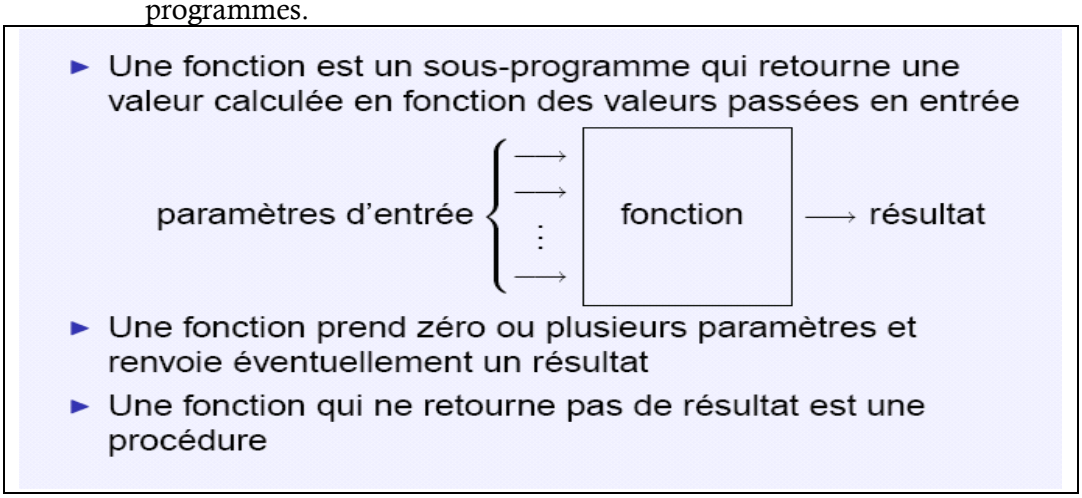
Chapitre N°1 : Fonctions et procédures

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules ;**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :
  - permettent de **"factoriser" les programmes**, càd de mettre en commun les parties qui se répètent ;
  - permettent une **structuration** et une **meilleure lisibilité** des programmes ;
  - **facilitent la maintenance** du code (il suffit de modifier une seule fois) ;
  - ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes.



1. Définition et appel d'une fonction en python

Définition d'une fonction :

```
def  nomdefonction ( argument1, argument2, … ,argumentk) :
    bloc_instructions
    return  valeur
```

Appel d'une fonction :

L'**appel de la fonction** prend la forme :

```
    nomdefonction( expression1, expression2, ... expressionk )
```

Exemple 1.1:

La fonction **sommeCarre** suivante retourne la somme des carrées de deux réels x et y :

```
def  sommeCarre ( x, y) :
    z = x**2 + y**2
    return z
```

L'appel de la fonction sommeCarre peut se faire :

```
>>> a=2
>>> b=3
>>> print(sommeCarre(a,b))
13
```

Exemple 2.1: la fonction suivante ne retourne pas de valeur, elle affiche le quotient et le reste de division de 2 nombres passés en paramètres.

```
def  division ( a, b) :
    q, r = a//b , a%b
    print('quotient de la division de ',a,' par ',b,' est ',q)
    print('Le reste de la division de ',a,' par ',b,' est ',r)
```

Remarque :

En Python, on peut utiliser les **tuples** pour renvoyer plusieurs valeurs.

```
def  division ( a , b ) :
    return a//b , a % b
```

L'appel de la fonction division peut se faire comme suit :

```
>>>a=5676
>>>b=668
>>>q , r = division ( a , b )
>>>print('Le quotient de la division euclidienne de ',a,' par ',b,' est ',q)
>>>print('Le reste de la division euclidienne de ',a,' par ',b,' est ',r)
```

2. Passage des paramètres en Python

Intéressons-nous au problème suivant : Si on envoie une variable en paramètre à une fonction ou une procédure, et qu'on la modifie dans la fonction, est-elle réellement modifiée après l'appel ?

Exemple 2.1 : Arguments de types immuables

Considérons la fonction **ajoute** suivante :

```
def   ajoute( a ) :
    a = a + 1
```

Elle devrait augmenter de 1 la valeur de son argument. Vérifions si c'est bien le cas.

```
# programme principal
b = 5
ajoute(b)
print(b)
```

En Python, le passage des paramètres est comparable à une affectation. Le fil d'exécution ressemble donc à ceci :

```
b = 5
# exécution d'ajout ( b )
a = b
a = a + 1
# retour au programme principal
print(b)
```

Sur cet exemple, il est évident que le programme affichera 5 (c'est **a** qui contient 6, pas **b**).
Tout se passe comme si le paramètre était passé par valeur : une copie du contenu de la variable est recopié de **b** vers **a** et c'est la copie qui est modifiée, pas l'original.
L'objet lié à **b** n'a donc pas changé (de toute façon, un entier est **immuable**) et **b** conserve sa valeur.
Ce mécanisme de transmission des arguments s'appelle le **passage par valeur**, car c'est seulement la valeur de **l'argument effectif** qui est transmise à la fonction appelée, et non **l'argument effectif** lui-même.

En ce qui concerne les paramètres, les paramètres de type non modifiable (immuable) sont passés par valeur (une modification à l'intérieur de la fonction n'a pas de répercutions à l'extérieur).

