Everything You Always Wanted to Know About Python\*

(\*but were afraid to ask)

# Introduction

This a very rapid run down of key things you should know or keep in mind when getting into Python programming as it relates to the basic coding skills that you’ll need for SXPS288. It's not exhaustive (so add to it as you learn). On its own it won't be nearly enough to learn Python - but it might help as a bit of revision. Python does, of course, provide a full set of documentation (Python Docs, n.d.). A word of warning - it’s a bit chatty and occasionally irreverent, but then lots of people who enjoy coding are like that. If you don’t mind rude words, have a search to see why lots of programming examples use the variable names “foo” and “bar” - don’t if you’re easily offended.

Learning programming can be great fun and very rewarding. With even a limited skill set you'll be able to analyse, manipulate and display complex data far better than with things like calculators and spreadsheets - but do bear in mind these primitive tools will still be useful for simple things. Some of you will find the coding experience exasperating, infuriating and sometimes almost impossible to comprehend (I did) but persevere - do you remember the first time you came across algebra (**What on Earth do you mean x=y ?????).** Stick with it and things will become clear(er).

The first thing you need to understand is that any programming language (and Python is no exception) is just a simple way of communicating with a computer that allows you to create a recipe that it will follow in order to do something. Now computers are just glorified calculators and have no intelligence. They really can't 'interpret' what you give them and say 'Oh, that's OK I think I know what you mean'. The syntax of the language is very precise and interpreted absolutely strictly. A simple misspelling, the use of a capital letter or a single space in the wrong place will 'crash' the system or far worse, seem to work, but not give you the answer you intended **(test all code thoroughly!)**.

Secondly, it’s worth realising that, whilst you're learning, however wrong you get something, it will not cause World War III - or even break the computer. So, feel free to experiment. Children don't seem to have any trouble picking up programming and part of the reason is they have no fear or preconceptions. They'll give anything a go out of curiosity and learn stuff by trial and error rather than lean by rote. If you're struggling - try and bring out your inner kid.

Finally, remember that it’s almost true that you can teach monkeys to program. There are just over 30 (33 last time I counted) keywords in Python and a very limited ‘grammar’. The average English speaker uses a vocabulary of around 20,000 words - and don't even try to explain the subtleties of grammar *"To whom am I speaking?"* but *"Who is at the door?"* (whom being the dative form of an old English 'who') ... I rest my case. Interestingly, you and I would have no difficulty in understanding *"Two who are I speking?"* or *"Whom is at the dore?"*, but then we aren't computers are we? So I maintain that if you can learn even a smattering of a foreign language, you can learn the basics of Python!

HOWEVER, learning to order your thoughts in a way that is necessary to write functional code is really, really hard! Anything you want to do has to be split up into simple instructions that can, in the simplest case, just be executed sequentially, or, much more interesting and usefully, repeated constantly until something useful changes and/or only executed if something useful has just happened. I simplify somewhat but that's about the gist of it.

SO, learning the syntax and grammar of Python is simple - learning to take a problem, turn it into a sequence of logical steps, loops and forks is not. Good luck!

In the rare even you see a box like this it will contain REALLY advanced concepts. Don’t go there unless you’ve had an excellent breakfast and have lots of coffee on hand

NOTE, as you read through, you’ll come across blue boxes like this. These usually contain extra, more advanced material that you can probably skip or skim through on a first reading

You’ll also see boxes like this:

1. Python = ‘easy’

A screenshot of a computer

Description automatically generated

Figure : Python 'shell' running in an Anaconda Prompt – this is green yours may well be white text on a black background:

These contain snippets of Python code. Sometimes they stand on their own and sometimes they may contain several lines including the Python prompt (>>>). Just ignore the line numbers. You might like to have a Jupyter Notebook (Jupyter Notebooks, n.d.) open, or an ‘Anaconda Prompt’ running a Python shell – or any other Python shell – to try things out as we go along. Figure 1 shows Python running in an Anaconda Prompt – when you run Python you will get a ‘>>>’ prompt that tells you Python is running and ready to take input from the keyboard. This allows you to run single Python commands interactively. Later when we come on to coding, you will be creating a list, or ‘script’, of commands in some form of editor (most probably a Jupyter Notebook’) which you can then ‘run’.

The first part of this document will look at basic Python statements and simple data types. From section 4, Coding we’ll get into programming and more advanced Python.

Just a final point: we’ll be talking about Python 3.x. You can still use Python 2.x but there are some differences – so be warned!

# Storing things in the computer

At its most basic, a computer:

* gets values in from the outside world (input)
* stores the values (numbers, characters, words etc.) in specific locations in its core
* manipulates these values in some way
* moves the resultant values to other storage locations
* gets values out to the outside world (output)

The earliest computers used switches and flashing lights for input and output - we're much more sophisticated now we use keyboards (lots of switches in a grid) and flat screen displays (loads of LED lights in a huge grid).

So, how does Python allow us to instruct the computer to perform these functions?

## Variables

First, we’ll look at how the computer stores things internally – out of sequence but a useful starting point for what’s to come.

Firstly, you get to name these values (or rather the location in which they’re kept) so you can always refer to them easily. These are then called '***variables***' The names can be pretty much anything you like (Figure 2):

Notice some things though:

Figure : Examples of variable names

*val, val1, avalue, fred, hello, world, Hello, hElLo\_WOrlD, pa55w0rd, Gubbins, tarantula, jdvuye, xxxx, first\_variable, my\_variable,*

1. You can't start with a number - 1a is not a valid variable name
2. You can't have spaces, or ! or : or ... In fact, just stick to letters, numbers and the underscore ('\_').
3. Hello is NOT the same as hello, or HeLLo or ... Capitalisation is important!
4. You shouldn't use one of the 33 (or so) keywords (print, and, True, is, False, None, as, if ...)
5. The names you use don't have to mean anything - but usually you should try to choose names that relate to what is stored as it makes reading the code easier and you're less likely to make a mistake.

## Types

So, now you've decided on a nicely named variable, how do you get a value into it? The answer is to use the ‘assignment’ ‘operator’ '='. ***Do please note this doesn’t mean ‘equals’ as in mathematical usage – we’re coding not doing calculus.***

1. my\_first\_value = 3.142

So, here is the first Python 'gotcha'. Read the '=' as 'assign to'. In the above line we 'assign' the value of 3.142 to the variable my\_first\_value. It does NOT mean that my\_first\_value is necessarily EQUAL to 3.142. This might seem a subtle point, but it is important because you can do things like this:

1. my\_first\_value = my\_first\_value +1

 This needs to be understood as ' take the value in my\_first\_value, add 1 to it and assign it to my\_first\_value'. So now the value in my\_first\_value is actually 4.142.

Now we need to talk about ***types.*** What values can we store in a variable? It's important to know this because we need to know what we can and can’t do with and to the stored value. As an example, adding two numbers together will obviously result in the sum of the 2 numbers (Figure 3) but what does adding two words together do (Figure 4)?

There are 4 basic 'types' of information we can store:

1. **Integers** (int) - these are whole numbers (1,2, -3456, ...)
2. **Floating point** numbers (float) - these are floating point number (1.0, 3.142, -0.000361, ...) NOTE you can also use scientific notation too. (1.23e2 is 1.23x102)
3. **Strings** (str) - contains characters, words, sentences ... (‘fred’, “1”, ‘World’, ‘3.142’). You’ll notice we can see these are strings by using “ “ or ‘ ‘. Also note that ‘1’ and “3.142” are strings and can’t be manipulated as if they were numbers.
4. **Boolean** (bool) - these are used for comparisons and only have 2 values possible True and False (NOTE the initial capital letter).

|  |
| --- |
| This is more advanced, but you can also represent imaginary numbers:  x = 3.2+0.2j  Get the real part: x.real returns 3.3  And the imaginary part: x.imag returns 0.2  And a ‘magnitude’ value (√(re2 + im2)): abs(x) = 3.2062439 |

There are some other, more technical types, but this will do for now. Also, in the not too far distant future we’ll introduce the idea of **‘lists’** which allow you to aggregate a number of variables together (into a list or array of values) and access members of this aggregate easily.

Now we can examine what difference the type of a variable might make by looking at a couple of examples:

Figure : Example showing addition of numbers

*Example1.*

*x = 1.0*

*y = 2.5*

*x+y will give the result 3.5 (which is probably what you’d expect)*

However,

Figure : Example showing concatenation of strings

*Example 2.*

*x = ’1.0’*

*y = ’2.5’*

*x+y will give the result ‘1.02.5’ This has joined the two strings – called CONCTENATION*

A bit of a warning: if x=1 and y="2", x+y will crash - you can't add an int to a str.

## Converting variable types

There will be times when you need to convert a variable type to another. You can do this as long as it makes sense.

So, if we have x="3.142" you can convert x to a floating point number (in order to do some sums) by typing:

1. x=float(x)

 This is a first example of a built-in ‘function’ – in this case **float()**

Conversely, if we have, say, a=2.718 we can convert it to a string by:

1. a=str(a)

Do be careful though: x="pi" is fine, but then float(x) will crash. The conversion of strings to int and float will come in very useful when we get to inputting stuff - next.

