

Introduction to Python

Colin Paterson *

August 8, 2024

* colin.paterson@york.ac.uk

1 Introduction

In 1981 I received my first computer, a BBC Model B microcomputer. It might seem difficult to believe now but this was a big deal. Computers were the future, they were exciting and the possibilities seemed endless. I plugged the machine into my TV and switched it I was rewarded with a couple of beeps and a blinking cursor waiting for me to type on the keyboard.

```
BBC Computer 32K
```

```
BASIC
```

```
> \_
```

This was the beginning of my programming adventure.

It might be over 40 years since that first computer, but for me, the joy I get from programming is still there. Learning to program that provided me with opportunities I couldn't have imagined and today the possibilities look bigger and more impressive than ever.

What I want to do through this short course is remove some of the mystery surrounding programming and show you that with very little effort your laptop can be so much more than a portable video player. I want you to take control of your computer and understand how easy and useful computer programming can be.

1.1 Who is this book aimed at?

This book is being written in the first instance to support post graduate students from non-scientific disciplines who are undertaking a multi-disciplinary PhD in Safe Artificial Intelligence. I assume that you have no previous programming experience, no experience of data analysis and no experience of building models using data.

By the end of this course however I expect you to be able to write computer

program to do all of these things. You will be able to create graphs for use in your thesis and you will even build and analyse a machine learning models.

1.2 So what are we going to learn?

Through this course you are going to learn how to program in Python. This isn't because I believe Python to be a wonderful language but because I believe it's a useful language in this context. It will allow us to use most of the commonly used programming structures and paradigms and it is widely supported in data intensive applications, such as artificial intelligence.

In the first part of the book we are going to concentrate on the basics of interacting with python. We will start by looking at how we interact with python in an interpretive environment before moving onto using a dedicated programming environment (VSCode) to write and debug our code.

We will then look at how we use python to store information introducing concepts like variables, data types, list and dictionaries before moving onto programming structures which allow us to make choices and create programs that can do the same thing many times.

2 First Steps

Let's dive straight in and see Python in action.

First we are going to launch the interactive Python environment, this requires us to get a command prompt. If you are on a windows computer then open a prompt from the start menu, alternatively if you are on Linux or a Mac you will run a terminal window. At the command prompt type `python` and you should see something like this:

```
Python 3.11.4 (main, Jul 5 2023, 08:40:20) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The number after the word Python tells you the version of the Python language you are running. The only thing that matters for this book is that it starts with a number 3. The computer is now waiting for you to give it an instruction. Try typing in each of these lines and press the return key after each line to see the result.

```
2+2
5*6
32/6.2
2*(2+3)
3**2
3 times 2
quit()
```

You can see that Python is working as a calculator but let's think a little more about what's going on here. When you press return Python is taking the characters you have typed and splitting it up into tokens it recognises, that is numbers and mathematical operators. We are limited in what we can type by the QWERTY keyboard we are using so we sometimes need to use special characters to tell Python what we need. For example line 2 means 5 multiplied by 6 but there isn't really a multiplication symbol on our keyboard so we have to use the asterisk (or star) symbol. Similarly / means divided by and ** means to the power.

Indeed Python has no real intelligence if it receives a command it under-

stands then it does what it's told, if it receives something it doesn't understand then we get an error, such as we see with line 6. The error on this line is known as a **Syntax Error**. Put simply it means that the words used, or the way the sentence is constructed, is not understood by the computer (or Python in particular). The error message provided tries to help us to see where we have gone wrong by telling us which line the error is on and pointing at the part of the line where the error exists.

```
>>> 3 times 2
      File "<stdin>", line 1
        3 times 2
          ^^^^^
SyntaxError: invalid syntax
>>> 
```

The final command we gave to Python, `quit()` tells the interpreter to close.

3 Your first Program

In the previous section we saw how to give commands to Python, but each command was unrelated. Writing computer programs typically requires us to use multiple commands, in a predefined order, to accomplish a task. This set of commands is called a computer program and, in this section, we will write our first program. This will require us to make use of a program editor and, in our case we are going to use Microsoft Visual Studio Code. I chose this editor because it's available for all computing platforms (there is even a web version and one for Raspberry pi computers).

So let's get started and launch Visual Studio Code. Create a new file and call it `my_first_program.py`. You might want to create a folder on your computer to store your programs from the book. The reason we give the program the extension ".py" is so that the computer knows this is a python file.

Because we are using Visual Studio Code and it knows what a python file so, if you haven't yet installed the python extensions to the editor, you will be asked if you'd like to, please go ahead and do so, it might take a few minutes. Now type in the following program. If you understand what the program

is doing then great, if not then don't worry we will learn what it means through the rest of the book.