Exemple 2.2 : Arguments de type muable

Voyons maintenant ce qui se produit lorsqu'on passe une **liste** en paramètre (**argument de type muable**) :

```
def   ajoute_liste( L , v ) :
    L.append( v )
```

```
# programme principal
lst = [5 , 3 , 6 ]
v = 90
ajoute_liste( lst , v)
print( lst )
```

Nous posons toujours la même question : que va afficher le programme principal :
[5 , 3 , 6] ou **[5 , 3 , 6 , 90]**.
Là aussi (ceci est toujours valable), le passage des paramètres est comparable à une affectation. Aussi, la machine exécute :

```
lst = [5, 3, 6]
# exécution de ajoute_liste(lst,42)
L = lst
v = 90
L.append(v)
# retour au prog principal
print(lst)
```

Cette fois-ci, c'est **[5 , 3 , 6 , 90]** qui va s'afficher, comme si la fonction **ajoute_liste** avait pu modifier la **liste originale**.
La liste **lst** a bien été modifiée. C'est tout à fait normal. Lors de l'appel de la fonction, l'objet **muable** [5, 3, 6] est lié à l'argument formel **L**. Cet objet est modifié à l'intérieur de la fonction. Mais comme **lst** est toujours liée à ce même objet, sa valeur se trouve changée.

Les paramètres de type modifiable (muable) sont passés par référence (une modification à l'intérieur de la fonction a des répercutions à l'extérieur).

3. Variables globales et locales

En Python, on distingue deux sortes de variables : les **globales** et les **locales**.
Par exemple, dans le programme suivant, **x** est une variable **globale** :

```
x = 7
print(x)
```

À l'inverse, la variable **y** dans la fonction **f** suivante est **locale** :

```
def   f( ):
    y = 8
    return  y
```

Après l'appel de la fonction **f**, la variable **locale y** *disparaît.*
En particulier, l'instruction suivante échoue en indiquant que la variable **y** *n'est pas définie* :

```
>>>print(y)
```

On dit que la *portée* de la variable **y** est limitée au corps de la fonction **f**.

Cela signifie qu'elle a une *portée réduite* à la fonction f, et, surtout, qu'elle est créée chaque fois que la fonction f est appelée et détruite chaque fois que la fonction f se termine.

Les variables **globales**, elles, ont une portée qui s'étend généralement sur l'ensemble du programme.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est visible de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

Remarque 3.1
Si l'on veut accéder à une variable **globale** à l'intérieur d'une fonction, on utilise le mot-clé **global** en Python.

Exemple 3.1
Par exemple pour écrire une fonction qui réinitialise la variable globale **x** à 0, alors il ne faut pas écrire :

```
def  reinitialise( ):
     x = 0
```

On exécute le code suivant :

```
x =7
reinitialise( )
print(x)
```

On observe qu'il affiche toujours la valeur **7**. Pour que la fonction réinitialise puisse avoir accès à la variable globale x, il faut désigner cette dernière comme telle :

```
def  reinitialise( ):
     global x
     x = 0
```

Ainsi, le programme :

```
x = 7
reinitialise()
print(x)
```

Affiche maintenant **0** et non **7**.

De manière générale, si elle n'y est pas explicitement déclarée comme globale, une variable **x** est locale à la fonction dans le corps de laquelle elle est affectée. Elle est globale si elle est utilisée dans la fonction sans être affectée ou si elle est déclarée globale.

Exemple 3.2

```
def  f( ):
     global a
     a = a + 1
     c = 2 * a
     return  a + b + c
```

Dans cette fonction, **a** est **globale** car elle est déclarée comme telle, **b** est **globale** car elle est utilisée mais non affectée et **c** est **locale** car elle est affectée mais n'est pas déclarée globale.

4. Fonction anonyme (lambda function)

Le mot-clé **lambda** en Python permet la création de **fonctions anonymes** (i.e. sans nom et donc non définie par **def**)

```
>>> f = lambda  x :  x * x
```

```
>>> f ( 3 )
9
```

Ou peut également préciser plusieurs arguments, voire même des valeurs par défaut :

```
>>> g =  lambda  x , y = 2 :  x * y
>>> g(5, 7)
35
>>> g ( 6 )
12
```

Cela permet de définir des fonctions courtes simplement.

5. Fonctions comme valeurs de première classe

En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être passée en argument, renvoyée comme résultat ou encore stockée dans une variable. On dit que les fonctions sont des valeurs de première classe.

Une application directe est la définition d'un opérateur mathématique par une fonction.

Par exemple, la somme : $\sum_{i=1}^{n} f\,(i)$, pour une fonction ***f*** quelconque, peut être ainsi définie :

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

Pour calculer la somme des carrés des entiers de 1 à 10, on commence par définir une ***fonction carre***.

```
      def   carre ( x ) :
            return  x*x
```

Puis, on la passe en argument à la fonction somme_fonction :

```
      >>> somme_fonction ( carre , 10 )
      385
```

On peut même éviter de nommer la fonction carre, puisqu'elle est réduite à une simple expression, en utilisant une ***fonction anonyme***.

Ainsi, l'exemple précédent se réécrit-il plus simplement :

```
      >>> somme_fonction ( lambda  x:  x * x , 10 )
      385
```

6. Fonctions récursives

Une fonction est dite récursive si elle s'appelle elle-même : on parle alors d'appel récursif de la fonction.

Exemple 4.6 : Fonction factorielle

La fonction factorielle qui calcule le produit des n premiers entiers positifs (n!= $\prod_{k=1}^{n} k$) est simplement définie par la relation de récurrence :

$$\begin{cases} 0! = 1 \\ n! = n.(n-1)! \end{cases} \quad \forall n \in \mathbb{N}^*$$

Version itérative :	Version récursive :

`def factorielle(n) :`
`  f = 1`
`  for k in range(2,n+1) :`
`  f = f * k`
`  return f`
	`def factorielle(n) :`
	`  if n==0 :`
	`  return 0`
	`  else:`
	`  return n* factorielle(n-1)`