## Lists - Aggregating variables

We often need to collect a number of variables together. For example, we might want to have a list of star coordinates we want to work on. Or we might have a series of data points to plot on a graph. We could do this by allocating a lot of similar variable names – coord1, coord2, coord3, … or x1, x2, x3, … This would work but would rapidly become unwieldy and would be difficult to automate any processing needed on all the variables.

This is where the concepts of lists come in. (also tuples, arrays and dictionaries – these are related data structures we’ll come to later). A list is indicated by square brackets ‘[]’. This is best seen with a simple line of code:

1. y\_values = [1.1, 3.9, 9.1, 16.5, 23.9, 36.0, 50.1, 63.2, 82.1, 102.3]

We now have a list containing some y values called ‘y\_values’ – lets have some x values.

1. x\_values = [1,2,3,4,5,6,7,8,9,10]

 You might like to consider what kind of curve this would make – we’ll see later.

### Accessing, modifying and deleting list values

The nice thing about using a list is that you can access/retrieve any value by using an ‘index’ into the structure. So, the first value has an ‘index’ of 0 (computer often start counting at 0, not 1), the second has an index of 1 … and so on. So to see, for example the fifth value of the y\_values list (36.0) we would ask for ‘**y\_values[4]**’ (remember indices start at 0).

Below is a Python console session showing how this works. The important thing to note here is what happens if you use an index that doesn’t exist, the line numbers, by-the-way won’t necessarily appear and are not important.

1. >>> y\_values
2. [1.1, 3.9, 9.1, 16.5, 23.9, 36.0, 50.1, 63.2, 82.1, 102.3]
3. >>> y\_values[2]
4. 9.1
5. >>> y\_values[0]
6. 1.1
7. >>> y\_values[8]
8. 82.1
9. >>> y\_values[10]
10. Traceback (most recent call last):
11. File "<pyshell#10>", line 1, in <module>
12. y\_values[10]
13. IndexError: list index out of range

 There are a number of ways to manipulate the data in a list.

Amongst these, you can find out how many things are in the list

1. >>> len(y\_values)
2. 10

 You can add (append) things to the list:

1. >>> y\_values.append(3.142)
2. >>> y\_values
3. [1.1, 3.9, 9.1, 16.5, 23.9, 36.0, 50.1, 63.2, 82.1, 102.3, 3.142]
4. >>>

 And there are a number of ways you can remove things from the list:

1. >>> y\_values.remove(3.142)
2. >>> y\_values
3. [1.1, 3.9, 9.1, 16.5, 23.9, 36.0, 50.1, 63.2, 82.1, 102.3]
4. >>>

 Or

1. >>> del y\_values[10]
2. >>> y\_values
3. [1.1, 3.9, 9.1, 16.5, 23.9, 36.0, 50.1, 63.2, 82.1, 102.3]
4. >>>

### Slicing and dicing lists

The final thing to look at is ‘slicing’ which allows us to get sub-sets of any list.

To do this we need an index for where we want to start the sub-list and an index for where we want to end it. Let’s say we want the sub-list of y\_values that contains the values 9.1, 16.5 and 23.9. The indices of these values are 2 and 4. Let’s say we want to assign this to a new shorter list called y\_sub\_list. Here’s how it’s done:

1. >>> y\_sub\_list = y\_values[2:5]
2. >>> y\_sub\_list
3. [9.1, 16.5, 23.9]
4. >>>

  So basically, it uses a start position, a colon, and a finish position within the square brackets ( **[start:finish] )**. The eagle-eyed amongst you will have spotted the inconsistency here – the start index seems fine, but the end index is one greater than what you might expect. Unfortunately, that’s the way it is, so best to just learn it – or you might like to think of it in terms of slice ‘limits’ (Figure 5**Error! Reference source not found.**)

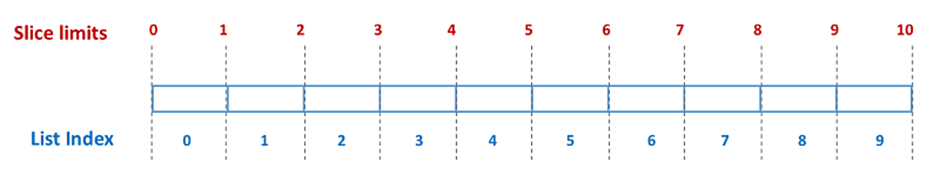


Figure : Introducing the concept of slice 'limits'

From this you can see that a sub-list from index 2 to index 4 is ‘book ended’ on either side by ‘limits’ of 2 and 5. Take your choice.

The final subtlety with slices is that if you are starting from the beginning or ending at the end you can omit the actual index or limit. So, in our case, the sub-list [1.1,3.9] can be obtained by:

1. y\_values[:2]

and the sub-list [63.2, 82.1, 102.3] by

1. y\_values[[7:]

|  |
| --- |
| There is also the concept of ‘striding’ that allows you to slice up the list but introduces a ‘step’ value as well that skips over ‘n’ values. Difficult to describe, so, by example (note the extra ‘:’):  Y\_values[::2]  (the whole array with a ‘step’ of 2) gives  [1.1, 9.1, 23.9, 50.1, 82.1] |

There are other subtleties, like negative indices but that’s for another time.

## Tuples, Dicts and Sets

There are other structures that allows you to aggregate things together and we’ll touch on them here. Dicts in particular can be very useful – they’re a bit like mini databases – but generally lists will be the main structure you’ll use.

### Tuples

These are just like lists – but you can’t change any values in them (you can add and delete values from the tuple but not change them. In the jargon they are ‘immutable’.

They are created in a similar way to lists:

1. my\_tup = (1,2,3,4,5) or just my\_tup = 1,2,3,4,5

and can be accessed in the same way:

1. my\_tup[2] gives 3

 but

1. my\_tup[2] = 9

will cause an error.

### Dicts (or Dictionaries)

These are collections of data that can be accessed by a reference ‘key’. So, each item is composed of 2 parts – a key and the actual data. They are created in a similar way to lists but using {}. So, let’s set up a dict that contains the atomic number of some elements. The key will be the element symbol and the data will be the atomic number, and these will be entered in pairs:

1. my\_dict = {'U':92, 'C':6, 'Cd':48, 'N':7, 'O':8}

Now we can search the dict as if it were a mini database:

1. my\_dict[‘N’] will return 7

You can check if something in the dict

1. ‘Cd’in my\_dict will return True, ‘Xe’in my\_dict will return False

You can add:

1. my\_dict[‘S’] = 16

And delete:

1. del my\_dict[‘N’] just deletes, val=my\_dict.pop[‘N’] deletes and returns the value

…and so on

### Sets

These are ***unordered*** sets of unique values and can be manipulated like mathematical sets. They are created in a similar way to dicts and have many of the same operations as lists. As well as that you can perform set operations. So, if we have:

1. s1 = {1,2,3,4,5} (any duplicate values will just be ignored and not added)
2. s2 = {3,4,12,15,99}

 Then

1. s1.union(s2) gives {1,2,3,4,5,99,15,12} (note the order is NOT important)

and

1. s1.intersection(s2) gives {3,4}

|  |
| --- |
| A warning note that if you do  list\_1 = list\_2,  then list\_2 IS JUST ANOTHER NAME FOR list\_1 and changes made to list\_2 will also change list\_1. In computer terms, both list\_1 and list\_2 ‘point’ to the same object.  If you don’t want this behaviour, then use the list copy:  new\_list = old\_list.copy() (doesn’t work with Python 2.x)  or, use slices:  new\_list = old\_list[:] (Obviously not as readable though)  also works with dictionaries.  We’ll revisit this later when talking about writing your own functions. |



# Getting simple stuff in and out

As we've come across earlier, a computer programme essentially comprises:

1. Getting stuff in (input)
2. Doing some computational wizardry on this stuff
3. Getting the results out (output)

There are a number of ways of getting data into a computer:

* Keyboard entry
* Reading files in from disk
* Reading information from over the Internet
* Reading data from attached sensors

In this brief section we'll be looking at keyboard entry but will do file and Web reading later.

Similarly, for output:

* Printing to the screen (and maybe a printer)
* Plotting data/images to the screen
* Saving such plots and data to file

We'll mostly be looking here at text output to the screen. Later on we'll talk about how to plot complex data to the screen as images and how to write files of data out onto disk.

## Keyboard input

This is fairly simple really – you use the inbuilt function **input().** Let’s assume you want the user to input their name. We also want to assign this to a sensibly named variable – how about ‘your\_name’. Here’s how.

1. your\_name = input()

 This works fine but is awfully user un friendly as the computer will simply wait for you to enter something – but won’t tell you so, nor prompt you for what it’s expecting. So, let’s make it issue a prompt.

1. your\_name = input(“Please enter your name ”)

 Much better, the user is prompted to enter a name

NOTE, the input value will be a string type – even if only numbers are entered. This is a problem if you want to do some calculation with it. SO – remember section 2.3 – if you want an number (integer or float) you need to convert it.

1. pi\_val = float(input(“Please enter a value for Pi “)

 NOTE, see how you can embed functions inside one another.

|  |
| --- |
| You can input more than 1 value in a number of ways:  x,y = input('Enter 2 values separated by spaces ').split()  Convert them to an appropriate type:  x,y = map(float, input('Enter 2 values separated by spaces ').split())  Or get them as a list  my\_list = list(map(float, input('Enter 2 values separated by spaces ').split())) |

We’ll be talking about reading things in from files – rather than having to type them in, later on.

