Python is a programming language, it can be used directly in with command line, or by writing and executing Python scripts. We start to use Python in command line by typing :

```
$ python3
... blabla
>>>
```

To exit, enter `exit()` or `Ctrl-D`. This is call an interpreter and we will start the tutorial within the interpreter. Then we will use a text editor (such as anaconda or SPYDER or PYCHARM) to edit the scripts using extension `.py`.

To use a script using a console type

```
$ python3 script.py
```

where the script is `script.py`. If your're using an IDE such as Anaconda, Spyder or pyCharm there is usually a RUN command.
Remark 1 : >>> just indicates that you're within the interpreter, DO NOT TYPE THEM!
Remark 2 : the pound sign (or hashtag) # are used to comment. Everything at the right of the pound is ignored.
Example : run the script `testscript.py` available in Moodle.

1 Basic training

This is for you to train and understand the basic mechanics DO NOT SPEND TOO MUCH TIME ON IT! Just make sure you understand what you are doing (by directly going to section 1.5 Exercices if you like).

1.1 Scalar types

Python can compute, test the following lines in the intepreter :

```
>>> 2*3
>>> 2**3
>>> 2/3
>>> 2//3          ## division integer
>>> 2%3           ## remainder modulo
```

Note the use of ** (double star) for power computation.

We can use variables when they are affected, i.e given a value. To see their value, we can type their name or within a script use the command `print` :

```
>>> a = 2    ## affectation
>>> a
>>> print(a)
```

To know the type of a variable use `type` :

```
>>> type(a) ## a integer
>>> a=2.0
>>> type(a) ## new type for a
```

Python can use classical affectation

```
>>> a += 2 # a = a + 2
>>> a *= 2
>>> a -= 1
>>> a /= 2
```

We can convert an integer to float and conversely via the functions `int` or `float`

```
>>> a = 2 ## integer
>>> type(a)
>>> b = float(a) ## float
>>> type(b)
>>> b = 1.5
>>> c = int(b) ## conversion integer
```

1.2 Strings

There are other objects that python can use different from scalar such as for example character strings (`str`). You can use both quote or double quote (but be consistent).

```
>>> s = "hed is red"
>>> ss = 'hed is red' ## equivalent notation
>>> type(s)
>>> type(ss)
```

We can operate on strings

```
>>> ss = s + ' and yop'
>>> print(ss)
>>> ss += s
>>> len(s)
>>> len(ss)
```

We can access each element (character) using **square bracket**. Indices starts at zero. In addition, python can use **slices** to obtain portion of strings. Try the following commands :

```
>>> s[0]
>>> s[1:5]
>>> s[:5]
>>> s[-1]
>>> s[5:]
>>> s[-3:]
```

We can read but not write for a string

```
>>> s[0]='Z' ## error
```

We can convert strings to number (and conversely) which is useful when reading files :

```
>>> x = "103"
>>> type(x)   ## x is a string
>>> y = int(x)
>>> type(y) ## y is an integer
```

```
>>> print(x,y)
>>> x = "1e-3"
>>> y = float(x)
>>> print(x,y)
>>> x = 12
>>> s = "{}".format(x) ## beware of the syntax
>>> y = 0.0001
>>> ss = "{0} {1}".format(x,y)
>>> ss = "{} {}".format(x,y) # assuming an order
>>> ss = "{1} {0}".format(x,y) # changing the order
```

The previous two commands are called format by default it uses the classical format for the variable depending on its type. You can change this :

```
>>> x = 3
>>> y = 1.001
>>> s = "{0} is an integer and {1} is a  float".format(x,y)
>>> ss = "{0:.3f}".format(123.01)
```

You can manipulate what you want printed very precisely there is a ton of explanation there `https://pyformat.info/`

In python, all types have methods – functions – that can be used. To access all the methods of an object use the command `dir`. To access documentation use the command `help` :

```
>>> dir("") ## "" is an empty string
>>> help("".upper)  ## documentation on the upper method
```

Try the following functions and try ONE use ;

- `count`
- `replace`
- `upper`

1.3 Lists

Python also provides a type of list that can contain anything. Initialization of a list requires square brackets `[]`.

```
>>> x = [] ## empty list
>>> type(x)
>>> x = [1,2,3]
>>> len(x)
>>> x[0]
>>> x[1:2]
```

Lists can contain any type of data

```
>>> x = ["abc",1,1e-3,[1,2,3]]
>>> type(x)
>>> type(x[0])
>>> type(x[1])
>>> type(x[2])
>>> type(x[3])
```

```
>>> print(x[3][1])
>>> type(x[3][1])
```

Some operations are available

```
>>> x = [0]
>>> y = x*10
>>> z = y + y
```

A lot of operations won't work on lists. Lists are not considered arrays.

