**Lecture 5 (week 3.2)**

Thursday 22nd 2016

Functions, Modules and Recursion

One of the fundamental concept in programming is the one of *functions*. You use functions all the time, e.g. **math.sqrt()** is the square-root function, and so are all methods of the **math** package. It is of course necessary that a user can define a *new* function. Most programing languages comes with the possibility to define new functions, but every language has its own syntax for doing this. In general, to run (i.e. execute) a new function **my\_function()**, the syntax is usually something like

**result=my\_function(arguments)**

where **result** is the outcome of your new function given some input **arguments**. In Python, the syntax is:

**def my\_function(arg1, ...) :** *# args are optional, the column “:” is mandatory*

**calculations**

**return value(s)** *# optional, may be several*

**def** is the Python keyword which defines new functions. Today you will learn how to create useful functions (i.e. function which returns a result), void functions (i.e. function which returns no result) and the concept of *recursion*. But first, I have to introduce a notation feature of Python called *indentation* which is imposed on the programmer (for a very good reason as you will see!).

Indentation

With practice, you will see that programs have a block structure (<https://en.wikipedia.org/wiki/Block_(programming)>). A block is a section of code which can be grouped together, although it is NOT a self sufficient (i.e. autonomous) section of code; this means that a block usually cannot be executed as a standalone piece of code. A block however has a structural meaning, for instance, a function *is* a block because it occupies a compact section of code in a program, it performs a specific task and can be called multiple times by the main program. The convention in Python is that whenever you enter a new block, you have to indent the text of this new block. It is a mandatory Python notation and if you do not do it you will get a syntax error. In Python, blocks are usually indented with four space characters, “ “, in practice any number of spaces is ok, but it is highly recommended to follow the four spaces guideline.

*Note*: never ever use **Tab** key or a mix of spaces and **Tab** for indentation. It might be useful to consult the Python style guide while you are building your programming skills: **<https://www.python.org/dev/peps/pep-0008/>**

Below is a generic example of a block structure of a program:

**main code starts here**

**here are statements of main code**

**block 1 starts here**

**here are statements that belong to block one**

**block 2 starts here**

**here are statements that belong to block 2**

**end of block 2** *# a block is automatically ended by the*

*last command in the blocks indentation*

*level.*

**some more statements from block 1**

**end of block 1**

**some more statements of main code**

**block 3 starts here**

**statements from block 3**

**end of block 3**

**some more statements of main code**

**end of main code**

Note that a block can also contain sub-blocks and so on (e.g. block 2 inside block 1 above). All or some blocks can share data values and arguments. Python requires the indentation of any new block, but the line spacing is free (I used empty lines above to clearly separate blocks but it is not mandatory practice). Example of blocks in Python are user defined functions, loops and conditional statements (there are other possibilities such as Python *class objects*, but we will not cover this for now). Why Python forces this notation on the programmer? well, take a look at the code structure above without the block indentation rule:

**main code starts here**

**here are statements of main code**

**block 1 starts here**

**here are statements that belong to block one**

**block 2 starts here**

**here are statements that belong to block 2**

**end of block 2**

**some more statements from block 1**

**end of block 1**

**some more statements of main code**

**block 3 starts here**

**statements from block 3**

**end of block 3**

**some more statements of main code**

**end of main code**

It is a mess, VERY hard to read, imagine thousands lines of code like this!!! So indentation is very good practice which good programmers have always used, and Python made it mandatory, which is a good thing. Now let’s practice indentation with a very simple function that writes “Hello world” to the prompt, no input arguments:

**In [1]: def mymessage():**

**...: print("Hello world")**

**...:**

**In [2]: mymessage()**

**Hello world**

When defining your new function **mymessage()**, Python automatically starts indentation with **“…”** (remember it is mandatory); to tell Python you are done defining the function, type **Return** twice, you return to the default prompt with no indentation (or the previous indentation if you are writing a sub-sub-block).

Now write the script **myscript.py** with these lines:

**def mymessage():**

**print("Hello world")**

you get this syntax error:

**In [1]: run myscript.py**

**File "/Volumes/TEACHING/PHYS210/2016/myscript.py", line 2**

**print("Hello world")**

**^**

**IndentationError: expected an indented block**

Instead, if you indent your script properly:

**def mymessage():**

**print("Hello world”)**

you get this:

**In [1]: run myscript.py**

**In [2]: mymessage()**

**Hello world**

it works!

