the sequence we are selecting from. We can perform different types of slices on a sequence, the most basic requires us to specify a start and end index and everything from the start up to but not including the end index are inserted into the new sequence.

These values are specified by putting them between a set of square brackets and separated by a colon. For example, `seq[2:8]` would create a new sequence containing the values that are in index 2, 3, 4, 5, 6, and 7 of the sequence `seq` (but not the value at index 8). Consider the following example of slicing in sequences.

- `l[2:4]` will give us a new list `[3, 4]`

- `t[2:4]` will give us a new tuple `(8, 8)`
- `s[2:4]` will give us a new string `"ll"`

We can leave out one or both of the values in the slice. If we do not include the first value, e.g. `seq[:8]`, then Python will assume that the number is 0 and include all of the values from the start of the sequence. If we do not include the second value, e.g. `seq[2:]`, they Python will assume that the number is `len(seq)` and include all of the values to the end of the sequence (including the last value).

If we do not include any of these values, e.g. `seq[:]`, they Python will assume that the first number is 0 and the last number is `len(seq)` and include all of the values in the sequence. This is effectively another way that we can create a copy of a sequence.

Advanced Slicing

We can also be more selective when we are choosing values from a sequence. By adding another value (separated by another colon) called the increment we can choose which values in the ranged to include. For example, if we wanted all of the values we would include the value 1, if we only wanted every second value we would include the value 2.

- `l[0:4:1]` will give us a new list `[1, 2, 3, 4]`
- `t[0:4:2]` will give us a new tuple `(6, 8)`
- `s[0:10:3]` will give us a new string `"Hlwl"`

The increment is not restricted to only positive values, it can also be used to reverse the order of values in a sequence. For example the slice `l[::-1]` would result in the list `[5, 5, 4, 3, 2, 1]`.

Searching

We are often required to search a sequence (or other data structure) for a particular value. This search can done in different ways, we may only want to know if the value is in the sequence, or we may want to know where it is in the sequence, or we may want to know how many times it is in the sequence.

Each of these tasks are different and must be achieved in a different way. The first is called membership and is only concerned with if a value is in the sequence or not. The second operation is called locating, and this is about locating where in the sequence (what index) the value is. Finally, we may be required to count the number of times the same value appears in the sequence.

Membership

To check if a value is contained in a sequence, we use the keyword `in`. This result of this keyword is represented as a boolean value, where `True` means the value is in the sequence and `False` means that the value is is not in the sequence. As the value returned is boolean, this keyword is very useful as a part of the condition of an if statement or while loop. Consider the following examples of testing for membership:

- `3 in l` will give us the result True
- `6 in t` will give us the result True
- `"l" in s` will give us the result True
- `6 in l` will give us the result False
- `3 in t` will give us the result False
- `"x" in s` will give us the result False

Locating

To find where a value is located in a sequence, we use a method named `index`. We use this method in a similar way to using the `insert` and `append` we studied in Chapter 7. The syntax is first the named of the sequence, followed by a dot (.) then the named of the method (`index`) with the value we are searching for passed as a parameter to the function.

For example, `l.index(2)` would search the list named `l` and return the first index where it can find the value 2. When we use the index method, we must be sure that the value is in the list. If the value is not in the list, this will cause an error that stops our program. Consider the following examples:

- `l.index(3)` will give us the result 2
- `t.index(6)` will give us the result 0
- `s.index("l")` will give us the result 2

Counting

To find how many times a value is repeated in a sequence, we use a method named `count`. Again, we use this method like the previous examples. For example, `s.count("l")` will return the number of times the character `"l"` appears in the string `s`. Consider the following examples:

- `l.count(3)` will give us the result 1
- `t.count(8)` will give us the result 2
- `s.count("l")` will give us the result 3

Equivalence

If we want to tell if two sequences are the same, we use the equivalence operator `(==)`. This is the same as we use for other values. When we compare two sequences, they are only equivalent if they have the same number of values, contain the same values, and also those values are in the same order.

This means that `l == [1,2,3,4,5,5]` would return the result `True`, but `l == [1,3,2,4,5,5]` would result in `False`.

8.3 Sequence Functions

There are a number of functions that can be used with sequences. First, we will look at the functions that can be used to change the type of a sequence.

Changing Type

The first function is named `list`. This can be used to create a list containing the same values as a sequence that is passed as a parameter. For example, `list( (6, 7, 8, 8) )` would give us the result `[6, 7, 8, 8]`, and `list("hello")` would give us the result `['h', 'e', 'l', 'l', 'o']`.

The second function is named `tuple`. This can be used to create a tuple containing the same values as a sequence that is passed as a parameter. For example, `tuple( [6, 7, 8, 8])` would give us the result `(6, 7, 8, 8)`, and `tuple("hello")` would give us the result `('h', 'e', 'l', 'l', 'o')`.

Other Useful Functions

We can also use the functions `min`, `max`, and `sum` directly on a sequence of values. The function `min` will find the smallest value in a sequence, `max` will find the largest value in a sequence, and `sum` will only work for numbers, but calculates the sum of all of the values in a sequence.

For each function, the syntax we use is the name of the function followed by the name of the sequence in a set of round brackets. E.g. `min(l)` will give us the answer 1, `max(t)` will give us the answer 11, and `sum(l)` would give us the answer 20.

Sorting

Python also provides a function called `sorted` that can be used to create a list containing all of the values from a sequence is sorted order. This function will always return a list, even if the parameter we pass is a string or a tuple.

For example, `sorted((23, 4, 1, 34))` would return the list `[1, 4, 23, 34]` and `sorted("hello")` would return the list `['e', 'h', 'l', 'l', 'o']`.

This function is not making any change to the sequence, it is always creating a new list. This means that if we want to use that later in the code we need to remember it in a variable. E.g. `sl = sorted(l)` would remember the sorted list using the variable `sl`.

Reversing

The sorted function allows an optional parameter with the name `reverse`. This parameter should be a boolean value where True means the data will be sorted in descending order (biggest to smallest) and False means that the data will be sorted in ascending order (smallest to biggest).

For example, `sorted((23, 4, 1, 34), reverse = True)` would return the list `[34, 23, 4, 1]` and `sorted("hello", reverse = True)` would return the list `['o', 'l', 'l', 'h', 'e']`.

8.4 Looping Sequences

It is very common to use data structures like a sequence and a loop together. We have already studied this practice in Chapter 7. Consider the following example:

```
1 i = 0
2 while < len(seq):
3   s = seq[i]
4   # do something with value of s
5   i = i + 1
```

Listing 8.11: Example of loop over all values in a sequence

In this example, we have a loop that will execute once for each value in the list. The loop counter (`i`) starts at 0 (the value of the first index) and will be increased by one every time the loop executes. In this example, we use a variable (`s`) to remember the current value in the sequence. At the beginning of each execution of the loop it is assigned the value from the current index.

This structure of looping through each value in a sequence is so commonly used that there is a more compact structure that we can use. The structure is called a for loop.

```
1 for s in seq:
2   # do something with value of s
```

Listing 8.12: Example of for loop

These two examples are functionally identical. The loops will access the same values from the sequence in the same order and perform some action. However, in the for loop we do not need to remember which index of the sequence we are accessing, instead this is managed by Python.

Each time the loop is executed, the next value from the sequence is remembered in the variable `s` (or whatever you named it). This is called a loop variable. For example, if we wanted to print every character in a string on a separate line we could use the following for loop:

```
for c in "Hello":
  print(c)
```

```
H
e
l
l
o
```

Figure 8.1: Example of for loop and output

The loop executes 5 times, once for each character in the string `"Hello"`. The first time, the variable `c` contains the value `"H"`, the second it is `"e"`, the third and fourth it is `"l"`, and finally the fifth time it is `"o"`.

Example

In order to demonstrate how a for loop can be used to simplify our code, we will return to the example from section 8.1. In this code we have a list containing tuples of student information. A for loop here would return a single tuple every time the loop executes.

First lets look at the relevant code.

```
i = 0
while i < len(students):
  final_result = students[i][1] * weights[0] + students[i][2] * weights[1] +
  ↪  students[i][3] * weights[2]
  print(students[i][0], "got final result", final_result)
  sum_quiz = sum_quiz + students[i][1]
  sum_exam = sum_exam + students[i][2]
  sum_prog = sum_prog + students[i][3]
  i = i + 1
```

Listing 8.13: Example completed using while loop

We can simplify our this example by replacing this with a for loop. The loop variable (named `s`) in this loop represents a tuple containing the information of a single student. In the previous example, we used `students[i]` to access the same information. So in the updated `s[0]` can be used instead of students[i][0] and similar in other places.

```
for s in students:
  final_result = s[1] * weights[0] + s[2] * weights[1] + s[3] * weights[2]
  print(s[0], "got final result", final_result)
  sum_quiz = sum_quiz + s[1]
  sum_exam = sum_exam + s[2]
  sum_prog = sum_prog + s[3]
```

Listing 8.14: Example complete using for loop

Chapter 9

Functions

Writing large programs can be very difficult. A common solution for making programs easier to solve is to break them down into smaller pieces. These smaller pieces can then be solved without needing to worry about the overall program. The main aim of this chapter is to introduce the concept of user defined functions and other related information. However, we will take this opportunity to revisit some topics we discussed in Chapter 1.

In the first chapter, a set of four activities that make up the development process were introduced. The activities were understanding our task, understanding how to complete it, explaining this to the computer and finally correcting any errors. In this chapter we will look at the first three activities through an example program. First we will discuss the understanding of the task, then we will look at a technique for breaking down tasks, finally we will translate this into code.

9.1 Example Task

The task we need to complete: Write a program to display a sequence of prime numbers on the screen. The prime numbers must be between 2 and a number entered by the user (including 2 and the users number if it is prime). These prime numbers should be printed on a single line.

Understanding

Before we can attempt to complete a task we really need to understand it. The main part that we need to understand in this task is what a prime number is. So after some searching, we find out that a prime number is a number that can only be divided evenly (integer division without a remainder) by the number 1 and the number it self (only two divisors). We search for examples, and find the number 7 (which is prime) and 9 (which is not prime). We see that if a number is prime or not is related to the quotient and remainder of the number and all of the numbers between 2 and itself.

Table 9.1: Calculations when a number is prime

Calculation	7 / 7	7 / 6	7 / 5	7 / 4	7 / 3	7 / 2	7 / 1
Quotient	1	1	1	1	2	3	7
Remainder	0	1	2	3	1	1	0

The remainder of the division by all of the numbers between 7 and 1 are highlighted (in a darker green). The important point is that none of the values are 0. This means 7 is prime.

Table 9.2: Calculations when a number is not prime

Calculation	9 / 9	9 / 8	9 / 7	9 / 6	9 / 5	9 / 4	9 / 3	9 / 2	9 / 1
Quotient	1	1	1	1	1	2	3	4	9
Remainder	0	1	2	3	4	1	0	1	0

Again, the remainder all of the division by all of the numbers between 9 and 1 are highlighted. In this case, we can see that the remainder of 9 / 3 is 0 and therefore this number is not prime.

Now that we have a good understanding of what a prime number is, we can consider that we understand the meaning of this task. Next, we need to understand how to complete the task. We investigate this using a technique know as top-down programming.

9.2 Top-Down Programming

Top-down programming is often called **stepwise refinement** or **divide and conquer**. This is a technique for solving problems and writing programs.

The main idea is that you break your problem into smaller sub-problems. Then for each sub-problem, we independently break that down into smaller steps again. This process is continued until we can no longer refine the problems any further. The final refinements are then combined together to give an entire solution to the original problem.

When we have a solution, we should check to make sure that it is sensible. Then we should work through the steps with some sample data to make sure that it works correctly. Finally, we translate the solution into a programming language.

First Refinement

We can easily break the problem down into the following parts without understanding it completely.

```
1 read number input by the user
2 calculate and output the prime numbers
```

Listing 9.1: First refinement of the task

Second Refinement

Reading the number input by the user is possibly the easiest of the steps. We can break it down to two steps, a prompt and reading the number. The calculation and output of the prime numbers can be done by looping through all of the numbers in that range and checking if each one is prime.

```
1 Display a message to the user prompting them to enter a number
2 Read the number into a variable named top
3 Repeat the following actions once for each number between 2 and top
4     Check if the number is prime
5     If the number is prime, display it on the screen
```

Listing 9.2: Second refinement

The first two lines match up with a single statement in Python, we will not need to refine this these parts any further. The loop is something that we could implement relatively easily, but the lines 4 and 5 still need to be refined a little more.

Third Refinement

We saw earlier that to check if a number is prime, we need to know that **all** of the numbers between itself and 1 have a remainder that is not zero when it is divided by them. We cannot know a number is prime by checking a single divisor, but we can know a number is not prime if the remainder is 0. So we can use a strategy where we assume that the number is prime, and then look for a number that divides evenly to prove that it is not prime.

Now that we understand what a prime number is, we can begin to break down the sub-problem check if the number is prime. The way we have to work is backwards, we need to look for a number that does divide evenly to prove that a number is not prime. Therefore, the first step is to assume that the number is prime.

```
Display a message to the user prompting them to enter a number
Read the number into a variable named top
Repeat the following actions once for each number between 2 and top
  assume that the number is prime (remember in variable)
  Execute the following steps for every divisor between 1 and the number
    if the number is evenly divisible by the divisor
      change our assumption to not prime
  If the number is prime
    display the number on the screen
```

Listing 9.3: Third refinement

We have refined this problem as far as we can. The next step is to manually go through the steps on pen and paper to make sure that the steps work correctly. Again, we would need to know in advance what the code should do (what the correct answer is) before we can know if it is working correctly.

Translating

While the steps in the pseudocode describe the instructions pretty well, we need to fill in details like converting the user input to a number. We also need to decide the type of variable to use for remembering if the number is prime or not. A boolean value would be suitable (`True` for prime, `False` for not prime), but we could use a number too.

```
counter = 2;
top = int(input("Enter a number: "))
# Calculate and output prime numbers
```

Listing 9.4: Translated code (part 1)

This part of the code takes care of declaring the variables, reading the number entered by the user and setting up the counter for the loop.

```
while counter <= top: # repeat operation once for every number
  prime = True # this means we assume the number is prime
  divisor = 2 # number to divide by
  while divisor < counter:
    if counter % divisor == 0: # check if number is divisible by divisor
      prime = False # this means we know the number is not prime
    divisor = divisor + 1
  if prime == True # number is prime
    print(counter, end = " ") # print the number
  counter = counter + 1
```

Listing 9.5: Translated code (part 2)

This part of the code takes care of the rest of the task. The outer loop executes once for each number from 2 up to the number that the user has entered. The inner loop executes once for each number from 2 up to the current counter of the outer loop. This is where we check each of these numbers to see if the counter of the outer loop is divisible by it. Finally, once this loop is finished, we will print the number when we have not found any divisors.

When looked at as a single program, this solution is large and complicated, however when it is broken down into smaller pieces each of those pieces becomes much more manageable in isolation. At the end, we only need to put all of our individually solved pieces together.