```
import math

# This program calculates the roots of a quadratic
# i.e.  $(-b \pm \sqrt{b^2 - 4ac})/2a$ 

a = 1.0
b = 1.0
c = -6.0

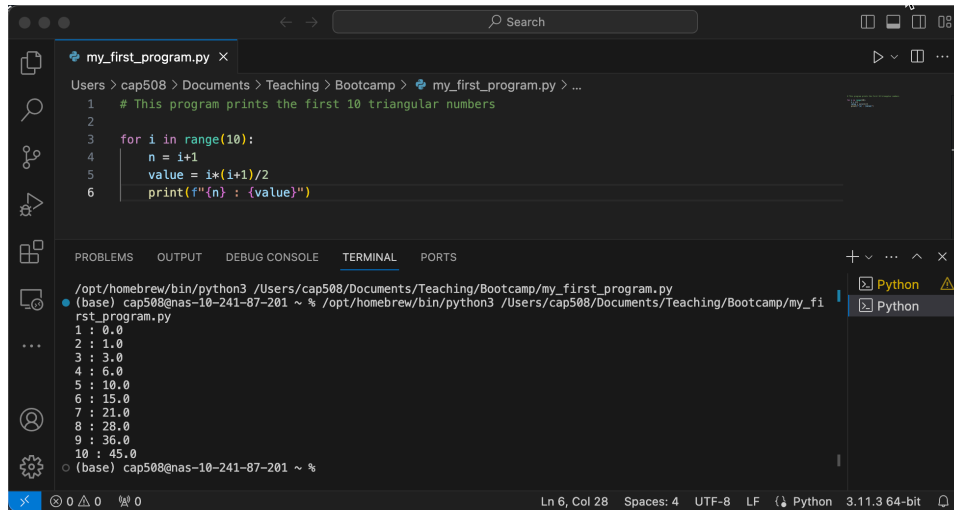
root1 = (-b + math.sqrt(b**2 - 4 * a * c))/(2*a)
root2 = (-b - math.sqrt(b**2 - 4 * a * c))/(2*a)

print(f"The roots are {root1}, {root2}")
```

Now save your program and look from the Triangle at the top of the screen this is the [Run Python File](#) button.



When you press this button you are telling VSCode to send your commands to the python interpreter. VSCode then opens a new window which contains a command window, just like the one you used earlier, and calls python passing it the file you created. You can see the output created by your file in the bottom window as seen below. Don't worry if your editor looks slightly different to this VSCode allows you to change colours and icons to suit your preferences. We won't spend much time looking at this here but you might want to explore the editor more by visiting the visualstudio website : <https://code.visualstudio.com/docs/introvideos/basics>.



The screenshot shows a code editor with a file named `my_first_program.py`. The code is a Python script that prints the first 10 triangular numbers. Below the code, a terminal window shows the output of the program, which is a list of 10 numbers: 0.0, 1.0, 3.0, 6.0, 10.0, 15.0, 21.0, 28.0, 36.0, and 45.0. The terminal also shows the command used to run the program: `/opt/homebrew/bin/python3 /Users/cap508/Documents/Teaching/Bootcamp/my_first_program.py`.

```
1 # This program prints the first 10 triangular numbers
2
3 for i in range(10):
4     n = i+1
5     value = i*(i+1)/2
6     print(f"{n} : {value}")
```

```
/opt/homebrew/bin/python3 /Users/cap508/Documents/Teaching/Bootcamp/my_first_program.py
(base) cap508@nas-10-241-87-201 ~ % /opt/homebrew/bin/python3 /Users/cap508/Documents/Teaching/Bootcamp/my_fi
rst_program.py
1 : 0.0
2 : 1.0
3 : 3.0
4 : 6.0
5 : 10.0
6 : 15.0
7 : 21.0
8 : 28.0
9 : 36.0
10 : 45.0
(base) cap508@nas-10-241-87-201 ~ %
```

Well done. You've started your programming journey. In the next section we are going to learn the basics ideas which are common to many different programming languages and see how we create and manipulate data using Python.

4 A place for everything ...

Programming is, at it's heart, about manipulating data. That data needs to be stored in the computer in such a way that we know what it is. Consider the following block of text.

Colin is a **Lecturer** in the department of **Computer Science**. They have a class of **2** students studying **Python**. The course starts on the **19th of September** and runs for **3** days. The names of the students are **Alex Smith and Charlie Jones**.

The structure of the block tells us something about how the values in bold interact. We can consider each item in bold as a placeholder for a value that could be changed without changing the intention of the text with is to inform you about someone's role and teaching responsibilities. So if we change only the values we might get something like this.

Tracy is a **Senior Lecturer** in the department of **Philosophy**. They have a class of **3** students studying **Ethics**. The course starts on the **24th of October** and runs for **2** days. The names of the students are **Bob, Carlos and Aled**.

You might think of the structure of the text block as a program.

Fire up the python interpreter in a command (or terminal) window and type in the following commands. The print command is one we will use quite a bit in this course, simply type print then put what you want to output in the brackets.

```
>>> print("Colin")
>>> print(4)
>>> print(3.5)
>>> print('Colin is a Lecturer in the department of Computer
↵ Science')
```

Notice that even though you have put quotation marks around some of the values the quotes don't show in the output. That's because we are using the

quotes to tell Python than the thing we are dealing with is a string of text character. Python doesn't mind if you use " or ' but what matters is that you use the same at each end of the string.

Indeed we can ask Python about what type of thing we are dealing with using the `type` function¹. So try typing each of the following in turn:

```
>>> type("Colin")
>>> type(4)
>>> type(3.5)
>>> type("3.5")
```

So we are being told that "Colin" is a string (`<class 'str'>`), 4 is an integer (`<class 'int'>`), 3.5 is a floating point number (`<class 'float'>`), and "3.5" is a string (`<class 'str'>`). Just think about that last one for a moment. Putting quotes round a number turns it into a string. Why does that matter? Well let's take a look at what happens if we try and use this value in a mathematical equation.

Challenge 1.

Can you work out what the type of `print` is?

Even though you might think of `print` as a special word, Python is pretty thick and let's you use the word and cause problems. try this.

Note: pressing the up arrow in the interpreter will allow you to run commands again.

```
print('Hello World')
print = 32
print('Hello World')
```

Suddenly `print` no longer works! This is because we've overwritten the *function* with a *variable* of the same name. All is not lost however. Try this.

¹We are using functions here, don't worry too much about that for now as we will get to them in a later chapter. For now simply type what is written and check that you match the brackets.