## Printing things to the screen

Rather conveniently we can print things out to the computer screen by using the function ‘print()’

1. print(‘The rain in Spain falls mainly on my holiday’)

 Or let’s say we have a variable called ‘your\_name’

1. print(‘Your name is’, your\_name)

Notice that you can print more than one thing at a time. You can also embed calculations or function calls:

1. >>> x=1
2. >>> y=2
3. >>> print('answer is', x+y)
4. answer is 3
5. >>>
6. >>> my\_list = [1,2,3,4,5,6,7]
7. >>> print ('The length of my\_list is', len(my\_list))
8. The length of my\_list is 7
9. >>>

So, there you have it, we can get stuff into and out of the computer!

|  |
| --- |
| Print Formatting  Given the following:  x=12.3  y=9  z=x/y  a\_str = ‘John’  then you can format the way it is printed out using the .format() function:  print('Hello {}<. x is {} and y is {} and z is {}'.format(a\_str,x,y,z))  gives:  Hello John. x is 12.3 and y is 9 and z is 1.3666666666666667  and  print('Hello {:10s}. x is {} and y is {} and z is {:06.3f}'.format(a\_str,x,y,z))  gives  Hello John . x is 12.3 and y is 9 and z is 01.367  Showing string padding (pad out to 10 characters) and float formatting (at least 6 digits with 3 decimal places).  Alternatively, you can use formatted string literals which are similar but less verbose. NOTE the ‘f’ before the string specifier – they’re often called ‘f strings’:  print(f'Hello {a\_str:10s}. x is {x} and y is {y} and z is {z:06.3f}')  gives the same result:  Hello John . x is 12.3 and y is 9 and z is 01.367  There are many other formatting options. |

# Coding

We now need to get into the nitty-gritty of programming. First, as we mentioned in the introduction, one of the hardest things to get your head around in the early days is the disciplined thinking needed to produce the detailed code used to solve the problem at hand.

Consider the task of opening your front door. You might describe it like this:

* Put the key in the lock
* Turn the key
* Open the door

This would work fine for most people. But consider an alien who knew little of our culture. You’d have to be much more precise. Maybe something like:

* Approach the door
* Identify the key hole.
* If there is a cover over the keyhole then move it out of the way
* Take the key and push the sharp end into the key hole.
* If the key isn’t fully inserted, push harder
* Now rotate the key clockwise – as viewed from the big end
* When the key can’t rotate further push on the left-hand edge of the door
* Keep pushing the edge of the door until it is fully open

You might see flaws even in this set of instructions, but the idea is you have to give precise step-by-step simple instructions, allowing for alternative options (keyhole covered or not) and for repetitious actions (keep pushing at the door until …). Programming requires you to precisely this, breaking down the computational task into simple steps that the computer can perform.

Whenever you approach a programming task, DON’T start programming but DO sit down, think about the task and draw/write some kind of plan that can THEN be translated into program code.

You could write simple steps as we have just done or you could use a flow chart, or there are other more formalised methods which we won’t go into here. **Error! Reference source not found.** shows a flow chart – slightly simplified – of the Alien’s instructions (with some annotations).

The blue annotations show a SELECTION and an ITERATION. The SELECTION shows a programming construct which shows that depending on the value of something (in this case whether the lock is covered (True) or not (False)), one thing (or block of code) or another will happen. The ITERATION shows a construct which shows an action (block of code) is continuously repeated until something (door is open) is satisfied. We’ll be returning to these constructs later.

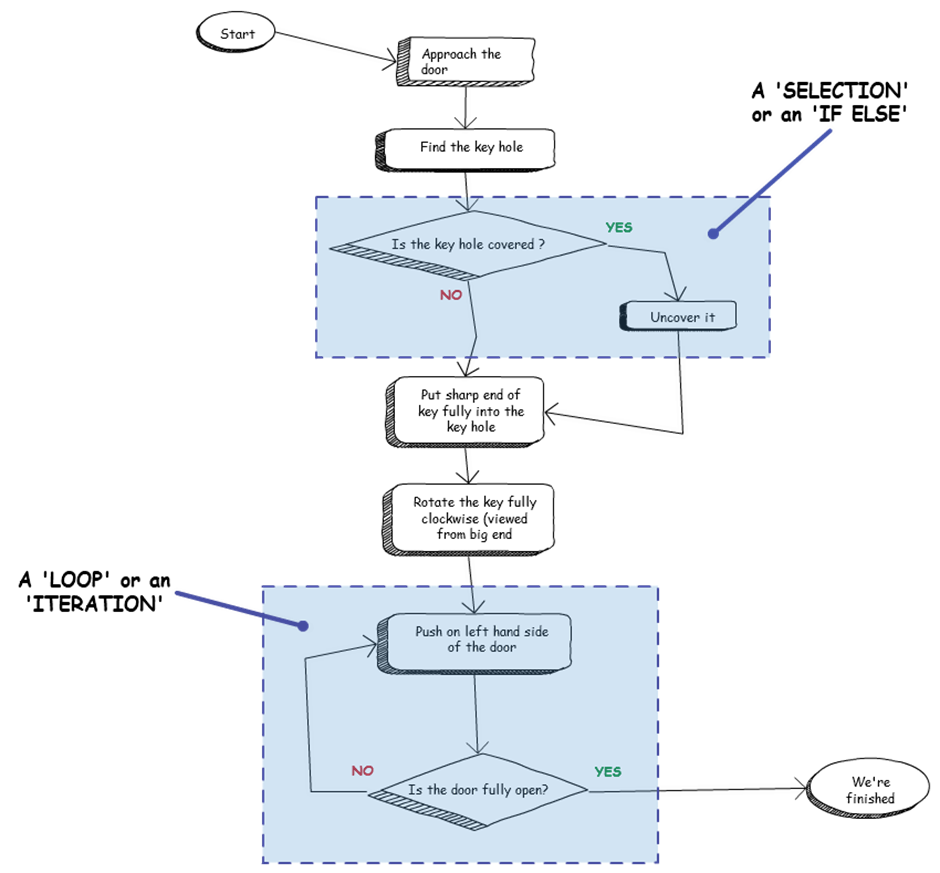


Figure : A simplified flow chart of Alien Instructions for opening a door

Now, finally, we can get down to some programming. We’ll be using the Jupyter Notebook with its built in editor, but if you’re feeling adventurous, try out the ‘Spyder’ IDE (which stands for Integrated Development Environment) that is also installed by default in the Anaconda setup. This provides a pretty easy to use environment that shows (amongst others) a widow for editing your code and a window for input and output. We’ll stick with Jupyter Notebooks for now.

Start a new notebook and, in a ‘Code’ cell, type in the following (ignoring the line numbers):

1. # A simple program to ask for 2 numbers, add them together
2. # and print out the result
4. x = float(input('Enter the first number '))
5. y = float(input('Enter the second number '))
7. print('The sum of the 2 numbers is', x+y)

 You should see something similar to Figure 7.

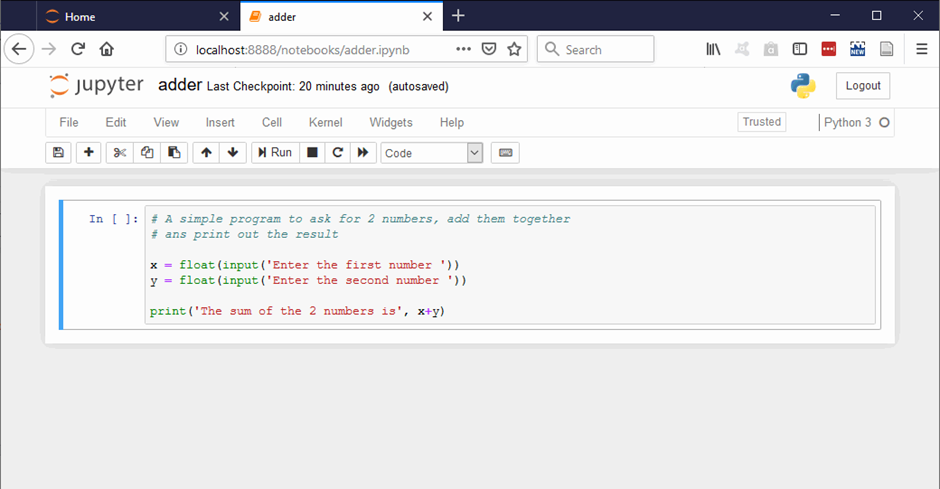


Figure : The first 'adder' program

Now run it (use the ‘Run’ button) and you should be prompted for 2 numbers and then shown the sum of these (Figure 8). From now on we’ll just show the essential Cells to avoid cluttering up the pages.

