Chapitre 7 : Les Fonctions et les méthodes

I. Les Fonctions :

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques.

Il nous est arrivé de travailler avec plusieurs fonctions sans qu'on nous rendons compte exemple : print() , input() , int() , type() , str() , range() , … donc les fonctions sont des pseudo-programmes qu'on définit à l'avance et qui font un traitement spécifique qu'on va utiliser après dans notre programme principale.

En python comme dans d'autres langages de programmation il existe plusieurs fonctions . Donc **d'où vient ces fonctions ?**

En général, les fonctions proviennent d'au moins trois endroits :

• à partir de Python lui-même - de nombreuses fonctions (comme print ()) font **partie intégrante de Python** et sont toujours disponibles sans effort supplémentaire de la part du programmeur ; nous appelons ces fonctions des **fonctions intégrées** ;

• à partir des **modules préinstallés** de Python - de nombreuses fonctions, très utiles, mais utilisées beaucoup moins souvent que celles intégrées, sont disponibles dans un certain nombre de modules installés avec Python ; l'utilisation de ces fonctions nécessite quelques étapes supplémentaires de la part du programmeur afin de les rendre entièrement accessibles (nous vous en parlerons dans un moment);

• **directement à partir de votre code** - vous pouvez écrire vos propres fonctions, les placer dans votre code et les utiliser librement ;

• il existe une autre possibilité, mais elle est liée aux classes, nous allons donc l'omettre pour l'instant.

1- Définition d'une fonction :

Pour définir une fonction, Python utilise le mot-clé **def :**

```
def message():
    print("Hello world")
```

Si on souhaite qu'elle renvoie ou retourne un résultat on ajoute le mot clé **return** :

```
def calcul_somme(x,y):
    return x+y
```

Notez que la syntaxe de **def** utilise les deux points comme les boucles for et while ainsi que les tests if, un bloc d'instructions est donc attendu. De même que pour les boucles et les tests, l'indentation de ce bloc d'instructions (qu'on appelle le corps de la fonction) est obligatoire

2- L'invocation de la Fonction (Appel) :

La fonction se déclare généralement avant le programme principale et pour l'appeler il suffit d'invoquer son nom au sein du programme principale

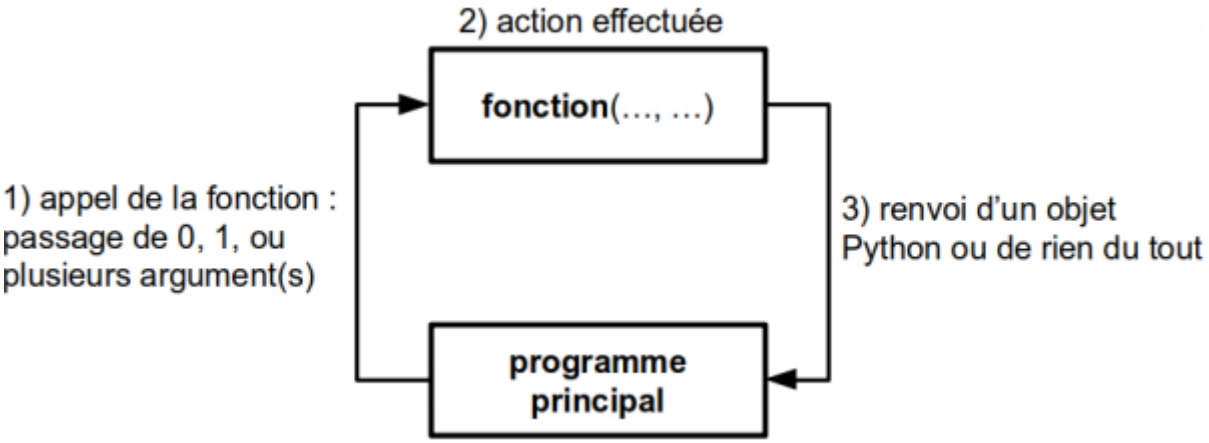
Exemple :

```
def message():
    print("Hello world")

def calcul_somme(x,y):
    return x+y

print("programme principale :")
message()
a=int(input("Entrez a :"))
b=int(input("Entrez b :"))
print("La somme est :",calcul_somme(a,b))
print("Bye")
```

Résultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Local\Programs\Python\Python38\serie_srepeti
ttives.py
programme principale :
Hello world
Entrez a : 5
Entrez b : 3
La somme est : 8
Bye
```

Comment ça fonctionne ?



Dans l'exemple précédent, nous avons passé un argument à la fonction calcul_somme() qui nous a renvoyé (ou retourné) une valeur que nous avons immédiatement affichée à l'écran avec l'instruction print(). Que veut dire valeur renvoyée ? Et bien cela signifie que cette dernière est récupérable dans une variable

Dans ce cas la fonction, message() se contente d'afficher la chaîne de caractères "Hello world " à l'écran. Elle ne prend aucun argument et ne renvoie rien. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction.

3- Passage d'arguments :

Le nombre d'arguments que l'on peut passer à une fonction est variable. Nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument. Dans les chapitres précédents, vous avez rencontré des fonctions internes à Python qui

prenaient au moins 2 arguments. Souvenez-vous par exemple de range(1, 10) ou encore range(1, 10, 2). Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui développe une nouvelle fonction. Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au « typage dynamique », c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution.

Exemple :

```
>>> def fois(x,y):
        return x*y

>>> fois(2,3)
6
>>>
>>> fois(3.1456,5.892)
18.5338752
>>>
>>> fois('to',5)
'tototototo'
>>>
>>> fois([1,3],2)
[1, 3, 1, 3]
```

L'opérateur * reconnaît plusieurs types (entiers, floats, chaînes de caractères, listes). Notre fonction fois() est donc capable d'effectuer des tâches différentes.

4- Renvoi des resultats :

Un énorme avantage en Python est que les fonctions sont capables de renvoyer plusieurs objets à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
        return x**2,x**3

>>> carre_cube(2)
(4, 8)
```

En réalité Python ne renvoie qu'un seul objet, mais celui-ci peut être séquentiel, c'est-à-dire contenir lui-même d'autres objets. Dans notre exemple Python renvoie un objet de type tuple, type que nous verrons plus tard.

Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube(x):
        return [x**2,x**3]

>>> carre_cube(5)
[25, 125]
```

Remarque 1 :

si une fonction n'est pas destinée à produire un résultat, l' **utilisation de l'** return **instruction n'est pas obligatoire** - elle sera exécutée implicitement à la fin de la fonction.

Quoi qu'il en soit, vous pouvez l'utiliser pour **terminer les activités d'une fonction à la demande** , avant que le contrôle n'atteigne la dernière ligne de la fonction.

Remarque 2 :

On peut utiliser le mot clé return pour interrompre l'exécution d'une fonction dans ce cas il suffit d'écrire return ou return 0

Exemple :

```
def happy_new_year(corona):
    print("trois..")
    print("deux..")
    print("Un..")
    if not corona:
        return
    print("Happy New Year")

reponse=input("Est ce qu'il ya toujours corona ?")
if reponse =="OUI":
    corona=False
else:
    corona=True
happy_new_year(corona)
```

5- Arguments positionnels et arguments par mot-clé :

- **Arguments positionnels :**

Jusqu'à maintenant, nous avons systématiquement passé le nombre d'arguments que la fonction attendait. Que se passe-t-il si une fonction attend deux arguments et que nous ne lui en passons qu'un seul ?

