2. Python - Variables, objects and methods

The aim of any numerical method we apply is to a use a series of step by step calculations to find a numerical answer to our problem. This step by step process is known as the *algorithm* and the purpose of our programme is to translate these series of steps, in the language of our choice (in this case python), to a form that can be executed by our computer. A key part of our programme is that we need to store the numbers we calculate in *variables* that we can store and use when executing our programme. (A bit like using the memory in a calculator).

In this notebook you will understand and learn how to use:

- the common data types (integers,floats,complex) used in numerical programming for storing variables,
- the methods used to identify and convert between data types,
- expressions with variables with mixed data types,
- the list data type and how to refer to individual elements in the list,
- use basic plotting functions from the Matplotlib package as an example of using lists.

2.1 Basic data types

There are four main data types (we will use others) that we will use in this unit.

2.1.1. Integers.

These can take positive, negative or zero values and are always whole numbers (no decimal points). In python they are defined as 'type int'. The size of an integer number in python is in principle unlimited and dynamically allocated. There a places where we must use them, for instance, in indexing for items in lists as we will see later.

2.1.2 Floating point

As a default these are stored as 'double precision' numbers. These will be sufficient for the calculations we will do in this unit. It is possible to 'declare' floating point numbers with higher precision to avoid rounding errors but this does not always help! If precision is a problem, it is better to think carefully about the algorithm you are using rather than assuming that forcing higher precision in your calculations will fix the problem. In python these variables are defined as 'type float'. To enter float numbers in scientific notation you use the syntax e.g. -2.54 or 3.546e-10

2.1.3 Complex

In python is is possible to do calculations with complex numbers of the format *a+bj* where $j = \sqrt{(-1)}$. The real and imaginary parts of the number are internally stored as two float objects. Provided we declare variables as complex numbers ('type complex') it is straight forward to do calculations using complex arithmetic and we will use this later.

2.1.4 Strings

Strings are used to store characters, for example, 'Hello World', that we used previously. The ability to manipulate and process strings is important, for example in writing *language compilers* but in this unit will be limited to simple tasks, for example, in formatting output. In python they are defined as 'type str'. Note, strings may be written as "string 1" or 'string 2' but the quotes should not be mismatched, i.e. 'string 3" is not allowed.

2.1.5 Other

There are a very large number of other variable types that can be used in python, especially when we use the numpy library. You will generally learn about these as you advance in the language when you have a partcular problem to solve. One that is common but often not noticed is the 'boolean' type with 'True' or 'False' values. We will look in detail at boolean types later.

2.2 More sophisticated data types

There are many more sophisticated data types that may be declared and used in python.

2.2.1 Lists

A list may contain an arbitary series of objects (numbers, strings other objects) and is enclosed in square brackets e.g. [1, 2.3, "Hello World]. We can mix data types in our lists (i.e. floats, strings etc.). They are a powerful way of handling data but by and large we will limit our use of lists to lists of numbers that we can use for vector calculations and some lists of number lists that we will use for matrix operations. If you are used to other languages you may be more familair with these as arrays. We will see how we declare and use lists as we go along.

2.2.2 Tuples

These may often be confused with lists as they look similar but are enclosed by round brackets e.g. (1, 2.3 ,"Hello World"). However, the values in a tuple may not be changed and are fixed when the programme runs. Attempting to use round rather than square brackets for a list is easily done and can lead to frustrating error messages.

2.2.3 Other sophisticated data types.

In python its is possible to create a whole hierarchies of collections of objects. For long programs and properly *object oriented programming* these are invaluable for creating clear and reliable code. We will not be using them here but the 'set' and 'dictionary' types are examples if you wish to investigate further.

2.3 Declaring and using variables

2.3.1 Declaring variable types

In other languages. such as C or Fortran, it is necessary or usual to 'declare' variable types before they are used. This is known as strong typing and makes sure the type of any variable used in the programme is clear. This is not the case in python and the data type is usually defined when the variable is first assigned. Look at and run the code fragment below.

