Fonctions

Ce chapitre a pour objectif de présenter les fonctions ainsi que leur utilisation sous Python.

I. Principe

Dès qu'un programme devient relativement volumineux, des blocs d'instructions identiques peuvent être exploités à divers endroits. Afin de rationaliser l'écriture du code, on regroupe ces instructions au sein de blocs identifiés par un nom: les fonctions.

II. Syntaxe

La syntaxe de la déclaration d'une fonction est la suivante:

```
def nom_fonction(parametre1, parametre2, ... , parametreN):
     # Corps de la fonction
```

Le mot-clé def est utilisé par Python pour indiquer la définition d'une fonction. Les paramètres sont utilisés pour pouvoir modifier le comportement de la fonction lors de son appel par le programme principal (ou par une autre fonction). Certaines fonctions n'utilisent pas de paramètre et la liste est vide.

Exemple 1:

Supposons qu'un programme Python nécessite d'afficher plusieurs fois les puissances de 2. Le code original correspondant est le suivant:

```
# Ensemble d'instructions 1
print("La liste des puissances de 2 est la suivante:"
for i in range(8):
     print("2^",i,"=",2**i)
# Ensemble d'instructions 2
print("La liste des puissances de 2 est la suivante:"
for i in range(8):
     print("2^",i,"=",2**i)
# Ensemble d'instructions 3
# Ensemble d'instructions 4
print("La liste des puissances de 2 est la suivante:"
for i in range(8):
     print("2^",i,"=",2**i)
```

On peut simplifier son écriture en utilisant une fonction appelée aff_puiss:

```
def aff_puiss():
     print("La liste des puissances de 2 est la suivante:")
     for i in range(8):
          print("2^",i,"=",2**i)
# Ensemble d'instructions 1
aff_puiss()
# Ensemble d'instructions 2
aff_puiss()
# Ensemble d'instructions 3
# Ensemble d'instructions 4
aff_puiss()
```

Comme on peut le voir, le code est beaucoup plus lisible.

Un autre intérêt des fonctions est qu'elles permettent de réduire un gros problème en plusieurs petits problèmes beaucoup plus simples à résoudre.

Exemple 2:

La fonction aff_puiss() est très pratique dans le cas de l'exemple précédant mais on peut l'améliorer. Il est très simple de modifier la base en utilisant un paramètre.

La définition de la fonction pourrait être la suivante:

```
def aff_puiss(base):
     print("Les puissances de ",base," sont:")
     for i in range(8):
          print(base,"^",i,"=",base**i)
```

Pour afficher les puissances de 3, il suffit d'inclure la ligne suivante dans le programme principal: `aff_puiss(3)`

Exemple 3:

Jusqu'à maintenant, notre fonction affichait les 8 premières puissances du nombre passé en paramètre. Nous allons ajouter un paramètre qui indique le nombre de puissances que l'on souhaite afficher. La définition de la fonction sera la suivante:

```
def aff_puiss(base,nb_puiss):
     print("Les puissances de ",base," sont:")
     for i in range(nb_puiss):
          print(base,"^",i,"=",base**i)
```

Ainsi, pour afficher les dix premières puissances de 7, on exploite la commande aff_puiss(7,10). On remarque que l'ordre des paramètres est très important. En effet, si on saisit aff_puiss(10,7), on affiche les 7 premières puissances de 10.

Exemple 4:

Si, la plupart du temps, on affiche les huit premières puissances du nombre base, il serait souhaitable de modifier la fonction aff_puiss et d'incorporer une affectation par défaut de nb_puiss à 8. Cela peut être réalisé de la façon suivante:

```
def aff_puiss(base,nb_puiss=8):
     print("Les puissances de ",base," sont:")
     for i in range(nb_puiss):
          print(base,"^",i,"=",base**i)
```

Ainsi, aff_puiss(3,8) et aff_puiss(3) afficheront le même résultat.

Exemple 5:

Une étude plus poussée a montré que la fonction aff_puiss est quasiment toujours utilisée pour afficher les 8 premières puissances de 2. Il devient donc intéressant d'utiliser une double affectation par défaut: l'une pour la base et l'autre pour nb_puiss. Le code source de la fonction devient le suivant:

```
def aff_puiss(base=2,nb_puiss=8):
     print("Les puissances de ",base," sont:")
     for i in range(nb_puiss):
          print(base,"^",i,"=",base**i)
```

Ainsi, les commandes aff_puiss() et aff_puiss(2,8) auront le même effet.
Il est aussi possible d'utiliser les appels suivants:
* aff_puiss(3) ou aff_puiss(base=3) : équivalents à aff_puiss(3,8),
* aff_puiss(nb_puiss=5): équivalent à aff_puiss(2,5).

III. Clause return

Jusque là, nos fonctions ne renvoyaient rien: elles se contentaient d'afficher du texte. La clause return permet d'enrichir nos fonctions en les autorisant à exporter un résultat.

L'utilisation de return est simple car il suffit de saisir return resultat_à_renvoyer.

Voyons son exploitation à l'aide de l'exemple 6:

```
def factoriel(n):
    resultat=1
    for i range(1,n+1):
        resultat=resultat*i
    return resultat
```

Cette fonction renvoie la valeur du factoriel du nombre n passé en paramètre. Dans le programme principal (ou dans une autre fonction), supposons que l'on souhaite récupérer

et afficher la valeur du factoriel de 5, il suffit d'utiliser les lignes suivantes:

```
temp=factoriel(5)
print("Le factoriel de 5 est ",temp)
```

Il aussi possible de saisir, plus simplement, print("Le factoriel de 5 est ",factoriel(5)).

IV. Passage de paramètres

Python utilise le passage par valeurs : un paramètre ne peut être modifié par une fonction.

Ainsi, le code suivant affichera deux fois le chiffre 5:

```
def tripler(x):
     x=x*3

truc=5
print(truc)
tripler(truc)
print(truc)
```

Par contre, les listes ne sont pas clonées donc toute modification au sein de la fonction aura un impact sur le programme principal (ou sur la fonction appelante).

Par exemple, à la fin de l'exécution du programme suivant, on aura tableau=[1,2,4].

```
def modif_liste(tab):
    tab[0]=1;tab[1]=2;tab[2]=4


tableau=[7,8,9]
modif_liste(tableau)
```