Lab1_Notebook

September 26, 2020

1 Lab 1: Introduction to Python: Basics, Variables and Functions

1.1 Python, what and why?

Python is an object-oriented, high-level programming language. It is relatively simple, so it is easy to learn since it requires a unique syntax that focuses on readability. Python can be used to process text, display numbers or images, solve scientific equations, and save data. The idea of computer algebra systems in general is to try to automate the process of mathematical manipulations. Advanced computer algebra systems such as Python have become very powerful tools for simplifying the process of algebra and calculus but can still need the help of the user to get results in a recognizable form.

As indicated above, there are several different ways of running Python code. This tutorial contains many snippets of code. Since you are viewing this tutorial using the Jupyter notebook, you will be able to run, and re-run, the code snippets within Jupyter, modifying and testing as you go.

To launch Jupyter Notebook, first launch the Anaconda Navigator and then click on the appropriate button for Jupyter. Either the Jupyter Lab or the Jupyter Notebook will work. Jupyter Lab has some nice functionality for handling files, but Jupyter tends to work more smoothly.

A more permanent approach to coding is to write code into a file (a Python script). You can do this within "developer environments" like Spyder (which is included in Anaconda). In this course we will use Jupyter exclusively, but all the coding you do in Jupyter can be converted into a single python code and run within Spyder. The Python you learn in this course can be used in any Python coding environment.

Let us proceed with this Lab 1 Jupyter notebook. Whenever you come a across a "coding" cell (which has `In [ ]` next to it, you can run that cell using the "Run" button at the top of the screen. You can also make modifications to see what will happen, and run it again! You should do this - it is how you will learn !

Now please work though the notebook below.

1.2 Getting Help

Python has extensive help files and tutorials which ought to be your first port of call when you run into difficulties. If you don't understand a Python procedure, for example, `list`, you can search the Python help files or access them directly from the Python terminal. This will call the built-in

Python help system. The page will be printed on the terminal, and if no argument is passed, Python's help utility starts on the console.

[ ]:
```
help(list)
```

1.3 The python notebook

At its very simplest, the Python notebook can be used as an "interpreter" and you can put one command in each cell. For example - it can be used as an advanced calculator.

The interpreter acts as a simple calculator: You can type an expression at it and it will write the value. Expression syntax is straightforward: the operators $+ - * /$ work just like in most other languages.

To get a power, like x^n, type $x ** n$. The integer numbers (e.g. 2, 4, 20) have type `int` and numbers with a fractional part (e.g. 5.0, 1.6) have type `float`.

Run these following cells.....(and do this for all coding cells that you see)

[ ]:
```
3*4
```

[ ]:
```
5/6
```

[ ]:
```
3**3
```

The result of the the last computation is stored in the _ symbol, for example:

[ ]:
```
_
```

[ ]:
```
_ + 5
```

The `math` library has many functions that are useful for programs that need to perform mathematical operations, that cannot be accomplished using the built in operators. These include trigonometric functions, mathematical constants and operators. Once you have imported the `math` library (you only have to do it once) it can be used at any point in the code that follows, or in the following cells in a Python notebook. Import the `math` library as follows

[ ]:
```
import math
```

[ ]:
```
math.pi
```

[ ]:
```
math.sqrt(9)
```

Python has a wide range of predefined functions we can work with. The more familiar ones are tabulated below as well as some examples of their usage.

Function Name	Python Command	Notes
Square Root	`math.sqrt`	
Sine	`math.sin`	Uses radians

Function Name	Python Command	Notes
Cosine	`math.cos`	Uses radians
Tangent	`math.tan`	Uses radians
Inverse sine	`math.asin`	Returns radians
Inverse cosine	`math.acos`	Returns radians
Inverse tangent	`math.atan`	Returns radians
Hyperbolic sine	`math.sinh`	
Hyperbolic cosine	`math.cosh`	
Hyperbolic tangent	`math.tanh`	
Exponential	`math.exp`	
Natural logarithm	`math.log`	Log with base e
Base 10 logarithm	`math.log10`	
Floor function	`math.floor`	Integer part

[ ]:
```
math.sqrt(2)
```

[ ]:
```
math.sin(math.pi/4)
```

[ ]:
```
math.asin(0)
```

[ ]:
```
math.floor(math.pi)
```

2 Variables in Python

2.1 Working with Variables

Computers are machines for manipulating data, so storing, and giving meaning to, data is a very important part of programming. As programmers, we need to be able to say "this bunch of 1's and 0's is a whole number (integer)", or "this other bunch of 1's and 0's is an image". The way this is done in programming is variables. A variable is a box, stored in memory, which has the following attributes:

- **Name**, a label we can use to refer to the variable
- **Type**, which describes what sort of data the box stores
- **Value**, the actual data which is in the box at any given point

Variables in programming have some things in common with variables in mathematics: both have a name or label (like "x"), and both have a type. In mathematics a variable type might be a number or a matrix or a set or something more complicated, and the same applies in programming. There are however some important differences which you should keep in mind. The most obvious one is that variables in programming can change value.