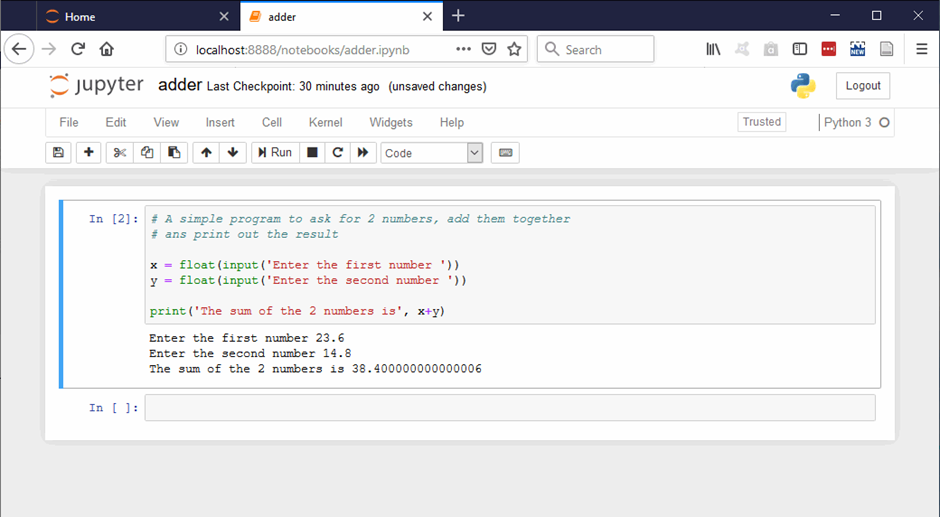


Figure : Running 'adder'

Hopefully you’ve now got a simple but functional program – congratulations!

|  |
| --- |
| The ‘adder’ just uses the ‘+’ maths operator but there are, of course more including the ones you’d expect: subtraction (-), multiplication (\*) and division (/). You can use brackets to clarify the calculations too:  3+2\*6 = 15, but, (3+2)\*6 = 30  There are others including:   * Sort of short cuts: x += 4 adds 4 to the previous value of x * Integer division (//) which gives the sum without remainder: 23//5 = 4 * Modulo (%) which gives the remainder: 23%5 = 3 * Power: (\*\*) raises to a power: 4\*\*3 = 64 |

## Branching and Looping (Selection and Iteration)

We now want to introduce the way we get our program to do something different depending on some state or value (Selection, branching, If-Else etc.) and to repeat sections of code while some state or value is true (Iteration, while loops etc). Just for fun we’ll do this by example before getting into the nitty-gritty – and we’ll base this all around a program to solve quadratic equations.

As I’m sure all SXPS288 followers could solve the equation ax2 + bx + c = 0 by using the formula:

To start, lets prompt for a, b and c and calculate the 2 roots (using the + and then the – alternatives). You need to enter something like this (Figure 9):

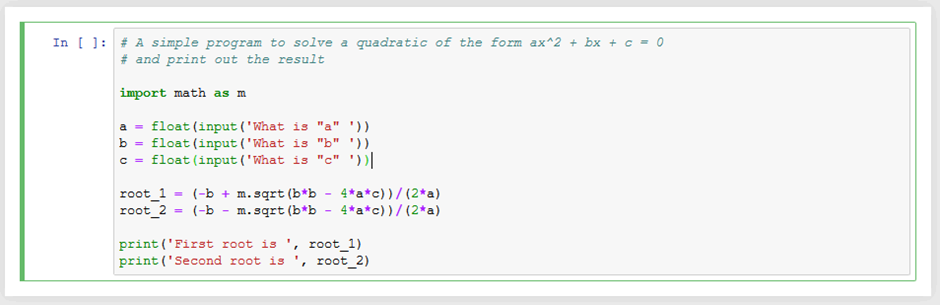


Figure : 'solver', a simple quadratic solver.

A couple of things to note:

* A line beginning with a ‘#’ is a comment line. Python ignores it and it is just some plain text to comment on the code. It’s just there to explain and/or clarify the code. Python will also ignore anything after the ‘#’ if it appears later in a line.
* There is a line ‘import math as m’. We’ll go into this later, but it is a way of ‘importing’ and using someone else’s code in our program. In this instance it is a load of mathematical functions. In our case we want to use the ‘sqrt()’ (square root) function. Notice we’ve imported math as ‘m’, so we use a ‘m.’ before the function name to access it (m.sqrt())
* You can’t just use 4ac and expect Python to know what you mean – you need to explicitly use the multiplication ‘operator’ – 4\*a\*c

Now you should find that if you enter the values: a=1, b=-5, c=6, you get the two (correct) solutions 2.0 and 3.0. Result! However, there are some issues …

### Selection (If – else)

The mathematically discerning amongst you will be wondering, though, what happens if the roots turn out to be imaginary? (Try a=-1, b=2, c=-3). You’ll find that the result is … not good!

Well we know that this happens when the (b2 – 4ac) bit is negative (and we don’t like negatives in a square root do we?). So we need to modify the code to first check if b2 is greater or equal to 4ac(b2 >= 4ac), only do the calculations if it is and just print a warning if it is not. We use the ‘**if – else**’ construct for this. Figure 10 shows how this looks.

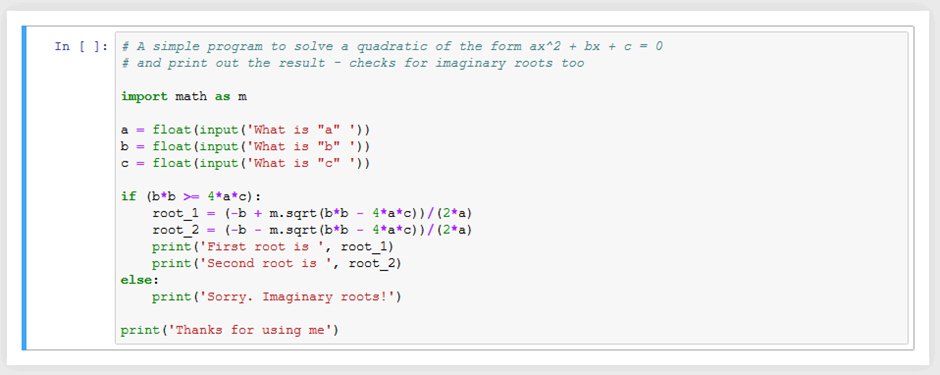


Figure : 'solver', now coping with imaginary roots

Let’s look at how the if-else construct works.

* Firstly the ‘if’ line. Note that it has an expression in the brackets (you don’t actually need these, but I think it looks clearer) which is using a ‘BOOLEAN’ operator (>=). This means the expression can only have two possible out comes True or False. Then we finish the line with a ’:’
* Now, if the expression we’ve just mentioned is True, Python will do the next block of code. (in this case a couple of calculations and a couple of prints(). IMPORTANTLY, the block of code is delineated by being indented (by convention, always use 4 spaces to do this – ignore this at your peril!).
* Next, if there is an ‘else:’ (don’t forget the colon) statement (and there doesn’t have to be), the next block (in this case only a single print() line) will be run if the original ‘if’ expression was False.

So, here, in this program we have 3 input() lines, which are always run, an if-else section that calculates and prints some answers OR a warning, and finally a single line, which always runs, printing a thank you message

|  |
| --- |
| There are a number of Boolean (true or false) operators which are pretty self-evident: <, >, >=, >=  Finally, though, ‘==’. This is the equals operator and you note it has 2 ‘=’ signs. So a single ‘=’ assigns a value and the double ‘==’ is a Boolean operator checking if two things are equal.  You can also make more complex Boolean expressions using ‘and’, ‘or’ and ‘not’. So, for example  if we wanted to check if a number was not in the range 1 to 8, we could use this line:  If not (x>=1 and x<=8):  Now if x is, say, 3, this will evaluate to False, whereas if x is 0.5 the expression will return True. |

|  |
| --- |
| If you want to check for a number of possible Boolean values sequentially, use the if-elif-else construct. This checks through a ‘list’ of possible values one after the other and stops if one of them is True. Probably easiest to see an example. Try running this:  x = int(input('Enter an integer number: '))  if x < 0:  print ('negative')  elif x < 20:  print ('less than 20 but greater than 0')  elif x < 30:  print('less than 30 but greater than 19')  else:  print('greater than 29')  This is quite like the ‘case’ statement you might find in other programming languages. |

### Looping (while …)

As a final tweak to our solver, it would be nice to repeat it multiple time so that we can solve multiple equations without having to restart it.

Now obviously, we could cut and paste all the lines to give us 2 goes at it, one after the other, or paste again to repeat 3 times and so on. But this rapidly makes the program huge and anyway, how many times do we want it to run? The answer is to put the main lines in a loop and repeat until we don’t want to any more. For this we use the ‘while’ loop. There are various ways to use this, but we’ll look at only one here (more later).

The form is similar, in a way, to the if statement – once again we’ll use an example Figure 11.