9.3 Subprograms

Programming languages generally provide some functionality that helps with the steps involved in breaking down a problem. In many languages this is implemented as a way of giving a name to a group of statements. Usually, the name we give will give us a hint to what the statements will do. Then when we want to use these statements, we use the name instead. The term **subprogram** describes a named group of statements.

Subprograms can be classified as either **procedures** or **functions**. A procedure is a named group of statements. When we use a procedure, we execute the associated statements. A function is the same as a procedure, except that a function **returns a value**. When we use a function, we execute the associated statements, but when they are finished we are given a result. It is very common for people to call both procedures and functions by the name function.

Using a procedure or a function is referred to as **calling** the procedure or calling the function. The code `print(num)` is an example of calling the function `print`.

Functions in Python

Both functions and procedures work in the same way, a procedure is implemented as a function that does not return a value. Though it can be difficult to tell the difference without reading the code or reading the documentation.

We have used many different functions in our programs so far. `print, input, int, len, sum` are all examples of functions that we have used. These are often called library functions because the come as part of the Python language libraries. The functions have been defined for us to use when we need them.

9.4 User Defined Functions

Python allows us to define our own functions. This is very useful when breaking down problems into smaller parts. We can only **define** a function once, but we can call a function many times. Additionally, functions can call other functions, which in turn may call other functions and so on.

There are two parts to every function, a **header** and a **body**. The header defines the name and any parameters that are required as input to the function. Parameters are specified by giving them a name that we can use to access the value.

The body of a function is a compound statement (multiple statements grouped by indentation). We can put any type of statement inside the body of a function. Inside the body of a function, we can access the values of the parameters that are passed when the function is called.

Function Syntax

```
1 def name(parameter-list):
2     # function body
```

Listing 9.6: Defining a function

The first part is they keyword `def`, this indicates that we are defining a function. The next part is the name of the function, just like variable names we get to decide this but we should choose something sensible. The parameter list can be left empty if the function does not required any parameters. If parameters are required, their names should be placed between the round brackets separated by commas. E.g. `(x, y)`. Inside the body of the function we can have any type of statement.

Where to Define Functions

In Python, you can define functions anywhere in your code. Typically, we will declare our functions at the top level of indentation (not inside any other function or statement).

It is also possible to define functions in another file which is then included using an import statement, this is how custom libraries work. We will study this in chapter 10.

One requirement is that functions must be defined before they are used. As the interpreter read files from top to bottom, the functions must be defined above the **first** place where they are used.

Returning an Answer

When the function we are writing is supposed to return an answer we must include a return statement. This means that in the body of the function we must have a statement that returns some value. This is done using the keyword `return`. E.g. `return 13` or `return max_value`. The value that is returned can be any valid Python expression, or simply the name of the variable containing the value.

The return statement marks the finishing point or one possible finishing point in the function body. When a return statement is reached, the function is stopped and no more code is executed. If we are using conditional statements, we can have multiple return statements in a function, each returning a value at different points, however only one of them can be executed when the function is called.

If a function does not return a value, but we try and remember the return value in a variable, the variable will contain a special value called `None`.

Example

To help our understanding of how to define functions, we will write a function to calculate the largest of two numbers. This function requires two parameters, one for each of the numbers we are going to compare. We will call the function `largest` because the function will be giving us the largest number.

```
1  def largest(a, b):
2    mx = a;
3    if b > a:
4      mx = b;
5    return mx
```

Listing 9.7: Function to find the larger of two integer parameters

The actual statements in the body of the function are pretty simple, we first assume that the biggest number is in the first parameter. Then we compare the two parameters to see if the second parameter is larger. If it is we change the value of the `mx` variable to match and return the value.

```
7  num1 = int(input("Enter a number: "))
8  num2 = int(input("Enter a number: "))
9  biggest = largest(num1, num2)
10 print("The biggest number is", biggest)
```

Listing 9.8: Using the user defined function max

This example shows the code that we execute when using the function we have defined. Because the function must be defined first, this code starts on line 7 of the same file. On line 9, the function `largest` is used to find the biggest of the two variables `num1` and `num2`.

This example was quite simple and it would be easier to have used an if statement to find the largest of the two numbers. To help justify why we might define a function to perform this task, we are going to have the user enter 4 numbers instead.

```
7  num = [0] * 4
8  num[0] = int(input("Enter a number: "))
9  num[1] = int(input("Enter a number: "))
10 num[2] = int(input("Enter a number: "))
11 num[3] = int(input("Enter a number: "))
12 biggest = largest(num[0], num[1])
13 biggest = largest(biggest, num[2])
14 biggest = largest(biggest, num[3])
15 print("The biggest number is", biggest)
```

Listing 9.9: Using the user defined function multiple times

This replaces the code we are using in example 6. Here we are using the the `largest` function three times. Each time it is passed two parameters and returns a single result to be stored in the variable `biggest`. Our final example is to show the `largest` function being used to find the biggest number in an array.

```
7  numbers = []
8  i = 0
9  while i < 10:
10   n = int(input("Enter a number: "))
11   numbers.append(n)
12   i = i + 1
13
14 i = 1
15 biggest = numbers[0]
16 while i < len(numbers):
17   biggest = largest(biggest, numbers[i])
18   i = i + 1
19 print("The biggest number is", biggest)
```

Listing 9.10: Using the user defined function in a loop

The code here can be viewed in two parts. The first part (lines 7 to 12) is a loop to read some number of values into a list (in this example 10 values). To change the number of values to be entered by the user, we just need to change the condition of the loop.

The second part of the example (lines 14 to 19) will find the biggest value in the list of numbers. It does not matter how many numbers are in the list, it will automatically check every item in the list and determine the biggest.

Parameter Names

When we are defining functions, it is impossible to know what the value of a parameter will be. This is because the value will be different every time the function is called. The names that we give parameters are just so that we can access the values, they have no meaning outside of the function.

When choosing names for a parameter, it is a good idea to use names that are not already used in your program. This is not a requirement of Python, but it can help reduce confusion. If you have two variables with the same name in different functions, it can become easy to confuse the two and make mistakes. Using different names makes it clearer exactly what each variable means in the program.

In the `largest` function, we have the parameters `a` and `b`. When we call the function in example 4, these parameters are given values. The first time largest is called the values are `num[0]` and `num[1]`, the second time they are `biggest` and `num[2]` and the final time they are `biggest` and `num[3]`.

9.5 Problem Decomposition With Functions

Now that we are able to define our own functions, we will return to the idea of top-down programming. The code can be made easier to understand if we define some functions.

The most complicated single sub-problem was checking if a number was prime. We can write a function to check if a single number is prime. We will give the function the name `isPrime`. The function requires only a single parameter, the number to check is prime or not. We can have the function return a boolean value when it is complete, `True` will mean that the number is prime and `False` will mean that the number is not prime.

```
def isPrime(num):
  divisor = 2
  prime = True
  while divisor < num:
    if num % divisor == 0:
      prime = False
    divisor = divisor + 1
  return prime
```

Listing 9.11: A function to determine if a number is prime

To make the problem easier, we can also write a function to calculate an output the prime numbers. This function will loop through the numbers in the specified range and display the prime numbers on the screen.

The function can be called `printPrimes`. It requires a single parameter, the highest number we need to check (we already know the lowest is 2). Because this function displays information on the screen instead of returning an answer it will not contain a return statement.

```
def printPrimes(highest):
  current = 2
  while current < highest:
    p = isPrime(current)
    if p:
      print(current, end=" ")
    current = current + 1
```

Listing 9.12: A function to print the prime numbers in a range

As long as these functions have been declared first, the final program becomes very small and simple.

```
top = int(input("Please enter a number: "))
printPrimes(top)
```

Listing 9.13: Using the `printPrimes` function to complete the task

9.6 Variable Scope

The location where we declare a variable has an effect on where it can be used. Obviously, we cannot use a variable before it has been defined. There are additional rules about where we can access variables.

In Python, we can declare variables in four places

1. Inside a function or another block of code - these are called **local** variables
2. Outside of all functions - these are called **global** variables
3. In the header of a function definition - these are called `formal parameters`
4. In a class definition - called instance variables

Variable Scope

Each of these kinds of variables has different rules about where they can be used. The **scope** of a variable is the area of code that it can be used in. Global variables, local variables and formal parameters have different scopes that they can be used in. Understanding the scope of a variable is important for understanding the programs that you write or others that you will read. Instance variables will not be studied in this book, they are not relevant to procedural programming that we are studying here.

Local Variables

If we declare a variable inside a function, then it can only be used inside that function. This is the scope of a local variable. Consider the examples of the `printPrimes` and `isPrime` functions from the previous section. The variables `current`, `highest` and `p` are all declared inside the `printPrimes` function, these variables cannot be used outside of this function. In the same way, the variables `prime` and `divisor` from the `isPrime` function cannot be used outside this function.

Local Variables With the Same Name

Because local variable have a reduced scope, it is possible to several local variables with different scopes but the same name in a program. Even though these variables share the same name, they are not the same variable and can have different values. If we do not understand the scope of the variables we declare we will not understand our programs correctly.

```
1  def bar():
2    i = 44
3    print("bar starting:  i =", i)
4    i = 55
5    print("bar finishing: i =", i)
6
7  def foo():
8    i = 0
9    print("foo function:  i =", i)
10   bar()
11   print("back in foo:   i =", i)
12
13 foo()
```

Listing 9.14: Multiple local variables with the same name

Here both variables named `i` are separate. In the `foo` function, the value of `i` is set at 0 and never changed. In `bar`, the value of `i` is initially set to 44 and then later changed to 55. This change has no effect on the value of `i` in the `foo` function.

```
1  foo function:  i = 0
2  bar starting:  i = 44
3  bar finishing: i = 55
4  back in foo:   i = 0
```

Listing 9.15: Output of previous example

Global Variables

Global variables are declared outside of any function. Global variables can be accessed from anywhere in the code, and changed if we use a special syntax. This is the scope of a global variable.

Because global variables can be accessed and changed from anywhere inside our programs, it can be hard to keep track of the value of a global variable. It is good programming practice to use the `minimum number` of global variables (preferably none) in a program.

```
1  def foo():
2    i = 55
3    print("i in foo =",i)
4
5  def bar():
6    i = 99
7    print("i in bar=",i)
8
9  i = 78
10 print("i outside =",i)
11 foo()
12 print("i outside =",i)
13 bar()
14 print("i outside =",i)
```

Listing 9.16: Use of global variable in multiple function

In that example, there were 3 `i` variables, one global and 2 local. This is because Python does not let you change a global variable accidentally.

```
1  i outside = 78
2  i in foo = 55
3  i outside = 78
4  i in bar= 99
5  i outside = 78
```

Listing 9.17: Output of previous example

You can let Python know you want to change a global variable using the keyword `global`. For example, if I want to change the global variable `i`, then first I must add `global i` in the function. Lets look at the previous example again, but this time we will use a single global variable.

```
def foo():
  global i
  i = 55
  print("i in foo =",i)

def bar():
  i = 99
  print("i in bar=",i)

i = 78
print("i outside =",i)
foo()
print("i outside =",i)
bar()
print("i outside =",i)
```

Listing 9.18: Use of global variable in multiple function

This time there is only one variable being used. This can make tracking the value of the variable more difficult because it can be changed by any function at any time.

```
i outside = 78
i in foo = 55
i outside = 55
i in bar= 99
i outside = 55
```

Listing 9.19: Output of previous example

Formal Parameters

Formal parameters are treated as local variables within a function. The scope of a formal parameter is the body of the function it is declared in. This means that formal parameters are used before global variables. You cannot declare a local variable with the same name and scope, the local variable must have a smaller scope.

```
def foo(i):
  print("i in foo =",i)
  i = 55
  print("i in foo =",i)

i = 78
print("i outside =",i)
foo(99)
print("i outside =",i)
```

Listing 9.20: Use of global variables with formal parameters of the same name

The global variable `i` has the largest scope, while the formal parameter `i` has the scope of the entire `foo` function. This allows us to know the result.

```
i outside = 78
i in foo = 99
i in foo = 55
i outside = 78
```

Listing 9.21: Output of previous example

Don't Use the Same Name

It is important that we understand the rules of variable scope. When we are reading programs that use the same name for different variables we must be able to know which value will be used.

However, in our own programs, it is easier to just **not** use the same name for different variables. This completely avoids any confusion that we might have.

9.7 Code Tracing

In this section we will look at an example of a program trace for a program that contains a function. The program for printing many primes would be too long, but lets loot at an example of checking if a single number is prime.

We will look at this function in two parts, first we will see how it works when the number we are checking is actually prime and then we will repeat the example with a number that is not prime. Note that we have changed the condition of the loop so that it ends as soon as we find out that a number is not prime. Additionally, while this part of the condition is written explicitly as `prime == True`, this can and usually would be shortened to `prime`. This is because the variable prime contains a boolean value that will already be `True` or `False`.

```
1  def isPrime(num):
2    divisor = 2
3    prime = True
4    while prime == True and divisor < num:
5      if num % divisor == 0:
6        prime = False
7      divisor = divisor + 1
8    return prime
9
10 x = 7;
11 if isPrime(x):
12   print(x,"is prime")
```

Listing 9.22: Program to check if a number is prime

Tracing a Prime Number

When python executes this code, it executes statements starting from the top of the file and moving down. The first statement is the function definition for `isPrime`. The purpose of this statement is for Python to remember the code for us to use later, this is why when we execute the code the first statement that will be traced is line 10. When we call the `isPrime` function then the code in the statement will be traced.

Table 9.3: Tracing code from Listing 9.22

Line	Code Executed	x	Next
10	`x = 7`	7	11
11	`if isPrime(7):`	7	1

Lines 10 and 11

When we reach the expression calling the isPrime function, we stop executing line 11 of the code and move instead to execute the function. This stop is only temporary and we will come back and complete the statement as soon as the function is finished. This is why the next line is listed as 1.

The isPrime Function

Here we are executing the isPrime function with the value of the parameter `num` set as 7.

Table 9.4: Tracing the isPrime function called with the parameter 7

Line	Code Executed	num	divisor	prime	Next
2	`divisor = 2`	7	2	?	3
3	`prime = True`	7	2	True	4
4	`while True: # True == True and 2 < 7`	7	2	True	5
5	`if False: # 1 == 0`	7	2	True	7
7	`divisor = 3`	7	3	True	4
4	`while True: # True == True and 3 < 7`	7	3	True	5
5	`if False: # 1 == 0`	7	3	True	7
7	`divisor = 4`	7	4	True	4
4	`while True: # True == True and 4 < 7`	7	4	True	5
5	`if False: # 3 == 0`	7	4	True	7
7	`divisor = 5`	7	5	True	4
4	`while True: # True == True and 5 < 7`	7	5	True	5
5	`if False: # 2 == 0`	7	5	True	7
7	`divisor = 6`	7	6	True	4
4	`while True: # True == True and 6 < 7`	7	6	True	5
5	`if False: # 1 == 0`	7	6	True	7
7	`divisor = 7`	7	7	True	4
4	`while False: # True == True and 7 < 7`	7	7	True	8
8	`return True`	7	7	True	11

No that the `isPrime` function has returned the answer `True`, we can finish executing the code on line 11.