We can assign a name to a value using the `=` operator, for example `a=1` will define a variable called `a` to have value 1, the value held by `a` can then be manipulated and modified and combined with other variables in calculations…

```
[ ]: a=1
```

```
[ ]: b=2
```

```
[ ]: a+b
```

```
[ ]: b**4
```

Assignments can be done on more than one variable simultaneously on the same line

```
[ ]: a,b = 4,5
```

```
[ ]: a*b
```

If a variable is not defined (assigned a value), trying to use it will give you an error.

```
[ ]: a*n
```

2.2 Variable Names

In mathematics, it's common to use single letters for variables like "x" and "y". You can do this in programming, but the result is not very understandable: you will end up with programs full of variables called "x", "y", "a", "b", etc. and have no idea what the variables mean. Much better is to give your variables descriptive names, so that it's obvious from the name what the variable is.

For example, say you wanted to write a program to calculate the orbit of a planet around a star. You could use "m" for the mass of the planet, "M" for the star, "v" for the velocity of the planet, and "V" for the velocity of the star, but then later when you try to understand what your program is doing it may not be obvious what variables correspond to the planet, and which to the star. Things will get much worse as you then try to add more planets to the program. Instead, it's better to give variables names like "star_mass", and "star_velocity", "planet_mass" and "planet_velocity", making it obvious what your variables represent.

There are some restrictions on what you can call your variables :
- Only use letters (upper or lower case), numbers, and underscore '_'

Note: Upper case (capital) letters are different from lower case letters: the variable "planet_mass" is different to "Planet_Mass", and "x" is not the same variable as "X"
- No spaces, tabs, or special symbols like +, -, *, /, $, %, & etc.
- Names must start with a letter or an underscore, not a number

2.3 Variable Types

Every variable in programming has a type, which describes what sort of data the variable contains. Some basic types you will come across are
- **Integers** which describe whole numbers (positive or negative)
- **Real numbers**, often called floating point numbers, which have a decimal point. This also

includes numbers written in scientific form like 1.602e-19 where the "e" means "times ten to the"
- **Strings**, which just means a string of characters i.e. some text.

It might seem strange that integers and real numbers are different types. This is due to the way real numbers are stored in computers, but also because they are used for quite different purposes. An integer is the answer to a "How many" question, and are used for counting. Real number are answers to "How much" questions, and are used for storing quantities which can have fractional values. One reason for this is that integers have a "next" number: the next integer after 9 is 10. Real numbers do not have a next number: what's the next number after 9.21? It could be 9.22, 9.211 or 9.2100000001 and so isn't defined.

In Python, you don't have to say what the type of your variables is (unlike many other languages including Fortran). Instead, the type of your variables is inferred automatically from the value it's given. For example :

[ ]:
```
 x = 2
type(x)
```

[ ]:
```
y = 4.5
type(y)
```

Python has figured out that x should be an integer, because it was set to the value 2, and y should be a floating point number because of the decimal point. You can use the `int( )` and `float( )` functions to change from one type to the other :

[ ]:
```
x = 3
print(x)
```

[ ]:
```
y = float(x)
print(y)
print(int(y/2))
```

In this last example, y/2 is 1.5, but `int( )` converts this to an integer, so everything after the decimal point is lost.

2.4 Our first codes and the print( ) function

Up to this point we have been using the python notebook as an "interpreter" - i.e. performing a single command in a cell which can give a single output. The cells above have multiple lines and are now actually snippets of python code (i.e. programs). Outputs from the pieces of code you write should be handled through the `print` function.

In the brackets after `print` you can output several items, separated by commas, and can combine text and variables…

[ ]:
```
print("The value of x is",x,"and y is",y,"and the integer part of y/2 is",int(y/
↪2))
```

2.5 The "=" sign

The equals sign in programming has a specific meaning - and it signifies an "assignment statement". It is the method through which values are assigned to variables and, importantly, those values can be overwritten…

For example, in programming it makes sense to write something like this:

[ ]:
```
x = 2
x = x + 1
print(x)
```

whereas in mathematics this would not make much sense. This is because the equals sign in programming has a different meaning. In maths, equals means "The left and right sides are equal", but in programming an equals sign means "Perform the calculation on the right, and put the result into the variable on the left". The command "x = x + 1" therefore means "take the value of x, add 1 to it, and put the result back into the variable x".