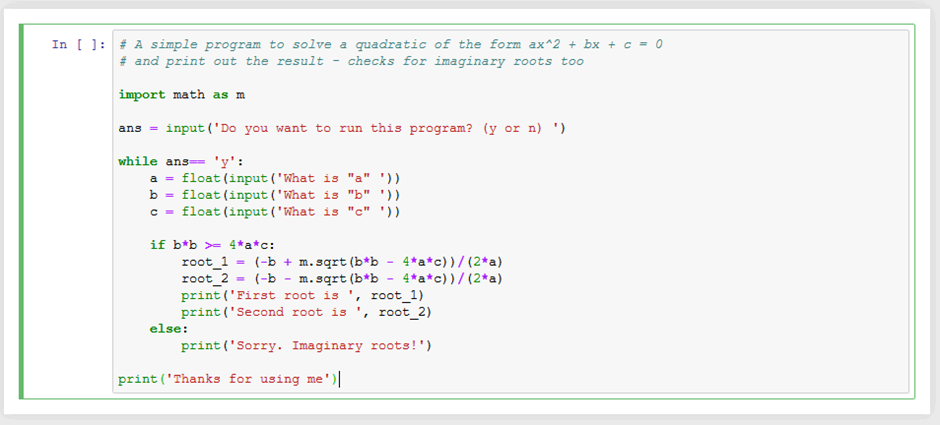


Figure : 'solver' - with a flawed loop

Now, right at the start, we’ve added an input to get a response into the variable ‘ans’ (it wants you to just input a ‘y’ or a ‘n’). The important part is the ‘while ….. :’ line This checks to see if ans is equal to ‘y’ (notice the ‘==’) and only if it actually is doe it execute all the following lines in the block – which is indented by 4 spaces. NOTE the if-else blocks are then indented from there – you can have indents within indents!), So if you entered ‘n’, Python would jump straight to the closing print() line, which is outside the indent.

However, as we’ve answered ‘y’, Python does exactly as it did before, prompts for input, does the calculations and prints the roots OR prints a warning. HOWEVER, now is restarts the ‘while’ block, re-checks that ‘ans’ is ‘y’ (which it still is as we haven’t done anything to change it) and repeats.

Great! Except it will keep doing this for ever! Not good. (actually, you can crash the thing by typing gobbledygook into one of the inputs, pressing Ctrl-c – but all that is cheating). What we need is another chance to input ‘ans’. And so we get to Figure 12.

Now we can exit the program at the end of each run through the calculations. Job done! There are other ways to do this, some of which are more elegant and some of which may be horrible, but this is one of the nice things about programming there aren’t any ‘right’ answers. If it works, then that’s fine. You may find prettier, quicker, more readable, safer (catches rubbish input) but just as long as it calculates the right answers, it’s fine!

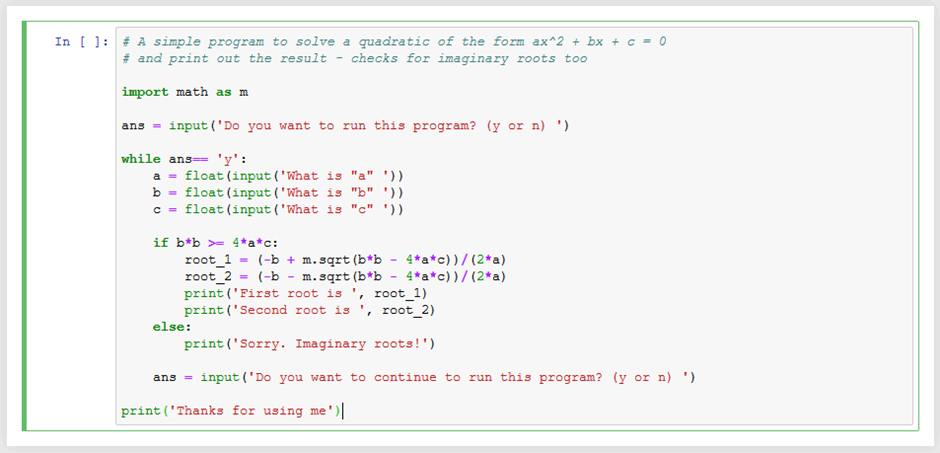


Figure : 'solver' - with a much better loop

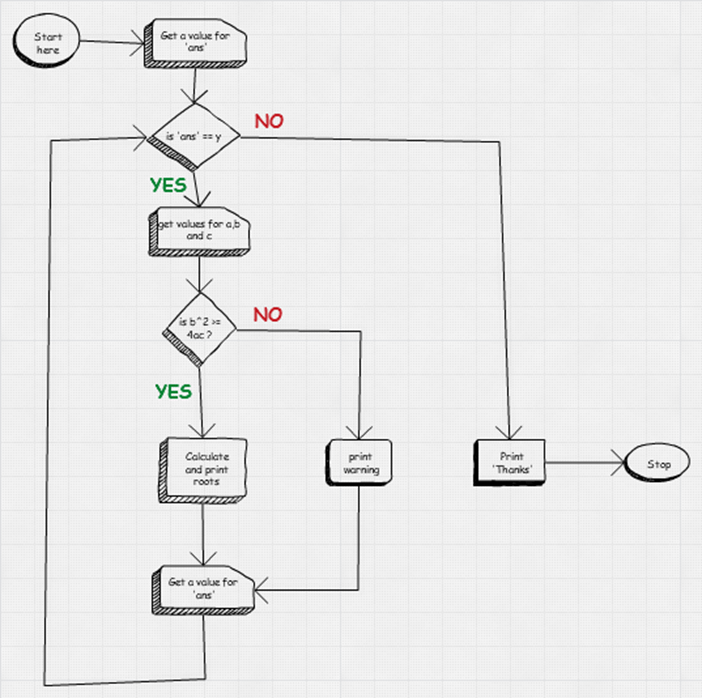


Figure : 'solver' – a flow chart

Although this is back-to-front – the best advice is to do the flow-chart first - Figure 13 shows what the design flow-chart might have looked like for the completed ‘solver’.

### Looping through a list – the ‘for’ loop

A very common task in any program is to do something to each variable in a list – even as simple as printing each value. This is such a common thing to do that Python provides a simple way to do this using the ‘for’ loop construct. Let’s say I’ve constructed a list of elements that fulfil some criteria or other:

1. els = [‘H’, ‘Ba’, ‘C’, ‘Kryptonite – oops how did that get in there’]

 and I’d like to now print out the list. Here’s how I could do it:

1. for element in els:
2. print(element)

The construct ‘for somename in list:’ cycles through all the members of the list assigning the list value to the ‘somename’ in turn.

### Looping – a fixed number of times

Sometimes we just want to loop through something a fixed number of times – say we wanted to print out the first 10 ‘squares’. We can do this using the range() function which is used to generate a sequence of numbers. It’s general form is range(start, stop , step] where the use of the ‘start’ and ‘stop’ are optional. So to print the first 10 squares, ***starting at 0 use***:

1. for x in range(10):
2. print(x\*x)

|  |
| --- |
| To demonstrate range() with its optional parameters:  for x in range(3,10):  print(x)  would give 3,4,5,6,7,8,9  for x in range(1, 15, 2):  print(x)  would give 1,3,5,7,9,11,13 |

**List Comprehensions**

As we've just looked at lists and looping through them, this is as good a place as any to talk about list comprehensions. This is an interesting construct that allows you to operate on a list (or lists) with a single line.

It's very powerful but can make the code harder to read - use at your discretion.

The form looks a bit like a list itself and goes something like:

new\_list = [an expression a\_for\_loop some\_conditions]

OK, that's as clear as mud. Let's have a look at a simple example.

Assume we have a list: my\_list=['the', 'boy', 'stood', 'on', 'the', 'burning', 'deck'], and we want to make all of the strings capitalised. We could write a loop to do this, but we can also use list comprehensions:

cap\_list = [wd.capitalize() for wd in my\_list]

This gives us the new cap\_list containing: ['The', 'Boy', 'Stood', 'On', 'The', 'Burning', 'Deck']

How about is we wanted to produce the cubes of the first 9 primes (a=[1,2,3,5,7,11,13,17,19])

print([x\*\*3 for x in a]) gives [1, 8, 27, 125, 343, 1331, 2197, 4913, 6859]

But there's more! let's say we have 2 lists a=[1,2,3,5,7,11,13,17,19] and b=[1,2,3,4,5,6,7,8,9,10] and we'd like to see what numbers are common to each: we can just use:

common\_list = [x for x in a for y in b if x=y]

Which gives - [1, 2, 3, 5, 7]

This brings us to the end of the first part of this roundup. You have covered enough to do pretty much anything you want in programming (told you monkeys could program) but the next part will introduce stuff that will just make your life easier.

# Reusing chunks of code - functions

Once you’ve written a nice bit of code, you may want to reuse several times over in a bigger program. Now, with a good editor, you could just cut and paste the block at the relevant place – and this would work. But there are a couple of fairly obvious issues with this:

* Your main program will rapidly become big and unwieldy.
* If you spot a nice modification to the original code you’ll have to remember to make the change wherever you’ve copied it to – a bit of a pain.

Most people who don’t program for a living are inherently inclined to make errors and anyway would rather do science than program. So, wouldn’t it be nice to ‘borrow’ someone else’s code that they’ve sweated over and got working well already?

We’ll now introduce the way we can do both of these things by talking about functions and external modules.