```
del print
print('Hello World')
```

The `del` keyword allows us to destroy the variable called 'print' and gives us access to the built in function again. Note: python won't let you create a variable called 'del'.

In the previous code we explored some simple maths operators in Python. Now I want you to type the following and try to explain what the program is doing².

```
>>> print(3+4)
7
>>> print(3*4)
12
>>> print("3"+"4")
34
>>> print(3*"4")
444
>>> print("3"*"4")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

The first two lines are simple arithmetic and we get returned the answers 7 and 12 for addition and multiplication respectively. The third line is interesting as Python is trying to be helpful. We can't "add" two strings together but it is quite common for people to want to concatenate strings so the people who developed Python decided the "+" symbol could be a nice shorthand way of doing this so "3"+"4" = "34". Similarly on the 4th line Python reads this as meaning you want to duplicate the string 3 times so 3*"4" = "444". Finally on the last line we are asking Python to multiply two string, unfortunately even Python doesn't know what you are asking for here and simply returns an error giving you a possible suggestion of how you went wrong.

²Note that you are only typing the bits preceded by >>>, the text on the other lines is the response from the Python interpreter.

Challenge 2.

Using what we've just learned see if you can write a single line of code that writes the word python 5 times in a row.

4.1 Variables

Having looked at how Python works with values let's consider a closely related idea, that of variables. Consider once again our text in the previous section and think about writing the block in a form where name of the lecturer might change. Here we would want to put a placeholder in for the name that we can set to be anything when the program is run. This placeholder is called a variable.

Challenge 3.

Look at the following code and see if you can work out what it will do before you run it.

```
>>> name = "Colin"
>>> print("The lecturer is called ", name)
>>> print(name, " works in the Department of Computer
↳ Science")
```

What we have done is created a variable called *name* and stored a string in it that we will use later. In fact having stored the value in a variable we can use it many times in lots of different places.

In fact you can ask the user to give you a value to use in your code. This makes use of the input command which is a little bit like the print command. Try the following.

```
>>> name = input("What is your name? ")
>>> print("Hello ", name, " nice to meet you".)
```

```
>>> answer = input(name, " what is your favourite colour?")
>>> print("Nice!, ", name, ". ", answer, " is my favourite colour
↳ too")
```

Challenge 4.

What is the variable type for answer?

If you said string, then well done, but what happens if we don't want a string? What happens when we want a number. Well luckily Python gives us functions that allow us to change the type of a variable. Try the following.

```
>>> int("3")
>>> float("3.5")
>>> int("3.5")
>>> str(3.5)
>>> int("Fred")
```

Notice that we can turn any number into a string, but we can't turn any string into a number. You should also be aware that making a number an integer will throw away the decimal part.

Challenge 5.

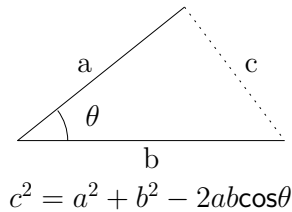
Write a program that asks for number, multiples it by 3 and prints out the answer.

4.2 Writing larger programmes

So far we have largely been looking at simple Python commands in isolation but the real power in programming comes when we link commands together to achieve more complex tasks.

Let's consider a simple mathematical problem. Calculating the length of one side for a triangle when we are given the lengths of the other 2 sides and the angle between them. Don't worry if you can't remember how to do this,

I'll show you. We can solve this problem the cosine rule as shown below.



Before we start to write our program let's think about the steps we will need to under take to solve this problem:

1. Get the length of the sides and the angle from the user
2. Because these will be strings we will need to convert them to numbers so we can do the maths.
3. We will get the angle in degrees but the cos function expect radians so we will need to convert it.
4. Apply the cosine rule
5. Display the length of the side.

This process of breaking a problem down before you start writing is crucial to writing computer programs and it's something I often do with pen and paper before even opening the editor. So having written down the steps we need to take, now open up VSCode and create a program called "Traingles.py". Once open type in the following program, note that I have included line numbers to help me explain the program, you do not need to type them.

```

1  """
2  This program calculates the length of the missing side of a triangle
3  when we are given two side lengths and the angle between them.
4  We do this using the cosine rule.
5  """
6
7  # The program will use mathematical operators from the maths library
8  # so we need to import them
9  import math
10
11 # Step 1: Get the values we will use from the user
12 side1 = input("Length of side one : ")
13 side2 = input("Length of side two : ")
14 angleDeg = input("Angle (degrees) between the two lines : ")
15
16 #Step 2: Convert the strings into numeric values so we can use
17 # them in calculations
18 side1 = float(side1)
19 side2 = float(side2)
20 angleDeg = float(angleDeg)
21
22 # Step 3: Convert the angle from degrees to radians.
23 angleRad = angleDeg * math.pi/180
24
25 # Step 4: Calculate the other side using the cosine rule
26 # Please note that **2 means to the power 2 (or squared)
27 side3 = math.sqrt(side1**2 + side2**2 - 2*side1*side2*math.cos(angleRad))
28
29 #Step 5: Print out your result
30 print(f"The length of side 3 is {side3}")

```

Comments: are used to explain what is going on in the code and is particularly important if you are sharing your code with others or if you might return to the code later, when you will have forgotten your design decisions. In the program shown comments are added in two ways. In Lines 1 – 5 a multi-line comment is started, and finished, with three quotation marks. This type of comment is common at the beginning of blocks of code to explain the rationale for the next section. Lines 7,8,11 etc. use a hash mark (#) and are used for short, single line, comments.

