CHAPITRE 4 : LES FONCTIONS

I. Définition d'une fonction :

Une fonction est un bloc d'instructions qui a un nom, dont le fonctionnement dépend d'un certain nombre de paramètres (**les arguments de la fonction**) et qui le plus souvent renvoie un résultat (au moyen de l'instruction **return**) ; dans le cas contraire, on parle plutôt de procédure que de fonction.

Les fonctions permettent de décomposer les programmes en sous-programmes et de réutiliser des morceaux de programmes.

Syntaxe :

```
def nomFonction( argument1,..., argumentn ):
    "Cette ligne explique à quoi sert cette fonction"
    bloc Instructions
```

- La ligne « **def nomFonction(argument1, ... , argumentn) :** » est appelée l'en-tête de la fonction ;
- On n'oubliera pas les **:** et l'indentation qui sont obligatoires en Python !

La définition d'une fonction :

- commence par le mot-clé **def**
- suivi du **nom de la fonction**
- et d'une liste entre parenthèses de paramètres formels ; cette première ligne se termine par des double-points :
- Les instructions qui forment le corps de la fonction commencent sur la ligne suivante, indentée par quatre espaces (ou une tabulation)
- La première instruction du corps de la fonction peut être un texte dans une chaîne de caractères ; cette chaîne est la chaîne de documentation de la fonction. On peut la visualiser dans un terminal en tapant l'instruction
  help(nomFonction)
- Le retour à la ligne signale la fin de la fonction

✍ Dès que l'instruction **return** est exécutée (si elle est présente), l'exécution de la fonction se termine ; la partie du code écrite après l'instruction **return** n'est jamais exécutée.

Exemple 1 : une fonction qui ne retourne rien

Ecrire la fonction (Procédure) **Affiche** qui prend en paramètre un nombre x et qui affiche le carré de ce nombre

```
def Affiche (x):
    "cette fonction affiche le carré d'un nombre"
    print(x**2)
```

Exemple 2 : une fonction qui retourne une valeur de sortie

Ecrire la fonction **double** qui prend en paramètre un nombre x et qui retourne le double de ce nombre

```
def double (x):
    "cette fonction calcule le double d'un nombre"
    return 2*x
```

Exemple 3 : une fonction qui retourne deux valeurs de sortie

Ecrire la fonction **Calcul** qui prend en paramètre deux entiers x et y et qui retourne la somme et la produit de x et y .

```
def Calcul (x,y):
    " " "cette fonction retourne
    la somme et le produit de deux nombres " " "
    s=x+y
    p=x*y
    return s,p
```

II. Appel d'une fonction :

Une fois qu'une fonction **nomFonction** a été définie, elle peut être utilisée dans une expression particulière qu'on nomme un appel de fonction et qui a la forme :

s1,s2,…,sm=nomFonction (v1 , v2 , …, vn)

où :

- v1 , v2 , …, vn sont des expressions dont la valeur est transmise au paramètres.
- s1 , s2 , …, sm sont les valeurs retournées par la fonction.

✍ Si la fonction n'a pas de valeur de retour, l'appel de la fonction a la forme :

nomFonction (v1 , v2 , …, vn)