## Functions – rolling your own

A screenshot of a cell phone

Description automatically generatedJust for demonstration purposes we’ll modify our quadratic solver program. We want to bundle the actual ‘solving’ code up and make it reusable. Now this isn’t particularly helpful in this instance, but it really might be in a longer program that calculates quadratic roots in a lot of places. Here’s how we go about doing this (Figure 14):

These are the things to note:

Figure : 'solver' with a function definition

* The function is introduced with the ‘def’ keyword, followed by a function name of some sort, some brackets (which may be empty but must be there) and the ending of the line with a ’:’
* In the brackets are 3 ‘parameters’ (a,b,c in this case). These are variables that have values passed into the function for it to work with. We use these variable names within the body of the function. NOTE. In this case the parameter name are the same as the ones we used outside. BUT they don’t have to be, I could have used x,y,z and just used x,y,z within the function.
* The extent of the code in the function is delimited by being indented (4 spaces, remember?) just like in the if-then and loop.
* The code within the function is basically just as it was before.
* We ‘call’ the function by just using its name, “roots”, and passing the appropriate values in the brackets of the call.

Now this is excellent as we have a block of code that does something useful and we can ‘reuse’ just by calling it. But it could be much more useful. It would be really nice if, rather than just seeing the roots printed out, we could get at the values and do other calculations with them. Well, as it happens, this is easily done by using a ‘return’ statement from within the body of the function

definition: As usual, by example (Figure 15)

A screenshot of a cell phone

Description automatically generated

Figure : 'solver' with a function that returns the roots to the caller

Here are the things to note:

* The couple of lines at the start of the function, delineated by ‘’’ (beginning and end), is another form of comment. This is called a ‘docstring’ and works the same way as any other comment except it starts with one ‘’’ and ends with another. It is possible to automatically create a simple form of program documentation using these docstrings.
* Instead of printing out the roots (or warning) we ‘return’ a list containing the 2 roots.
* If the roots are imaginary, we return ‘None’. This is a special Python object which is, essentially an official way of showing the absence of anything. In other words, we are not returning any roots at all. It is easy to check for this later.
* The calling line is “root\_list = roots(x, y ,z)” This calls the function with the values of the variables x, y and z (which is what the function wants and needs. NOTE they don’t have to have the same names as the function parameters) and assigns the returned object (a list in this case) to the variable ‘root\_list’.
* Finally, we just print it out. We could have printed out the first root by using ‘root\_list[0]’ and the second using ‘root\_list[1]’. Remember lists and how to access them (Section 2.4.1)?

|  |
| --- |
| A warning about passing parameters:  If you pass a simple variable (int, float, string) as a parameter into a function, changes you might make to these inside the function don’t affect the variables you originally pass in.  HOWEVER – if you pass in a list, changes made inside will be reflected outside – use list. Copy()  def changeit(a, b, alist):  print('passed in ', a, b, alist)  a += 1  b += 1  alist.append(99)  print('at end ', a, b, alist)  x = 22  y = 33  my\_list = [1,2,3]  print('before call ', x, y, my\_list)  changeit(x,y,my\_list)  print('after call ', x, y, my\_list)  Gives:  before call 22 33 [1, 2, 3]  passed in 22 33 [1, 2, 3]  at end 23 34 [1, 2, 3, 99]  after call 22 33 [1, 2, 3, 99]  You avoid this by making a copy of the list and working on that:  def new\_changeit(a, b, alist):  local\_list = alist.copy()  …  local\_list.append(99)  … |

|  |
| --- |
| You can, in fact, return more than one value at a time. So, instead of returning a list we could have returned both roots separately:  return root\_1, root\_2  And you access these by assigning the output from the function call to 2 separate variable:  r1, r2 = roots(x, y, z) |

|  |
| --- |
| You can include default values for parameters in your function definition, which means that if you call the function without a particular parameter value the default will be used, but if you do include a value this will override the default. So something like:  my\_func(a, b, c=23):  will expect to have at least 2 parameters when called, and if this is the case will use c=23. However, if called with 3 parameters it will use the third as the value for c. So, by calling it like:  my\_func(3, 6, 12)  the function will use a=3, b=6 and c=12 |

|  |
| --- |
| Python 3 introduced the concept of ‘argument unpacking’ this allows a function with multiple parameters to be called with a ‘pointer’ (indicated by a ‘\*’ symbol) to a list of values. This list is then ‘unpacked’ and assigned to the individual parameters. Easy to see an example:  def my\_func(a, b, c):  Do stuff with a, b and c  my\_list = [1, 2,4]  my\_func(\*my\_list]  you can also pass a tuple in a similar way.  You can pass a dict, where the keys are the same as the parameter names using a ‘\*\*’ prefix:  my\_dict = {‘c’:1, ‘a’:2, ‘b’:4}  my\_func(\*\*my\_dict) |

|  |
| --- |
| **Lambda ‘Functions’**  Python also has a concept of one-line functions called ‘lambda functions’. These allow you so define a simple function with a single line. They are, so called, ‘anonymous’ functions which means, as you might deduce, functions without a name – but you can assign them to an object (variable). They just perform a simple operation and won’t return anything.  They are of the form: lambda <parameter list> : expression. Let’s define a simple lambda function that takes the first parameter it gets and adds this to the square of the second, and we’ll assign this to an object called calc\_it (I know, I know, I’ve got no idea why you’d want to do this but still …)  calc\_it = lambda x,y : x+y\*\*2  Now we could use this like a named function:  calc\_it(2,3)  Which would give a result of 11.  These are often used as ‘throw away’ functions, though, were you only want them at that instant and won’t use them again, so they don’t need a name. Probably easy to use an example. Consider I have a list of tuples:  my\_list = [('a', 26), ('b', 5), ('c', 105), ('d', 67), ('e', 45)]  I want this sorted on the second, numeric, part of the tuple. You could, produce a function to do this but there is no need.  my\_list.sort(key = lambda x : x[1])  Does it in one go, giving a ‘new’ my\_list of:  [('b', 5), ('a', 26), ('e', 45), ('d', 67), ('c', 105)]  As those annoying meercats say ‘simples’ |

Now we’ll look at using someone else’s hard work.

## Modules – reusing someone else’s code (introducing numPy)

So, we’ve written some code to solve quadratic equations, which we could re-use anywhere, and it works quite well but doesn’t handle complex roots in a particularly sophisticated way. Wouldn’t it be nice if someone else had written a better version – and you could ‘borrow’ their code? Well, they have and you can – for free. This is where the concept of importing modules arises.

As already mentioned, Python is an extremely popular programming language and there are huge numbers of people out there using it to do specialist things. Python is managed by the ‘Python Software Foundation’ (PSF, n.d.)which guides its development and offers it as part of the more general Open Source movement. This encourages the development of software to be distributed under a license that allows its use free to all. In that vein, many of the aforementioned specialist have made their code available for free to all.

There are lots of modules written either specifically for Physics/Astronomy and Data Science or more generally for science and technology (as well as Machine learning, A.I., natural language processing, web development … and so on). In SXPS288 we’ll make use of a number of these (NumPy, SciPy, Matplotlib, Pandas, AstroPy, lmfit etc.). The reason being that they make life easy – why re-invent the wheel?

Here we’ll introduce just one of them – numPy (pronounced, apparently, ‘NumPie’ – although I prefer ‘Numpi’, like jumpy (http://www.numpy.org/, n.d.). We’ll do this, as usual, by example – rewriting our solver.

Now it happens that the module NumPy has a quadratic solver – in fact it solves polynomial functions, AND it calculates the complex roots, and it’s called roots() too. To use it we’ll have to ‘import’ it just as we imported ‘math’ way back in our original ‘solver’ (Figure 9). Conveniently, the Anaconda distribution makes NumPy available as standard. By Example (Figure 16):

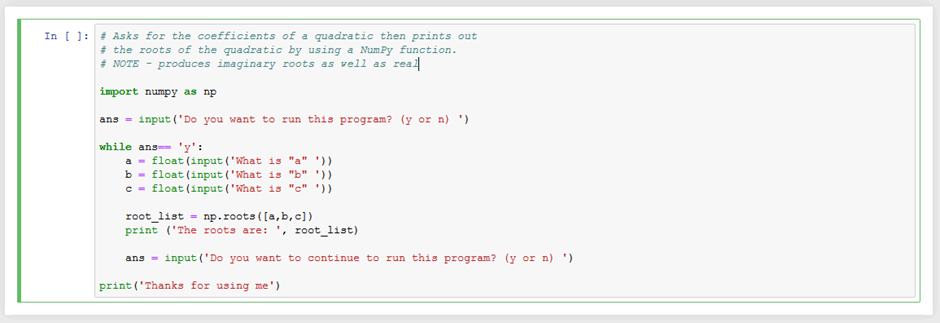


Figure : 'solver' using numPy

You can see how much simpler this is. We just needed to know that numPy existed and that it contained an equation solver. It’s also been really well checked for errors!

Thigs to note:

* We import numPy using ‘import numPy as np’ – it doesn’t have to be ‘np’ but by convention often is. There are other, similar, ways of importing a module, that alter, subtly, the way we call the module functions (see blue box). By doing it this way, we call any numPy function using a ‘np.’ prefix (np.roots()). This may seem a bit long-winded, but it has the advantage that we know we are calling a numPy function and can’t get confused with, say, a function called roots() that we’ve declared somewhere, or any other module’s function called roots().
* We call it, and deal with the function’s return value(s) in much the same way as with our self-declared function. The only real difference is that we have to send the parameter values as a list ([a, b, c,]) rather than as separate values. This just happens to be the way the numPy function has been written and, if you think about it, allows variable numbers of parameter values to be sent and allows the function to determine the ‘order’ of the polynomial you want a solution for.
* You will find that this code will return imaginary roots and also, if you modify it, it will solve polynomial equations of a higher order.