Now, I want you to read this introduction to functions, sections 3.1 to 3.12 in <http://greenteapress.com/thinkpython2/html/thinkpython2004.html>

Functions

There are two types of user defined functions, those which don’t return any value(s), called void functions and those which return value(s) called fruitful functions:

**In [1]: def printHello():**

**...: print("Hello!")**

**...:**

**In [2]: printHello()**

**Hello!**

**In [3]: type(printHello())**

**Hello!**

**Out[3]: NoneType**

Note that the void function **printHello()** does not return a value but has the type **NoneType.** Now let’s look at a function which returns a value:

**In [1]: def square(x):**

**...: return x\*x**

**...:**

**In [2]: square(3.4)**

**Out[2]: 11.559999999999999**

**In [3]: a=square(3.4)**

**In [4]: a**

**Out[4]: 11.559999999999999**

**In [5]: type(a)**

**Out[5]: float**

The function has the type of the returned argument(s). When several values of different types are returned, the function has also a mixed type, we will cover this soon in class. In Python, and nearly all computer language, a function-argument is blind to any external variable with same name:

**In [1]: a=1**

**In [2]: b=2**

**In [3]: def f(a):**

**...: print("Inside f: a=",a)**

**...: return a\*a**

**...:**

**In [4]: print("f(b)=",f(b))**

**Inside f: a= 2**

**f(b)= 4**

**In [5]: c=2.7**

**In [6]: print("f(c)=",f(c))**

**Inside f: a= 2.7**

**f(c)= 7.290000000000001**

It is possible to alter the value of a variable external to a function using the **global** keyword, but we will not look at this for now (if you are interested in reading more on this topic, see <http://www.python-course.eu/python3_global_vs_local_variables.php>)

Note also that function-arguments can be preset to some values, so if they are not called it does not generate an error:

**In [1]: def my\_func(x, y=0.0, z=1.0):**

**...: a=x+y**

**...: b=a\*z**

**...: return b**

**...:**

**In [2]: my\_func(1.,3.,2.)**

**Out[2]: 8.0**

**In [3]: my\_func(2.,z=3.)**

**Out[3]: 6.0**

**In [4]: my\_func(z=3.)**

**---------------------------------------------------------------------------**

**TypeError Traceback (most recent call last)**

**<ipython-input-5-059a1b08eba7> in <module>()**

**----> 1 my\_func(z=3.)**

**TypeError: my\_func() missing 1 required positional argument: 'x'**

Modules

Very often, you want to define a function which will be used by different codes. It is then common practice (with *any* programing language) to write separate pieces of code in different files (i.e. different scripts), which can be used by different programs. In Python, these are called *modules*. You have used a module in this course before, the **math** module. You import modules with the command **import**. There are variations on how to import modules:

**In [1]: import math**

**In [2]: math.sqrt(3.)**

**Out[2]: 1.7320508075688772**

**In [3]: sqrt(3.)**

**---------------------------------------------------------------------------**

**NameError Traceback (most recent call last)**

**<ipython-input-3-f9cf3212a7e5> in <module>()**

**----> 1 sqrt(3.)**

You can also rename the module:

**In [1]: import math as m**

**In [2]: m.sqrt(3.)**

**Out[2]: 1.7320508075688772**

You can also import all methods of the module:

**In [4]: from math import \***

**In [5]: math.sqrt(3.)**

**Out[5]: 1.7320508075688772**

which differs from **import math** in the sense that now all methods of **math** are directly accessible from the list of functions names at the Python prompt, so you can call them directly without using the prefix module name **math**:

**In [6]: sqrt(3.)**

**Out[6]: 1.7320508075688772**

This is EXTREMELY dangerous because the method name could accidentally be used to define another object, e.g. now if you write:

**In [7]: sqrt=4.**

**In [8]: sqrt(3.)**

**---------------------------------------------------------------------------**

**TypeError Traceback (most recent call last)**

**<ipython-input-8-f9cf3212a7e5> in <module>()**

**----> 1 sqrt(3.)**

**TypeError: 'float' object is not callable**

and then you have just lost the **sqrt()** function and **math.sqrt()** is not accessible! So I would recommend *never ever* use the **from … import …** syntax.

Now let’s look at how user defined functions can be used as new nested modules. Imagine you write script **A.py** with:

**import B**

**print("I am inside A”)**

write script **B.py** with:

**import C**

**print("I am inside B”)**

write script **C.py** with the function **f(x)** which returns the square of x

**print("I am inside C")**

**def f(x):**

**return x\*\*2**

Now, you can import **A.py**

**In [1]: import A**

**I am inside C**

**I am inside B**

**I am inside A**

**In [2]: A.B.C.f(3.)**

**Out[2]: 9.0**

You have just defined the new method **f()** of modules A, B and C. You should experiment with **run A.py** and different **from … import …** statements to get an idea of how the **import** command works.

Recursion

Recursion is the action of a function calling itself. If not taken care of, a recursive call could cause the function to call itself endlessly. A recursive call must end with a conditional statement, usually when a certain variable has reached a certain value. The example below shows you how to calculate factorial of integer **n** with the user defined function **fact()**. It uses the conditional **if** statement, which we haven’t properly introduced yet, but it is sufficiently simple to understand in this example. Note the indentation inside the **if** statement:

**In [1]: def fact(n):**

**...: if n > 0:**

**...: return n \* fact(n-1)** *# Recursive call*

**...: return 1** *# exits function when n reaches 0*

**...:**

**In [2]: fact(9)**

**Out[2]: 362880**

**In [3]: fact(1)**

**Out[3]: 1**

As an exercise, how would you modify this function so that it never stops calling itself? what happens then?