In [ ]:

```
a = 10
b = 11.8
c = 3.24+6.5j
d = 'help'
print("the type of a is ",type(a))
print(type(b))
print(type(c))
print(type(d))
print(a)
print(b)
print(c)
print(d)
a=11.2
print(type(a))
```

In this example the type() function returns the type of the variable in the enclosed brackets. Here you can see that 'a' is recognised as an integer as there is no decimal point when it was first used, 'b' is a float as it contains a decimal point, 'c' is a complex number as it contains a 'j' and 'd' is a string as its values is given between quotes. Note, however, that the type of variable a was not fixed. The minute we assigned it to float number its type changed!

Exercise. You may like to modify the print statements in the code above to make the output a little clearer (as for example in the first print statement).

If you are unfamiliar with computer programming you may find the use of '=' a bit strange. In python and other programming languages this does not mean a mathematical equality but should rather be interpreted as 'a' is assigned the value of 10 (in this case).

To see how this works run the code fragment below.

In [ ]:

```
a=10
print(a)
a=a+5
print(a)
a+=1
print(a)
a+=3.5
print(a)
a-=2
print(a)
```

In this case the line 'a=a+5' does not make mathematical sense. However, it is better understood as 'take the value stored in the variable a, add 5 to it, and then assign the result to the variable a. Hopefully the result from this example should show you how it works.

The += operator looks confusing and you don't have to use it if don't want to.

a += *n* is translated as take the variable a and increment it by _n_.

it is equivalent to *a* = *a* +*n*

The related operator *a -= n* does the same except the variable _a_ is decremented by _n_. it is equivalent to *a = a - n*

Did you note where the type of the variable _a_ changed in the above code? Beware and try not to overwrite data types - there can be unforeseen consequencces, but fortunately not here.

2.3.2 Changing variable types

The data type of variables can be changed explicitly using the int(), float(), complex() and str(). At least when we do this we are taken a deliberate action. For example look at the following code fragment to see how the types may be changed.

In [ ]:

```
a=10
b=print(complex(a))
c=float('3.3453')
print(c)
print(int(c))
d=complex("2.3+5.2j")
print(d)
```

However, we clearly can't convert a complex variable to an integer or float type. We can however find the real and imaginary parts of a complex number as follows.

In [ ]:

```
a=34.5+2.3j
print(a.real,a.imag)
```

You can find more of the objects and methods by using the the dir() function (see notebook 1) on our variable _a_. See if you can find the simple way of finding the complex conjugate of _a_ (if you can't see'll see it shortly). You can also see all the methods that may be appled to complex numbers, for example, find the method for finding its absolute value. If you wish, type *dir(a)* into the following cell to find the methods and attributes associated with _a_.

In [ ]:

```

```

2.3.3 Calculations using mixed data types

Run and look carefully at the following code fragment below (you will get an error!)

In [ ]:

```
a=10
b=3
c=2.345
d=3.2+3.4j
e= 2.1*1.4j
f='1.234'
print(a/b)
print(a//b)
print (c*d)
print (d*e)
print (d/e)
print (a*c)
print (d*d.conjugate())
print (c*float(f))
print (c*f)
```

There are a few points to note.

In mixed arithmetic a calculation that includes a complex type will always return a complex answer and a calulation that involves a float and an integer will always return a float answer.

The case of integer division is interesting. In many languages a division involving two integers would result in an integer (rounded) value (indeed this is what happened in the older version of python (version 2 that is now obsolete)!). In python (version 3) it results in a variable of float type. To maintain purely integer arithmetic you must use the integer division operator // as shown. In contrast multiplying two integers returns an integer result.

WARNING. As noted above, old code written in Python 2, will behave differently or may not even run in Python 3. If you are using your own installation of python make sure you are using python 3.

Unless a number, stored as a string, is converted into a numeric data type (with int(),float() or complex()) python will produce an error, it will not try to do the conversion automatically. Note in the above how the line number in which the error occurs is shown. Note however, the error message is a little obscure. A quick google search however does give better explanation. Try searching google, or other, for this error message and see what you find out. You may wish to correct this error before you proceed.