Libraries: Line 9 shows an example of including a library. Libraries are key to the power of Python and we will learn more about them as we progress through the book, but for now just understand that including a library gives us access to functionality that is not in the core language. In this case we are saying we are going to use mathematical functions (like cosine). Having

imported the library can then use the "math" to access values and functions defined in the library, e.g. "math.pi" and "math.cos(...)".

Formatted Printing: I've also taken this program as an opportunity to introduce to you formatted printing. We have already come across the print function, but here it's slightly different. Inside the brackets I have used the letter "f" directly before the string to say this is a formatted string. This means that wherever there are curly brackets in the string it means I want you to replace this with the value stored in a variable. e.g. "print(f"{name}")" means print out the value stored in the "name" variable.

Readability: You might be tempted to combine lines to shorten the program. This is of course possible, but be warned, simple is good when it comes to maintaining code. Trying to fix a complicated programme might take more time than writing it did in the first place.

Consider the following program which does the same as Triangles.py. I hope you can see that whilst the functionality is exactly the same the choices we made in laying out the code impact our ability to digest what is doing wrong.

```
1 import math
2
3 side1 = float(input("Length of side one : "))
4 side2 = float(input("Length of side two : "))
5 angle = float(input("Angle (degrees) between the two lines : ")) * math.pi/180
6
7 print(f"The length of side 3 is {math.sqrt(side1**2 + side2**2 - 2*side1*side2*math.cos(
```

4.3 Collections

So far we have only considered 3 simple data types i.e. integers, floating point numbers and strings. In this next section we will look at variables that can hold a number of objects in a collection.

Lists: We will start our exploration of collections by looking at lists. A list can hold a number of things in order and each thing need not be of the same type.

Open your python interpreter (in a terminal or command window) and type the following:

```
>>> students = ["Alex", "Charlie", "Sam"]
>>> marks = [67, 58, 84]
>>> mixed_list = [1, "first", "primary", 1.0]
```

In each case we have created a single value that has multiple values. To use a value stored in the variable we need to tell python which element of the list we are interested in. We do this using the square brackets and an integer index, starting at 0. Try the following:

```
>>> students[0]
'Alex'
>>> marks[2]
58
>>> mixed_list
[1, 'first', 'primary', 1.0]
```

Notice that if we don't specify an index then we get the entire list returned.

Now let's try changing the values in a list:

```
>>> students[0] = "Jackie"
>>> mixed_list[3] = "one"
>>> marks[3] = 67
```

The first two lines changed the value in the list, even though in the second case we were inputting something of a different type. In the final line however we get an error. The problem is we don't have an element number 3 because we start counting at zero, so we can only see marks[0], marks[1] and marks[2].

We can however add things to a list once we've created it using the append function. Try this ...

```
>>> students.append("Hasan")
>>> marks.append(72)
>>> mixed_list.append(marks)
```


Have a look at each new list and you will see that the new *element* has been added. You can also see that we are allowed lists of lists!

Challenge 6.

Here's a challenging little question, "What do you think the following snippet of python will do?"

```
>> name = "Alex"
>> my_list = [1, name]
>> name = "Sam"
>> print(my_list[1])
```

Once you have decided type it into the interpreter and see if you were right.

Tuples: Tuples are a lot like lists but they are an ordered set of "things". A tuple uses the round brackets instead of the square brackets for assignment. Unlike lists however they can't be changed, you can't change the values in them and you can't append stuff to them. So, why would you bother? well there are two reasons. Firstly they are more memory efficient and quicker to process so if it's important that your code runs fast you might well consider them. Secondly sometimes you don't want the values in the collection to change, for example days of the week, months of the year etc.

```
>> days = ("mon", "tue", "wed", "thu", "fri", "sat", "sun")
>> days[2]
>> days[2] = "fred"
>> days.append("sun2")
```

Challenge 7.

Double check that you can create a tuple containing mixed variable types.

Dictionaries: The final type of collection we're going to consider for now is the Dictionary. This type of collection use a *key:value* structure meaning that we can look up values by providing a key.

It's easier to show than explain so try entering the following.

```
>> marks = {  
    "Alex": 67,  
    "Charlie": 58,  
    "Sam": 84}  
>> print(students)  
>> print(students["Mark"])
```

What we've created here then is a collection where the student names are the keys and the marks are the values. To access information in this collection we don't use an integer index, like we did with lists, but instead use the key.

We can also create dictionaries using a dictionary constructor function like this.

```
>> thisdict = dict(name = "Sam", age = 36, country = "UK")  
>> print(thisdict)
```

Adding things to a dictionary uses a slightly different format to the method we saw for lists.

```
>> students["Jo"] = 97  
>> students["Sam"] = 42  
>> students["sam"] = 86  
>> print(students)
```

This shows us a couple of things. Firstly if the key doesn't exist then a new item will be added to the list. Secondly if the key exists then the value will be updated and Finally *case* matters, i.e. "S" is not the same as "s" for the computer.

4.4 Some useful bits and pieces

To finish off this section on variables I want to introduce you to a couple of useful things you might come across.

The first of these is **Boolean** values. A boolean value can hold one of two

values true or false and for this reason the words "True" and "False" are reserved words in the language.

Try the following.

```
>>> x = True
>>> y = False
>>> print(f"True is {x} False is {y}")
>>> print(f"3*True is {3*x} 3*False is {3*y}")
```

The last line of code shows you something quite interesting there, that True can be interpreted as the integer 1 whilst False is 0. This little trick can itself be quite useful on occasions.

Although we have already introduced strings (and f strings) I want to introduce another interesting language feature, that of the **multi-line string**. This isn't another variable type, as such, but it's away of holding long strings in your program (remember multi-line comments?).