We can create list automatically :

```
>>> x = list(range(10))
>>> y = list(range(1,20,2))
>>> z = list(range(20,0,-1))
```

Like strings, there are tons of methods for lists. Use `dir` and `help` to understand what these functions do. Try one case for each :

- `sort`
- `append`
- `pop`
- `extend`

1.4 Loops and conditions

Condition expressions are used to perform tests. A condition returns the value `True` or `False` which is called a boolean. This is a type in Python and you can store this value in a variable.

```
>>> 1<2
>>> 1==1
>>> 2>=2
>>> 3<2
>>> "ab"=="ab"
>>> 3<4 or 4<3
>>> 3<4 and 4<3
>>> type(1>3)
```

To make conditional statement we use `if`. The conditional blocks are defined via tabulation in python. The use of a text editor is easier in this case. You can now switch to your text editor.

Use the following syntax :

```
if condition :
    ... ## code executed if condition is True
```

The bloc ends when the last tabulation is recorded. Optionally, we can add an `else` or `elif` block (`else` followed by an `if`, we can add a final `else` too).

```
if condition :
    ... ## code executed if condition is True
else :
    ... ## code executed if condition is False
```

or

```
if condition :
    ... ## code executed if condition is True
elif condition2 :
    ... ## code executed if condition is False AND condition2 is True
```

A simple parity test is then

```
x = 3
if x%2 ==0:
    print("x is even")
else :
    print("x is odd")
```

Loops works also using tabulation. We loop on an iterator such as a list or a string. Keyword is `for` in conjunction with `in` with the following syntax :

```
for index in iterable_object :
    ... ## loop code
```

Code is repeated as many times as there are elements in the iterator and `index` is equal to the corresponding value :

```
for x in range(10):
    print(x) ## print all the x in the range list
```

Loops and conditions can be combined together. You just need to be careful with the indentations (tab).

```
for x in range(10):
    if x%2 ==0:
        print ("x = %d is even "%x)
    else :
        print ("x = %d is odd"%x)
```

Alternatively, the `while` block will be executed as long as the conditions is fulfilled :

```
while condition :
    ... ## code
```

1.5 Exercises

Do the following exercises (keep them in scripts) :

1. Create a string of 1000 "a" and of 1000 "ab" (hints : use a loop)
2. Create a list of 100 integers equal to 1
3. Make a list that contains all the even number ≤ 100 (`pair100.py`)
4. Compute the sum of the 100 first integers (`sum100.py`)
5. Compute the sum of the odd integers ≤ 100 (`sumimpair100.py`)
6. Let `a` be the list from 0 to 100. What is a [::2] ? What sum(a[::2]) ?
7. Create a list of number by yourself and find the maximum of that list. Can you also find the index of that maximum ? (example : a = [1, 2, 30,0.2, 10.0, 2.1], maximum is 30 its index is 2)

2 Basic I/O

Python allows reading and writing in files using the function `open` which syntax is :

`open(filename,method)`

The filename is a string and the method can either be 'r', 'w' or 'a' depending on the desired access : read ('r'), write ('w') and append ('a').

```
>>> f = open("test.txt","w") ## open a file for writing
>>> f = open("test.txt","r") ## read access
```

For read access the call is an error if the file does not exist. For write access, if the file exists it erases its content else it is created. Create a file "test.txt" and put anything inside. Try :

```
>>> f = open("test.txt","r")
>>> type(f) ## what is the type
>>> dir(f) ## what methods
>>> f.close() ## close it
```

Opened files are iterator on the lines :

```
>>> f = open("test.txt","r")
>>> for l in f:
>>>     print(l)
>>> f.close()
```

there are also other methods :

```
>>> f = open("test.txt","r")
>>> l = f.readline()  ## read the first line
## and positon the stream on the next line
>>> l = f.readline()  ## second line
>>> ll = f.readlines() ## all remaining lignes
>>> type(l)
>>> type(ll)
>>> len(ll) ## number of lines (minus 2)s
>>> f.close()
# don't forget to always close a file!
```

In write access, we can use `write` which prints the string in parameter

```
>>> f=open("retest.txt","w")
>>> f.write("writing␣test\n")
>>> f.close()
```

Check if your sentence is in the file.

