TP n° 3 : Rappel sur les fonctions en Python

1. Un exemple

Il arrive souvent qu'un script fasse appel plusieurs fois à un même bloc d'instructions :

Exemple : Soit $f : \begin{cases} \mathbb{R} \mapsto \mathbb{R} \\ x \mapsto x^2 - 10x + 1 \end{cases}$

Pour remplir le tableau de valeurs suivant :

x	1	2	3	4	5	6	7	8	9
$f(x)$									

On pourrait le faire ainsi :

```
>>> x=1
>>> x**2-10*x+1
-8
>>> x=2
>>> x**2-10*x+1
-15
>>> x=3
>>> x**2-10*x+1
-20
```

et continuer ainsi jusqu'à **x = 9**.

On préférera l'approche suivante :

```
>>> def  f(x):
...         return x**2-10*x+1
...
>>> for i in range(1,10):
...         print(f(i),end=" ")
...
-8 -15 -20 -23 -24 -23 -20 -15 -8
```

Nous avons donc utilisé ce que l'on appelle une **fonction** : une même instruction étant répétée plusieurs fois, nous l'avons isolée et lui avons donné un nom.

Cette fonction peut être appelée par ce nom à n'importe quel endroit du programme et autant de fois que l'on veut

2. Fonctions

Définition : Une fonction est définie par :

- son **nom**, que l'on choisira suffisamment explicite
- ses **arguments formels** (ou **paramètres**) : ce sont des variables utilisées dans le corps de la fonction, et dont la valeur est donnée au moment où la fonction est appelée
- éventuellement une **valeur de retour**, communiquée au programme par la fonction en fin d'exécution.

Dans l'exemple précédent, le **nom** de la fonction était **f**, son **argument x** et la **valeur de retour x**2-10*x+1**.

Une fonction est caractérisée par sa **signature** (ou en-tête), c'est-à-dire son nom suivi de la liste des paramètres formels (ou arguments) de la fonction.

Remarque : print vs **return**

```
>>> def  f(x):
...         print( x**2-10*x+1)
...
>>> f(1)+f(2)+f(3)
-8
-15
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'

>>> def  g(x):
...         return  x**2-10*x+1
...
>>> g(1)+g(2)+g(3)
-43
```

print ne constitue qu'une commande d'affichage, alors que nous aurons souvent besoin de réutiliser le résultat d'une fonction. On utilisera pour cela la commande **return**.

On évitera d'utiliser la commande **print** à l'intérieur d'une fonction (sauf si on vous le demande). La commande **print** reste utile, mais surtout dans les scripts, à l'extérieur des fonctions

Remarque : Dans une fonction, tout s'arrête après l'instruction return: inutile de placer des instructions après un return !

```
>>> def g(x):
...         print("Les instructions avant le return sont effectuées")
...         return x**2-10*x+1
...         print("Le reste est inutile !")
...
>>> g(2)
Les instructions avant le return sont effectuées
-15
```

Remarque : Une fonction sans **return**, exécute un bloc d'instructions sans retourner de valeur autre que None. On parle de **procédure**.

Portée des variables

Les variables utilisées dans une fonction sont classées en deux catégories :

- les variables **locales** : ce sont les arguments formels de la fonction, ainsi que les variables qui sont affectées au cours de la fonction

- les variables **globales** : ce sont les variables qui apparaissent dans le corps de la fonction sans être affectées (mais qui sont utilisées par exemple lors de calculs).

```
>>> x=3
>>> def f(x):                    #définition de la fonction f
...         x=x+1
...         y=x**2                 # y est une variable locale
...         return y**2-4*y+12
...
>>> f(1)                         # appel de la fonction f
12
>>> x
3
>>> y
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Remarque : Faire très attention avec les objets de type "**modifiable**" en Python (notamment les listes) :

