Functions in Python

A function in Python is the equivalent of a mathematical function: an object that take as an input some data, process it, and provides an output. In this regard, although there is an injective relation between input and output, that is, for any input you can only have one output, the output itself does not have to be a single object, but can be many (however, this sequence of objects is the same, given the same input).

Let us first look at the general declaration of a function and then make a simple example to clarify the meaning of its different parts. So, here is the general form:

```
def nameOfFunction( argument1, argument2, ..., keywordArgument1 = X, keyword
Argument2 = Y, ...):
    '''A description of how the function works. Typically, this would includ
e the meaning of the
    various argument, and their data type, as well as their output (and its
 data type), all
    included between inverted commas like a normal string
    '''

    Command1
    Command2
    ...

    return output1, output2 (,...or more than two!)
```

Note that a function that returns 2 outputs must be given two variables to store their value, so the typical use in a Python code would be:

```
var1, var2 = nameOfFunction( arg1, arg2, ..., keyArg1 = X, keyArg2 = Y, ...)
```

The example above illustrates the general structure of a function:

1 - It starts with the protected word `def` , then comes the name of the function, followed by parenthesis that contains the input (so-called arguments) of the function, followed by a the sign `:` .

Note that there are two types of input. Normal arguments, which have no default value and must be provided first. These can then be followed by *keyword arguments* (only if necessary!) to which a default value **must** also be assigned. When later using a function, normal arguments must be provided, whereas for keyword arguments the rule is that if it they are provided, their input value will overwrite the default value. If not, the keyword argument will take the default value given in the function declaration.

2 - Right after the function, in between inverted commas and indented with respect to the `def` protected word, one can usually find a short description of the function.

This description is not necessary, **but it is good practice to put it so that others can see and quickly understand what this function does**. It constitutes the so-called *documentation* of the function, and in order to see it you just need to type the following command: `print( name_of_function.__doc__ )` or `help( name_of_function )` .

3 - Following the documentation, one finds the **body of the function**, where a series of operations is given (whatever you wish the function to do!). Note that, as for the `if` or `for` and `while` statements, the body of the function **must be indented with respect to the protected word `def`** .

4 - Finally, one can notice that the function ends with a `return` statement. After `return`, typically one gives the value of the outputs that the function will return (separated by commas).

Note that providing a `return` value is not really necessary. In fact, sometime the only reason to call a function is to perform some operation on its input and nothing more, in which case `return` is not followed by anything.

An explicit example

To make an example, let us define a function called `average` that takes two numbers as input and returns both their arithmetic and geometric mean. However, if a third number is provided, it takes the mean and average over these three numbers.

```
def average( a, b, c = None ):
    '''This function takes two numbers a and b and returns their arithmetic
 and geometric average.
    If a third number c is given, the average is calculated for the three of
them
    input =  a, b: numbers (float or int)
    optional input = c: number (float or int)
    output: number (float or int)
    '''

    if c == None:
        out1 = ( a + b ) / 2.0
        out2 = ( a * b )**( 1.0 / 2.0 )
        #np.sqrt is the pre-defined square root function
    else:
        out1 = ( a + b + c ) / 3.0
        out2 = ( a * b * c)**( 1.0 / 3.0 )

    return out1, out2
```

for example, we could call this function as:

```
v1, v2 = average( 2, 3, 3 )
```

after which the values of the variables would be `v1 = 2.666..., v2 = 2.0`. We could have avoided to provide the third keyword argument **if** we wanted just the average of two numbers. In this case `c` would have taken the default value of `None`, a *protected word* that means the variable does not have an assigned value. In practice, we could have called the function as:

```
v1, v2 = average( 2, 7 )
```

after which the values of the variables would be `v1 = 4.5, v2 = 9.0, c = None`.
Note that when calling a function, as well as in its declaration/definition, keyword arguments **must** always be given **after** the compulsory ones.

Another important thing to notice is that when calling the function the input values are assigned to the variables of the function assuming the same order as in the function definition. In the example above, when we write `v1, v2 = average( 2, 3, 3)` the value 2 is assigned to the variable `a`, 3 to the variable `b` and the second `3` to the variable c. The following declaration is equivalent and could have also been used `v1, v2 = average( a = 2, b= 3, c = 3)`. In this case, the input can be provided in any order so we could have also written for example `v1, v2 = average( b= 3, c = 3, a=2)`.

If one wants to use a "mixed" declaration using both non-keyword and keyword arguments, the non-keyword arguments must always be provided first and in the correct order, followed by the keyword arguments assigned explicitely. For example, `v1, v2 = average( 2, 3, c = 3 )` is another equivalent and valid declaration, but `v1, v2 = average( c=3, 2, 3 )` would make Python **raise an error**.

Functions are very powerful objects and they allow us to do all sorts of things. To understand their use better, one must practice with them. Look into the Jupyter Notebook provided for further interesting details about functions and try to experiment with them!

In [ ]:

```
1
```