Exemple 1 :l'appel de la fonction Affiche

```
Affiche( 3 )
```

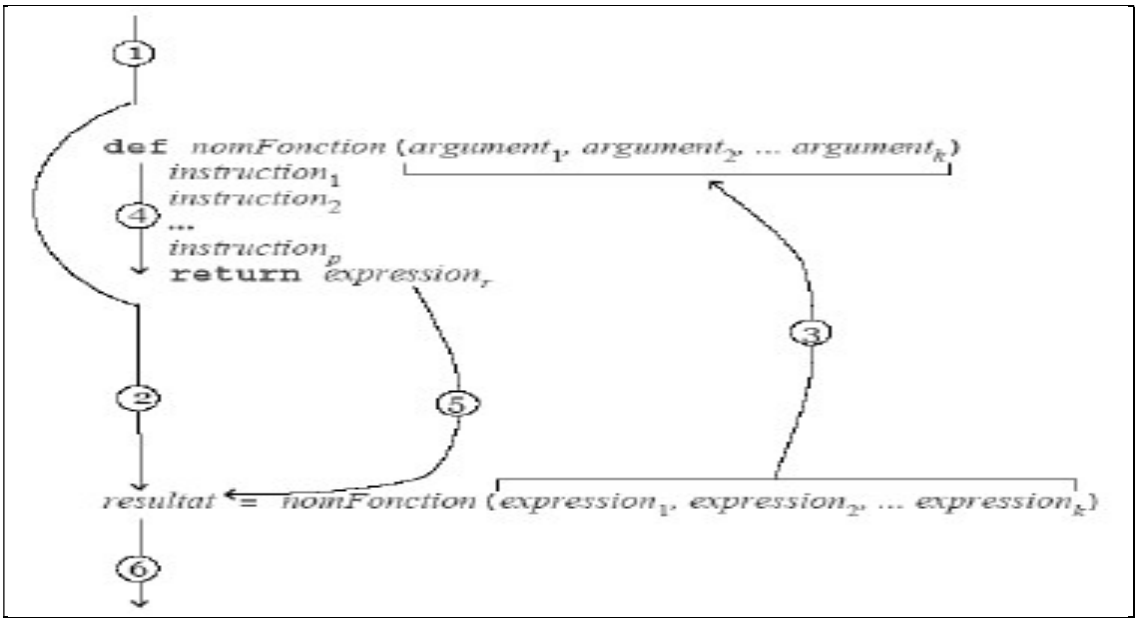
Exemple 2 :l'appel de la fonction double

```
x=double(3)
print(x)
```

Exemple 3 :l'appel de la fonction calcul

```
x,y=calcul(3,10)
print(" la somme est : ",x)
print(" le produit est : ",y)
```

III. Chronologie de l'exécution d'un script comportant une fonction



Exemple :
Définition de la fonction "somme " qui prend en paramètre un entier n et qui retourne la somme des entiers de 0 à n

```
def somme( n ):
    s=0
    for i in range(n+1) :
        s=s+i
    return s
```

un appel de la fonction "somme"

```
n=10
s = somme(n)
print(' la somme de 0 à ', n ,' est :', s )
```

un autre appel de la fonction "somme"

```
n=int(input('Entrer un entier'))
s = somme(n)
print(' la somme de 0 à ', n ,' est :', s )
```

IV. Valeurs par défaut :

On peut, à la fin de la suite des arguments préciser que certains arguments ont une valeur par défaut lorsque cette valeur n'est pas transmise

Exemple :

def somme(a=1,b=5) : return a+b s=somme(10,20) print(s) s=somme(10) print(s) s=somme() print(s)	le script affiche : 30 15 6

V. Arguments nommés :

Il est parfois commode de donner un nom aux différents arguments, cela permet d'appeler la fonction de façon assez naturelle et en transmettant les arguments dans un ordre quelconque.

Exemple :

def somme(a=1,b=5) : return a+b s=somme(b=20,a=10) print(s)	le script affiche : 30

VI. Variables locales et globales :

En Python, on distingue deux sortes de variables : les variables globales et les variables locales.

Les variables qui sont introduites dans la définition d'une fonction peuvent être utilisées dans la suite de la définition mais pas à l'extérieur de la fonction. Ces variables sont dites locales par opposition aux variables globales qui sont introduites à l'extérieur de la définition d'une fonction et qui peuvent être utilisées à l'intérieur comme à l'extérieur de cette définition.

Pendant l'exécution d'une fonction **NomFonction**, Python crée une table locale des symboles, où seront inscrits :

- les noms des arguments de la fonction **NomFonction**;
- les noms des variables et fonctions créées pendant l'exécution de la fonction **NomFonction**.

Au cours de l'exécution, si Python a besoin de la valeur d'une variable x , il va chercher le nom x dans la table locale des symboles ;

- si x ne s'y trouve pas, il consultera la table globale ;
- si x ne s'y trouve pas non plus, il enverra un message d'erreur.