Lines 11 and 12

Table 9.5: Continuing to Trace code from Listing 9.22

Line	Code Executed	x	Next
11	`if True: # isPrime(7)`	7	12
12	`print(7,"is prime")`	7	-

As the function returned the value `True` the code on line 12 is executed and we see a message telling us that 7 is prime.

Tracing a non Prime Number

Now let us repeat the exercise only this time instead of setting the value of `x` to 7, we will set it to 9. This will show us how the function behaves differently when the number is not prime.

Lines 10 and 11

Table 9.6: Tracing code from Listing 9.22

Line	Code Executed	`x`	Next
10	`x = 9`	9	11
11	`if isPrime(9):`	9	1

Again, we pause executing line 11 and instead start executing the code of the function.

The isPrime Function

Here we are executing the isPrime function with the value of the parameter `num` set as 9.

Table 9.7: Tracing the isPrime function called with the parameter 9

Line	Code Executed	`num`	`divisor`	`prime`	Next
2	`divisor = 2`	9	2	?	3
3	`prime = True`	9	2	True	4
4	`while True: # True == True and 2 < 9`	9	2	True	5
5	`if False: # 1 == 0`	9	2	True	7
7	`divisor = 3`	9	3	True	4
4	`while True: # True == True and 3 < 9`	9	3	True	5
5	`if True: # 0 == 0`	9	3	True	6
6	`prime = False`	9	4	False	7
7	`divisor = 4`	9	4	False	4
4	`while False: # False == True and 4 < 9`	9	4	False	8
8	`return False`	9	4	False	11

No that the `isPrime` function has returned the answer `False`, we can finish executing the code on line 11.

Table 9.8: Continuing to Trace code from Listing 9.22

Line	Code Executed	`x`	Next
11	`if False: # isPrime(9)`	9	-

This time, because the result of the function was False, we are finished and do not print any messages.

Chapter 10

Modular Design

Building large scale software systems has many difficulties. The larger the system the more complications that we have to deal with. One of the problems we face is dividing the tasks between large numbers of developers all working on the same project.

The same principle applies to other engineering disciplines. Consider the NASA space shuttle, it is a very complex system containing more than 2.5 million parts. Figure 10.1 shows how we might break down the engineering tasks in its construction. The major components are the orbiter vehicle, a large external liquid-fuel tank, and two solid rocket boosters. If we look more closely at the orbiter vehicle, it is composed of several subsystems, which in turn may be composed of sub-subsystems.

Some technicians may understand the shuttles overall design, but only a small number of individuals need to understand the detailed design implementation and testing of a specific subsystem. This is the principle that we use in modular design of software systems.

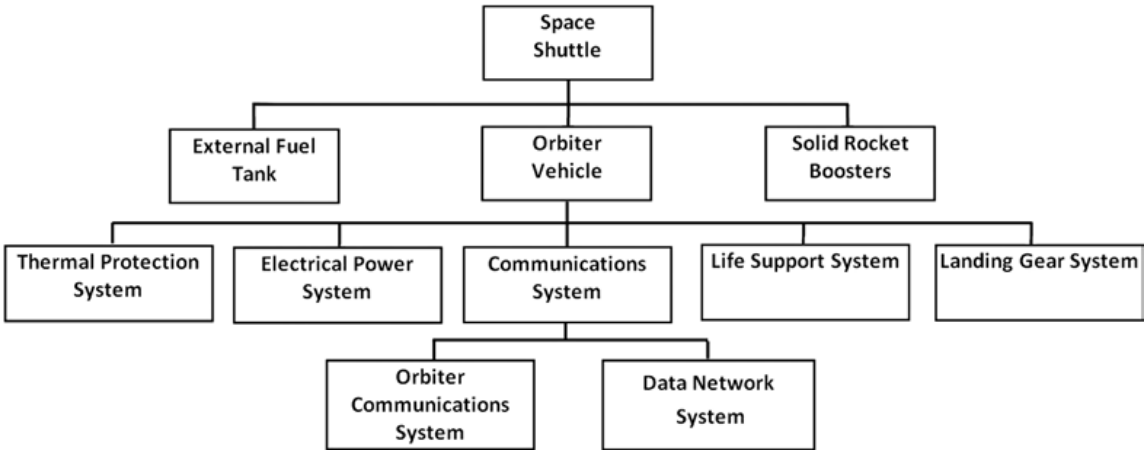


Figure 10.1: Modular Design for the Space Shuttle

10.1 Modules

Good software is usually designed as a collection of modules. Module broadly speaking, refers to the design of specific functionality to be included in a program. In reality what this means is that modules usually contain of a collection of functions or classes. These functions or classes can be used together to perform some related tasks or functions.

There are many benefits to the programmer from using modules when creating software. Firstly, it usually results in software with a better design. Modular design makes it easier for us to divide up tasks between different teams or individual programmers (each is assigned to complete a piece of the software). Additionally, using a modular design makes it easier to test that parts of our system are working correctly.

On of the greater benefits of modular design is code reuse. Writing software as modules allows us to reuse code in many programs, thereby reducing the amount of time that we spend writing some code. To make it easier to reuse our modules in other software, we usually include some information with each function we write to explain how it should be used. This information is known as a docstring.

Docstrings

A docstring is a triple-quote string literal inserted directly into the source code of a function immediately after the header of the function. A string literal that is not assigned to a variable is ignored in the source code of our programs, but this specially placed string is treated differently by the system.

```
1 def numPrimes(start, end):
2   """ Returns the number of prime
3   numbers between start and end """
```

Listing 10.1: A function header and docstring

We do not need to read the code of a module to see the docstring of a function. We can access it in the code using the code `__doc__`. For example, if we wanted to print the docstring of the `numPrimes` function, we would use the code `print(numPrimes.__doc__)`. This would print the text `Returns the number of prime numbers between start and end` to the screen.

10.2 Top Down Programming

Choosing how to break a piece of software into modules is not always easy. This is the same problem as we face when trying to understand how we break a simpler program down into functions.

The idea we use with functions was top down programming, this can be applied to the choice of how to break a system into modules. Top down design is developing the overall design of a system first, then adding the more detailed parts (like actual code) later in the process. To get an example of how we would apply this top-down approach to design, we will look at using it to design a text based minesweeper program.

Step one of designing is always understanding the task that we want to achieve, so before we start breaking the design down we need to have a thorough explanation of the requirements for this minesweeper program. Note the following details are the actual text of a introductory programming assignment used in the past.

Minesweeper Description

MineSweeper is a classic puzzle game where a number of mines are randomly located within a two-dimensional grid. The purpose of the game is to find the location of all of the mines and clear

all of the other locations. There are moves the player can make, they can select a square on the grid they believe is clear or they can place a flag on a square the believe is a mine.

Placing a flag on a square, only marks it for you to help you remember. If you are incorrect and there is not a mine, you can remove it by repeating the action.

Selecting a grid square that you believe is clear can have three possible outcomes;

1. There is a mine - In this case, the game is over
2. There are mines in some of the surrounding 8 squares - In this case, the square reveals how many mines are contained in the 8 squares around it by showing a number between 1 and 8.
3. There are no mines in the surrounding 8 squares - In this case, a chain reaction is set off. All spaces that are directly touching this one and also have no mines in the surrounding squares are cleared as well as the mines around them that show a number.

Lets look at an example of each of these outcomes using an example mine field. Figure 10.2a shows the starting minefield. The coordinates of the grid are shown along the top and left sides. The x coordinate is shown on the top and the y coordinate is shown on the left.

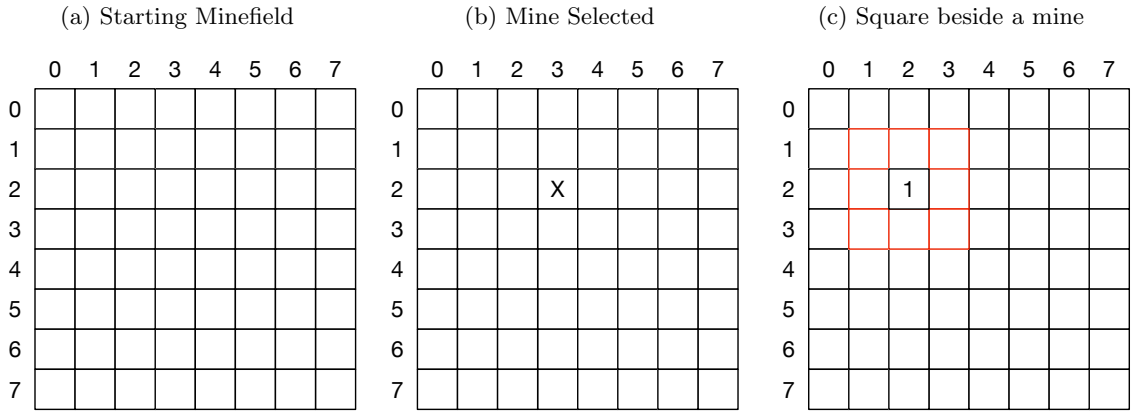


Figure 10.2: Demonstration of Clearing Outcomes

Example 1: Selecting a Mine

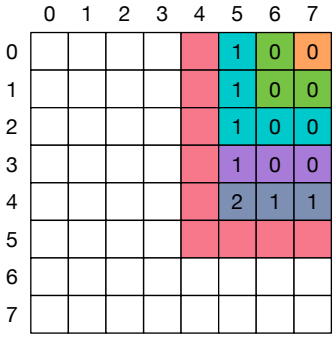
Lets assume that we select the grid square (3, 2), but that square contains a mine, then we would see the output shown in figure 10.2b. At this point the game is over and we need to start again.

Example 2: Selecting a clear square with a mine as a neighbour

If instead of selecting (3,2) we had chosen (2, 2) then we would see the output shown in figure 10.2c. The number in grid square tells us how many of the 8 neighbour squares (shown in red) contain a mine. In this case, there is only one. However we do not know which of these 8 squares may contain the mine.

Example 3: Selecting a clear square with no mines around it

If we choose a square and there are no mines in any of it's neighbouring squares, then the square is cleared (in our example we will show this as a 0). After this, the neighbouring squares are automatically selected and cleared, if those squares also have no mines as neighbours, then the squares neighbouring those squares are cleared. This action is repeated for all of the neighbouring squares, stopping only when we reach a squares with a mine as a neighbour.



The steps that the algorithm goes through are highlighted in different colours. Orange shows the first selected square, green shows the squares that are selected because they are neighbours and then the blue shows the squares that are selected because they are neighbours of the green squares. Because the algorithm is only repeated when there are no neighbours, the grid squares (4, 0), (4, 1), (4, 2) and (4, 3) are not cleared. Only the purple and then grey squares are cleared.

Making Moves

There are only 2 moves that can be made. Placing a flag on a square when you think there is a mine there and clearing a grid square.

Placing a Flag

Given the grid below, we can work out that there must be a mine in location (5, 0). We know this as grid square (6, 1) shows a 1 and all other neighbouring squares are already cleared. In cases like this, we want to put a flag on the location to mark it.

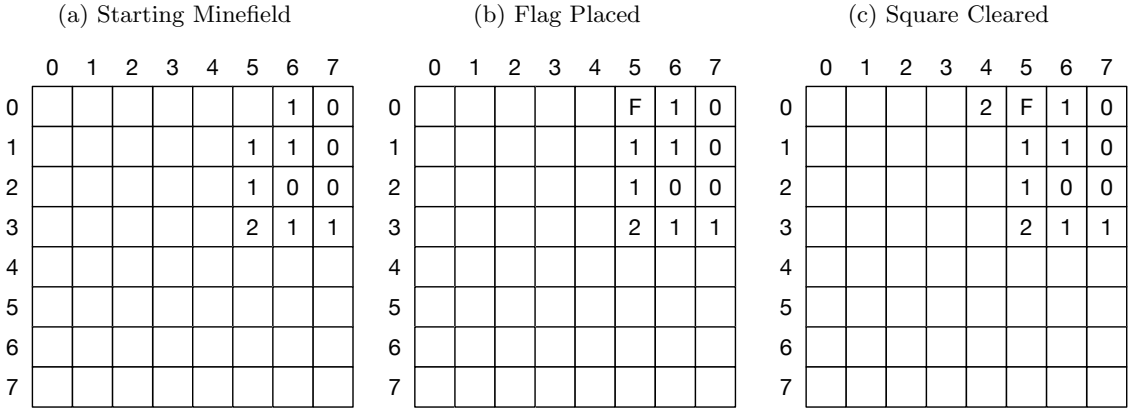


Figure 10.3: Demonstration of Flag Placement

When the game asks for the next move, we use the command `F x y` to instruct the game to places a flag on the grid square (x, y). This should only work if the grid square (x, y) has not already been cleared, if the grid square has already been cleared no action should be performed. The output of the command F 5 0 is shown in figure 10.3a.

Clearing a Square

After looking at the grid after our last move, we realise that there are squares that we know are not mines. (4, 0) cannot be a mine, because grid square (5, 1) shows only 1 neighbouring mine and we know (5,0) is a mine. So we will want to clear the mine. To clear a grid square, when the game

asks for our next move we use the command `x y` to instruct the game to clear the grid square (x, y). So our command would be 4 0, the result of this action is shown in figure 10.3c.

Text Version

The expected version of the game is a text version, you should not implement a version of the game that plays using the mouse. Here is an example of how the game might look when you make a text version:

```
Welcome to Minesweeper.
Please choose your difficulty:
  1: Beginner 8 x 8 grid with 10 mines
  2: Intermediate 16 x 16 grid with 40 mines
  3: Expert 24 x 24 grid with 99 mines
1
Mines: 10    Flags: 0   No visible squares: 0
     0   1   2   3   4   5   6   7
   +---+---+---+---+---+---+---+---+
0  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
1  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
2  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
3  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
4  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
5  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
6  |   |   |   |   |   |   |   |   |
   |---+---+---+---+---+---+---+---+
7  |   |   |   |   |   |   |   |   |
   +-------------------------------+
   Please enter your move:
```

Listing 10.2: Typical output of menu and display of mine field

Example

To get an example of the how the minesweeper game plays, the game can be played on most windows computers and can be found for phones and other systems on the internet. For information about the game see the following website http://www.minesweeper.info/wiki/Windows_Minesweeper

Difficulty Levels

The game should have a menu that is presented to the user giving them an option of what difficulty level to play. The game should have three difficulty settings, beginner: 8 x 8 grid with 10 mines, intermediate: 16 x 16 grid containing 40 mines and expert: 24 x 24 grid containing 99 mines. An example of how the menu should look and function is given here:

Top Down Design of Minesweeper

Now that we understand how the game should work, we can start breaking the code down in to parts that we could define as functions. Figure 10.4 shows the initial breakdown of our code, it is

only separated into two parts, the function that displays the menu and the function that plays the game.