2.1 Exercises

You can use as an example the file `fruit.txt` available in Moodle.

1. make a script that counts the number of lines, words and characters of a file (`wc.py`). Hints : you can use the string method `split`

2. make a script that capitalize all the characters of a file (and write it in another file) (`cap.py`). Hints : you can use the string method `upper`
3. make a script which reverse the order of the lines of a file (`rev.py`). Can we do it for words ? Characters ?

3 Functions

We have already encountered functions - especially in the form of methods. Functions in Python obey the same rules as the functions in programming : they take something as an argument and return something. Some rules

— They can take a variable number of arguments (or no argument at all)
— The body of the function is indented (tab)
— They can return several things at the same time

For the rest, they are similar and use the same keyword `return`.

The syntax is as follows :

```
def functionname (arg1,arg2,...,argkw1,argkw2,...):
        ... ##  fonction code
        return something ## return optional
```

The syntax may seem a bit obscure let's take a simple example, a function that returns the square of a number

```
def square(x):
    return x**2
```

The keyword `def` is followed by the name of the function and in parenthesis the list of arguments (here only one) and finally the two points to start the block. The block itself only return the square. We check that it works :

```
>>> print(square(10))
```

Then we want to do a power function :

```
def pow(x,n):
    return x**n
```

No problem, we check that it works :

```
>>> print(pow(10,3))
```

3.1 More on functions

Previously we have to make formal arguments that is to say they will be taken in order. We also note that there is no type checking a priori, so the arguments could be anything. Suppose we try with a list :

```
>>> x = list(range(10))
>>> print(pow(x,3)) ## error
```

It does not work. We can add a boolean parameter that the user will set at `True` if they want to pass a list and `False` otherwise

```
def pow(x,n,l):
    if l:
        d = []
        for k in x:
            d.append(k**n)
        return d
    return x**n
```

We check

```
>>> print(pow(10,3,False))
>>> print(pow(range(10),3,True)) # does it work?
>>> print(pow(list(range(10)),3,True))
```

We note in passing that we can return a list. We must add a new argument for our function. Moreover, we must add it even if we want the power of a normal number. To avoid this, we can give a default value to the third parameter :

```
def pow(x,n,l = False):
    if l:
        d = []
        for k in x:
            d.append(k**n)
        return d
    return x**n
```

so we just have to write :

```
>>> print(pow(10,3))
>>> print(pow(range(10),3,True))
```

Now, the concern is on the type of return : it is either a scalar or a list. To avoid worries, we would like to return the result as a string, but optionally. For this it is necessary to convert, the list in chain and the number also :

```
def pow(x,n,l = False, asstring):
   if l:
        if asstring :
            s = ""
            for k in x:
                s += "␣%f"%k**n
            return s
        else :
            d = []
            for k in x:
                d.append(k**n)
            return d
    if asstring :
        return "%f"%x**n
    return x**n
```

Here the test returns us an error

```
def pow(x,n,l = False, asstring):
SyntaxError: non-default argument follows default argument
```

Indeed, if we can omit an argument, the interpreter does not know which argument we are referring to, so we have to modify the header of the function :

```
def pow(x,n,l = False, asstring=False):
```

It works well. But now we have to remember what order the arguments are. To avoid this, Python predicts the arguments by keyword (argkw see above) because the following combinations are now feasible :

```
>>> print(pow(10,3)) ## default
>>> print(pow(10,3,asstring=True)) ## default number + string
>>> print(pow(range(10),3,l=True,asstring=True))
>>> print(pow(range(10),3,asstring=True,l=True)) ## any order
>>> print(pow(range(10),3,True)) ## in order
>>> print(pow(range(10),3,True,True)) ## in order
```

In addition, Python allows to return several values (or types) You just need to separate the various variables by a comma :

```
def div(a,c):
    return a/c, a%c
```

Yielding

```
>>> q,r = div(10,3)
>>> print(q,r)
```

Functions can call other functions and call themselves. They can receive any type of argument (even other functions). For example

```
def square(x):
    return x**2

def g(x,f):
    return f(x)
print(g(10,square))

def recursion(k):
  if(k > 0):
    r = k + recursion(k - 1)
  else:
    r = 0
  return r
print(recursion(6))
```

Python finally provides a set of functions that facilitates the use of functions on lists : `map`, `filter` and `reduce`. Do a `help` of these functions to see what they consist of.

