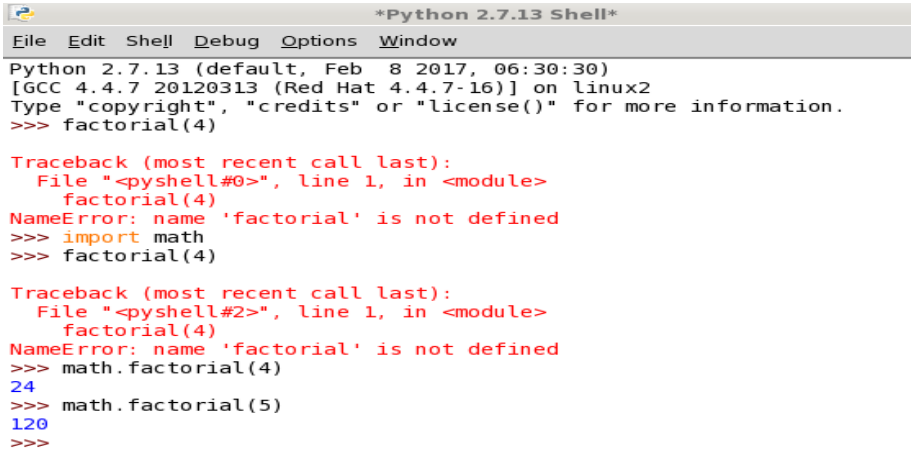
I. FUNCTIONS

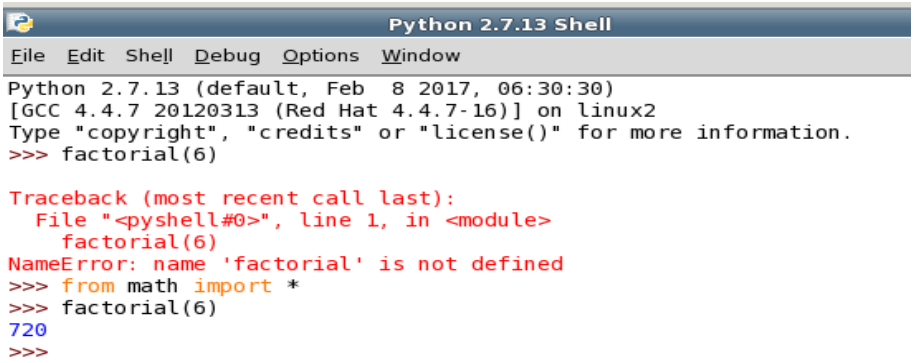
A function is a set of statements that instructs the computer to do something. We have already used the built-in functions named **type()** and **print()**. Recall that **type()** was used to determine whether a particular variable was an integer, float or string. If the function accepts an input - the value inside the brackets - it is called the *argument* of the function. If the function sends back a value it is called the *return value*.

A. Libraries and Built-In Functions

Some functions - such as the **type()** and **print()** functions discussed above are *built-in* functions and can be called at any time. Other functions are not built-in and must be *imported* into IDLE or a python script before they can be used. Most of the mathematical functions you may want to use exist in a library (module) called **math**. (Actually there are bigger libraries called **numpy** - numerical python - and **scipy** - scientific python - that contain all the **math** functions in addition to others that are useful to scientists.) There are two ways to be able to use the functions in the math library. The first is shown in the following figure. As shown below, if you try to use **factorial()** before the **math** library is imported you will receive an error message. If you import using the statement **import math**, it will be necessary to use *dot notation* : **library.function()** as shown below.



You do not need to use dot notation if you if you import using the **from** statement, and the *wildcard* (*), as shown below. The command **from math import *** means import every function from the math library.



B. Defining your Own Functions

Many times you will want the computer to do something that is not already a built-in function. As an artificial example, consider writing a function that will print a string variable four times. Do this using **gedit** and save it as a script ***.py** file which you can run. Again the wildcard (*) means you are free to call the function any allowable name (NOT *.py) The *keyword* **def** marks the beginning of the function **def**inition.

print_four.py

```
def print_four(inp_str):
    print inp_str
    print inp_str
    print inp_str
    print inp_str

# the main body follows; note that it uses the defined function "print_four"
test='It is OK\n'
print_four(test)
print('we have returned to the main function')
```

Note that the **def** line ends with a colon implying that the following line(s) of code defines the function. Note the statements that the function will carry out are indented. It is crucial to have the same level of indentation for each line of the function . You can use two spaces or four spaces or anything - you just have to be consistent.

> **EXERCISE: To convince yourself, try writing the function with a different indentation level for one of the four print statements. What happens when you try to run the script?**

A blank line and a return to the non-indented lines marks the end of the function definition. Note the order of execution of the statements:

1. **test = 'It is OK'** assigns a string to the memory location called **test**
2. **print_four(test)** sends the argument **test** to the function **print_four()**. Note that inside **print_four()** a memory location called **inp_str**, which is a *local variable* , contains the string value **'It is OK'**. Local variables will be defined below.
3. the control has been transferred to the function **print_four()** which executes all its statements
   (a) **print inp_str**
   (b) **print inp_str**
   (c) **print inp_str**
   (d) **print inp_str**

   after the completion of the last function statement, which is the end of the function definition, the control is returned to the main routine.
4. **print('we have returned to the main function')**

The statements inside the function definition are not executed until the function is called in the main program. Try running the program and you will see that a function is like a detour in the flow of execution. Once the function is called - all the statements in the function are executed and only afterwards is the next statement in the main program executed.