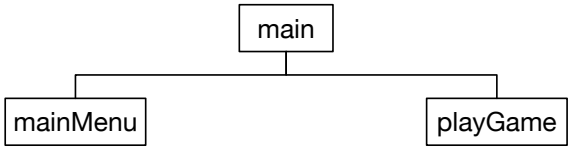


Figure 10.4: First breakdown of the functionality

Obviously, we will need to go into greater detail as to what happens in each of these functions.

We saw in the example output that the menu allows the user to choose the difficulty level of the game. This means that based on what number the user enters, the `mainMenu` function will be responsible for setting up the minefield for the game. We can put this in another function.

The `playGame` function can be much more varied, at the very least the function will have to get a move from the player many times so that should probably be a function. When the player makes a move, we will need to be able to show them the result so we will need to have a function that can display the currently visible minefield, also if the game is over we will need a function to display the whole minefield.

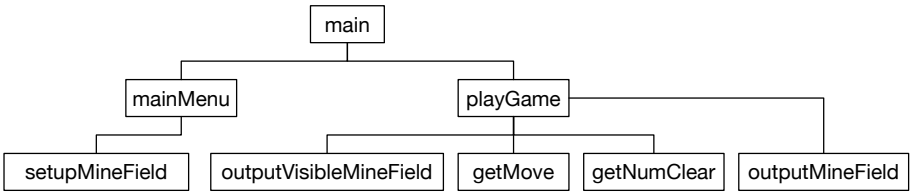


Figure 10.5: Second breakdown of the functionality

Returning to these functions, when setting up the minefield we will need to know the correct count of nearby mines for each of the squares in the grid. When getting a move from the user there are several things that we need to do. Firstly, we would need to check if the coordinates they entered are valid for the gird we are playing. Secondly we need to implement the functionality for the two moves that we can make, placing or removing a flag and clearing a square in the grid.

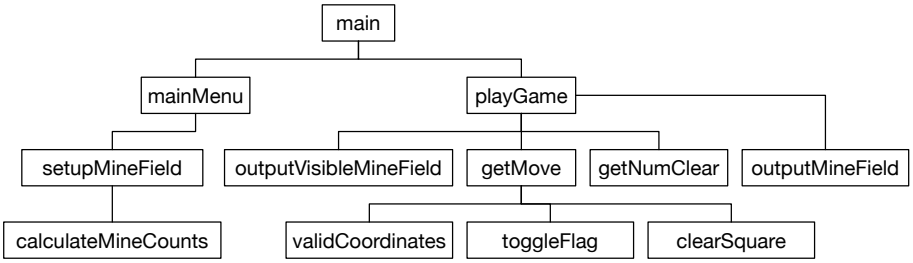


Figure 10.6: Third breakdown of the functionality

At this point most of the functionality has been described in one of the functions we have discussed. There are a couple of little pieces remaining, for example to set up the minefield we would need to count all of the mines around each square, so we should probably write a function to do this. As a part of this code, we would need to know which squares are our neighbours, the number of neighbours changes based on our position so we would need to be able to calculate which squares are neighbours for any given square. This will also be useful when we are implementing the `clearSquare` function.

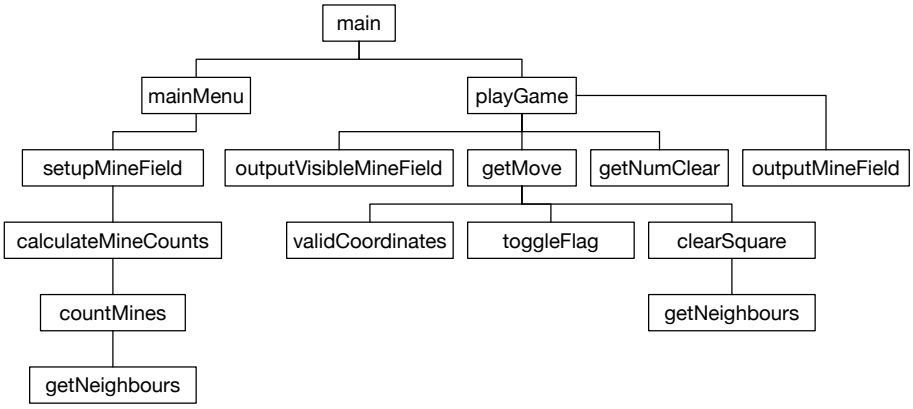


Figure 10.7: Fourth breakdown of the functionality

Based on the descriptions that we have made and the final diagram in figure 10.7, we could start creating this program by defining the functions and documenting what they will do.

```
1  def mainMenu():
2    """ Shows the main menu and gets the difficulty setting """
3
4  def playGame():
5    """ Plays the game of minesweeper until the field is cleared or a mine
6    is selected """
7
8  def setupMineField(difficulty):
9    """ Sets up the playing field to the correct size and fills in the
10   mines """
11
12 def outputVisibleMineField():
13   """ Prints the minefield to the screen where only the squares the player
14   have cleared are shown """
15
16 def getMove():
17   """ Asks the player to enter their next move and checks if it is valid
18   and then performs the move """
19
20 def getNumClear():
21   """ Counts the number of squares in the field that have been cleared """
22
23 def outputMineField():
24   """ Prints the minefield to the screen including the location of all of
25   the mines """
26
27 def calculateMineCounts(mineLocations):
28   """ sets the correct count for each square representing the number of
29   neighbours that are mines """
```

Listing 10.3: Definitions and Documentation of Minesweeper Functions

```
31  def countMines(mineLocations, x, y):
32    """ Counts the number of neighbours that are neigbours and also contains
33    mines for the square (x, y) """
34
35  def validCoordinates(x, y):
36    """ Returns true if these coordinates (x, y) are valid for the difficulty
37    level being played """
38
39  def toggleFlag(x, y):
40    """ places a flag (marks player thinks this is a mine) on the coordinates
41    (x, y) or removes a previously placed flag """
42
43  def clearSquare(x, y):
44    """ Clears the square defined by the coordinates x and y. If there is a
45    mine at this location the game is ended. If there are no mines in
46    neighbouring squares, then these are also cleared """
47
48  def getNeighbours(x, y):
49    """ Returns a list of tuples containing the coordinates of the neighbours
50    of this square (x, y) """
```

Listing 10.4: Definitions and Documentation of Minesweeper Functions

10.3 Python Modules

In Python we can use modules to help us break down large programs into smaller pieces. This is typically more useful in programs that are larger than the minesweeper program we used as an example in the previous section.

A module in Python is a file that contains some code (definitions and statements). Usually this will be the definition of some functions or classes (we are not studying classes in this book) and some code that may be used to prepare these functions for use.

When we execute a Python program, we do this by executing a file containing some code. This file is considered the main module of the program. The main module of a Python program can import (use) any number of other modules that are useful. This is how a larger Python program is created.

As a module is just a file containing some code, we can easily create and import our own modules. What we need to understand is what happens when we do so and how we can then make use of the code that we have imported.

When we import a module into our code, the code of the module is executed once. This usually means that there are a number of functions that will be defined for us to use. There may also be some code that is not inside a function definition, this code will be executed when we import the module.

The Python standard libraries contain a set of predefined modules that we can use. Some of these are very general purpose, like the math library containing functions for performing calculations and the random library that can be used to generate random numbers.

Before we study the details of modules and how they are imported, we will create a very basic example module.

The code on the left is saved in a file named `hello.py`. The text on the left shows some code and its result when executed in the interpreter. When the module is imported (the name imported (`hello`) matches the name of the file) the whole file is executed, meaning that the print function is executed and two functions are defined for us to use. These functions can then be used with a special syntax.

```
print("Hello")

def f1():
  print("Function 1")

def f2():
  print("Function 2")
```

```
>>>import hello
Hello
>>>hello.f1()
Function 1
>>>hello.f2()
Function 2
```

Figure 10.8: Example of module and output when imported

Namespaces

When we start to import modules, we are introducing a lot more names into our programs. In the previous example, the module only added two names of functions to be remembered in our program, but it could have contained many more. One problem we might face when importing code is that we might accidentally, import something (a function, variable, class, or something else) that has the same name as a variable or function that we have defined. This problem is known as a name clash and Python uses namespaces to prevent this from happening.

Name Clash Example

Consider the following example, we are writing some large and complicated code for processing lists of numbers. We come to a part where we are required to create a list containing all of the values of another list doubled, e.g. instead of `[1,2]` we need `[2,4]`. We find a module (`module1`) that contains a function (`double`) that we can use to do this and we import it into our code.

```
def double(lst):
  """ returns a new list with each number doubled, for
  example [1, 2, 3] returned as [2, 4, 6] """
```

Listing 10.5: `module1` a module containing a function named double

Later, while working on anther piece of functionality in the same program we come to a part where we are required to create a list of tuples based on the values in another list, in this case we want each item in the list to be a tuple containing the same value twice, e.g. instead of `[1,2]` we need `[(1,1),(2,2)]`. We find a module (`module2`) that contains a function (`double`) that we can use to do this and we import it into our code.

```
def double(lst):
  """ returns a new list with each number duplicated, for
  example [1, 2, 3] returned as [(1,1), (2,2), (3,3)] """
```

Listing 10.6: `module2` a module containing a function named double

However, we know realise that both functions have the same name, without some way to distinguish between them there would be no way to know what the result of the following code would be.

```
import module1
import module2

l = [3,8,14]
res = double(l)
```

Listing 10.7: Code using the two modules

This is the reason that Python uses namespaces to organise the names we can use in our programs. Each module we import has its own namespace. This is where the names of all of the variables, functions and other stuff in this module are remembered. We can use this namespace when we want to access or use these items.

The syntax we use is the dot operator (we have seen this used with lists). The first part is the name of the module (the name that we imported) followed by a dot (.) which is then followed by the item we wish to use (variable or function).

```
module_name.item_name
```

Listing 10.8: The dot operator for using an item in a module

Using this syntax we can correct the previous code and use both of the imported functions without any name clash. We get to choose which function we want to use because we must use the full name, e.g. `module1.double(l)` or `module2.double(l)`.

```
import module1
import module2

l = [3,8,14]
res = module1.double(l)

res2 = module2.double(l)
```

Listing 10.9: Code using the two modules

Namespaces in Python

This example of a name clash and how it is prevented in Python introduces us to the idea of namespaces, but how many are there and how are they related to each other? In Python, there are different levels of namespaces.

At the top level there is the built-in namespace. This contains all of the built-in functions that we can use (`print`, `int`, `input`, etc.). This is created automatically when we start executing a program.

For every module that we use in our program, a global namespace is created. These are stored in the built-in name space. The file that we are executing (the main module) also has a global namespace created for it. These global namespaces contain the names of all of the variables and functions in the module. Figure 10.9 shows a representation of the namespaces that would be created by the code in the last example.

The main module has the name `__main__` and the other imported modules use the names they are imported as. In the global namespace of the main module, we can see that the names of the

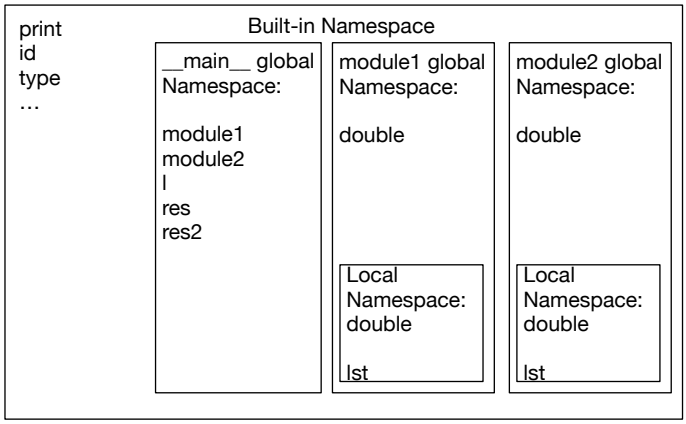


Figure 10.9: Example of built-in namespace containing several global namepaces

imported modules are included as well as the names of the variables declared in the code and there would also be the names of functions if they were declared in this module too.

The final type of namespace is a local namespace. Local namespaces are created when a function is called. A local namespace contains all of the local variable that are defined in the function. As soon as the function is finished, the local namspace is removed. These local namespaces are created inside a global namespace.

Importing Modules

Depending on how we want to use a module or the functions inside it we can import it in different ways. If we want to use many of the functions in the module, we can import the whole module. If there is only part of the module that we want use, we can use an from import statement.

Full Import

This is what we have used in the previous examples. Here is an example of the syntax, we use the keyword import followed by the name of the module (name of the file without the extension).

```
1 import mod_name
```

Listing 10.10: Importing a Whole Module

In this example we have imported the module named `mod_name`. This will be loaded from the file `mod_name.py`. When we use this type of import statement, a global namespace is created for the module containing all of the variables and functions in the module.

Figure 10.9 shows us an example of this. It is important to understand that these names do not become a part of the namespace of the main module. We can only access these values or functions using the name of the module that they are contained in.

This means that when we import `math` module, we must use the functions (like `factorial`) by first using the name of the module e.g. `mat.factorial(5)`. This is because factorial is not in the main modules namespace, but the module `math` is. Using the name of the math module, we can find the factorial function that is in it's namespace.

From Import

If we only want to use part of a module and not import the whole module, we can use the from import statement. Here is an example of the syntax, we use the keyword from followed by the

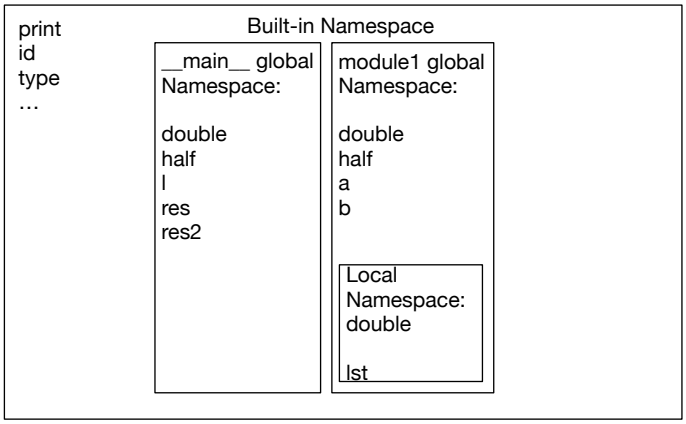


Figure 10.10: Example of built-in namespace after from import statement

name of the module and they keyword import. After the keyword import we list the parts of the module that we want to use. These are separated by commas if there are more than one.

```
1 from module import something
```

Listing 10.11: Importing only part of a Module

There are several differences between the two import statements. Firstly, we do not have access to any of the other parts of the module in our code, only the part that we named is available to use. Secondly, when we use a from import statement, the items we import become a part of our namespace. The consequence of this is that we do not need to use the name of the module to use the item.

For example, if we use the code `from math import factorial` instead of importing the math module we can use the factorial function directly using its name. E.g. `factorial(5)`

Consider the following example. Figure 10.10 shows a representation of the namespaces that would be created by the code being executed.

```
1 from module1 import double, half
2
3 l = [3,8,14]
4 res = double(l)
5 res2 = half(l)
```

Listing 10.12: Importing functions `double` and `half` from module

There are several differences that should be noted between this representation and that of figure 10.9. Firstly, the name of the imported module is not a part of the main modules namespace, but the name of the functions `double` and `half` are. This means that we can use these functions, but have no way to access any other part of the module they are imported from.

