"Everything should be made as simple as possible,
but not one bit simpler ."
-- A. Einstein

ITI 1120
Module 5: Program Structure

General Concepts:

1. Python functions.
2. Passing parameters.

General Objectif: Understand the transfer of paramètres to functions.

Learning Objective:

1. Definition and usage of functions.
2. Passage of lists as parameters and returning lists as results.

Theme 1. Python Functions

Sub-theme: Use of several sub-programs

- So far we have used programs composed of few functions and one main program.
- If an algorithm (function) contain a complex bloc of imbricated in another bloc, you can transform the imbricated bloc into a separate algorithm (function).
- Thus each algorithm/function can itself invok other algorithms/functions. That way, you can keep our algorithms/functions simple, short and clear.
- It is thus possible to divide a complex problem into several tasks which themselves can also be divided (top-down design)

Sub-theme: Python program structure

- Import modules if necessary.
- If there are function definitions, they are not executed, but the definitions are stored in the memory. They will be executed later, if they are invoked.
- The execution starts with the first main program line until the last one.
- Function définitions must be before their invokation.
- For clarity purpose, all function definitions should be at the beginning of the file.

Sub-theme: A function definition

```
def nameOfFunction(parameters list):
      "comments optionels"
       bloc of instructions
```

Simple function without parameters:

```
def table7():
      i = 1  # local variable
      while i < 11 :
            print(i * 7, end =' ')
            i = i + 1

table7()  # main programme
          # invoks the function without parameters
          # which does not return anything
          # prints 7 14 21 28 35 42 49 56 63 70
```

Sub-theme: A function that call other functions

```
def table7():
    i = 1
    while i < 11 :
        print(i * 7, end =' ')
        i = i + 1
    print()

def table7triple():
    print('The table by 7 in triplicate :')
    table7() # 3 inkocations of the other function
    table7()
    table7()

table7triple()
```

Will display:

```
The table by7 in triplicate :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Sub-theme: Functions with a parameter

```
def table(base):
 # take a parameter, the base, and display the table
 # of values
    i = 1
    while i < 11 :
       print(i * base, end =' ')
       i = i + 1
```

We can ise with different values: arguments.

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130
>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Sub-theme: Functions with several parameters

```
def tableMulti(base, start, end):
  print('Fragment of the multiplication table by', base,':')
  i = start
  while i <= end:
    print(i, 'x', base, '=', i * base)
    i = i + 1

tableMulti(8, 13, 17)
```

Whose display will look like :

```
Fragment of the multiplication table by 8:
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Sub-theme: locales variables locales and global variables

- When variables are defined inside the body of a function, they are accessible only to that function. They are categorized as **locales variables** to that function.
- It is the case for variables base, start, end and i in the previous exercise.

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

- The symbol base is unknown in the main program

Locales variables locales and global variables

- Variables definied outside a function are global variables. Their content is visible to the inside of the function but the function can not modify them.

```
def mask():
    p = 20  # another p, local variable
    print(p, q)

p, q = 15, 38
mask()
print(p, q)
```

How to avoid creating a locale variable, when there is a global variable

```
>>> def goUp():
    global a
    a = a + 1
    print(a)

>>> a = 15
>>> goUp()
16
>>> goUp()
17
>>>
```

Sub-them: Functions that return a value

```
def cube(w):
    return w*w*w

b = cube(9) # store the result
print(b)    # will display 729
```

Sub-theme: functions documentation

```
def my_function():
  “function description"
  …
>>> help(my_function) # display the description
Help on function my_function in module __main__:

my_function()
    function description
```

Specify the parameters type and the returned value

```
def prime(n):
     '''(int)->bool
     returns True if n is prime, False otherwise
     Precondition: n is appositive integer
     '''
     if(n==1):
          return False
     for i in range(2,n):
          if(n%i == 0):
               return False
     return True
```

```
>>> help(prime)
Help on function prime in module __main__:
prime(n)
    (int)->bool
    returns True if n is prime, False otherwise
    Precondition: n is a positive integer
```