```
>>> def fois(x,y):
        return x*y

>>> fois(2,3)
6
>>> fois(5)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    fois(5)
TypeError: fois() missing 1 required positional argument: 'y'
```

On constate que passer un seul argument à une fonction qui en attend deux conduit à une erreur.

Définition : Lorsqu'on définit une fonction def fct(x, y): les arguments x et y sont appelés arguments positionnels (en anglais positional arguments). Il est strictement obligatoire de les préciser lors de l'appel de la fonction. De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction. Dans l'exemple ci-dessus, 2 correspondra à x et 3 correspondra à y. Finalement, tout dépendra de leur position, d'où leur qualification de positionnel.

- **Argument avec mot-clé :**

Python propose une autre convention pour le passage d'arguments, où la signification de l'argument est dictée par son nom, et non par sa position - c'est ce qu'on appelle le passage d'argument par mot clé.

Exemple :

```
def Introduction(firstname , lastname):
    print("Bonjour Je suis :",firstname,lastname)


print("Programme Principale")
Introduction("joairia","lafhal")
Introduction(firstname="joairia",lastname="lafhal")
Introduction(lastname="lafhal",firstname="joairia")
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Loc
ttives.py
Programme Principale
Bonjour Je suis : joairia lafhal
Bonjour Je suis : joairia lafhal
Bonjour Je suis : joairia lafhal
```

Le concept est clair - les valeurs transmises aux paramètres sont précédées du nom des paramètres cibles, suivi du signe =.

La position n'a pas d'importance ici - la valeur de chaque argument connaît sa destination sur la base du nom utilisé.

Remarque : je dois respecter l'écriture du mot-clé comme je l'ai écrit dans la définition du la fonction je dois l'écrire dans l'appel

- **Mélanger les arguments positionnels et avec mot clé :**

Vous pouvez mélanger les deux modes si vous le souhaitez - il n'y a qu'une seule règle incassable: **vous devez mettre les arguments positionnels avant les arguments mot-clé**.

Pour vous montrer comment cela fonctionne, nous utiliserons la fonction simple à trois paramètres suivants :

```
def adding(a,b,c):
    print(a,"+",b,"+",c,"=",a+b+c)
```

Son but est d'évaluer et de présenter la somme de tous ses arguments.

La fonction, lorsqu'elle est invoquée de la manière suivante :

adding(1, 2, 3)

affichera :

1 + 2 + 3 = 6

C'était - comme vous pouvez le soupçonner - un pur exemple de **passage d'arguments positionnels**.

Bien sûr, vous pouvez remplacer une telle invocation par une variante purement mot-clé, comme celle-ci :

adding(c = 1, a = 2, b = 3)

Notre programme affichera une ligne comme celle-ci :

2 + 3 + 1 = 6

Notez l'ordre des valeurs.

Essayons maintenant de mélanger les deux styles.

Regardez l'invocation de fonction ci-dessous :

adding(3, c = 1, b = 2) Analysons-le:

• l'argument (3) pour le paramètre a est passé en utilisant la manière positionnelle ;

• les arguments pour c et b sont spécifiés comme mots-clés.

Voici ce que vous verrez dans la console :

3 + 2 + 1 = 6

Soyez prudent et méfiez-vous des erreurs. Si vous essayez de passer plusieurs valeurs à un seul argument, vous n'obtiendrez qu'une erreur d'exécution.

Regardez l'invocation ci-dessous - il semble que nous ayons essayé de définir a deux fois :

```
>>> adding(3,a=2,b=1)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    adding(3,a=2,b=1)
TypeError: adding() got multiple values for argument 'a'
```

6- Les fonctions paramétrées (valeur par défaut) :

Il arrive parfois que les valeurs d'un paramètre particulier soient utilisées plus souvent que d'autres. Ces arguments peuvent avoir leurs **valeurs par défaut (prédéfinies)** prises en considération lorsque leurs arguments correspondants ont été omis.

Exemple :