Import Everything

We can also use this syntax to import everything from a module into the namespace of our module. This is done by using the character asterisk (*) instead of the name of the items you want to import. This is known as a wildcard and results in all of the contents of the module being included in our namespace.

The drawback of using this type of import statement is that it makes name clashes more likely. This is because we may not know the names of all of the items that we are importing into our namespace. The result of this is that we might replace something we have created or imported with something else.

This is something that we can always do. For example we might accidentally replace a built in function like `sum` with a variable using the same name. The result of this is that when we try to use the function it will not work.

```
1 l = [1,2,3,4]
2
3 sum = 0
4
5 total = sum(l)
```

Listing 10.13: Replacing the function sum with a variable

In this example, the final line of code would normally result in the sum of all of the numbers in the list being stored in the variable `total`. However, because at some point before this the function sum was replaced with a variable it does not work. Instead this causes our program to crash.

Renaming

When we are importing modules or functions we can rename them. This can be done to shorten or simplify the code we have to write or to prevent name clashes from happening.

If we want to rename a module, we add a little more code to the import statement. After the name of the module we are importing we add the keyword `as` and the name we want to use for the module. For example we the statement `import math as m` imports the match module but renames it to `m`.

The consequence of this is that when using something from this module we only need to use the name we have given it. Fr example to use the factorial function we would use the code `m.factorial(5)`.

We can also rename functions that we import using the from import statement. This is done using the same syntax, we add the keyword `as` and the new name to the statement. For example, we could solve the name clash in our earlier example by renaming the functions we import. E.g. `from module1 import double as d1` and `from module2 import double as d2`.

Executing Modules

When the code we write is imported as a module the statements in the file are executed. This allows us to perform any necessary set-up like creating variables that will be required. Additionally, this will execute the statements defining the functions that are a part of this module.

This is also the case when we execute our file as the main module of a Python program. There is one small difference between when we execute our code as the main module, and when code we write is imported as a module. This is that the name of the module is different.

When a module is executed (as the main module or when imported) it is given a name. This name is remembered in the variable `__name__`. When we execute a python file in the interpreter it is given the name `"__main__"`, however when we import a module it is given the name of the file it is stored in. E.g. if we import the code from `sean.py` it will be given the name `"sean"`.

This information is often not very important, but it can be used to write some code that is not executed when the module is imported. It is very common to see modules containing some code at the end that is used to test if the functions written are working correctly. This code is designed to only be executed when it is the main module and not when it is imported.

This can be done using an if statement with the condition `if __name__ == "__main__":`. Here we are checking if the name we have been given is the same as `"__main__"`. As this will only be

the case when we are the main module and not when we are imported, we can execute our test code in this if statement.

```
1 def calcNumber():
2   return 8
3
4 print("module loaded")
5
6 if __name__ == "__main__":
7   if calcNumber() == 8:
8     print("calcNumber working")
```

Listing 10.14: If statement to prevent test code being executed when imported

Consider this example. If we execute this code, several things happen. First the statement starting on line 1 defines the function `calcNumber`. Second the print statement on line 4 results in the text `module loaded` being output to the screen. Finally, we execute the if statement starting on line 6. The value of the variable `__name__` matches the string literal we are comparing it to so the result of the comparison is `True`. This means that we execute lines 7 and 8 and the text `calcnumber working` is output to the screen.

```
1 module loaded
2 calcNumber working
```

Listing 10.15: Output of the previous example when executed

Alternatively, lets consider what happens when someone imports this code. Lets assume that it is stored in a file named `calc.py`. This means that we are importing the module using the code `import calc`.

The code is still executed and several things happen. First the statement starting on line 1 defines the function `calcNumber`. Second the print statement on line 4 results in the text `module loaded` being output to the screen. Finally, we execute the if statement starting on line 6. The value of the variable `__name__` is `"calc"` and does not match the string literal we are comparing it to so the result of the comparison is `False`. The code on lines 7 and 8 is not executed and there is no more output.

```
1 module loaded
```

Listing 10.16: Output of previous example when imported

Chapter 11

Objects in Programming

One of the important ways in which humans see the world is as being made up of objects. Objects are just things, but we can describe them by their attributes and behaviours. This allows us to group things together so we can more easily understand the world.

The attributes of these objects are things we can use to describe them. For example, when we are describing people we might use height, weight, name etc. because these are common to everyone. I.e. everyone has a height, everyone has a weight, everyone has a name.

The behaviours of these objects are the things that they can do. For example, when describing people we could use move, sleep, talk, etc. These are common to all people, everyone can move, everyone can sleep, everyone can talk.

Grouping objects like this allows us to understand more about the world by seeing the similarities between things. For example, when I see a car I have never seen before, because I know it is a car I know what kind of attributes and behaviours it has. I know it will have a colour, weight, height, top speed, even if I do not know what those values are. I also know it will have behaviours like drive forward and reverse even though I do not know exactly how fast it will do those.

11.1 Object-Oriented Programming

In this book we have been studying a technique of programming known as procedural programming. This is where the creation of programs is based on defining and using functions (procedures) in our code.

Object-oriented programming is a technique of programming where the creation of programs is based on defining and using objects. These objects would generally have a special type of function inside them, called methods.

Python is a language that can be used to write programs using the procedural technique. However, Python can also be used to write programs using object-oriented techniques.

It can be difficult to learn to fundamentally different techniques of programming at the same time. This book only covers a little about objects and their use. Other books can be studied to become more familiar with the concept of object-oriented programming.

Objects In Programming

Objects in programming are based on how humans view the world. That means that software objects are things with attributes and behaviours. Attributes of objects are data or variables that they store, while behaviours are the things it can do or the methods it contains.

In Python, all values are objects. This is one of the reasons that we must have a little understanding of object even though we are studying a different technique of programming. Every value represented in Python programs is an object, this includes all the data types that we have used such as numbers, strings, list, and tuples. This means that all of these values can be viewed in terms of their attributes and behaviours.

In this chapter, we will be focusing only on the behaviours of objects. This is what we need to know and understand to be able to interact with the objects in our programs. Behaviours in objects are represented by methods. These are a type of function, the difference between functions and methods is that a method belongs to an object. This means that the method will have access to the data inside the object, where a function would need this data to be given as a parameter.

Lets compare these two approaches to the task of sorting a list of numbers. Python has the ability to use both techniques to achieve this task. First, we will see how a list of numbers could be sorted using a function. Then we will see how this can be done using a method.

To sort a list of numbers we can use the function `sorted`. This function takes a list as a parameter and returns a list containing the same values sorted into ascending order. This example shows how this is achieved. This is the procedural technique of sorting.

```
1 lst = [5,6,3,6,2,4,5]
2 print(lst)
3 lst = sorted(lst)
4 print(lst)
```

Listing 11.1: Sorting a list using a function

To sort a list of numbers we can also use the method `sort`. This method does not take any parameters because it has access to all of the data in the list already. It does not need to return a value because it can change the values in the list.

```
1 lst = [5,6,3,6,2,4,5]
2 print(lst)
3 lst.sort()
4 print(lst)
```

Listing 11.2: Sorting a list using a method

This example shows the use of the dot notation to call a method. What is important is that the method belongs to the list object, this is why we do not need to supply it with any more data. We have seen examples of other methods being used in earlier chapters of the book.

Methods such as `append`, `insert`, `count`, and `index` which we used with list or sequence objects are examples of methods which require parameters in addition to having access to the items in the sequence.

11.2 Turtle Graphics

This section of the book introduces a module called turtle that provides a basic graphics library for Python. This purpose in this section is to introduce several objects that have different behaviours that can be used to help understand the concept of objects.

Using this library we can draw on the screen, we do this by moving a turtle around the screen. Each turtle is an object with methods we can call to control it. We can even have many turtles on the screen at the same time. But before we can do this we must import the module and set up the screen.

Set-up

To create the turtle window on our computer, we must follow a few steps. First we need to import the turtle module. Second we need to use the function in this module called `setup`. Once this function is executed we should see a window appear, this is where our drawings will be.

If we want to make any changes to the setup of the screen, like changing the text on the window, we need to access the window object. This is returned by the function `Screen` in the turtle library. The window object only has a small number of methods we can use, there is a `setup` method that can change the size and shape of the window, a `bye` method that will close the window and a `title` method that will set the text on the top of the window.

```
1 import turtle
2 turtle.setup()
3 window = turtle.Screen()
4 window.title("Seans Magic Turtle")
```

Listing 11.3: Creating the Turtle Graphics Window

This example shows how the turtle graphics window can be created shown on the screen. Getting the window object is only necessary if we want to change the text of the window or to change its size. If we do not need to do this, then only the code on lines 1 and 2 is required.

Figure 11.1 show how the window will appear on the screen. This image is annotated to show the coordinate system used in turtle graphics. The origin is shown in the center of the screen with the coordinates of the 4 corners also shown. The window may be a different size depending on the operating system you are using.

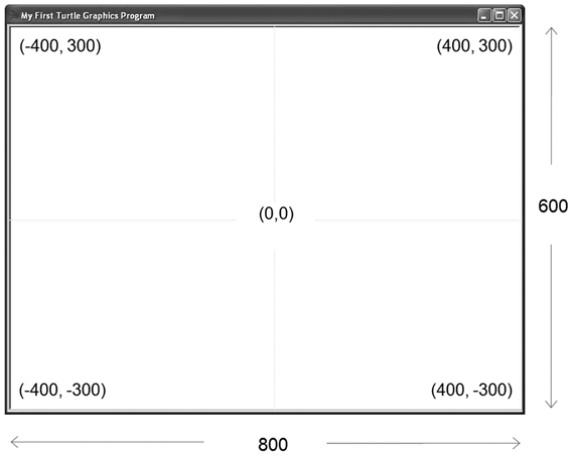


Figure 11.1: The Turtle Graphics Coordinate System

The Turtle

The turtle library is for drawing on the screen by moving a turtle around the screen. To be able to move the turtle, we need access to the object to be able to call its methods. When we call the `setup` method, a single turtle is created by default. To access get access to this object, we need to use the function `getturtle` from the turtle module and save the result in a variable. E.g. `myturtle = turtle.getturtle()`. This will also make the turtle appear on the screen.

The turtle graphics library is all about drawing on the screen. The turtle object is the tip of a pen, we draw by moving the turtle around the screen. The movement is achieved by calling methods of the turtle object and passing parameters. The methods in the following table are what we use to move the turtle. When the window is created, the turtle will be initially placed at the origin (coordinates (0,0)). Any movement after this point will result in a change to the location of the turtle and some drawing on the screen.

Table 11.1: Turtle Object Movement Methods

Method	Explanation	Example
`setposition`	Moves the turtle to the coordinates passed as parameters	`myt.setposition(0,0)`
`speed`	Changes the speed that the turtle moves at, parameter is a number between 0 and 10	`myt.speed(0)`
`forward`	Moves the turtle forward in the direction it is facing by a certain distance (parameter)	`myt.forward(20)`
`backward`	Moves the turtle backwards in the opposite direction than it is facing by a certain distance (parameter)	`myt.backward(20)`
`setheading`	Changes the direction that the turtle is facing to a set direction (degrees as parameter)	`myt.setheading(90)`
`left`	Turns the direction the turtle is facing left by a number of degrees (parameter)	`myt.left(20)`
`right`	Turns the direction the turtle is facing right by a number of degrees (parameter)	`myt.right(20)`
`heading`	Returns the turtles current heading in degrees	`h = myt.heading()`
`xcor`	Returns the turtles current x coordinate	`x = myt.xcor()`
`ycor`	Returns the turtles current y coordinate	`y = myt.ycor()`

Absolute Movement

The first method we will learn about is `setposition`. This method allows us to move the turtle to a set of coordinates on the screen. These coordinates must be passed as parameter when the method is called, x first and then y. The turtle will move in a straight line between its current location and the coordinates. This will result in a line on the screen.

We can use the `setposition` method many times to draw a number of lines. We can draw a shape this way, but we must know the coordinates of all of the lines before we can move the turtle. The following example, shows how we can draw a square by having the turtle move to 4 sets of coordinates in sequence.

```
4 myt = turtle.getturtle()
5 myt.setposition(100, 0)
6 myt.setposition(100, 100)
7 myt.setposition(0, 100)
8 myt.setposition(0, 0)
```

Listing 11.4: Drawing a square using absolute movement

Speed

We can set the speed that the turtle moves when it is animate on the screen. To do this we can use the method `speed` with an integer parameter between 0 and 10. The numbers relate to speed with 10 being fast and reducing down until 1 which is the slowest speed. However, the number 0 represents the fastest possible speed. We can set the speed of each turtle independently (if we have more than one), e.g. `myt.speed(0)`

Relative Movement

Much of the rest of the methods relating to moving the turtle are for relative movement. This means that instead of telling the turtle a specific set of coordinates to move to, we give it instructions like move forwards, move backwards, turn left, and turn right. This type of control of the movement is more flexible because we can use the same code to draw a shape at any position on the screen without the need to recalculate the correct coordinates.

For motion we have the methods `forward` and `backward`. Both of these methods take a single parameter which is the distance that the turtle should move. The turtle will move in the direction that it is facing (for forward) or the opposite direction (for backward), this means that we also need some measure of control over the direction the turtle is facing.

We can set the absolute direction the turtle should face using the `setheading` method. This method takes a number parameter between 0 and 360 that represents the angle that the turtle should face in degrees. Alternatively, we can use the methods `left` and `right` which allow us to turn the turtle to the left or right by a number of degrees.

```
4  myt = turtle.getturtle()
5  myt.speed(0)
6  i = 0
7  while i < 4:
8      myt.forward(100)
9      myt.left(90)
10     i = i + 1
```

Listing 11.5: Drawing a square using relative movement

This example draws the same square by having the turtle move and turn set amounts instead of moving to a set of coordinates. The difference is that this code will work to draw a square on the screen no matter where the turtle is located and which direction it is facing.

Locating the Turtle

The final three methods in the table relate to finding out about the turtle. We may want to know in our code where the turtle is or which direction he is facing so that we can make a decision about what to do next.

The method `heading` will tell us the current heading of the turtle in degrees. We can remember this in a variable, use it in calculations or use it to decide where to go next.

The methods `xcor` and `ycor` both return a single number representing the turtles current x or y coordinate respectively. These values can be used again to make decisions or direct the turtle on the next place to go.

Turtle Appearance

We can change the appearance of the turtle on the screen or even show and hide it. This does not have any effect on the drawings we are making, but we can use it to create animations.

Table 11.2: Turtle Object Shape Methods

Method	Explanation	Example
`shape`	Changes the icon of the turtle on the screen. Acceptable values are: `"arrow"`, `"turtle"`, `"circle"`, `"square"`, `"triangle"`, and `"classic"`	`myt.shape("turtle")`
`hideturtle`	The turtle is no longer shown on the screen	`myt.hideturtle()`
`showturtle`	The turtle is shown on the screen	`myt.showturtle()`

The appearance of the turtle can be changed using the `shape` method. This method accepts a list of pre-set parameters that chooses one of the defined shapes to be shown on the screen. If we do not want the turtle to be shown on the screen, we can use the method `hideturtle` and it will no longer be visible. If we want to return the turtle to visibility, we can use the `showturtle` method. Neither of these methods require any parameters.