NumPy also has a much-improved version of Python’s list – the numPy ‘array’ and we’ll see that in action in the section on File IO (Section 6 )

|  |
| --- |
| You'll see a number of ways of importing modules. let's use 'math' as an example (and later numPy)  import math  This works fine and if we want to use sine we need to use:  math.sin()  That is, in effect, we need to tell Python where to 'find' the sin() function we want.  If we use:  import math as mt  We can use a slightly shorter form:  mt.sin()  If we're really lazy we could use:  from math import \*  Now we can just use:  sin()  Superficially this seems great - why not use it all the time? The problem comes if we also define a function called sin(), or we import numpy \* as well (also contains a sin() function). Which version should Python use? **\*My advice is always use the 'import math as mt' for as it really does avoid ambiguity.** Of course, for short, sweet and dirty code to just do something quickly then you might use '\*' to save a few seconds.  Finally, if we only want a few of the functions in a module we can call them specifically and this might useful for unique function names that we won't confuse later:  from math import sin, cos, tan |

So, be lazy, don’t reinvent the wheel, let someone else do the code checking.

Moving on, let’s start to extend the functionality you can use in Python.

## Numpy Arrays

In this section we’ll look at numPy arrays. We’ll only touch the surface to give a flavour of them. Later when we look at file input and output, we’ll use some of their functionality in anger. Be aware there is much more to them than we’ll look at here.

NumPy has a lot of useful mathematical functions that generally work more quickly and efficiently than the standard Python ‘math’ library (even if that has and equivalent function, have a look at the documentation (SciPy.org, n.d.)).

It also has a dedicated data structure, that looks and feels like a list, that is ideal for holding and manipulating scientific data – even newer modules like Pandas which you’ll be using are based heavily on numPy arrays. These are n dimensional grids of data. They can be 1D, 2D or higher dimensions. Think of a 2D array as being like a spreadsheet and a 1D array being like a single column or row.

Let’s start with a 1D Array (we’ll assume we’ve used ‘import numpy as np’)

1. x = np.array([1,2,3,4,5,6,7])

 Now we can manipulate its contents:

1. x[4] will return 5
2. np.append(x,99) will return a new array of [1,2,3,4,5,6,7,99]
3. np.delete(x,3) will return a new array of [1,2,3,5,6,7]

NOTE: they don’t modify the exiting array. Also, these are simple forms of the functions. N Dimensional forms will also have modifiers to explicitly define which dimension they apply to.

You can manipulate all the elements of a numpy array at once and the procedure is very quick – much quicker than using loops to do the same for a standard list.

A couple of examples say we have n\_a = [1, 12, 3 ,7,2, 14] and n\_b=[1,2,3,4,5,6]

1. n\_a+1 gives [2,13,4,8,3,15]
2. n\_a + n\_b gives [2,14,9,11,7,20]
3. n\_a \* n\_b gives [1, 24, 9,28,10,84]
4. n\_b\*\*2 gives [1,4,,9,16,25,36]

and so on

1. n\_a.sort gives [1,2,3,7,12,14]

Let’s now look at a 2D array – which as we’ve said is a bit like a spreadsheet sheet.

1. x2d = np.array( [[11, 12, 13, 14, 15],
2. [21, 22, 23, 24, 25],
3. [31, 32, 33, 34, 35]])

This will produce an array with 3 rows of 5 ‘columns’.

NOTE: We’ve split this over 3 lines for clarity, you don’t have to.

You can access a value in them in like this:

1. x2d[2,3] will return 34 - in other words, use x2d[row,column)

 And you can do slicing – obviously a bit more complex, to get a sub-array:

1. x2d [1:3,1:4]

will give:

1. [[22, 23, 24],
2. [32, 33, 34]]

WARNING: If we do something like:

1. x2d\_2 = x2d[1:3,1:4]

x2d\_2 is actually pointing to a portion of the ORIGINAL array, x2d. So changes you make to x2d\_2 will be reflected in x2d. This is actually quite useful if you are working on large data sets as you can access and manipulate subsets of the main data set without having to copy the underlying data – but you can see the opportunities for Gotchas!

To be safe, you can use the copy() function:

1. x2d\_2 = x2d[1:3, 1:4].copy()

# File IO

We’ve seen how we can get the user to type in data from the keyboard. Fine, but what if we have hundreds or thousands of data values, collected from, for example a sky survey, or a single run of the LHC at CERN? Keyboard input will just not cut it and we’ll need to read data in from a file (and then write it out to a file). Or, possibly from a link over the Internet.

To start with we’ll read in a file containing the values we described at the beginning of Section 2.4. We’ll do this from numPy and use numPy arrays. Later in the module we’ll probably be using the ‘Pandas’ module which is often used for reading in scientific data.

You can, of course, read files directly from basic Python, but as we’ll mostly be using numPy (and later Pandas) we’ll concentrate on numPy

## Using numPy to read from local files

For starters let’s assume we have the following comma delimited file (CSV), in the same folder as the Python code, called ‘squares.csv’ and containing the 2 lines of data (x values and y values):

1,2,3,4,5,6,7,8,9,10

1.1,3.9,9.1,16.5,23.9,36,50.1,63.2,82.1,102.3

We can read this is in easily with the numPy function genfromtxt():

1. import numpy as np
2. sq\_data = np.genfromtxt('squares.csv', delimiter=',')

 and if we print sq\_data out we’ll get:

1. [[ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. ]
2. [ 1.1 3.9 9.1 16.5 23.9 36. 50.1 63.2 82.1 102.3]]

 Now, the important thing to notice is it seems to be a list containing 2 lists. This is, in effect, true but the more interesting point is that this is a ‘***numPy array***’ These are very similar to the lists we’ve met already, and you can access the values in them much like lists BUT they have many more useful properties which we’ll come to later.

We can access the first list – which are the x values by just using sq\_data[0] (remember indices start at 0) and the second, y values, by using sq\_data[1]. Now we can, should we want to, separate these into x and y variables – which themselves will be numpy arrays, although of a single dimension.

1. x = sq\_data[0]
2. y = sq\_data[1]

We’ll use something like this later on when we look at plotting graphs.

### NumPy array re-shaping – when data doesn’t fit

Sometimes, your data might not quite be in the form you want it. Let’s say your CSV file is actually in this form – not uncommon:

1,1.1

2,3.9

3,9.1

4,16.5

5,23.9

6,36.0

7,50.1

8,63.2

9,82.1

10,102.3

When we read this in to sq.data we’ll get the following array:

1. [[ 1. 1.1]
2. [ 2. 3.9]
3. [ 3. 9.1]
4. [ 4. 16.5]
5. [ 5. 23.9]
6. [ 6. 36. ]
7. [ 7. 50.1]
8. [ 8. 63.2]
9. [ 9. 82.1]
10. [ 10. 102.3]]

 Now this is not as easy to extract our x and y values. But numPy provides away:

1. sq.data.transpose()

 This does the trick and produces:

1. [[ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. ]
2. [ 1.1 3.9 9.1 16.5 23.9 36. 50.1 63.2 82.1 102.3]]

Which is where we started.

NOTE in the next section on reading files over the Internet (Section 6.2), we’ll see a way of reading data of this form directly into sensible numPy arrays.

|  |
| --- |
| There is another way (although not quite as easy to ‘see’) to get x and y when the columns are arranged like this, and that is to use array slicing and the slightly different way that numPy address 2D arrays.  Consider a numPy array (called, say, my\_array) that looks like this:  [[11, 12, 13, 14,15]  [21, 22, 23, 24,25]  [31, 32, 33, 34, 35]]  We can access the value on the 2nd row and 4th along (13) by using a construct that is pretty much an extension of how we access elements in a list:  my\_array[1,3] will point to the value 24 (as usual index starts at 0)  Using this, we can do without the transpose() function and get x and y in our example above by:  x = sq.data[:,0]  y = sq.data[:,1]  The choice, as they say, is yours1 |

Next, how can we make pretty plots of the data?

## Using numPy to read from the Internet

Often, data files will be located on the internet. Now you could download these and use the techniques we’ve discussed to read in the data. However, it is possible to read directly from the Internet and this may is often the better way. We’ll look at doing this using numPy and, in the next section show how a new module ‘Pandas’ makes things easier.

To illustrate this, we’ve got a form of the ‘squares.csv’ we’ve used before located on an internet server with a URL of 'http://greymamba.uk/Datasets/squares.csv' (NOTE THIS NEEDS TO BE AN OU SERVER REALLY) in the following form

1. x\_values,y\_values
2. 1,1.1
3. 2,3.9
4. 3,9.1
5. 4,16.5
6. 5,23.9
7. 6,36.0
8. 7,50.1
9. 8,63.2
10. 9,82.1
11. 10,102.3

 Note that it has a ’header’ row with data descriptions of the columns (‘x\_values’ and ‘y\_values’) as well as the actual data. We’ll be using these names to actually access the data – which is a neat trick!