Exercise. In the code cell below you should experiment with different variable types, how you convert them and using it to test some of the basic python maths operators. You may find you need to restart the kernel and clear the output if you start to find errors you don't understand.

In [ ]:

```

```

2.4 An introduction to lists.

The list is a sophisticated data type that we will often use in this unit. It is therefore useful to get the basic idea of using them early on. We will do this in particular by seeing how we use the contents of lists to plot data.

Look at the following code fragment (I recommend you restart the kernel and clear output before you do this. If you do make sure you return to this point before executing any code).

In [ ]:

```
a=[1,2,3,4,5,6,7,8,9,10]
print (a)
print(type(a))
print(a[4])
b=[3,4,2,1,8,7,10,5,6,9]
print(b)
print(b[4])
```

Note that in both cases we have created a list that contains the numbers 1-10. Note, how the order in which they occur in the list is also preserved when we print the list. We can also see the method by which we can refer to an individual element of the list i.e. a[4] and b[4].

Note. the index for an element in an array must be an integer. This is a good reason for using integer variables where approrpiate! In the above code, try using a float number as an index (i.e. *b[4.5]* and observe the error you get.

IMPORTANT . A common cause of error and misunderstanding when using lists is the element to which the index number (the one in square brackets) refers. As can be seen, in the code fragment above a[4] corresponds to 5 and b[4] refers to 8, both of which are the 5th items in the list. Why? The index number of lists starts from 0 and not 1. Hence the elements in the list are indexed a[0],a[1] ... a[9]. (The convention is not the same in all languages - beware!)

We can refer to and operate on invidual items in the list using the appropriate index. Look at the code fragments below and inspect the answers. Make sure that you understand how the new values in 'a' have been obtained.

In [ ]:

```
a=[1,2,3,4,5,6,7,8,9,10]
a[3]=a[3]**2
a[1]=a[2]*a[8]
print (a)
```

There are several standard methods that can be used with the list type. For example, the following code fragment sorts in ascending order and then reverses the order of the elements in the list a.

In [ ]:

```
a.sort()
print(a)
a.reverse()
print(a)
```

If you would quickly like to see the methods and attributes for a list type *dir(a)* in the code cell below and run.

In [ ]:

```

```

2.5 An example of using lists - plotting data using matplotlib

A common task in numerical modelling is to plot the results of calculations. it may even involve some input of experimental data for plotting as well. In python there is a very sophisticated library for plotting data. The library is extensive and can produce plots of quality sufficient for scientific publication. Full details may be found at https://matplotlib.org (https://matplotlib.org) along with plenty of examples and tutorials for producing a wide range of types of graph. Fortunately, even given its versatility, it is still straightforward to produce some simple plots. Here we will priduce some very simple plots using lists. To import the packages we require we must include the following at the start of our code.

import matplotlib.pyplot as plt

This only imports the 'pyplot' part of matplotlib. Look at and run the code fragment below.

In [ ]:

```
import matplotlib.pyplot as plt
data=[1,2,3,4,5]
plt.plot(data)
plt.show()
```

Here we have a simple set of data. If this is the only list 'passed' to the plot function it will assume it is y-data and the corresponding x values are [0, 1, ..4] (remember our indexes start from zero). As you can see the default plot is a line connecting the data points (not obvious from this graph). We can change the colour of the line by using

plt.plot(data,"r") (red)

plt.plot(data,"g") (green)

try modifying the code above to chnage the colour of the line. (Other colours are available - look them up).

Run the code fragment below

In [ ]:

```
import matplotlib.pyplot as plt
data=[1,2,3,4,5]
plt.plot(data,"ro")
plt.show()
```

As you can see the additional parameter passed to the plot function means the data is now plotted as points rather than as a connected line. In this case red circles. In the two code fragments above try changing the values in the 'data' in the list and observe the change in the graph. You should also try changing the point plotting. Instead of "ro", try "go","b+","b*","cx". You may wish to look at the the Matplotlib web site to find more symbol types and colours.