```
>>> def   echange_debut_fin(t):
...         """Echange le premier et le dernier element de la liste t"""
...         t[0], t[-1] = t[-1], t[0]
...
>>> x = [ 10, 20, 30, 40, 50 ]
>>> echange_debut_fin(x)
>>> x
[50, 20, 30, 40, 10]
```

3. Exercices

Exercice 1. Écrire une fonction **somme_chiffres(n)** qui calcule, pour $n \in IN$, la somme des chiffres (en écriture décimale) de l'entier n.

Exercice 2. Écrire une fonction **premier_plus_grand(M)** prenant en entrée un réel M et renvoyant le plus petit entier $n \in IN$ tel que $n4 - 5n3 + 4 \geq M$.

Exercice 3. Ecrire une fonction **parfait(n)** qui vérifie si l'entier n est parfait ou non. Un entier est dit parfait s'il est égal à la somme de ses diviseurs.

Exemple: 6 = 3 + 2 +1

Exercice 4. La distance de Hamming entre deux chaines de caractères de mêmes longueurs est égale au nombre de caractères, à la même position, qui sont différents.

Exemple :

La distance de Hamming entre « sure » et « cure » est **1**.

La distance de Hamming entre « aabbcc » et « xaybzc » est **3**.

Ecrire une fonction **distanceH(s1,s2)** qui calcule et retourne la distance de **Hamming** entre deux chaînes de caractères s1 et s2.

Exercice 5. Ecrire une fonction **supprime_occurence(s,c)** qui permet de supprimer toutes les occurrences du caractère c dans une chaîne de caractère s.

Exemple : si on a une chaîne de caractères s='agadir' et qu'on supprime toutes les occurrences de 'a', la fonction doit retourner le résultat :'gdir'

Exercice 6. Ecrire une fonction **recherche_mot(m,t)** qui, étant donné deux tableaux m et t, permet de déterminer la position de la première occurrence de m dans t, s'elle existe, et qui renvoie None sinon.

Exemple : si on m =[1,2,3] et t=[2,1,4,1,2,6,1,2,3,7] la fonction **recherche_mot** renvoie 6 : [2,1,4,1,2,6,1,2,3,7]

Exercice 7. On considère le fichier *mots.txt*, qui contient sur chaque ligne un mot.
Ecrire une fonction :

- **nombre_mots(chemin_fichier)** qui détermine le nombre de mots dans le fichier.
- **mot_plus_long(chemin_fichier)** qui détermine le mot le plus long dans ce fichier.
- **extraction7(chemin_fichier)**qui extrait tous les mots de 7 lettres du fichier mots.txt et les écrit dans un fichier mots7.txt.

Exercice 8. Écrire une fonction itérative **facto(n)** prenant en entrée un entier n et qui calcule sa factorielle en utilisant une boucle **for**: $n! = 1 \times 2 \times \ldots \times (n-1) \times n$.

4. Fonctions récursives : Des fonctions qui s'appellent elles-mêmes

Exemple : $n! = 1 \times 2 \times \ldots \times (n-1) \times n$.

```
def facto_rec(n):
    assert n >= 0
    if n==0:
        return 1
    else:
        return n*facto_rec(n-1)
```

Exercice 9. Écrire une fonction itérartive **puissance(x,n)** prenant en entrée un entier n et un réel x et qui calcule x^n

Écrire une fonction récursive **puissance_rec(x,n)** prenant en entrée un entier n et un réel x et qui calcule x^n

Exercice 10. On considère la suite définie par $u_0=2$, $u_n=3*u_{n-1}-1$.

- Ecrire une fonction itérative qui prend comme paramètre un entier naturel n et calcule le terme u_n de la suite en utilisant une boucle **while**.
- Ecrire une fonction récursive qui prend comme paramètre un entier naturel n et calcule le terme u_n de la suite.

Exercice 11. Écrire une fonction récursive qui calcule le quotient de la division euclidienne d'un nombre entier naturel par un autre, bien entendu sans utiliser l'opération **/** du langage.