There are many ways of reading this data from over the Internet but perhaps the simplest way is to just use the ‘genfromtxt()’ function we’ve already used(Figure 17).

**This approach actually downloads the data to your hard drive – so use it with caution for huge data sets!**

A screenshot of a cell phone

Description automatically generated

Figure : Simple file IO from the Internet

A couple of notes:

* We’ve introduced another parameter ‘names=True’ to the genfromtxt() function. This tells it that the first row contains descriptors of the column data. This will be very useful very shortly!
* The data is in a numPy array with each row being a tuple containing the column values.

Next, we’ll use the column ‘names’ to greatly simplify getting the x values and y values we might need to further process our data (Figure 18). We’ll use this data when we talk about producing graph plots using a module called ‘matplotlib’ (Section 7 ).

A screenshot of a cell phone

Description automatically generated

Figure : File IO from the Internet - using column 'names' to access the data

So, using the instruction ‘sq\_data[‘x\_values’]’ we can directly access the column of data that had a column header of ‘x\_values’

We’ll use this approach later when we look at plotting out our data.

## Introducing Pandas

Pandas is another module that is pretty much dedicated to scientific data IO. Its main data structure is the dataframe – which, essentially, builds extra functionality into a numPy 2D array. Once again, let’s see it at work (Figure 19).

The Notes:

* ‘pd.read\_csv’ reads a csv file into a datframe. The ‘sep=’\s+’ parameter is used because the columns happen to be separated by whitespace characters.
* We’ve printed out some of the first 5 lines of data
* Just for cleanlinest in producing the H-R diagram, we want to drop any data where the star is too far into the ‘red’ (arbitrarily, B-V > 2.0). We do this with the star\_data.drop() function. Note that the statement acts on the whole dataframe.
* We’ve used matplotlim.pyplot.scatter() this time and used scaled ‘dots’ for markers (s=3) along with a colour mapping of blue to red (c=t and cmap=’coolwarm’) – other maps are available or you can roll your own.
* The rest is pretty standard stuff

A screenshot of a social media post

Description automatically generated

Figure : A Hertzsprung–Russell diagram from Hipparcos data - introducing Pandas

Section end

# Graphing and Plotting (Matplotlib)

Graph plots are easily made using another module, ‘matplotlib’ (matplotlib, n.d.). This can be used very simply but with increasing sophistication as you learn the way it works. We’ll start of with a very simple program based on the ideas we developed in the File IO (section 6). As usual, an example (Figure 19) showing the code and its output:

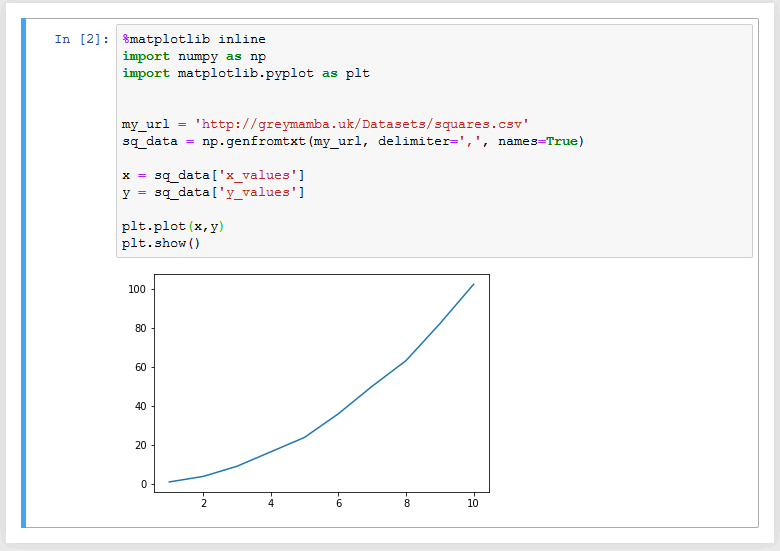


Figure :Matplotlib at its simplest

Things to note:

* The first line is just a controlling statement that indicates where matplotlib plots will appear in Jupyter Notebooks – it’s nothing to do with the Python.
* We import matplotlib.pyplot as plt. NOTE pyplot is a sub section of matplotlib.
* Get 1D NumPy arrays for x and y from the input array (transposed due to its format).
* Now all we have to do is construct the plot (plt.plot(x,y) and make it visible (plt.show()). And that’s all there is to it.

But we can do better, we can add all the kinds of things like titles, axis labels and legends if we want to do so.

As usual the easiest way to see this is with by extending our example (Figure 20).

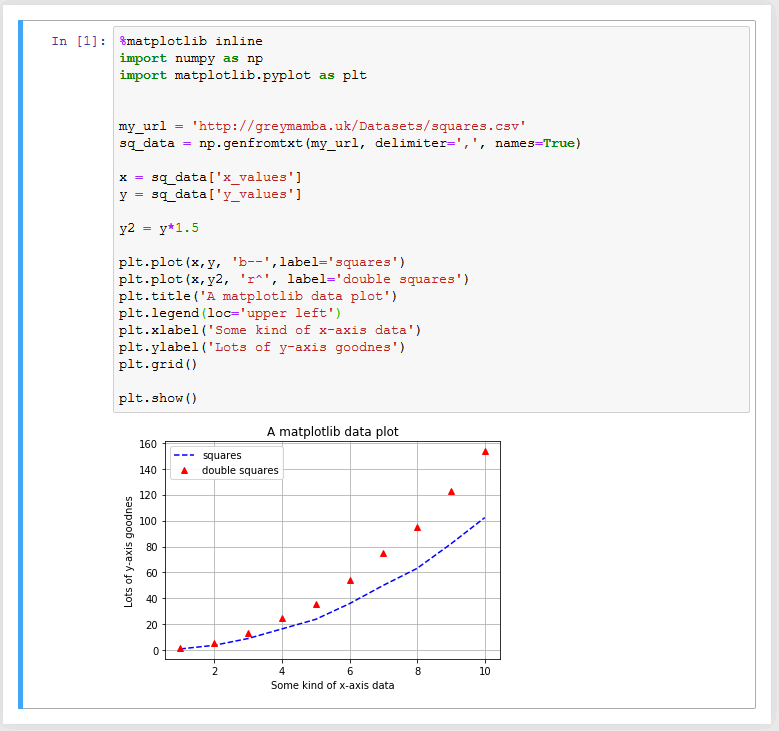


Figure : Matplotlib with Title, axis labels, legend and grid

Here are some notes:

* We’re plotting 2 lines, the original one and one where the y values are 50% larger.
* plt.plot(x,y, 'b--',label='squares') – use a blue dashed line (b--) and provide a label for the legend.
* plt.plot(x,y2, 'r^', label='double squares') – plot red triangles (r^) for the data points.
* plt.legend(loc='upper left') – put the legend here
* The rest is pretty self explanatory!

And sub-plots:

A screenshot of a social media post

Description automatically generated

Figure : Figure with simple sub-plots

There are many more ways of stylising and customising the matplotlib plots, but for the time being we’ll leave it there – except for one final thing. You can also save your plots to files for use in reports (or TMAs). It’s simple:

Just include a line like, plt.savefig(‘figurename.png’) or plt.savefig(‘figurename.pdf’) – you choose the figurename of course (and it can include a full folder path).

Later on you may well want to use another plotting Module called ‘Bokeh’. It has certain advantages over matplotlib, and isn’t as well known or used, – but we’ll come to that later in the Topic.

# References (citations & list need tidying)

http://www.numpy.org/ (n.d.) *NumPy* [Online]. (Accessed 24 March 2019).

Jupyter Notebooks (n.d.) *The Jupyter Notebook — Jupyter Notebook 5.7.6 documentation* [Online]. Available at https://jupyter-notebook.readthedocs.io/en/stable/notebook.html (Accessed 13 March 2019).

matplotlib (n.d.) *Matplotlib: Python plotting — Matplotlib 3.0.3 documentation* [Online]. Available at https://matplotlib.org/ (Accessed 29 March 2019).

PSF (n.d.) *Python Software Foundation* [Online]. Available at https://www.python.org/psf/ (Accessed 24 March 2019b).

Python Docs (n.d.) *Python Docs* [Online]. Available at https://www.python.org/doc/ (Accessed 24 March 2019a).

SciPy.org (n.d.) *Overview — NumPy v1.16 Manual* [Online]. Available at https://docs.scipy.org/doc/numpy-1.16.1/ (Accessed 28 March 2019).

# Bibliography

Kazarinoff, P. (2018) *Problem Solving with Python 3.6 Edition: A beginner’s guide to Python & open-source programming tools*, [Online]. Available at <https://problemsolvingwithpython.com>.

Kinder, J. M. and Nelson, P. (2015) *A Student’s guide to PYTHON for physical modelling*, Princeton and Oxford, Princeton University Press.

Mckinney, W. (2012) *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, Sebastopol, O’Reilly Media.

Saunders, M. (2017) *Coding, Computer programming (beginners onward)*, Yeovil, Haynes Publishing.

Scopatz, A. and Huff, K. D. (2015) *Effective Computation in Physics: Field Guide to Research with Python*, Sebastopol, O’Reilly Media.