```
def Introduction(lastname, firstname ="Mohamed"):
    print("Bonjour Je suis :",firstname,lastname)


print("Programme Principale")
Introduction("Lafhal")
Introduction("lafhal","joairia")
Introduction(firstname="joairia",lastname="lafhal")
Introduction(lastname="lafhal")
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppDat
ttives.py
Programme Principale
Bonjour Je suis : Mohamed Lafhal
Bonjour Je suis : joairia lafhal
Bonjour Je suis : joairia lafhal
Bonjour Je suis : Mohamed lafhal
```

On remarque que lorsqu'on appelé la fonction avec un seul paramètre ça n'a pas créé des problèmes car python a déjà une valeur par défaut pour le 2ème argument .

7- Les variables locales et globales :

Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite locale lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction. Une variable est dite globale lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.

Exemple :

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

print("Programme principale")
z= int(input("Entrez un nombre :" ))
print(x)
y=carre(x)
print(y)
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Local\P
ttives.py
Programme principale
Entrez un nombre :5
Traceback (most recent call last):
  File "C:\Users\TOSHIBA\AppData\Local\Prog
ves.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
```

Python ne connait pas la variable **x** car elle est créé dans la fonction et manipulée dans la fonction est détruite une fois le traitement de la fonction est terminé donc les variables locales n'ont aucun rôle en dehors la fonction.

Par contre les variables **z** et **y** sont des variables global elles sont connu partout dans le programme principale et même dans une fonction

Exemple :

```
def carre(x):
    print(a)
    return x**2

a=5
print("Programme principale")
z= int(input("Entrez un nombre :" ))
y=carre(z)
print(y)
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Local\Pr
ttives.py
Programme principale
Entrez un nombre :6
5
36
```

Remarque : Python reconnait les variables globales dans la fonction mais on ne peut pas modifier la valeur de la variable dans le programme principale

Exemple :

```
def carre(x):

    print("Incrementation de a dans la fonction :")
    a+=1
    print(a)
    return x**2

a=5
print("Programme principale")
z= int(input("Entrez un nombre :" ))
y=carre(z)
print("a=",a)
print("y=",y)
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Local\Programs\Python\Python38\
ttives.py
Programme principale
Entrez un nombre :6
Incrementation de a dans la fonction :
Traceback (most recent call last):
  File "C:\Users\TOSHIBA\AppData\Local\Programs\Python\Python38\ser
ves.py", line 11, in <module>
    y=carre(z)
  File "C:\Users\TOSHIBA\AppData\Local\Programs\Python\Python38\ser
ves.py", line 4, in carre
    a+=1
UnboundLocalError: local variable 'a' referenced before assignment
```

- **Le mot clé « global » :**

Il existe une méthode Python spéciale qui peut **étendre la portée d'une variable d'une manière qui inclut les corps des fonctions** (même si vous voulez non seulement lire les valeurs, mais aussi les modifier).

Un tel effet est provoqué par un mot-clé nommé global :

global name

global name1, name2, ...

L'utilisation de ce mot-clé dans une fonction avec le nom (ou les noms séparés par des virgules) d'une ou de plusieurs variables, force Python à s'abstenir de créer une nouvelle variable à l'intérieur de la fonction - celle accessible de l'extérieur sera utilisée à la place.

En d'autres termes, ce nom devient global (il a une **portée globale** et peu importe qu'il fasse l'objet d'une lecture ou d'une affectation).

Exemple :

```
def carre(x):
    global a
    print("Incrementation de a dans la fonction :")
    a+=1
    print(a)
    return x**2

a=5
print ("a avant fonction : ",a)
z= int(input("Entrez un nombre :" ))
y=carre(z)
print("a apres la fonction ",a)
print("y=",y)
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Local\Prog
ttives.py
a avant fonction :  5
Entrez un nombre :6
Incrementation de a dans la fonction :
6
a apres la fonction  6
y= 36
```

8- La valeur None :

Il n'y a que deux types de circonstances qui None peuvent être utilisées en toute sécurité :

• lorsque vous l'**affectez à une variable** (ou le retournez comme résultat d'une **fonction**)

• lorsque vous le **comparez à une variable** pour diagnostiquer son état interne.