Example

```
def divisors(n):
    '''(int)->None
    print all the divisors of n
    Precondition: n is a potive integer
    '''
    for i in range(1,n+1):
        if(n%i == 0):
            print(i, end=" ")
    print()
```

Example

```
def is_eligible(age, citizenship, prison):
    '''(int, str, str)->bool
    Returns True if the person is eligible to vote,
    and False otherwise
    Precondition: age non-negative
    '''
    citizenship = citizenship.lower()
    citizenship = citizenship.strip()
    prison = prison.lower()
    prison = prison.strip()
    if((citizenship =='canada' or
          citizenship =='canadian')
          and (age>=18) and (prison=='no')):
        return True
    else:
        return False
```

Sub-theme: Modules of functions

We can put function definitions in a Python module, and the program that uses them in another module.

Exemple : We want to produce a serie of drawings using the module ***turtle.*** Write the following code lines and save them in a file whose name is ***drawings_turtle.py*** :

```
from turtle import *

def size, color):
 "function that draws a square for a given size and color"
  color(color)
  c = 0
  while c < 4:
    forward(size)
    right(90)
    c = c + 1
```

Import the module of functions

```
from ddrawings_turtle import *

up()             # raise the crayon
goto(-150, 50) # move up and to the left
# draw ten red aligned squares:
i = 0
while i < 10:
  down()             # crayon down
  square(25, 'red') # draw a square
  up()               # raise the crayon
  forward(30)        # move + further
  i = i + 1
```

How to import modules?

```
import math
x = math.sqrt(3)  # use the name of the module
print(x)


from math import sqrt     # import a function
# from math import *      # or all

x = sqrt(3)
print(x)
```

Question:

What does the following Python code display on the screen?

```
def mask():
    a = 50
    b = 60
    return b

a, b = 10, 20
res = mask()
print(a, b, res)
```

Possibles responses (chose one)

a) 10 20 60
b) 50 60 60
c) 10 20 20
d) error

Theme 2. Parameter passage

Sub-theme: Lists as parameters

- A list is a type of reference; i.e. that is accessed via a reference variable.
- A list is not passed from a function to another, it is its reference (i.e. the content of a reference variable) that is passed to the function (or returned to that function).
- That is, we have (temporary) 2 references to the same list.
- Eventhough the invoked function can not modify the original reference variable, it can modify the content of the list. The changes brought into the list will remain even after the return from the invokation.
  - The copy of a variable that represent a single value (such as int, float) is destroyed when the function returns.
  - For a list, it is a copy of the reference variable that is destroyed during the return.

Sub-theme: Functions that return a list of values

```
def table(base):
    result = [] # empty list result
    i = 1
    while i < 11:
      b = i * base
      result.append(b) # add to the list
      i = i + 1
    return result

res = table(3)
print(res)
```

Sub-theme: Functions that take lists as parameters

```
def my_function(a):
    a[0] = 100
    a[1] = 200

list1 = [1,2,3,4,5]
my_function(list1)
print(list1)    # display [100, 200, 3, 4, 5]
```

During the function execution, `a` and `list1` are two references of the same list. When `a` is modified, `list1` is modified permanently.