2.6 User input

It is possible to ask the user to provide the value for a variable. The `input( )` command is used and, by default, it assumes that you input is text (i.e. a "string") and not numbers. If you wish to input a number as an integer or a float, use the `int()` or `float()` command. For example :

[ ]:
```
x = input("Please enter a value for x : ")
print("the value of x is ",x," and the type of variable is ", type(x))
```

[ ]:
```
y = int(input("Please enter a integer value for y : "))
print("the value of y is ",y," and the type of variable is ", type(y))
```

[ ]:
```
z = float(input("Please enter a value for z : "))
print("the value of z is ",z," and the type of variable is ", type(z))
```

So, y is now an integer variable, z is a floating point variable. Now try:

[ ]:
```
answer1 = y + 6
answer2 = y + z
print('answer1 is', answer1, 'answer2 is', answer2)
```

Here answer1 is an integer variable, since it is the result of adding two integers together. However, answer2 is a float, since we have added and integer and a float (if you involve any floating point number in a calculation, or do any division at all, the answer will be a float).

3 Functions

3.1 Introduction to functions

There is an introduction to functions in the Lab 1b video - make sure you watch this. As functions are extremely useful and will be used often in your Python codes this year, it is important that you understand the basics.

We often want to define our own functions and treat them as we would the pre-defined functions in Python. We can think of this function as a seperate piece of code that does a specific task, given a specific set of inputs. In computing this is known as a "black box" - since you do not need to know the inner workings of this code (i.e. it can be "opaque"); you simply need to know what it does and what input it needs to do it. Indeed, we happily use (call) the math functions in Python like `math.sin()`, `math.cos()` or other functions like `type()`, `float()`, `print()`, etc, without needing to know how these functions work. If we put some things into this box (i.e. pass inputs/arguments into the function), the box does its job and you can take something completely different out of the box at the end (return an output).

However, in order to use this function, we need to call it, otherwise it will just hang about doing nothing! In programming terms, we call a function by typing the function name, specifying the inputs we want to pass into the function between two parentheses.

Using functions in Python is a great way to reduce the complexity of our programs, and it does so by breaking code into blocks that do specific tasks. This is one of the most powerful ideas in computing. Functions allow us to wrap up a fragment of code so that we can reuse it again and again without having to type it out every time.

Say we have an equation, and want to calculate some values, say the equation

$$y = (x-1)^2 + \frac{1}{x-1}$$

We want to calculate this for x = -1, -0., 0, and 2. We could write a program like this:

[ ]:
```
print ((-2 - 1)**2 + 1. / (-2 - 1))
print ((-1 - 1)**2 + 1. / (-1 - 1))
print ((0 - 1)**2 + 1. / (0 - 1))
print ((2 - 1)**2 + 1. / (2 - 1))
```

but this is time-consuming, and it's quite easy to make a mistake like forgetting to change one of the numbers. Better would be to use a variable in the formula, and just change the variable each time:

[ ]:
```
x = -2
print ((x - 1)**2 + 1. / (x - 1))
x = -1
print ((x - 1)**2 + 1. / (x - 1))
x = 0
print ((x - 1)**2 + 1. / (x - 1))
x = 2
```

```
print ((x - 1)**2 + 1. / (x - 1))
```

This is better, and is less likely to have an error in it, but still involves us copying and pasting a chunk of code each time. If we wanted to use this many times it would quickly get tedious. This is where functions come in. If you find yourself copying and pasting chunks of code, you probably need a function.

Now look at this piece of code:

[ ]:
```
def calc(x):
    return (x - 1)**2 + 1. / (x - 1)
print (calc(-2))
print (calc(-1))
print (calc(0))
print (calc(2))
```

Here we use the special word `def` to define a function. We give it a name (here 'calc'), and a list of input variable names (only one in this case, called 'x'). The code that is inside this function is indented fro the rest of the code. The code inside the function is run every time this function is used ("called"). At the end of the function it "returns" a value, which is the result of the function. In this case, there is only one line of code, which calculates the value of $(x-1)^2 + \frac{1}{(x-1)}$ and returns it.

[ ]:
```
import math
print(math.sin(math.pi/2))
```

We just need to call it with an input, and it will return the result to us. We can therefore write a function once, or use a function someone else has written, and re-use it in many places.

3.2 Function inputs

The inputs to a function (known as "arguments") are between the round brackets after the function name e.g.

`def someFunction():` # No inputs
code

`def anotherfunction(x):` # One input, called 'x'
code

`def yetanother(x, y):` # Two inputs, 'x' and 'y'
code

[ ]:
```
def hello():
    print('hello there!')
```

[ ]:
```
hello()
```

Notice here we have put the function definition in its own cell in the notebook, and we can run that first. Once it is defined, we can use it in the following cells.