The Pen

We can change the effect that the turtle has while drawing by changing the settings of the pen. This can be done by calling some of the following methods on the turtle object.

Table 11.3: Turtle Object Pen Methods

Method	Explanation	Example
`penup`	Lifts the pen from the screen so we will not draw as we move	`myt.penup()`
`pendown`	Places the pen against the screen so we will draw as we move	`myt.pendown()`
`pensize`	Sets the thickness of the line drawn to an integer value (parameter)	`myt.pensize(1)`
`pencolor`	Changes the colour of the line drawn based on a string parameter. String parameter can contain the name of the colour, such as `"red"`, `"yellow"`, and `"blue"`	`myt.pencolor("red")`

The methods `penup` and `pendown` are used to stop and start drawing. We can use them to change the position of the turtle without leaving a line between our current position and the location we are moving to. This can be done by lifting the pen up, moving the turtle and then placing the pen down.

The methods `pensize` and `pencolor` allow us to change the thickness and colour of the lines that we draw. This allows us to create more interesting pictures on the screen.

Multiple Turtles

By default, a single turtle object is created for us to draw on the screen. However, we can use multiple turtles at the same time to draw in different locations on the screen. When we need another turtle we can add one using the function `Turtle` in the turtle module. Just like the `getturtle` function, we will need to remember the turtle object in a variable so that we can control it using the methods available.

The code `t2 = turtle.Turtle()` can be used to create another turtle. This will appear at the same location as the default turtle.

Example

Lets look at an example of drawing on the screen with multiple turtles. This example will simultaneously draw spirals on the screen and requires the use of methods from all of the sections that we have seen so far. Figure 11.2 shows the result of the code.

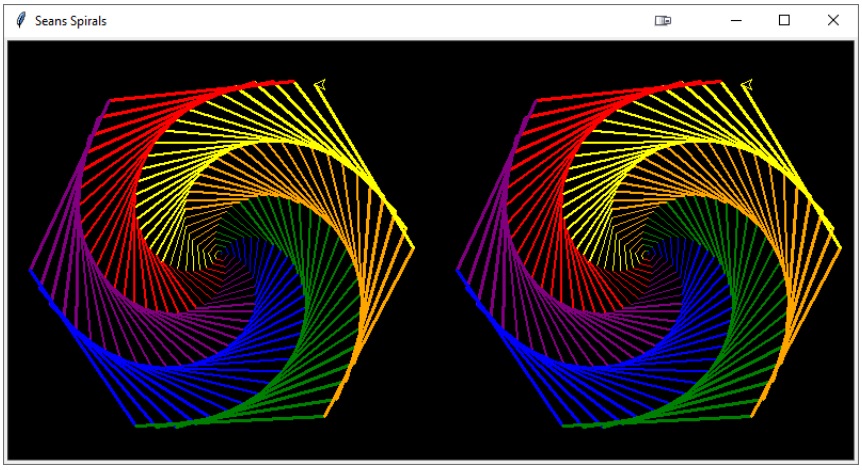


Figure 11.2: Turtles coloured spirals

We will complete this example in several parts, first we will need to do some set-up. We need to import the library, set the text of the window and then prepare for drawing the spirals.

In order to draw the spirals at the same time, we need to have multiple turtles, we will remember these objects in a list. This will make performing the actions for each turtle easy as we can use a loop. The spirals are drawn in many colours, so we need to change the colour frequently. We can remember the name of the colours in a list to help make this easier.

```
1  import turtle
2  turtle.setup(width = 800, height = 400)
3  window = turtle.Screen()
4  window.title("Seans Spirals")
5  colours = ['red', 'purple', 'blue', 'green', 'orange', 'yellow']
6  turtle.bgcolor('black')
7  turtles = [ turtle.getturtle() , turtle.Turtle()]
```

Listing 11.6: Setting up for Drawing Spirals

We will also want to be able to move the turtles to their starting positions without drawing any lines, we can write a function to do this. This function can also take care of little details like hiding the turtle and setting its speed to the fastest setting.

```
9  def moveto(t, x, y):
10     t.hideturtle()
11     t.speed(0)
12     t.penup()
13     t.setposition(x,y)
14     t.pendown()
```

Listing 11.7: Function to move a turtle to coordinates

This function can be use to move a turtle (first parameter) to a set of coordinates (second and third parameters). Using a function this way means that every turtle can be moved using only a single line of code.

Next, we will want write the code to draw the spiral. To draw the spiral for a single turtle would return a loop. We are repeating the following set of actions:

- Change the pen colour
- Change the width of the pen
- Move forward by a changing amount (x)
- Turn left by an amount (59)
- Increment the amount we move by (x)

The first 4 actions must be repeated for every turtle, so we in reality need to use nested loops. But before we can begin drawing the spiral, we need to move the turtles into the correct positions.

```
16  moveto(turtles[0], -200,0)
17  moveto(turtles[1], 200,0)
18  x = 0
19  while x < 180:
20      for turt in turtles:
21          turt.pencolor(colours[x%6])
22          turt.width(x/100 + 1)
23          turt.forward(x)
```

Listing 11.8: Function to move a turtle to coordinates

This code uses a clever trick to choose the colours from the list in sequence. There are 6 colours in the list, so they have indexes 0 to 5. The first line of code in the loop sets the colour to index of `x` modulus 6 which will always be a number between 0 and 5. This way the colours are changing with every line drawn, but they always appear in the same sequence.

Chapter 12

File and String Processing

This chapter introduces the process of reading information from files and writing information to files. First we will study a little bit about files and how they are stored, then we will look at the functions and methods we can use to both read information from files and write information into files. Finally, we will study how we can process the data that we have read from files using a number of methods of string and lists.

12.1 Files

Every day there are huge amounts of data generated by billions of computers. This data usually ends up stored in files on computers or servers somewhere. But how is this data stored?

This data is represented as binary information which can be processed by and stored on computers. Binary is all that computers can understand, but it is less useful to us because it is difficult to understand. Usually when we look at this type of data in a file, we view it in hexadecimal instead of binary. This is because 4 bits of binary can be represented with a single character of hexadecimal.

To make things even more difficult, the numbers represented in these files usually have a special meaning. For images, each number might represent the brightness of a particular colour for one pixel of the picture. For a sound file, each number could represent the amplitude of the sound wave at an instant in time.

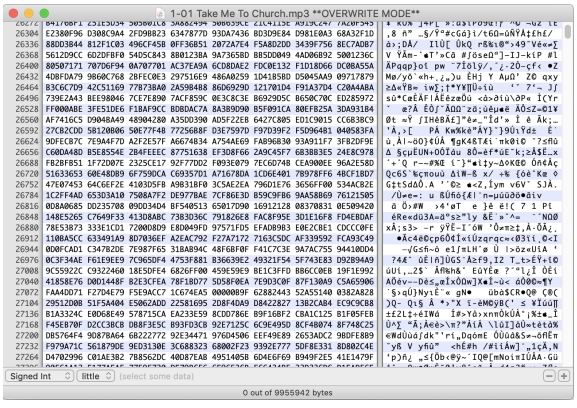


Figure 12.1: Hexadecimal data contained in a mp3 file

Thankfully, applications on our computers can understand these numbers and their meanings. When we open and image file or a music file (like shown in figure 12.1) the application automatically converts this data into something that we can understand.

Text Files

A common type of file used is called a text file. Text files are almost universally understood by all programs and operating systems. These files are represented in the computer in binary (just like the other file types we have seen), however, the meaning of the data in this files is very well understood. So almost any application can display the contents of a text file.

Text files are just data files where we use an encoding scheme such as ASCII or Unicode to store our data. Figure 12.2 show a text file when viewed using a hex editor. Hex editors show the data in a file represented using hexadecimal. This particular hex editor shows the individual values on the left each byte of data shown by two characters and a representation of the data as text on the right side. In this example, because the data is text, we can understand the right side of the screen.

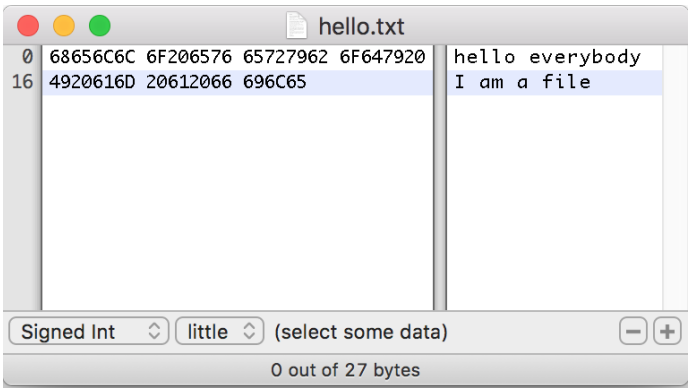


Figure 12.2: A text file when viewed using a hex editor

The important point to remember is that the computer does not see a file containing "hello" it sees the hexadecimal bytes `68 65 6C 6C 6F`, these bytes are the numbers 104, 101, 108, 108 & 111 in decimal format. The computer automatically translates these bytes into "hello" for us to understand, however they are just 5 bytes of data.

ASCII

ASCII - American Standard Code for Information Interchange, is a character encoding standard for electronic communication. ASCII codes represent text in computers, telecommunications equipment, and other devices.

Figure 12.3 shows the table of ASCII codes. These can be used to represent the letters of the Roman alphabet as well as numbers and common punctuation. All of the numbers below 32 are invisible characters and have a special meaning.

The new line character (`\n`) is represented by number 10 (hex 0A) on Apple or Linux computers and by the numbers 13 and 10 (hex 0D0A) on Windows computers. This difference is generally handled automatically by the systems reading the file, however sometimes if this is not handled a fill may appear to have all of its contents on a single line.

These new line characters are used to define the structure of text files. What inside the computer is just a single long sequence of numbers is represented to us as a number of individual lines of text in a file. Each of these lines of text either ends with a new line character or is the final line in the file. When reading text information from files, the data is usually read or processed one line at a time.

Text files can be given any name or extension. The extension `.txt` is most commonly used, but we can also use other extensions such as `.csv`, `.py`, `.c`, `.java` and many many more. What is common about text files is how the information is encoded inside the file.

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

asciicharstable.com

Figure 12.3: ASCII Codes for text representation

12.2 File Input and Output

When we first learned to program, we learned how to make our programs more useful by allowing them to interact with the user. This allows the programs that we write to be used again and again with different input and therefore different results. This is done using input and output functions such as `input` and `print`.

Input and output with files is a little different in Python. The big difference is that we are using methods and file objects to perform input and output instead. Following after this, we will need to use functions and methods to process the data to get the information we want from it.

The following are the typical steps that we use when we are working with files;

1. Open the file
2. Check that it has opened correctly
3. Read or write some data
4. Close the file

Opening Files

The biggest difference between reading and writing information on the terminal and reading and writing information in files is that there is a single terminal to read/display information, but there are many different files that we can read/write. When we want to read information from a file or write information to a file we have to specify what file we want to use. This process is know as **opening** a file.

To open a file, we use the function `open`. This function requires two parameters, both of which are strings. The first parameter is the name (and path) of the file we want to open, including the extension. The second parameter is the mode that we want to open the file in, this determines what we can do with the file once it is opened.

File Modes

There are several different modes to choose from when opening a file. The mode we choose is based on what we want to do with the file.

This different modes are useful in different situations. If for example, we wanted to read a file named `myfile.txt` we would open it using the code `open("myfile.txt", "r")`. Normally, we will only use the reading (`"r"`) and writing (`"w"`) modes.

Every time we call the open function a file object is created and returned (if the file is found successfully). This is how we can interact with the file, so we should store this object in a variable. E.g. `file_var = open("myfile.txt", "r")`. When we need to work with multiple files we can create many file objects, we then choose which file we will read/write based on the name of the variable.

Closing Files

When we are finished working with a file, it should be closed. This is not very important if we are only reading a file, but when we are writing it makes sure that all information is written into the file before the program finishes. To do this we use the method `close`. The method, does not require any parameters because we call it using the file object that we want to close. E.g. `file_var.close()`

If `close` is forgotten when we are finished writing to a file, some data may not be written into the file as expected.

Opening and Closing Files Safely

When we attempt to open a file for reading, it must exist before we do so. If the file does not exist, is in a different location, or the name is spelled differently then the code will not work. The following example shows the output when we try to open a file named `"example.py"`. When this error happened it also crashed the Python program it was a part of.

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    open("example.py","r")
FileNotFoundError: [Errno 2] No such file or directory: 'example.py'
```

Listing 12.1: Error shown opening a file that doesn't exist

There is a statement called a `with` statement we can use in Python to safely open the file we want to read or write. The with statement is a compound statement, after the first line of the statement, we add the code that should be executed when the file is opened correctly. This could should be indented by one level more than the with statement.

If the file is opened correctly, our code will be executed and the file will be closed automatically when it is complete. If the file is not opened correctly, then the code is not executed at all.

The with statement replaces the call and assignment of the open function. In addition, when using a with statement we do not use assignment to remember the file object. Instead, we put the name of the variable after the keyword as.

```
with open("file_name.txt", "r") as var_name:
    # read data
    # proccess data
```

Listing 12.2: With statement

12.3 Reading Text From a File

In this section, we will study how the methods of the file objects can be used to read text from the a file.

Order

File objects are pretty smart and one of the things that they do automatically when reading or writing is they remember their place in the file. This is done automatically in the background so we never have to consider it.

The file objects remembers the text we have read so that when we ask for some more, it will give us the next piece of text. The methods we will be studying are based around lines of text. Each time we call the method, the file object will give us the next line of text. By repeating this action again and again we can read all of the text in the file.

Reading a Line of Text

The method `readline` can be used with a file object to read a single line of text from the file. `readline` does not require any parameters because the file object contains all of the required information for the method to work. This line of text we read from the file will also include the new line character from the end of the line. When we have reached the end of the file and there is no more text to read, the method will return an empty string (`""`). We can use this method in a loop to read all of the text in a file.

```
1 with open("file_name.txt", "r") as var_name:
2     line = var_name.readline()
3     print(line)
```

Listing 12.3: Read a single line of text from a file and print it

Reading a Whole File

If we want to read all of the text in a file, there are different methods we can use. We could use the `readline` method again and again in a loop. This way we can process the data in a single line of the file before reading the next line of the file. Lets look at how we would structure this code by studying an example that reads all of the names stored in a text file.