You simply surround your string with three marks, so try typing this in.

```
>>> longstring = """ This is an example of
... a long string. It uses three quote marks
... at the begining and the end"""
>>> print(longstring)
```

Notice that once we start typing this in we get three dots each time we press the return key, this tells us that we are still entering the same line of code. When you print the variable out you will notice that we get the text we types with a "\n" each time we pressed return. This sequence is the new line command and tells python the text should go to a new line at this point.

That's it for data types and variables, in the next section we're going to look at how we control the flow of execution in a program and how we compare the values we have stored in our variables.

5 Controlling the Flow

In the previous section we introduced variables which allow us to keep a record of information at a particular point in time. We showed how we could amend this data and combine it using arithmetic operations. In this section we are going to look at how we use that information to make decisions about the flow of data through a program.

5.1 Conditional Statements

Let us consider a problem where we only want to show some information to a person if they have the authority to see it. This is a common problem and one that we come across every time we use a cashpoint. In plain english we might right ...

if the pin code is correct then show then the balance on their account.

This is what we call a *conditional statement* and most programming languages have a structure that allows us to write commands of this form and in most languages it also makes use of the same "if" keyword. Open VScode and create a program called `balance.py` containing the following code, be careful to make sure you insert spaces (or a tab) at the beginning of line 6.

```
1 realPin = 1024
2 balance = 10457
3 pin = input("Enter your pin : ")
4
5 if (int(pin) == realPin):
6     print(f"Your Balance is {balance}")
```

Note here that input returns a string so we need to make it an integer before we compare it with realPin. In fact maybe we would be better making realPin a string, think about a pin of 0004.

Note also that we do nothing here to check the format of the input I could

enter anything, try entering a name instead of a number. This isn't a problem for a cashpoint machine of course as we physically restrict what the user is able to input, but we would need to add code to make sure the pin was in the right format. We're not going to worry about that for now, but instead let's think about giving a message to someone who enters the wrong number.

In this case we can make use of the *else* construct which means whenever the if isn't true do something else. Update your balance program as follows.

```
1 realPin = 1024
2 balance = 10457
3 pin = input("Enter your pin : ")
4
5 if (int(pin) == realPin):
6     print(f"Your Balance is {balance}")
7 else:
8     print("incorrect pin")
```

Before we move on a quick note on indentation as this is the first time we've seen it and it can cause some confusion in Python. Look at the following code and see if you can work out what will happen before you run it.

```
1
2 a = 5
3 if (a > 2) :
4     print("The number you entered is greater than 2")
5     print("It might not be much bigger")
6
7     if (a < 5) :
8         print("I've just checked and it's less than 5")
9         print("when will this statement get printed")
10 print("and how about this one")
```

Hopefully you can see that we might want to run more than one line of code when our conditional condition holds. Indentation tells us which lines to run and when to return to the main flow of the program. Having different levels of indentation is a feature of Python, unlike many other languages which ignore it completely, and can be a point of confusion.

What we have looked at is fine when we have a simple yes no choice to make

but let's consider a situation where we have lot's of choices depending on the value of a variable. Consider a simple grading program where we tell you your grade based in your mark.

```
1 boundaries = [40, 50, 60, 70, 80]
2 result = ["3rd", "2:2", "2:1", "1st", "Distinction"]
3
4 grade = input("Please enter the grade as an integer : ")
5
6 if (int(grade) > boundaries[4]):
7     print(f"Your result is a {result[4]}")
8 elif (int(grade) > boundaries[3]):
9     print(f"Your result is a {result[3]}")
10 elif (int(grade) > boundaries[2]):
11     print(f"Your result is a {result[2]}")
12 elif (int(grade) > boundaries[1]):
13     print(f"Your result is a {result[1]}")
14 elif (int(grade) > boundaries[0]):
15     print(f"Your result is a {result[0]}")
16 else:
17     print("Your failed your exam")
```

You can see here that we can use greater than ($>$) in if statements. We can also use less than ($<$), greater than or equal to ($>=$) and less than or equal to ($<=$) You can also see this new keyword *elif*, which is short for "else if". You can think of the code falling through this construct and exiting when it hits a condition that is true.

We can also chain conditions together to make more complex decision boundaries. For example

```
if ((age > 18) and (country == "UK")) or ((age > 21) and
    ↪ (country == "USA")):
    print("Drinking allowed")
```

Challenge 8.

Write a program to ask them their name and age. Tell them how long it is until their 35th birthday if they are less than 35 and how long since their 35th birthday if they are over 35. If they are 35 then respond by telling them it sounds like a good age to be.

5.2 Loops

One of the really powerful things about computers is their ability to do the same thing over and over again and in this section we are going to look at the programming constructs which allow us to do this for ourselves.

Take a look at the following program.

```
1 print("Counter : 1")
2 print("Counter : 2")
3 print("Counter : 3")
4 print("Counter : 4")
5 print("Counter : 5")
6 print("Counter : 6")
7 print("Counter : 7")
8 print("Counter : 8")
9 print("Counter : 9")
10 print("Counter : 10")
```

Well technically there is nothing wrong with it, what happens if we want to count to 100 or more? are we really going to just add more and more print lines? The answer is of course not. Repetition of this sort is common in programming and spotting patterns of repetition is a key skill in improving your programming.