3.2 Exercises

1. Make a function that tests if the integer sent as a parameter is divisible by 3
2. Use the previous function to get the list of multiples of 3 less than 1000 (optional : you can use `filter`)

3. In the same way, list the quotients obtained by dividing by 7 numbers less than 1000 (optional you can use `map`)
4. Do the factorial function $n! = 1.2.3 \ldots n$ with a loop and recursively and test how far it can go.

4 Module

Python allows access to several hundred separate functions in modules. They are accessible via the import of the module to which they belong :

```
>>> import modulename
```

The modules are actually python files where classes and functions are coded. The Python library contains several modules as standard and imports some of them at boot time. A developer can simply create modules and distribute them easily. The import allows to use the constants, classes and functions defined by the module via the point operator ".".

```
>>> import modulename
>>> modulename.function()
```

4.1 `math` module

This is the standard math module :

```
>>> import math
>>> dir(math)
>>> help(math)
>>> print(math.pi)
>>> print(math.e)
>>> math.sqrt(2)
>>> x = map(math.sqrt,range(100))
```

Note : we will see others numerical calculus modules such as `numpy` or `scipy`.

4.2 `random` module

This is the random generator module :

```
>>> import random
>>> dir(random)
>>> help(random)
>>> x = random.randint(0,10) ## random int between 0 et 10
>>> y = random.normal(0,1) ## normal law
```

1. Create a list of size n of elements drawn from a Uniform $[0, 1]$ and then a Gaussian $\mathcal{N}(0, 1)$ distribution.
2. Compute the mean and variance the two lists. See the evolution of theses values according to n.
3. Make a program that randomly changes the order of words in one line for all lines in a file (use `random.shuffle`)

4.3 Dictionnary

Python has another type called the dictionary. As the name suggests its data structure is key/data. We create such a structure via the curly bracket (accolade). Each key / data pair is separated by commas, the key being separated from its data via colon " :" (deux points)

```
>>> d = {} ## empty dictionary
>>> d2 = {'a':123,'1':'ABA',12:[1,2,3]}
>>> print(d2['a'])
>>> print(d2[12])
>>> print(d['1'])
>>> d[10]= 'one'
>>> d['10']= 'other'
>>> print(d)
>>> print(d.keys())
```

1. Make a program that counts all the occurrences of all words in a file (you can use the file `fruit.txt` on Moodle). Hint : you can test if a key is in the dictionary using the `in` keyword.

```
if word in dictionary.keys():
        # if true word is already in dictionnary keys
```

5 Advanced lists

Some more advanced commands for list generation :

```
>>> a = [x**2 for x in range(10)]
>>> b = [x/2 for x in [x**2 for y in range(10)]]
>>> c = [(x, x+10) for x in range(10)]
>>> d = [x**y for x,y in c]
>>> e = [ x>10 for x in range(20)]
```

1. Make a program that creates a list of all words in a file in one line
2. Make a function that takes a list as an argument and a function and applies the function to each element of the list and returns it. In one line (for the function)

5.1 Copy

Python can copy objects. Many are copied by value and others by reference :

```
>>> x =3
>>> y = x   ## copy by value
>>> x =2
>>> print(x,y)
>>> x = [1,2,3]
>>> y = x    ## copy by reference
>>> x[0] = -1
>>> print(x,y)
```

```
>>>  x =list(range(1,4))
>>> y = x[:]  ## copy by value
>>> x[0]=-1
>>> print(x,y)
>>> x = [[1,2],[3,4]]
>>> y = x[:]
>>> x [0][0]=-1
>>> print(x,y)  ## ah ah
```

Look at the doc on the `copy` module and try to solve the problem of copying tables of tables.