```
1 with open("names.txt", "r") as names:
2     line = names.readline()
3     while line != "":
4       print(line, end = "")
5       line = names.readline()
```

Listing 12.4: Read all of the names in a file

The steps carried out after the file is opened are important. First we must read a line of text. This is assigned to the variable line, which would not exist if we had not done so. This line of text must be checked in the condition of the loop to make sure that the file was not empty. Because we are comparing the line to an empty string literal the loop will only execute if a line of text was successfully read from the file.

In this example the processing of the line of text is simply to print it, but in later examples we will look at this in more depth. However, once that processing is complete, the only other

step we must do is to read the next line of text from the file. This should always be the last step inside the loop so that when we next check the condition of the loop we are checking if the method successfully read a line of text.

Alternatively, Python provide a method that we can use to read all of the text in a file in one single statement. The method `readlines` instead of returning a single line of text as a string returns a list containing all of the lines of text from the file in order. We can then loop through all of the strings in the list to process the text.

```
1 with open("names.txt", "r") as names:
2     lines = names.readlines()
3     for line in lines:
4       print(line, end = "")
```

Listing 12.5: Read all of the names in a file

The structure of this code is much easier because we do not have to think about checking when we have reached the end of the file. However, it may be very slow to use this method when the file contains a lot of data. This is because all of the data would have to be read before we could start processing it. Using the first method, we process each line of data as it is read from the file.

12.4 Writing Text Files

When we want to write data into a file, we can use the method `write` in the file object. This method takes a string as a parameter and writes that text into the file. This text is placed after the last text that was written into the file and a new line character is not added automatically (unlike the `print` function). If the data we are writing into the file should be placed on separate lines, then we need to add the new line characters tot he file ourselves.

```
1 with open("test.txt", "w") as test:
2     test.write("Sean really likes programming\n")
```

Listing 12.6: Write a single line of text into a file

Example - Sorting Names

Lets study an example where some actual processing of the data takes place and the results are written into another file. The example file we have been using contains several names, but these names are not in alphabetical order. We will look at an example that opens this file, reads the names, sorts them and then writes the names into another file.

```
1 with open("names.txt", "r") as names:
2     lines = names.readlines()
3     lines.sort()
4     with open("sortednames.txt", "w") as snames:
5         for name in lines:
6             snames.write(name)
```

Listing 12.7: Sorting the names

Here we can see that first we open the names file for reading and use the `readlines` method to read all of the lines of text in the file and store them in a list. We use the `sort` method in the list object to order the strings in the list alphabetically before we write them into another file.

The second file is opened in the same way, but uses the file mode `"w"`, then we loop through all of the lines in the file and call the `write` method once for each string. We do not need to add any new line characters in this example because the strings read from the file in the first part already contain the new line characters.

```
Sean Russell
Abey Campbel
David Lillis
Vivek Nallur
Shen Wang
Ruihai Dong
```

```
Abey Campbel
David Lillis
Ruihai Dong
Sean Russell
Shen Wang
Vivek Nallur
```

Figure 12.4: Contents of files `names.txt` (left) and `sortednames.txt` (right)

12.5 Processing Strings

One of the things missing from the previous sections is how to read and write other forms of data like numbers from files. The reason this is missing is because we can't, we instead need to do some processing on the text we read to convert it to the correct type. We have been doing this using the `int` and `float` with the strings we got from the user with the `input` function.

In this section we will study several of the methods we can use with string objects to help process the text data. Because functions like `int` and `float` can only work for a single value (e.g. `"123" but not "12 34"`) we will need to be able to break strings into several pieces that we can process independently.

Splitting Strings

The `split` method can be used to break a string into multiple pieces. The method takes a string as a parameter (know as a delimiter) and uses this to create a list containing smaller pieces of the original string. Lets look at an example `"Sean is my name".split(" ")` will return a list with the following contents `["Sean", "is", "my", "name"]`.

What happens is that every time we find the parameter in the string (in this case a space character) we break the string at that point. In the example there were three space characters, the point where each one was in the string is now the start and end of the separated strings.

The delimiter that we use is never included in any of the strings that are in the list returned. It is less obvious in the previous example because spaces are invisible. Lets look at another example, the code `"abcdeffgfde".split("f")` would return the list `["abcde", "", "g", "de"]`. Here it is much more apparent that the f character is not contained in the strings returned.

Joining Strings

The `join` method is the reverse of the `split` method. Instead of splitting a string a list of smaller strings using a delimiter, the join method joins a list of strings (parameter) together using the string as a separator. Here, what would be the delimiter in split is the string we use the method on, the list that was the result is the parameter and our result is a single string.

The code `" ".join(["Sean", "is", "my", "name"])` would return the result `"Sean is my name"`. The space is inserted between every string in the list and they are joined into a single string. Similarly, `"f".join(["abcde", "", "g", "de"])` would result in `"abcdeffgfde"`.

Stripping Strings

Another useful method we can use with strings is the `strip` method. This allows us to remove characters that we do not want from the start and end of a string. This is particularly useful when reading data from a file and it starts or ends with spaces or a new line character. The method takes a string containing the characters we want removed as a parameter and returns a copy of the string with the those characters removed.

The code `" hello! ".strip(" !")` would return the result `"hello"` and the code `"goodbye\n".strip("\n")` would return the result `"goodbye"`.

Other String Methods

There are many methods that we can use with strings to process them in different ways. For each, method there will be a brief explanation and some examples to show how it works and the results.

isalpha

This method can be used to check if a string contains only letters, it returns `True` if there are only letters in the string and `False` if any other character is present (including spaces).

Table 12.1: The isaplha method for strings

Data	Method	Result
`s = "Hello"`	`s.isalpha()`	`True`
`s = "Hello!"`	`s.isalpha()`	`False`

isdigit

This method can be used to check if a string contains only digits, it returns `True` if there are only the characters 0 - 9 and `False` if any other character is present (including spaces).

Table 12.2: The isdigit method for strings

Data	Method	Result
`s = "123"`	`s.isdigit()`	`True`
`s = "12A!"`	`s.isdigit()`	`False`

islower

This method can be used to check if a string contains only lowercase letters, it returns `True` if there are only lowercase letters (a - z) and `False` if any other character is present (including spaces).

Table 12.3: The islower method for strings

Data	Method	Result
`s = "sean"`	`s.islower()`	`True`
`s = "Sean"`	`s.islower()`	`False`

isupper

This method can be used to check if a string contains only uppercase letters, it returns `True` if there are only uppercase letters (A - Z) and `False` if any other character is present (including spaces).

Table 12.4: The isupper method for strings

Data	Method	Result
`s = "SEAN"`	`s.isupper()`	`True`
`s = "Sean"`	`s.isupper()`	`False`

`lower`

This method can be used to copy the contents of a string, but replace all uppercase letters with lowercase letters.

Table 12.5: The lower method for strings

Data	Method	Result
`s = "SEAN"`	`s.lower()`	`"sean"`
`s = "Sean"`	`s.lower()`	`"sean"`

`upper`

This method can be used to copy the contents of a string, but replace all lowercase letters with uppercase letters.

Table 12.6: The upper method for strings

Data	Method	Result
`s = "sean"`	`s.upper()`	`"SEAN"`
`s = "Sean"`	`s.upper()`	`"SEAN"`

`find(w)`

This method takes a string as a parameter (w) and returns the index of the first occurrence of w in our string. If the contents of w are not in the string, the the method returns -1.

Table 12.7: The find method for strings

Data	Method	Result
`s = "hello"`	`s.find("l")`	`2`
`s = "goodbye"`	`s.find("l")`	`-1`

12.6 Example

In this section we will look at an example that uses techniques from many of the sections in this chapter. Lets assume that we have a file containing information about the students in a class (`results.csv`). The file is a type of text file called a comma separated values file (csv). It contains information about student where one line of text is the information about a single student and the individual values on a single line are separated by commas. On each line we have the following information, the students id number, the students name, their result in an exam and their result in their homework.

What we want is a list of the students numbers and their final grades in the course written into the file `grades.csv`. This information should be formatted correctly for this type of file. We can assume that grades can be calculated using a function named `getgrade`, and the weight of each part of the course is 50%. Additionally, we also want to print the name and result of the student who performed best in the exam and homework respectively.

To complete this task, we must achieve the following steps:

- Open the file `results.csv`
- Open the file `grades.csv`
- Read the contents of the results file to a list
- Loop through each line of the file we have read
- assume 0 is the best score for exam and homework results we have seen
- For each line in the list, repeat the following
  - Split the line into pieces
  - Process the pieces and convert where necessary
  - Compare the exam result to the biggest (change if necessary)
  - Compare the homework result to the biggest (change if necessary)
  - Calculate the students grade
  - Write the students id number and grade into the file separate by a comma
- Print the best exam and homework results (and the students names)

Code Tracing

The code to complete this task in Python is shown in Listing 12.8. Compared to the examples we have seen so far, this is quite a complicated piece of code combining almost all of the concepts we have learned about in the class so far.

```
15 with open("results.csv","r") as results:
16   lines = results.readlines()
17   bestExam = (0, "")
18   bestHome = (0, "")
19   with open("grades.csv","w") as grades:
20     i = 0
21     while i < len(lines):
22       line = lines[i]
23       lst = line.split(",")
24       sid = lst[0]
25       name = lst[1]
26       exam = int(lst[2])
27       home = int(lst[3])
28       if exam > bestExam[0]:
29         bestExam = ( exam, name)
30       if home > bestHome[0]:
31         bestHome = (home, name)
32       out = sid + "," + getgrade(exam*.5+home*.5) + "\n"
33       grades.write(out)
34       i = i + 1
35   print("The best exam result was", bestExam[0], "by", bestExam[1])
36   print("The best homework result was", bestHome[0], "by", bestHome[1])
```

Listing 12.8: Read and process student results

We are going to look at an example of tracing through this code. We will assume that the text shown in Listing 12.9 is the contents of the file that will be read and analysed. As this will be the largest example of code tracing we have seen so far it gives us an understanding of the kind of information we need to be able remember when we are writing code.

```
1 192372101,Sean Russell,50,60
2 192372102,Vivek Nallur,89,40
```

Listing 12.9: Contents of the results.csv file

The result of the code tracing is shown in Table 12.8. Because it has many variables, we will not show the contents of all of them.

Table 12.8: Tracing the code in Listing 12.8

Line	Code Executed	bestExam	bestHome	i	line	sid	name	exam	home	Next
15	`with open("...","r") as results`									16
16	`lines = [...]`									17
			Current value of `lines` is `["192372101,Sean Russell,50,60","192372102,Vivek Nallur,89,40"]`							
17	`bestExam = (0, "")]`	(0, "")								18
18	`bestHome = (0, "")`	(0, "")	(0, "")							19
19	`with open("...","r") as grades`	(0, "")	(0, "")							20
20	`i = 0`	(0, "")	(0, "")	0						21
21	`while True: # 0 < 2`	(0, "")	(0, "")	0						22
22	`line = "192372101,Sean Russell,50,60"`	(0, "")	(0, "")	0	"..."					23
23	`lst = line.split(",")`	(0, "")	(0, "")	0	"..."					24
					Current value of `lst` is `["192372101","Sean Russell","50","60"]`					
24	`sid = "192372101"`	(0, "")	(0, "")	0	"..."	"192372101"				25
25	`name = "Sean Russell"`	(0, "")	(0, "")	0	"..."	"192372101"	"Sean Russell"			26
26	`exam = 50`	(0, "")	(0, "")	0	"..."	"192372101"	"Sean Russell"	50		27
27	`home = 60`	(0, "")	(0, "")	0	"..."	"192372101"	"Sean Russell"	50	60	28
28	`if True: # 50 > 0`	(0, "")	(0, "")	0	"..."	"192372101"	"Sean Russell"	50	60	29
29	`bestExam = (50, "Sean Russell")`	(50, "Sean Russell")	(0, "")	0	"..."	"192372101"	"Sean Russell"	50	60	30
30	`if True: # 60 > 0`	(50, "Sean Russell")	(0, "")	0	"..."	"192372101"	"Sean Russell"	50	60	31
31	`bestHome = (60, "Sean Russell")`	(50, "Sean Russell")	(60, "Sean Russell")	0	"..."	"192372101"	"Sean Russell"	50	60	32
32	`out = "192372101,C\n"`	(50, "Sean Russell")	(60, "Sean Russell")	0	"..."	"192372101"	"Sean Russell"	50	60	33
							Current value of `out` is `"192372101,C\n "`			
33	`grades.write("192372101,C\n")`	(50, "Sean Russell")	(60, "Sean Russell")	0	"..."	"192372101"	"Sean Russell"	50	60	34
34	`i = 1`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372101"	"Sean Russell"	50	60	21
21	`while True: # 1 < 2`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372101"	"Sean Russell"	50	60	22
22	`line = "192372102,Vivek Nallur,89,40"`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372101"	"Sean Russell"	50	60	23
23	`lst = line.split(",")`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372101"	"Sean Russell"	50	60	24
					Current value of `lst` is `["192372102","Vivek Nallur","89","40"]`					
24	`sid = "192372102"`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372102"	"Sean Russell"	50	60	25
25	`name = "Vivek Nallur"`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	50	60	26
26	`exam = 89`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	60	27
27	`home = 40`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	28
28	`if True: # 89 > 50`	(50, "Sean Russell")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	29
29	`bestExam = (89, "Vivek Nallur")`	(89, "Vivek Nallur")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	30
30	`if False: # 40 > 60`	(89, "Vivek Nallur")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	32
32	`out = "192372101,C\n"`	(89, "Vivek Nallur")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	33
							Current value of `out` is `"192372101,C\n "`			
33	`grades.write("192372102,C\n")`	(89, "Vivek Nallur")	(60, "Sean Russell")	1	"..."	"192372102"	"Vivek Nallur"	89	40	34
34	`i = 2`	(89, "Vivek Nallur")	(60, "Sean Russell")	2	"..."	"192372102"	"Vivek Nallur"	89	40	21
21	`while False: # 2 < 2`	(89, "Vivek Nallur")	(60, "Sean Russell")	2	"..."	"192372102"	"Vivek Nallur"	89	40	35
35	`print("...", 89, "by", "V...r")`	(89, "Vivek Nallur")	(60, "Sean Russell")	2	"..."	"192372102"	"Vivek Nallur"	89	40	36
35	`print("...", 60, "by", "S...l")`	(89, "Vivek Nallur")	(60, "Sean Russell")	2	"..."	"192372102"	"Vivek Nallur"	89	40	-

Chapter 13

Advanced Data Structures

In chapters 7 and 8 we studied the list, tuple and string data structures. These three data structures are collectively called sequences and have many common factors in how they work and can be used.

These sequences are all a type of data structure known as indexed linear data structures. The indexed part of the name means that each value in the data structure is stored at a particular location and we use an index to access it. The linear part of the name describes how the data is stored in order. These are not the only type of data structure, in this chapter we will study two different unordered data structures.

13.1 Unordered Data Structures

An unordered data structure stores elements without any particular order. Values are not accessed using an index. In this chapter we will study two different unordered data structures, the first are associative data structures (commonly called maps or dictionaries), the second is the set data structure.

Associative data structures do not use an index to access values, instead we use another value know as a **key** to choose which value we want to access. This is similar to the idea of using variable names to remember values instead of memory addresses. In Python, the dictionary data type is an example of an associative data structure. The dictionary is a common data type, but it is often called a map or associative array in other programming languages.

The set data structure in python does not require the use of keys, but also does not use indexes. The items we store in a set are unordered, but they are also guaranteed that there are no duplicates in the set.

13.2 Dictionary

Keys and Values

The important feature of the dictionary data type is the use of keys. A key is a **unique** value that we can use to insert and access the data. We can use any **immutable** type as a key, but strings are most commonly used.

Consider an example of storing the exam scores of a class. In a list, each value would be stored at a particular index. Some extra logic or data would be required to remember which value belonged to which student. However, using a dictionary would allow the students id number to be used as a key to insert and access the data. This works because every students id number is unique, so when we need to find the exam score of a student I can use the student id to find it.

Creating a Dictionary

There are several ways to create a dictionary in Python. We can declare a literal dictionary in our code, or create an empty dictionary. Creating an empty dictionary is easiest, we can use the

function `dict` or we can use a pair of empty curly brackets ({}) to create the dictionary and then assign the value to a variable. E.g. `values = {}` or `values = dict()`.

The syntax of declaring a dictionary containing data is a little complicated. The data must be surrounded by a set of curly brackets and the items must be separated by commas. However, each item is made up of two parts, a key and a value. Each value must be placed with the key that matches it and they must be separated by a colon (:).