Ainsi, s'il existe une variable locale et une variable globale portant le même nom, la variable globale ne sera pas accessible pendant l'exécution de la fonction

Exemple 1: dans ce programme : x est une variable globale et y est une variable locale :

x = 7 #variable globale
def f():
  y = 8 #variable locale
  return y+x
s= f()
print(s,x)
	le script affiche :
	15 7

Exemple 2:

def f(x):
  y=2
  return x + y
 	
print (f(3))
print(y)
	le script affiche :
	Traceback (most recent call last):
	  File "D:/CPGE/Python sources/varlocale.py", line 6, in <module>
	  print(y)
	NameError: name 'y' is not defined

VII. Module de fonctions :

- On peut ranger les définitions de fonctions se rapportant à une même application au sein d'un script commun baptisé **module**. Un module est sauvegardé sous forme d'un fichier dont le nom a la forme :

  Nom_du_module.py

- Pour utiliser un module, il faut se servir:
  - Soit de l'instruction : import **Nom_du_module**
  - Soit de l'instruction : from **Nom_du_module** import **NomFonction**

  L'exécution de cette instruction consiste à exécuter le script définissant le module (ce script peut contenir des instructions autres que des définitions de fonctions).
- Pour importer un module, Python a besoin de connaître le chemin qui permet d'accéder au fichier correspondant. Ce chemin doit apparaître dans la liste des chemins possibles stockés dans la variable **path** du module **sys**.

Exemple : Les étapes à suivre pour créer un module nommé calcul.py

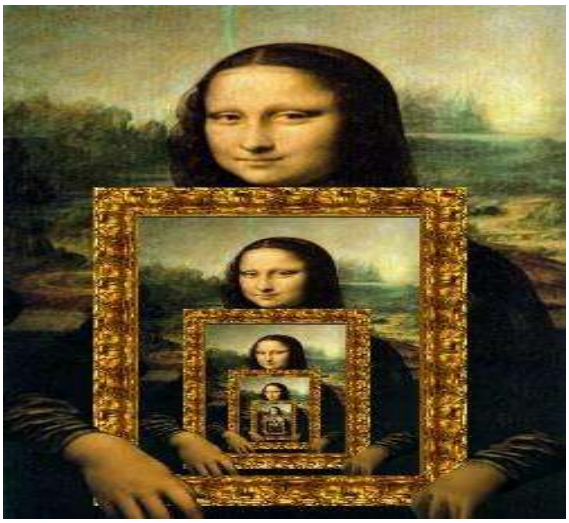
- définir les fonctions à utilisées

```
def somme(a,b) :
    return a+b

def Soust(a,b) :
    return a-b

def Mult(a,b) :
    return a*b
```

- Sauvegarder le fichier dans 'c:\ calcul.py'
- Importer la variable **path** du module **sys** : >>> **from sys import path**
- Ajouter le chemin de votre module dans la variable path : **>>>path=path+['c:\\']**
- Importer votre module : **>>>import calcul**
- Ensuite, pour utiliser les objets introduits dans le module, on les désigne par **Nom_du module.nom_de_objet** :
  >>> calcul.somme(3,7)
  >>>10
- Pour éviter de devoir préfixer les noms des objets par le nom du module
  - on peut pour l'importation utiliser l'instruction :
    >>> from calcul import somme
    >>> somme(3,7)
    >>>10
  - Pour importer tous les objets du module, on utilise l'instruction :
    >>> from calcul import *
    >>> somme(3,7)
    >>>10
    >>> Mult(3,7)
    >>>21

VIII. Les fonctions récursives

Une fonction récursive est une fonction dont la définition contient un (ou plusieurs) appel à elle-même.



En Python, toute fonction peut appeler toute fonction dont elle connaît le nom. En particulier, elle peut **s'appeler elle-même**. Il est donc possible d'écrire des fonctions récursives.

Prenons l'exemple le plus connu en matière de récursivité : la factorielle :

- **n! = n x (n-1) x (n-2) x (n-3) x ……x 2 x 1**
- **avec le cas particulier 0! = 1**

Cette fonction factorielle peut s'écrire de la manière suivante :

```
def factorielle(n):
    |
    |if (n == 0)
    |    |return 1
    |else:
    |    |return n* factorielle(n-1)
    |
```

Le résultat de chaque appel est stocké dans la pile. (PILE = zone mémoire accessible seulement en entrée (fonctionnement de type LIFO). LIFO = Last In First Out (dernier entré, 1er sorti)). Les appels s'arrêtent avec une « condition d'arrêt ». Les résultats empilés sont dépilés en ordre inverse.

Remarque :

Avant de rédiger une fonction récursive :

- on cherche d'abord une définition récursive
- on s'assure que cette définition possède une ''condition d'arrêt"
- et après seulement, on écrit le code correspondant