C. Keyboard Input

As an another artificial example, assume that we are interested in calculating the **sin** of an arbitrary angle. You want to be able to input which angle you are interested in. There is a built in function called **raw_input('instruction \n')** that does two things. First it prints out the "instruction" for the input. For example the instruction could be 'What is your name?' Note that the **\n** produces a carriage return. Secondly **raw_input()** will

return, whatever it is you want to input, as shown in the figure below.

simple_calculation.py

```
from math import *

#note that pi is defined in the math library
def to_radians(angle):
    out_angle=angle*pi/180.0
    return out_angle

in_variable=raw_input('enter the angle in degrees\n') #note that the return value will be a string
float_angle=float(in_variable)
out_variable=sin(to_radians(float_angle))
print ('the sine of ',in_variable,' degrees is', out_variable)
```

Note that **raw_input()** always returns a string, no matter what the nature your input is. So you will have to change the string to a float or integer if necessary using the **float()** and **int()** functions, respectively. The trigonometric functions in python assume that the argument is in radians, so if we enter the angle in degrees we will have to change it to radians before calculating the sin. In the python script shown above, the conversion of the input angle to radians occurs in the function **to_radians()**.

> **EXERCISE: To convince yourself, try writing the script without converting the input to a float. What happens when you try to run the script?**

D. Functions and Local Variables

If a variable is used within a function, it is not accessible to outside the function, as shown below. **print(cat)** and **print cat** should do the same thing. They are both instructions to print the value that is located in memory location called **cat** to the screen. Note that the **print cat** statement works inside the function, but outside the same instruction **print(cat)** generates an error message. This is what is meant by saying that **cat** is a *local* variable to the function **put_together**. Once the function has finished execution, all the memory associated with the function is destroyed. The code also exhibits that addition of two string variables concatenates the strings. Try creating the code either in interpreter (IDLE) or in script mode and see what happens.

```
>>> def put_together(string1,string2):
        cat=string1+string2
        print(cat)


>>> input1='the cow jumped '
>>> input2='over the moon.'
>>> put_together(input1,input2)
the cow jumped over the moon.
>>> print cat

Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    print cat
NameError: name 'cat' is not defined
>>>
```

II. SIMPLE REPETITION

One of the best reasons for using computers is that they can do repetitive boring tasks accurately and quickly. Consider yet another artificial example that we wanted to write a function that would print something out 100 times. While it would be possible to write a function that contains 100 print statements, it would be stupid. A way to perform repetitive tasks simply is to use the **for loop**. The syntax is similar to a function definition. That is, note the colon and the indentation. Any symbol can be used for the loop counter (**i**)

```
File  Edit  Shell  Debug  Options  Window
Python 2.7.13 (default, Feb  8 2017, 06:30:30)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-16)] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> for i in range(10):
        print 'hello!'


hello!
hello!
hello!
hello!
hello!
hello!
hello!
hello!
hello!
hello!
>>>
```

III. EXERCISE: NUMERICAL INTEGRATION

Consider the problem of trying to calculate the definite integral

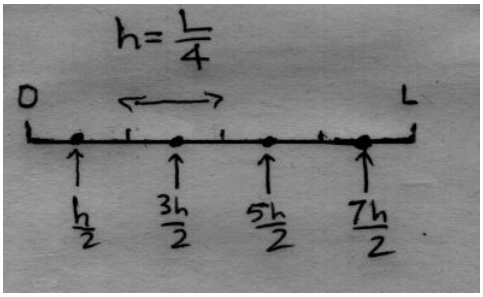
$$\int_x^{x+L} f(x)dx$$

You probably know that there are many approximations of varying accuracy such as:

1. Midpoint Rule $\int_x^{x+h} f(x)dx \approx hf(x+\frac{h}{2})$
2. Simpson's Rule: $\int_x^{x+h} f(x)dx \approx \frac{h}{6}\left[f(x)+4f(x+\frac{h}{2})+f(x+h)\right]$
3. Two-point Gaussian Rule

$$\int_x^{x+h} f(x)dx \approx \frac{h}{2}\left[f(x+\frac{h}{2}(1-\sqrt{3}))+f(x+\frac{h}{2}(1+\sqrt{3}))\right]$$

For example, consider trying to calculate $\int_0^L f(x)dx$ by breaking up the interval of integration into 4 pieces as shown below :



Using the midpoint rule where $h = \frac{L}{4}$,

$$\int_0^L f(x)dx = \int_0^h f(x)dx + \int_h^{2h} f(x)dx + \int_{2h}^{3h} f(x)dx + \int_{3h}^L f(x)dx$$
$$\int_0^L f(x)dx \approx h\cdot f(\frac{h}{2}) + h\cdot f(\frac{3h}{2}) + h\cdot f(\frac{5h}{2}) + h\cdot f(\frac{7h}{2})$$

.

Write a program that will calculate the electric field at position (2.2m,0,0) for a line of length $L = 2$m and linear charge density 10 C/m that is lying on the positive x-axis with one end at the origin. Starting by assuming $h = 0.5 = L/4$, where $N = 4$ and reducing the step size by 2 until $h = \frac{L}{1024}$. Print your calculations in two columns: $N, E(N)$ separated by three tabs: (**"\t"**)You should be able to do the calculation analytically and observe that the computation is getting closer to the exact value as h is decreased.