When a function is used ("called"), its inputs are first calculated, then they are matched to the input variables based on their position in the list. For example:

[ ]:
```
def power(x,y):
    return x**y
a = 2.0
print(power(a, 2*a))
```

When Python sees this, it does the following:

1. The value of the inputs 'a' and '2∗a' are calculated turning "power(a, 2∗a)" into "power(2.0, 4.0)"
2. Inside the power function, the variables 'x' and 'y' are created. Because 'x' comes first, it's given the value of the first input (2.0), and because 'y' is second it's given the value of the second input (4.0)
3. These input variables 'x' and 'y' can be used to perform calculations, but only exist inside the power function: they are "local".

3.3 Function outputs

When a function has finished, it can send back results ("return") to wherever they were called from. In the "power" function above only one value was calculated (x ** y) and returned. In general though a function can return in more than one way different ways. For example:

[ ]:
```
def maximum(a, b):
    if a > b:
        return a
    else:
        return b
```

(We will learn about "if" statements in the next lab session!) This function can return in two ways. It first checks if `a` is greater than `b`, and if so returns the value of `a` which finishes the function. If `a` is not greater than `b`, then the function skips to the next line which returns the value of `b`. The result is that it always returns the maximum of `a` and `b`. This function can then be used e.g.

Note - a function will only execute the `return` statement once. As soon as a `return` statement is encountered, it will return this value and exit the function

[ ]:
```
print(maximum(2,3)) # Result 3
x = 2
y = 1
print(maximum(x, y)) # Result 2
print(maximum(2, maximum(3,1) )) # Result 3
```

This last example uses the result of one 'maximum' call as input to another. This is because first the value of the inputs is evaluated, so `maximum(3,1)` is calculated and gives result 3. Then `maximum(2,3)` is calculated, giving result 3. It's the same as this code:

What if we need a function to produce more than one output? In Python, just as a function can have more than one input, a function can also return more than one result:

[ ]:
```
def reorder(a, b):
        if a > b:
                return b, a
        else:
                return a, b
```

This function `reorder` now returns one of two results, depending on the order of the values: This could be used to do something like:

[ ]:
```
a = 10
b = 2
a, b = reorder(a, b) # make sure b >= a

print(a,b)
```

The first value returned has now been put into `a`, and the second value has been put into `b`.

4 Random Numbers

For many applications in the physical sciences we want to use random numbers. In reality, of course, a computer can't ever produce a truly random number because the methods available to the computer are all deterministic but methods are available to produce $psuedo-random$ numbers which are to all intents and purposes indistinguishable from truly random numbers.

The simplest way to produce a random integer in Python is using the `random` function. This generates a random number between 0 and 1. The `uniform` and `randint` commands can also be made to return a procedure which returns random numbers or integers from a particular range.

[ ]:
```
from random import *
random()
```

If you try these commands you will get a different result because `random` will produce a different number every time.

It is also possible to get Python to choose between the items in a list.

[ ]:
```
random()
```

[ ]:
```
uniform(1,10)
```

[ ]:
```
randint(1,10)
```

[ ]:
```
randint(1,20)
```

[ ]:
```
choice('abcdefghij')
```

```
[ ]: choice('abcdefghij')
```

```
[ ]: sample([1,2,3,4,5,6,7,8,9],5)
```

5 Lab 1 Tasks (complete these in the lab session)

5.1 Task 1 : Function Examples

Download and run the python notebook Lab1_Function_Examples.ipynb.

There should be a series of things written out to the Python Shell. Make sure you understand every part of this script. Play with some of the arguments to the functions. If you don't understand anything, please ask one of the demonstrators… that's what they're there for!

5.2 Task 2 : Calculating your weight on different planets

Download and run the python notebook Lab1_Planets.ipynb.

Your task is to create a function called massPlanet that calculates and returns (give as an output) the expected reading on a set of bathroom scales if a 60 kg person were to stand on the scales whilst stood on any planet, star or moon.

You are given some example code to help you. Your task is to complete the code in the last two cells of the notebook.

The solution is provided at the bottom of the VLE page. Get a demonstratr to help you if you are stuck, and show the demonstrator when you have completed the task.

5.3 Task 3 (Optional - if you have time!): Mike's restaurant bill and the tipping function

Download and run the python notebook Lab1_Tipping.ipynb

This starting code for this exercise can be found in Tipping.py. Here, you need to make two functions that calculates how much Mikes restaurant bill is (he's in the USA) after adding a tip onto the original value, and having accounted for tax. Read the instructions in the notebook carefully.

Again you are given a starter code.

The solution is provided at the bottom of the VLE page. Get a demonstratr to help you if you are stuck, and show the demonstrator when you have completed the task.

6 Assignment 1 : Random number generator

You should now complete the Lab 1 Assignment (see VLE Page). You can start this during the lab session if you like and get some help from the demonstrators if needed.

[ ]: