VI – Fonctions et procédures

Jusqu'ici, on a écrit des programmes qui traitent des données fournies par l'utilisateur. Pour exécuter des tâches plus complexes, il faut que les données traitées par des programmes soient directement utilisées par d'autres programmes, sans aucune intervention humaine. Les fonctions et les procédures vont nous permettre de DONNER VIE À CETTE ENVIE !

Ces structures fondamentales sont, en quelque sorte, des sous-programmes qui traitent les données qu'on leur fournit en entrée. La différence fondamentale réside dans le fait que les fonctions possèdent des valeurs de sortie, contrairement aux procédures.

1/ Notion de fonction

1.1) Généralités

Définition

*Une **fonction** est une suite d'instructions plus ou moins complexes.*
*Elle peut recevoir des paramètres à traiter, que l'on appelle **arguments**.*
*Elle renvoie un résultat, que l'on appelle **valeur de la fonction**.*

Remarque : cela correspond bien à la notion de fonction vue en mathématiques.

Intérêt des fonctions

- En mathématiques, une fonction est un raccourci qui permet de simplifier les notations. C'est exactement la même chose en informatique : quand une même suite d'instructions est utilisée à plusieurs reprises dans le programme, l'utilisation d'une fonction permet de simplifier le code et de le rendre plus lisible.

  On parle alors de **factorisation de code**.
- Ainsi, si l'on doit corriger la fonction, il suffit de modifier sa définition sans avoir à rechercher toutes ses utilisations dans le programme.

  C'est très pratique lorsque l'on procède à des essais et que l'on ne sait pas encore si la méthode employée est la meilleure.
- Les fonctions sont également faciles à lancer et à exécuter depuis la console, que ce soit pour une utilisation ou des tests en cours de développement.
- Enfin, quand on traite un problème complexe, on découpe le programme en plusieurs fonctions que l'on écrit et teste séparément. Cela permet de procéder par étapes, de planifier les tâches et même de travailler en équipe.

Inutile de préciser qu'à partir de maintenant, nous recourrons massivement aux fonctions !

1.2) Écriture en Python

En Python, une fonction s'écrit

```
def fonction(arguments):
    bloc d'instructions
    return résultat
```

On a deux nouveaux mot-clefs ici :

- `def` définit le nom de la fonction et les arguments utilisés.
- `return` définit la valeur de retour de la fonction et **interrompt son exécution**.

La fonction peut utiliser plusieurs arguments qui peuvent être de n'importe quel type, tout comme sa valeur de retour. Elle peut aussi n'admettre aucun argument.

Exemple : voici la valeur absolue et l'addition de deux vecteurs du plan, entre autres ...

```
def val_abs(x):
    if x>0:
        return x
    return -x
```

```
def add_vect(u,v):
    (a,b)=u
    (c,d)=v
    return (a+c,b+d)
```

```
def Cesar():
    return 'Ave !'
```

Pour exécuter une fonction, il suffit de taper `fonction(valeurs)`, en fournissant en entrée les arguments attendus. On obtient alors la valeur de retour, qui s'utilise comme n'importe quelle donnée. **Attention toutefois à l'ordre et au type des arguments.**

Exemples :

1) On peut taper par exemple `a=val_abs(-4)` ou bien `add_vect((3,1),(7,4))`.
2) L'instruction `val_abs("t")` provoque une erreur, de même que `add_vect(7,2,4)`.

Remarque : la variable x utilisée dans la fonction `val_abs` n'a pas de sens en dehors de cette fonction (Python va nous dire qu'elle n'est pas définie). On dit que c'est une **variable locale**.

1.3) Documentation

On peut insérer de la documentation dans une fonction en insérant du texte entre triple guillemets (ou bien triple apostrophe) juste après la ligne du `def`.

- Cela permet de placer des informations facilement lisibles dans le code.
- On récupère ces informations en tapant `help(fonction)` dans la console.

Exemple :

```
def somme(liste):
    """
    calcule la somme des éléments d'une liste de nombres
    """
    somme=0
    for e in liste:
        somme=somme+e
    return somme
```

2/ Notion de procédure

Définition

*Une **procédure** est une fonction sans valeur de retour, mais avec en général un **effet de bord**. Ceci signifie qu'elle modifie quelque chose en dehors de la procédure, par exemple :*

- *en provoquant un affichage ;*
- *en transformant des données déjà existantes.*

Remarque : quand on affiche le résultat d'une procédure avec `print()`, on obtient `None`.

Exemples : voici deux procédures utiles au quotidien.

```
def table7():
    for n in range(1,11):
        print('7 * ',n,'=',7*n)
```

```
def tablemulti(b,d,f):
    for n in range(d,f+1):
        print(b,'*',n,'=',b*n)
```

Comme les procédures et les fonctions ont des syntaxes extrêmement proches, on parlera simplement de fonctions dans tous les cas. Les fonctions ont quand même un gros avantage : **la valeur de retour permet d'affecter facilement le résultat des opérations à une variable.**

3/ Portée des variables

3.1) Utilisation de variables à l'intérieur d'une fonction

Définition

*Lorsque l'on définit une variable à l'intérieur d'une fonction, on obtient une **variable locale**. Elle n'existe qu'à l'intérieur de cette fonction et n'est plus accessible en dehors de celle-ci.*

Remarque : les variables locales sont les homologues des **variables muettes** présentes dans des expressions comme $\sum_{k=0}^{9} 2^k = 1023$, $\int_0^1 3t^2 \,\mathrm{d}t = 1$ ou bien $\lim_{x\to+\infty} \arctan(x) = \pi/2$.

Définition *Une variable définie en dehors de toute fonction est une **variable globale**.*

Remarque : on peut utiliser cette variable à l'intérieur d'une fonction, mais pas la modifier. Une affectation sur le même nom crée une variable locale homonyme, et toutes les opérations portent sur cette variable locale tandis que **la variable globale demeure intacte**.

Exemple : observer l'effet produit par les codes suivants

```
x='navet'
def soupe():
    y='céleri'
    print(x,y)
soupe()
print(x,y)
```

```
x,y='poireau','carotte'
def soupe2():
    x='navet'
    print(x,y)
soupe2()
print(x,y)
```

Proposition

Si l'on veut modifier une variable globale dans une fonction, il faut la déclarer globale en début de fonction à l'aide de l'instruction `global`.

Exemple : ajouter `global x` au début de la fonction `soupe2()` et observer le résultat.

Résumons ce que l'on peut dire des deux types de variables.

Proposition

Au sein d'une fonction :

- *une variable est **globale** si elle est utilisée dans la fonction sans y être affectée, ou bien si elle est déclarée globale au début ;*
- *une variable est **locale** si elle est affectée sans être explicitement déclarée globale.*

Remarque : malgré les apparences, ce n'est pas du vice. Cela permet de ne pas se soucier des noms de variables utilisés à l'intérieur d'une fonction, puisque l'on sait que cela n'a aucune incidence sur les variables locales des autres fonctions et sur les variables globales du programme. PYTHON EST GRAND !

Exemple : que se passe-t-il lorsque l'on exécute le programme suivant ?

```
def f():
    global x
    x=2
    y=1
```

```
def g():
    y=4*x+1
    print(y)
def h():
    x=3
```

```
x=1
f()
g()
h()
print(x)
```

Proposition (cas des fonctions imbriquées)

Si une fonction `g` *est définie à l'intérieur d'une autre fonction* `f`*, l'instruction* `nonlocal` *permet de rendre les variables de* `g` *visibles dans la fonction* `f`*, mais pas en dehors.*

Remarque : cette instruction est quand même d'un usage peu fréquent.

3.2) Bon usage des variables

Pour éviter les confusions et faire bon usage des différents types de variables, on essaiera de respecter les conventions suivantes :

❶ Les variables globales sont des constantes du problème ; elles sont définies en dehors des fonctions (idéalement en début de programme) et portent des noms explicites.

❷ Les variables locales portent des noms courts et ne servent que dans des calculs précis ou bien dans des boucles.

4/ Passages d'arguments

4.1) Valeurs par défaut

On peut attribuer des valeurs par défaut à certains arguments lors de la déclaration d'une fonction. Il faut impérativement les mettre **à la fin** de la liste d'arguments.

Exemple : les instructions de droite ont le même effet

```
def f(a,b=2):
    return a+b*1j
```

```
f(3,2)
f(3)
```

4.2) Arguments nommés

On a jusqu'ici utilisé des **arguments positionnels** : on passe les arguments à la fonction dans l'ordre où ils sont définis lors de la déclaration de la fonction. On peut aussi se servir des noms des arguments utilisés à ce moment : on parle alors d'**arguments nommés**.

Exemple : les instructions de droite ont le même effet

```
def f(a,b):
    return a+b*1j
```

```
f(3,2)
f(b=2,a=3)
```

On peut mélanger les arguments nommés et les arguments positionnels, mais ces derniers sont pris dans l'ordre où ils arrivent : il faut être alors très vigilant.

Exemple : si l'on reprend la fonction précédente, `f(2,a=1)` renvoie une erreur.

4.3) Arguments multiples

Quand le nombre d'arguments est indéterminé, on peut utiliser des listes ou des tuples comme variables. On peut aussi employer la **forme étoile** : on déclare `*args` comme variable, ce qui place les arguments dans un tuple `args`. On exécute alors la fonction avec la syntaxe habituelle `f(arg1,arg2,...)` ou avec la syntaxe `f(*t)`, `t` étant un tuple d'arguments.

Exemple : les instructions de droite ont le même effet

```
def somme1(l):
    s=0
    for e in l:
       s=s+e
    return s
```

```
def somme2(*args):
    s=0
    for e in args:
        s=s+e
    return s
```

```
somme1([1,2,3,4,5,6,7])
somme2(1,2,3,4,5,6,7)

t=(1,2,3,4,5,6,7)
somme2(*t)
```

Exercices

Exercice 1.
Créer une fonction `factorielle(n)` qui calcule `n!` pour tout `n` $\in \mathbb{N}$.

Exercice 2.
Concevoir une fonction `div_eucl(a,b)` qui retourne le quotient et le reste de la division euclidienne de l'entier a par l'entier b.

Exercice 3.
Écrire une fonction `divisible(a,b)` qui renvoie `True` si l'un des deux entiers `a` ou `b` est divisible par l'autre, et `False` sinon.

Exercice 4.
Créer une fonction `Boule(R)` qui donne la surface et le volume de la boule de rayon R.

Exercice 5.
Concevoir une fonction `Boite()` telle que :

1) `Boite(a,b,c)` est le volume d'un parallélépipède rectangle de dimensions `a` $\times$ `b` $\times$ `c`.
2) `Boite(a,b)` est le volume d'un parallélépipède rectangle de dimensions `a` $\times$ `b` $\times$ `a`.
3) `Boite(a)` est le volume d'un cube de côté `a`.

Exercice 6.
Écrire une fonction `puissance(x,n)` qui, pour `x` $\in \mathbb{R}$ et `n` $\in \mathbb{Z}$, calcule $\texttt{x}^{\texttt{n}}$ sans utiliser `**`.

Exercice 7.
Créer une fonction `racines(a,b,c)` qui renvoie la liste des racines de $aX^2 + bX + c$.
Attention, il faut qu'elle fonctionne aussi pour des polynômes de degré 0 ou 1.

Exercice 8.
Soient `u` et `v` deux vecteurs (ce sont des tuples) de même dimension.

1) Écrire une fonction `scalaire(u,v)` qui calcule le produit scalaire de `u` et de `v`.
2) Créer une fonction `norme(u)` qui retourne la norme euclidienne du `u`.

Exercice 9.
Écrire une fonction `somme_fonction(f,n)` qui renvoie la valeur de la somme $\sum_{i=0}^{n} f(i)$.
Utiliser cette fonction pour calculer $\sum_{i=0}^{10} (i^2 - 3i + 4)$ puis $\sum_{i=14}^{37} \cos(i)$.

Exercice 10.

1) Créer une fonctions `maxi(liste)` qui donne le maximum d'une liste de nombres réels.
2) Écrire une fonction `index_maxi(liste)` qui renvoie l'index de ce maximum.

Exercice 11.

1) Créer une procédure `applique(fonction,liste)` qui affiche les images par `fonction` des éléments de `liste`.
2) Écrire une procédure `tableurP(fonction,debut,nbre,pas)` qui affiche les images par `fonction` de `nbre` points partant de `debut` avec un `pas` constant.
3) Concevoir une procédure `tableurN(fonction,debut,fin,nbre)` qui affiche les images par `fonction` de `nbre` points régulièrement répartis de `debut` à `fin`.

On utilisera ensuite ces procédures pour visualiser la limite en $+\infty$ de $f : x \longmapsto x \sin\left(\frac{12}{x}\right)$.

Exercice 12.

1) Écrire une procédure `derive(f,h)` qui prend en argument une fonction `f` et un réel `h`> 0 et qui retourne la fonction $g : x \longmapsto \dfrac{\mathtt{f}(x+\mathtt{h}) - \mathtt{f}(x)}{\mathtt{h}}$.
2) Vérifier que pour `h=0.001`, on retrouve de bonnes approximations des dérivées usuelles.

Exercice 13.
Que se passe-t-il lorsque l'on exécute les programmes suivants ?

```
def g(x):
    global a
    a=10
    return 2*x
def f(x):
    v=1
    return g(x)+v
a=3
print(f(6)+a)
```

```
def g(x):
    global v
    v=100
    return 2*x
def f(x):
    v=1
    return g(x)+v
a=3
print(f(6)+a)
```

```
def f(x):
    while True:
        x=x+1
        if x==1000:
            return 2*x
print(f(100))
print(f(500))
```

Exercice 14.
Créer une fonction `modif(liste)` qui retourne la liste modifiée comme suit :

- si `liste` est de longueur paire, on intervertit ses deux premiers éléments ;
- sinon, on supprime son dernier élément.

Exercice 15.

1) Écrire une fonction `compte_car(ca,ch)` qui donne le nombre de fois que l'on rencontre le caractère `ca` dans la chaîne `ch`.
2) Créer une fonction `change_car(ca1,ca2,ch)` remplace tous les caractères `ca1` par des caractères `ca2` dans la chaîne `ch`.