So python gives us two constructs for repetition. The first is called the *for* loop and can be used with the collection structures we used earlier. Try this program.

```
1 students = ["Alex", "Charlie", "Sam"]
2 n = 0
3 for x in students:
4     n = n + 1
5     print(f"Student {n} : {x}")
6 print("Student listing complete")
```

Which produces this output:

```
Student 1 : Alex
Student 2 : Charlie
Student 3 : Sam
```

A couple of things to note. Firstly we didn't know how many things were in our list, we simply said that for however many there are do this code. This means that this code will work regardless of how many elements are in the list. Secondly having a counter (n) in the loop is often very useful and the code on line 4 is very common to keep a running total of how many times we have gone round the loop. Finally note the indentation. Just like the *if* statement we saw earlier the indentation tells us how much of the code that follows is included in the loop.

Let's think a little bit about our counter example we started this section with. I really don't want to create a list of a particular size to print out 10 lines (or worse 100). So how can we do that?

Well Python has a nice function which creates a sequence of numbers that the for loop can use. Let's use this to make a much smaller program to do exactly the same functionality. Try the following.

```
1 for x in range(10):
2     print(f"Counter : {x}")
```

Challenge 9.

This isn't quite what I wanted because it starts at 0.

- Change the program to count from 1 to 10.
- Can you also make it count in 2's?

Now you may have chosen to change your print statement, but I'm also going to tell you that range can take up to three numbers.

Change your program with different numbers as the parameters of range to work out what they do. e.g. `range(1,7, 3)`.

I said there were two different loop structures in python so let's move on to the second one, the *while* construct.

Remember the if statement that we used earlier? well while leverages that conditionality behaviour to decide how many times to repeat something. This could be useful if you don't know in advance how many times you're going to need to loop. Think of a program to guess someones age, we could use a higher and lower strategy but it's still going to be difficult to say in advance how many guesses we will need. Anyway let's not get ahead of ourselves and let's start slowly. Try this.

```
1 i = 10
2 while (i > 0):
3     print(i)
4     i = i - 1
5 print("blast off")
```

Notice again we are using the counter idea this time counting down rather than up. Notice also that when $i = 0$ we don't meet the condition of the loop and the indented code is not executed, instead we move to the print statement in the final line.

A quick note of **caution**, if you forget to change the counter you will never leave the loop and the program will hang. Although it's obvious in this case to see that the loop will never end, sometimes these errors are much

harder to find and, unlike syntax errors, the python editor can't warn you about these in advance. If this does happen you can normally interrupt your program using Ctrl-C.

OK, let's put everything we've learnt so far into writing a program to guess ages.

```
1 # This program uses a binary search to guess the players age
2 print("Hello and Welcome to my age guessing program")
3
4 # Set up some variables we will use in the program
5 maxage = 100
6 minage = 1
7 age = int((maxage - minage)/2)
8 correct = False
9
10 while (correct == False):
11     # Guess the players age
12     x = input(f"Are you {age} years old (Y/N)")
13     if (x == "Y"):
14         correct = True
15     else: # if our guess was wrong then were we too high or low
16         higher = input(f"Are you older than {age} (Y/N)")
17         if (higher == "Y"):
18             minage = age
19             age = int(age + (maxage-age)/2)
20         else:
21             maxage = age
22             age = int(age - (age-minage)/2)
23
24 print("Hurray I got it")
```

Well done for getting this far. You now know how to write programs to manipulate data in python and make decisions based on that data. In the next section we're going to look at how we can structure our code to make larger programs more manageable and reduce the amount of repeated code we need to write.

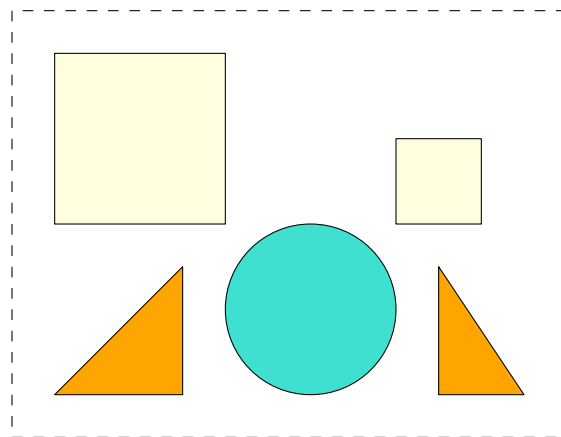
6 Put all of this in a box

With the tools we have learnt so far we can write large and complex programs but there will come a point in time where the single file linear flow model becomes a problem.

If we can break a program down into smaller parts we can:

1. Improve the readability of our code
2. Reuse useful pieces of code
3. Reduce errors by only having to change one piece of code
4. Improve testing of our code

Thinking about programs as interconnected blocks also allows us to plan our programs. Consider a program which will calculate the total area of a set of primitive shapes within a region. The image below is an example of what you might see.



So we might think of a flow something like

1. Load in an image

2. Create data structures to hold our information
3. Find all of the shapes in the region
 - Scan the region
 - Calculate the area of the shape
 - Decide what shape it is
 - Depending on the shape calculate it's area
 - Update our total area

Wouldn't it be nice if the Python program to do this was as easy to read as the list you created. Maybe something like this

```
filename = input("Please enter the file name")
load_image(filename)
data = initialise_data()
area = find_shapes(filename, data)
```

Similarly we might create a smaller program called *FindShapes* which our program calls. *FindShapes* might look something like this.

```
shapes = ScanRegion(filename, data)
for s in shapes:
    sType = shape_type(s)
    area = calculate_area(s, sType)
    update_data(data, area)
```

Unfortunately Python doesn't have specialist functions to CalculateArea but it does give us the capability to split programs up like this and to create blocks of reusable code which are stored in our own functions.

Note: you've already been using functions, like print, so calling them is easy. We're now going to start creating some of our own.