```
1 var_name = { key1 : value1, key2 : value2 }
```

Listing 13.1: Declaring a dictionary containing data

When storing the results of students in an assignment, we could use dictionary. If we know all of the grades in advance, we can directly assign the values and keys together.

```
1 scores = { "19372101" : 95, "19372102" : 80, "19372103" : 100, "19372104" :
  ↪  83 }
```

Listing 13.2: Creating a dictionary for remembering students results

Accessing Values

When we want to access a value in a dictionary, we need to use its key. The syntax is very close to accessing a value in a list or sequence, however instead of placing the index between a set of square brackets we place the key of the value we want. For example, `dict_name[key]` would access the variable stored using the key `key` from the dictionary remembered using the variable `dict_name`.

In our example, to find the result of the student with the id `"19372101"` from the variable `scores`, the code would be `scores["19372101"]`. This value could then be used in a calculation or printed to the screen like any other variable. E.g. `print(scores["19372101"])`. The key that we use can be any immutable value. In our code we can write this as a literal (number or string literal) or the key can be the result of an expression or the value of a variable.

Inserting Values

The previous example shows how to create a dictionary when we already know the keys and values that we want to insert. When we want to insert a new value into a dictionary, we use assignment and again we need to use the key. For example, `dict_name[key] = value` could be used to add another value to a dictionary.

In our example, to add the result (86) of a student with the id `"19372105"` into the dictionary `scores`, the code would be `scores["19372105"] = 86`. The value 86 will now be remembered in the dictionary and can be accessed again using the key `"19372105"`.

If there was already a value in the dictionary that was inserted using the key `"19373105"`, then that value is replaced and is no longer available. There can only ever be a single value in a dictionary with any key.

Removing Values

Just like with lists, to remove a value from a dictionary we use the keyword `del`. The keyword should be followed by the name of the dictionary with the key in a set of square brackets. E.g. `del dict_name[key]` would remove the value stored using the key `key` from the dictionary remembered using the variable `dict_name`.

In our example, if we wanted to remove the result of the student with id `"19372102"` from the dictionary named `scores`, the code would be `del scores["19372102"]`. The key and the value associated with it will both be removed from the dictionary.

Keys and Checking Keys

When we try to access an item in a sequence that does not exist it causes an error in our program. This is relatively easy to prevent because we can calculate the acceptable indexes based on the length of the sequence (0 to len(sequence)-1). However, when we are using a dictionary it is not so simple, the keys we use to access the values can be different types and can be any value. So we need a different way to check that the key we are going to use is acceptable.

To achieve this check we can use the keyword `in`. We previously studied this operator for use checking if a value was contained in a sequence or list. When we use this with a dictionary it instead checks only if the value is a key in the dictionary. E.g. `key in dict_name` would tell you if there is a value stored in the dictionary `dict_name` using the key `key`.

The keyword returns a boolean value that we can use in an if statement to make decisions. For example we may choose to access the value if it exists and do something with it or insert a new value if it does not exist.

Looping over Dictionaries

It is very common to want to perform the same action for every value in a dictionary (just like we did with sequences). With dictionaries there are three different ways that we can do this, we can loop through all of the keys, we can loop through all of the values or we can tuples containing both keys and values. All three of these techniques are suitable for different situations and involve the use of a for loop.

Looping by Keys

The default way is to loop through the keys in the dictionary, this is done with a very simple for loop. When we loop this way we need to access the value from the dictionary (using the key) if we want to use it. Otherwise in the loop we will only have access to the key.

```
1 for k in dict_name:
2   v = dict_name[k]
3   # do something with k or v
```

Listing 13.3: Looping Through Keys of a Dictionary

In this example here, the first line of the loop is to access the value from the dictionary using the key (`k`). Every time the loop executes the variable `k` will contain another key from the dictionary. Because the dictionary is not ordered like a list, result may be calculated or printed in a different order.

Looping by Values

If we want to loop through all of the values in a dictionary, we need some way to access them. Dictionary objects have a method named `values`, this method takes no parameters and returns a list containing all of the values in the dictionary. We can then use this list in a for loop to loop through all of the values.

```
for v in dict_name.values():
  # do something with v
```

Listing 13.4: Looping Through Values of a Dictionary

It should be noted here that there is no way to find the key that belongs to a value. So this is only useful when we do not care what the keys of the values are.

Looping by Both

If we want to loop through the contents of the dictionary and we need to have access to both the key and the value, we need some way to access them. Dictionary objects have a method named `items`, this method takes no parameters and returns a list containing all of the keys and values in the dictionary. The list is structured as a list of tuples, where the first index in each tuple is the key and the second index in each tuple is the value. We can then use this list in a for loop to loop through all of the keys and values.

```
for i in dict_name.items():
  k = i[0]
  v = i[1]
  # do something with k and v
```

Listing 13.5: Looping Through Keys and Values of a Dictionary

Tuple Unpacking

We can simplify this code a little bit by using a technique called tuple unpacking. This is where instead of using a single variable name to represent the name of the tuple that will be taken from the list (or returned by a function) we instead use two variable names separated by commas. Python automatically unpacks the tuple and places the item from index 0 in the first variable, the item from index 1 in the second and so on.

In this case we have a tuple containing two values, first the key and second the value. We can replace the variable `i` from the previous example with the variable `k` and `v` and then we no longer need the code in the loop that retrieved these values.

```
for k, v in dict_name.items():
  # do something with k and v
```

Listing 13.6: Looping Through Keys and Values of a Dictionary

These two examples would have exactly the same result when executed

13.3 Dictionary Example

Lets consider the example from chapter 12, it allowed the user to read student information from a file and processed it and stored the output in another file. For this example, we will start with the same input data, however this time we will create a program that allows the user to query for information about a single student based on their id number. First we will complete the task using a list, then we will complete the task using a dictionary.

Using a List

First we need some code to read the data from the file and store it in the data structure. For the example using the list, we will store the information about each student as a tuple, the first index will contain the id, the second will contain the name and the remaining ones will contain the students results. This will be appended to a list and then when the function is finished the list will be returned.

```
1  def read_results():
2    result_list = []
3    with open("results.csv","r") as results:
4      lines = results.readlines()
5      for line in lines:
6        lst = line.split(',')
7        result_list.append((lst[0], lst[1], int(lst[2]), int(lst[3])))
8    return result_list
```

Listing 13.7: Function to read values into list of tuples

Now using this function we can read the contents of the file into a list. Following from this we can have the user enter a student id number. We will need to search through the list to match the student number the user entered against a student number contained in a tuple before we know the information belongs to the student.

```
10 reslst = read_results()
11 sid = input("Enter student id: ")
12 for tpl in reslst:
13     if tpl[0] == sid:
14         print(tpl[1],"got results", tpl[2],"and",tpl[3])
```

Listing 13.8: Finding the name and results based on id number

This code allows the teacher to type the id number of a student and then the program will print the name of the student as well as their results in their assignment and exam.

Using a Dictionary

We will use a similar technique to complete the task using a dictionary, first we will write a function to read the data from the file and store it in a dictionary.

```
1  def read_results():
2    result_dict = {}
3    with open("results.csv","r") as results:
4      lines = results.readlines()
5      for line in lines:
6        lst = line.split(',')
7        result_dict[lst[0]] = (lst[1], int(lst[2]), int(lst[3]))
8    return result_dict
```

Listing 13.9: Function to read values into list of tuples

Comparing this to the previous version of the function, we can see that the differences are very small. We use different code to create an empty list or dictionary and inserting into a dictionary is a little different to appending to the end of a list. However, the overall structure of the code is basically the same; read the file, loop over and process each line adding to the data structure.

Completing the task will require similar code again, we will need to use the function to read the file and ask the user to enter the id of the student they want to search for. After this point the code is a little different, instead of needing to loop through the list until we find the right student, we can simply ask the dictionary to give it to us. However, to do so safely we need to first check if there is a value stored in the list using that key.

```
10 resdic = read_results()
11 sid = input("Enter student id: ")
12 if sid in resdic:
13   tpl = resdic[sid]
14   print(tpl[0],"got results", tpl[1],"and",tpl[2])
```

Listing 13.10: Finding the name and results based on id number

13.4 Sets

A set is a mutable data type containing non-duplicated unordered values. We can use sets in a similar way as lists, we can add and remove data as required. However, if we attempt to add the same value to a set twice it will not work. Additionally, when we loop through the contents of a set we do not know what order this will happen in, whereas in a list it would be ordered by index.

Sets are often used if we want to remove duplicates from data or in cases where we want to use common mathematical set operations like union and intersection.

Creating a Set

There are two ways to create a set in Python. We can declare a literal set in our code containing values or we can create an empty set. Creating an empty set is the easiest, we use the function `set` to create the set and then assign the value to a variable. E.g. `values = set()`.

The syntax for declaring a set containing data is similar to a list, however instead of square brackets we use curly brackets. Each individual value must be separated by commas, just like when declaring a literal list.

```
1 var_name = { value1, value2, value3 }
```

Listing 13.11: Declaring a set containing data

We can also use the `set` function to create a set containing data from a sequence. If we pass a sequence (list, tuple or string) as a parameter to the function, the contents of the sequence will be used to fill the set and duplicates will be ignored. For example, the code `st = set([ 1, 1, 2, 3 ])` would return the following set `{ 1, 2, 3 }` and the code `st = set("hello")` would return the following set `{ "h", "e", "l", "o" }`.

When viewing all of the operations that can be performed with set, we will use the same example sets in the demonstrations.

Each of the examples given in the following sections are assumed to be starting with these values and showing you the result of the operations.

Table 13.1: Example sets we will use to demonstrate the functionality of sets

Name	Contents
a	{ 1, 2, 3 }
b	{ 3, 4, 5, 6 }
c	{ 1, 5, 6 }

Inserting Values

To add a value to a set we use the method `add`.
The examples show us two scenarios, in the first the value that is added to the set is not already present in the set and so it is added. In the second example, the value 1 is already in the set so there is no change made to the set.

Operation	Result
a.add(4)	{ 1, 2, 3, 4 }
a.add(1)	{ 1, 2, 3 }

Removing Values

To take an item from a set we use the method `remove`.
In these examples, the value passed as a parameter is removed from the set. However, we should be careful as this will cause an error in our program it the value is not currently in the set. We should always check if the value exists before using this method.

Operation	Result
a.remove(3)	{ 1, 2 }
b.remove(4)	{ 3, 5, 6 }

Checking Membership

To check if a value is a member of a set (if it is in the set) we use the keyword `in`.
This operator returns a boolean value and so can be used in an if statement to check if a value exists in a set before removing it or applying other operations.

Operation	Result
1 in a	True
1 in b	False

Set Size

To find the number of items in a set, we use the function `len`. This function returns an integer value that represents the number of items in the set.

Operation	Result
len(a)	3
len(b)	4
len(c)	3

Union of Two Sets

To get the union of two sets we use the operator `|`.
The union contains all of the values of both sets with no duplicates.

Operation	Result
a | b	{ 1, 2, 3, 4, 5, 6 }
a | c	{ 1, 2, 3, 5, 6 }
b | c	{ 1, 3, 4, 5, 6 }

Intersection of Two Sets

To get the intersection of two sets we use the operator `&`.
The intersection contains only the values that appear in both sets with no duplicates.

Operation	Result
a & b	{ 3 }
a & c	{ 1 }
b & c	{ 5, 6 }

Difference of Two Sets

To get the difference of two sets we use the operator `-`. The difference contains only the values from the set on the left side of the operator that do not appear in the set on the right side of the operator.

Operation	Result
`a - b`	`{ 1, 2 }`
`b - a`	`{ 4, 5, 6 }`
`a - c`	`{ 2, 3 }`

Looping over a Set

We can loop over the values in a set in the same way as we loop over a sequence. The big difference is that we do no know what the order that the values will be printed in.

```
1 s = set("a quick brown fox")
2 for v in s:
3   print(v, end=' ')
```

Listing 13.12: Looping over values in a set

This code results in the following (on my computer):

```
1 n r f   a c w x u b q i o k
```

Listing 13.13: Output of the previous example

13.5 Set Example

In this section we will look at an example that can be completed by making use of both sets and of dictionaries. Students complete a quiz as part of a course, this quiz can be attempted many times by each student if they wish to improve their grade. Assuming we have a file containing the list of attempts (student id and grade), our task is to find the number of students who attempted the quiz, the number of times the quiz was attempted, and the number of students that achieved each of the grades.

First we will need some code to read the data from the file and return it. We will do this as a function that returns a list of tuples.

```
1 def read_grades():
2   result_list = []
3   with open("quiz.csv","r") as results:
4     lines = results.readlines()
5     for line in lines:
6       lst = line.split(',')
7       result_list.append( (lst[0], lst[1].strip()) )
8   return result_list
```

Listing 13.14: Function to read values into list of tuples

Now with this information available, the next task we will complete is to find both the number of attempts made on the quiz and the number of students who attempted the quiz. The first is easy because it is the length of the list returned by the function. To calculate the second, we only

want to count each student once. If we add all of the student ids to a set, each value can only be added once and the size of the set will be our answer.

```
10 results = read_grades()
11 students = set()
12 for sid, grade in results:
13   students.add(sid)
14 print("The quiz was attempted",len(results),"times")
15 print(len(students), "students attempted the quiz")
```

Listing 13.15: Find the number of students and attempts

The final task is to find the number of students that achieved each of the grades. Using a set we could find number of unique grades that were achieved by the students, but this would not be very useful. We could instead use a dictionary to remember how many times we have seen each grade. Using the grade as a key, we can store an integer as a value which we change every time we see a new grade.

```
17 grades = dict()
18 for sid, grade in results:
19   students.add(sid)
20   if grade in grades:
21     grades[grade] = grades[grade] + 1
22   else:
23     grades[grade] = 1
24 print("This is the breakdown of the grades:")
25 for grade, num in grades.items():
26   print(grade, num)
```

Listing 13.16: Find the distribution of grades in the quiz

List of Source Code Examples