If we wish to plot x-y data we need to supply two lists one list with the x-values and one with the the y-values. It is important that each list has exactly the same number of items and that there is a one-to-one correspondence between the elements in lists. i.e the 5th data point is xval[4], yval[4] if the the lists xval and yval that contain the data.

Run the code fragment below

In [ ]:

```
import matplotlib.pyplot as plt
xval=[1.3,1.9,3.2,3.9,5.2]
yval=[2.3, 3.4, 5.6, 6.3, 9.4]
plt.plot(xval,yval,"ro")
plt.plot(xval,yval)
plt.show()
```

Here we have introduced two lists, one with x values, one with y values. We have plotted the data as red data points and linked them together with straight lines. Try changing the colour of the line and the points.

The list of x values does not need to be numerical. In the example below our list of x values refers to fruit and the yvals the histogram value for each fruit. Note how we have changed the type of plot to 'bar'.

In [ ]:

```
import matplotlib.pyplot as plt
xval=["apples","pears","plums","oranges","lemons"]
yval=[2.3, 3.4, 5.6, 3.3, 9.4]
plt.bar(xval,yval)
plt.show()
```

As you see once you have the data in appropriate lists it is easy to produce different plot types. You may wish to play around with the code fragments above to change the way the data is plotted. What we have produced here is a very tiny fraction of the possible ways of plotting data using Matplotlib. We will come across others as we go through the course but you can always refer to https://matplotlib.org (https://matplotlib.org) for more examples.

Look at and run the code fragment below. There are several things you might note. We have controlled the appearance of the graph (labels, titles etc.) by setting further attributes with the plt methods. We have set the axes range using the *plt.axis* method and we have drawn a grid on our graph with the *plt.grid* method. For the latter note how we have used the boolean value 'True'. You may wish to edit the code fragment to test the effect of changing some of these attributes.

You may also note that we have imported the math package to give us the constant π and the $\sin$ function. This is a very clumsy and laborious way of plotting the $\sin$ function and we will see better methods later. However, it does show how you may use the index of the array x to generate the y values at each point.

In [ ]:

```
import matplotlib.pyplot as plt
from math import *
x=[0, 0.2*pi,0.4*pi,0.6*pi,0.8*pi,pi,1.2*pi,1.4*pi,1.6*pi,1.8*pi,2*pi]
y=[sin(x[0]),sin(x[1]),sin(x[2]),sin(x[3]),sin(x[4]),sin(x[5]),sin(x[6]),sin(x[7]),sin(x[8]
plt.plot(x,y,"*")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.title("A simple plot of sin(x)")
plt.axis([0,2*pi,-1.2,1.2])
plt.grid(True)
plt.show()
```

We may wish to save the plot to an image file that we might incorporate into other electronic documents. To do this use the *savefig* pyplot attribute as shown in the code below. Note, how the plt.savefig() function replaces the plt.show() in the previous example. There is much more that can be done with the savefig function (change file formats for example, e.g. "test.pdf") but this is enough for what we will be doing.

In [ ]:

```
plt.plot(x,y,"*")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.title("A simple plot of sin(x)")
plt.axis([0,2*pi,-1.2,1.2])
plt.grid(True)
plt.savefig("test.jpg")
```

2.6 Summary

in this notebook you have been introduced to:

- the different data types that may assigned to variables in your programmes,
- seen how you can identify and use different data types,
- the results that are obtained when using mixed type calculations,
- the concept of a list in python and how to reference different elements in a list,
- the Matplotlib plotting package and how plots may be created from data in lists.

Exercise

Go to the web site: https://matplotlibguide.readthedocs.io/en/latest/Matplotlib/types.html (https://matplotlibguide.readthedocs.io/en/latest/Matplotlib/types.html)

This is a more detailed tutorial on the possible plot types in matplotlib (including pie charts, polar plots etc.). You may need to wait a little to understand all of the python code shown but in the meantime see if you can include a simple example of each type of plot in to this notebook, or if you prefer, a new notebook. You may find this a useful reference for plotting in future courses, and of course, you can add more hints and tips of your own as you get more experienced.