6.1 The anatomy of functions

Let's start by having a look at a function and then breaking down what it's doing. Try typing in the following code

```
1 # At the beginning of the program we will create functions
2 # that the rest of the program will use
3 def addition(a, b):
4     answer = a + b
5     return answer
6
7 # Having defined our function we can now call it
8 x = addition(5, 2)
9 print(f"The answer is {x}")
10 x = addition(3, 2)
11 print(f"The answer is {x}")
12 x = addition(5, 7.3)
13 print(f"The answer is {x}")
```

So, the function here is rather useless because it simply adds two numbers but remember the code in the function can be anything you want.

The `def` keyword here tells Python we are defining a function and the label that follows it is the name of the function we are defining, in this case `addition`.

Next we are going to have **brackets** which contain **parameters**. These parameters are values that the function will use. In this case we are going to send in two variables which are called `a` and `b`.

At the end of the definition line we add a colon (`:`) which, just like the `if` command says the indented lines below it belong to the function. In this case the function has only two lines. The first does the work of the function adding the two variables together and the second uses the `return` keyword to pass control back to the main program and the along with an optional values, in this case the variable `sum`.

Having created our function Python treats it like any other built in Python function, so on line 8 , 10, and 12 we can call it and pass in two values and get the result.

Challenge 10.

Create a function that takes a number and prints out the multiples of that number up to 12. For example if you give me 2 I will output $1 \times 2 = 2$ $2 \times 2 = 4$ etc.

6.1.1 Variable Scoping

Variables defined inside a function are only known inside the function, we call this local scope. It means that when the function finishes all of the information that wasn't passed back with the `return` command is forgotten.

Conversely variables defined outside of the function are available for use inside the function. For example take a look at this program.

```
1  gv = "Hello World"
2
3  # At the beginning of the program we will create functions
4  # that the rest of the program will use
5  def addition(a, b):
6      answer = a + b
7      print (f"the variable gv says {gv}")
8      print (f"the variable y is {y}")
9      return answer
10
11 # Having defined our function we can now call it
12 y = 32
13
14 x = addition(5, 2)
15 print(f"The answer is {x}")
16 x = addition(3, 2)
17 print(f"The answer is {x}")
18 x = addition(5, 7.3)
19 print(f"The answer is {x}")
20
21 print (answer)
```

As we would expect the value of `gv` is printed out inside the function correctly and the value of `answer` is unknown on the final line (we get an error).

What's rather more surprising is that the value of y is printed correctly inside the function. That's because even though we write the function at the top of the program the code isn't run until it is called on line 16 by which time y has been defined.

This scoping problem can cause particular confusion if we aren't careful with our variable names. Try this and think about what the value printed out will be for each print statement.

```
1 i = 10
2
3 def my_func():
4     i = 3
5     print(f"Inside the function the value is {i}")
6
7 print(f"before the function is called the value us {i}")
8 my_func()
9 print(f"after the function is called the value us {i}")
```

6.1.2 Optional parameters

It's quite common to have parameters for a function that are optional this allows us to write richer functions that are more useful. For example maybe we want a function that adds not only two numbers but also lists of numbers. We could create a separate function or we could have one function that accepts either two numbers or one list.

Let's change our program to be like this.

```
1 def addition(a, b=None):
2     if b == None:
3         answer = 0
4         for x in a:
5             answer = answer + x
6     else:
7         answer = a + b
8     return answer
9
10 x = addition(5,2)
11 print(f"The answer is {x}")
12
13 my_list = [1,2,3,4,5]
14 x = addition(my_list)
15 print(f"The answer is {x}")
```

So what have we done? Firstly we changed our second parameter to have a default value `b=None`. This means that if I don't receive a second parameter then assume it has no value. **None** is a special word in python which allows us to check whether something has been defined. We of course set any default value, e.g. `b=1` would mean if we were sent only one number we would expect the program simply to add one to the value and return the result.

Having said that `b` defaults to `None` we need to check for this case and we do this on line 2 using our if code. if this case holds then we use our `for` loop to add up all the values in the list.

7 Classes

Now I think this is an advanced topic in programming so we won't go too deep but I think it's interesting to understand what a class is because it might help you understand how some of the libraries we use later are created.

So far we have thought about writing programs as a sequence of simple commands, but there is another way we can think about our programs. We can consider them as sets of things, or objects, which interact with each other. This is known as object oriented programming.

We might for example think of a 'graph' as an object, a 'data set' as an object etc. In fact a graph might need a collection of data sets, axes, titles etc. Rather than have a single program that operates on all the data in these objects we could give each object some functionality and data that it uses itself.

Maybe that all sounds a bit complicated so let's just dive into some code. Open up VSCode and create a program called `Person.py`. Then enter the following code

```
1 class Person:
2     x = 5
3
4 me = Person()
5 print(f"The value of x stored in me is {me.x}")
```

Well so far this doesn't look a lot different to a function. Well let's introduce the `__init__` function. Change your program as follows.

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6 p1 = Person('alex', 32)
7 p2 = Person('charlie', 16)
8
9 print(f"{p1.name} is {p1.age} years old")
10 print(f"{p2.name} is {p2.age} years old")
```

What we've done here is said that we will pass some values to the object Person which will be used to initialise some variables it holds. The `self` keyword is used to access the class and is used to access variables and functions that belong to the class. talking of functions let's extend our program one last time.