Comme ici:

```
value = None
if value is None:
print("Sorry, you don't carry any value")
```

N'oubliez pas ceci : si une fonction ne retourne pas une certaine valeur en utilisant une return clause d'expression, on suppose qu'elle **renvoie implicitement** None. (On peut l'utiliser dans l'exercice 6 pour tester si a et b sont saisis)

9- Les Fonctions récursives :

En mathématiques, une suite (Un) n∈N est récurrente lorsque le terme un+1 est une fonction du terme un. En informatique, une fonction f est récursive lorsque la définition de f utilise des valeurs de f. Chaque fonction récursive est construite sur une relation de récurrence.

Exemple :

Le factoriel : La fonction factorielle est une fonction récursive car elle appelle elle-même dans le traitement

Si on souhaite calculer le factoriel de 5 c'est

5 != 5*4*3*2*1

Et 4*3*2*1 c'est le factoriel de 4

Donc 5 !=5*4 !

Et donc 5 !=5*4*3 ! est ça continue comme ça donc en peut écrire

Exemple :

```
def factoriel(x):
    if x==0:
        return 1
    return x*factoriel(x-1)

print("Programme principale :" )
a=int(input("Entrez un nombre :"))
print("le resultat est :",factoriel(a))
```

Résultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Loca
ttives.py
Programme principale :
Entrez un nombre :5
le resultat est : 120
```

10- Passage par adresse et passage par valeur :

En python c'est toujours le passage des paramètres est par valeur pour les variables ça se fait automatiquement donc on n'aura pas de problèmes en python par contre il faut faire attention dans les autres langages de programmation

Exemple :

```
def f(x):
    x+=1
    print("dans la fonction:",x)

print("Programme principale :" )
a=2
print("a avant la fonction:",a)
f(a)
print("a apres la fonction:",a)
```

Resultat :

```
= RESTART: C:\Users\TOSHIBA\AppData\Lo
ttives.py
Programme principale :
a avant la fonction: 2
dans la fonction: 3
a apres la fonction: 2
```

On remarque que la modification de la variables ne se propage pas en dehors de la fonction est donc le passage des variables en parametre se fait automatiquement par valeur

11- Exercices

Exercice 1 :

créez une fonction gen_pyramide() à laquelle vous passez un nombre entier N et qui renvoie une pyramide de N lignes sous forme de chaîne de caractères. Le programme principal demandera à l'utilisateur le nombre de lignes souhaitées (utilisez pour cela la fonction input()) et affichera la pyramide à l'écran.

Exercice 2 :

Ecrire une fonction qui accepte comme paramètre le jour, le mois et l'année qui détermine si la date est correcte.

Exercice 3 :

Ecrire une fonction **chiffrePorteBonheur(nb)** qui permet de déterminer si un nombre entier **nb** est chiffre porte Bonheur ou non.

Un nombre **chiffre porte Bonheur** est un nombre entier qui, lorsqu'on ajoute les **carrés** de **chacun** de ses chiffres, puis les carrés des chiffres de ce **résultat** et ainsi de suite jusqu'à l'obtention d'un nombre à un seul chiffre égal à 1 (un).

Le calcule s'arrête lorsque le chiffre devient inférieur à 10

le résultat doit être comme suit :

```
Nombre de départ: 913
9^2=81
1^2=1
3^2=9
Nouveau: 81+1+9=91
9^2=81
1^2=1
Nouveau: 81+1=82
8^2=64
2^2=4
Nouveau: 64+4=68
6^2=36
8^2=64
Nouveau: 36+64=100
1^2=1
0^2=0
0^2=0
Nouveau: 1+0+0=1
Le chiffre: 913 est un porte boneur
```

Exercice 4 :

Définir une fonction qui renvoie les nombres parfaits qui se trouve entre deux nombres entiers a et b. tel que a < b.

Exercice 5 :

Définir une fonction qui calcul le nombre des mots dans une phrase passée en paramètre