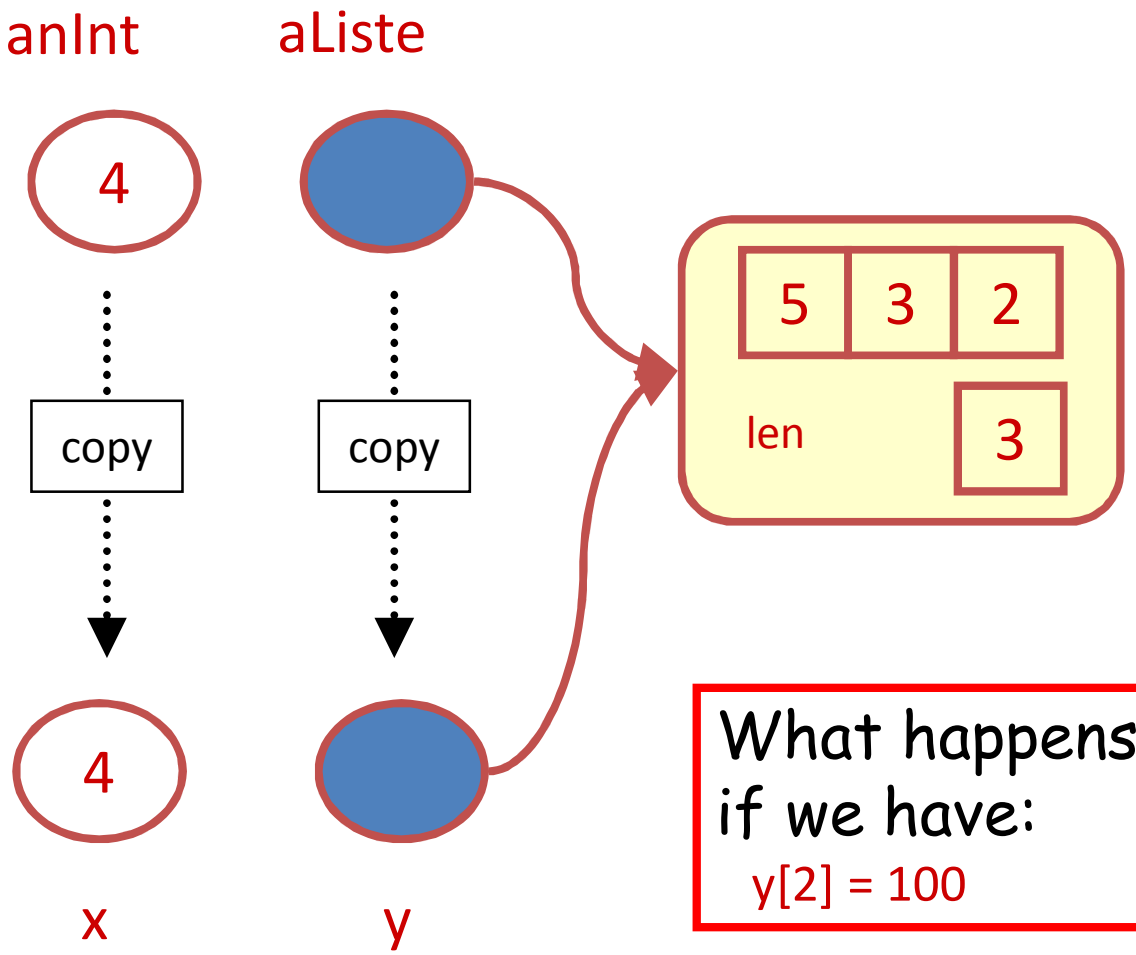
Sub-them: Passing as parameters individual values versus lists

```
def m(x, y):
  x = 5
  y[2] = 100 # y is a second reference
             # a aListe, but temporary

anInt = 4
aListe = [5,3,2]
m(anInt, aListe)
print(anInt)  # display 4, no change
print(aListe) # [5,3,100] permanent change
```

Passage of simple values and reference

Invocator:
m(anInt, aListe)



Invoked method:
m(x, y)

What happens if we have:
y[2] = 100

Question

What is being displayed with the following Python code?

```
def m(a, b):
    a = a + 1
    b[0]= -2

a = 100
b = [10,20,30,40]
m(a,b)
print(a,b)
```

a) 101 [-2, 20, 30, 40]
b) 100 [-2, 20, 30, 40]
c) 101 [10, 20, 30, 40]
d) 100 [10, 20, 30, 40]

Question – Solution:

Correcte response b)

Explication: the local variable a changes, but not the global variable a.
The list has changed via the local variable b, which is a second reference to the list, and the change stays.

Conclusion

- We reviewed Python program structures, the function définitions (including how to pass them parameters) an the concept of locale and global le concept de variables.