```
1 class Person:
2     voting_age = 18
3     can_vote = False
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7         self.check_vote()
8
9     def check_vote(self):
10        if self.age >= self.voting_age:
11            self.can_vote = True
12        else:
13            self.can_vote = False
14
15    def info(self):
16        s = f"{self.name} can't vote"
17        if self.can_vote:
18            s = f"{self.name} can vote"
19        print(f"{self.name} is {self.age} years old. {s}")
20
21
22 if __name__ == '__main__':
23     people = []
24
25     people.append(Person('Alex', 32))
26     people.append(Person('Charlie', 16))
27
28     for p in people:
29         p.info()
```

Let's look at this program in a little detail.

On line 2 and 3 we've introduced a couple of variables that will be used within the class but aren't really intended for use outside the class.

On line 9 we've created a new function called `check_vote`. It is indented to indicate that it belongs to the class and, like all functions in a class it has as its first (and in this case only) parameter the keyword `self`. This means

that the function can access any other variables and functions that belong to the class. In this case it is going to use the `age` and `voting_age` to set the `can_vote` variable.

Notice that even though we defined this on line 9 we were able to call it in line 7 (inside the `init` function).

On line 15 we then create an `info` function which will write out the information that we have about the people we create using this class. Notice that this means I can call this function for every person I create and I don't need to write print statements each time I create a new person.

Line 22 is a bit odd and I'll explain it in the next section. For now just believe me that it's useful

The last part of the program then creates a collection (list) of people initialises them and prints out their information.

Challenge 11.

Create another function in your class to change the voting age. Make sure when you call the `info` function you still get the right results.

8 Is there a box for my boxes?

Do you remember when we created our first python program we had a line in our code which said

```
import math
```

Well that single line allowed us access to a vast number of functions that someone else had written. Well we can do exactly the same.

In you VS Code editor I want you to create a new file and call it mylib.py

```
1 # This function takes a list and tells you lots of
2 # things about the data
3 def data_summary(a):
4     print("#####")
5     print(f"Number of data points in list : {len(a)}")
6     print(f"Minimum value in list : {data_min(a)}")
7     print(f"Maximum value in list : {data_max(a)}")
8     print(f"Average value of items in list : {data_average(a)}")
9
10 def data_min(a):
11     return min(a)
12
13 def data_max(a):
14     return min(a)
15
16 def data_sum(a):
17     val = 0
18     for x in a:
19         val = val + x
20     return val
21
22 def data_average(a):
23     return data_sum(a)/len(a)
24
25
26 # A quick test of my function
27
28 my_list = [-1, -9, 32, 1,2,3,4,5,6]
29
30 print(f"Minimum in list is {data_min(my_list)}")
31 data_summary(my_list)
```

So as we might expect this creates a set of functions and then executes them. But what makes this interesting is if we create a new file called `data_analysis.py` and make sure you have saved your `mylib.py`.

```
1 import mylib as mine
2
3 # A quick test of my function
4
5 my_list = [-1, -9, 32, 1,2,3,4,5,6]
6
7 print(f"Minimum in list is {mine.data_min(my_list)}")
8 mine.data_summary(my_list)
```

Just like we did with importing the `math` module we have told our program to load to our own module and give it a friendly name to use in our code, i.e. `import mylib as mine`. This friendly name is just an alias and is often used to shorten library names which can be quite long.

As long as the program can find the file called `mylib.py` in the directory (or somewhere else it knows). Then it can make use of the functions inside that file.

But something odd has happened when we run this code we are seeing the output twice! This is because the file we created the functions in `mylib.py` is executed when we `import` it. But it was handy having that test code at the bottom of my file to test the functions so I don't really want to delete it.

Well, remember that strange `if __name__ == '__main__':` line in the last section? It has an important role. It says only run the code belonging to this condition if the file is being called directly, not when it is imported.

With that in mind amend your `mylib.py` program to have your test code defined like this.

```

1 def data_summary(a):
2     print("#####")
3     print(f"Number of data points in list : {len(a)}")
4     print(f"Minimum value in list : {data_min(a)}")
5     print(f"Maximum value in list : {data_max(a)}")
6     print(f"Average value of items in list : {data_average(a)}")
7
8 def data_min(a):
9     return min(a)
10
11 def data_max(a):
12     return min(a)
13
14 def data_sum(a):
15     val = 0
16     for x in a:
17         val = val + x
18     return val
19
20 def data_average(a):
21     return data_sum(a)/len(a)
22
23
24 # A quick test of my function
25 if __name__ == '__main__':
26     my_list = [-1, -9, 32, 1,2,3,4,5,6]
27
28     print(f"Minimum in list is {data_min(my_list)}")
29     data_summary(my_list)

```

Save it and then run your `data_analysis.py` file again. Hopefully you will now only see one set of calls to the functions.

One final note is that libraries can become very large over time and we probably won't want to use all the functions in it. Loading the whole library will be memory inefficient and may slow down our program so sometimes we want to be able to choose the one or two functions we actually need.

```

1 from mylib import data_min, data_summary
2
3 my_list = [5,6,7,8,9]
4 print(f"Minimum in list is {data_min(my_list)}")
5 data_summary(my_list)
6
7 print(f"Maximum in list is {data_max(my_list)}")

```

A couple of things to note here. Firstly line 7 will throw an error because our program doesn't know what `data_max` is, we didn't import it. Secondly even though `data_summary` uses all of the functions in the module we don't need to include them on our import, python will take care of these dependencies for us.

8.1 Finishing off

Well done, you are well on your way to being a Python programmer. I hope that over the next couple of sessions you will see why that is useful.

If you are keen to learn more about programming then I would recommend that you start with the *gitbook* which supports our undergraduate module on software which was written by Dr. Lilian Blot. <https://drililianblot.gitbook.io/introduction-to-programming-with-python/>