1 Écrire une fonction en Python

Un programme complexe est construit à partir de différentes fonctions qui effectuent chacune une tâche précise, assez simple.
Une fonction nécessite la plupart du temps des paramètres d'entrée qui sont les données sur lesquelles travailler, et elle produit en général un résultat en sortie.

> La **signature** d'une fonction, introduite par le mot clef `def`, donne le nom de la fonction et le nom des **paramètres formels** (notés entre parenthèses) qu'attend la fonction en entrée.
> La sortie est obtenue avec le mot clef `return`.

Exemple simpliste :

```
def somme(a, b):
  return a + b
```

> Après avoir défini une fonction, on peut l'**invoquer** (ou l'**appeler**) n'importe où dans le programme. On appelle une fonction en lui passant des **arguments** en entrée. Les paramètres formels de la fonction prendront alors pour valeurs celles des arguments passés (ce sont les paramètres **effectifs**).
> Lors de cet appel, l'**expression** `nom_fonction(arguments)` **vaut** alors le résultat de l'évaluation de la sortie renvoyée par la fonction.

Exemples : `somme(-3, 7.2)` **vaut** 4.2.
Les paramètres formels `a` et `b` ont pris les valeurs des arguments -3 et 7.2 (ou paramètres effectifs).

Remarque importante ! Si l'argument d'une fonction est un objet **mutable**, tel qu'une liste-Python en particulier, alors le paramètre formel correspondant ne prend pas la valeur de l'argument, mais désigne une **référence** vers cet objet. Toute modification du paramètre formel dans le corps de la fonction sera donc répercutée sur l'objet lui-même (même sans aucune valeur de retour introduite par `return`). Dans ce cas, on dit que l'objet est modifié **en place**.

Exemple :

```
>>> def modif(liste):
        liste[0] = 5 # modification du 1er élément de la liste

>>> tableau = [1, 2, 3]
>>> modif(tableau) # la fonction ne renvoie rien, l'expression vaut None
>>> tableau        # on vérifie que le tableau a été modifié en place
[5, 2, 3]
```

2 Spécification d'une fonction

La **spécification** d'une fonction consiste à définir ce que doit faire la fonction, ainsi que les **préconditions** sur les données d'entrée (leur type et leur domaine de valeur en particulier), et quelles sont les **postconditions** sur les données de sortie.
Il est bien de décrire la spécification dans une « **docstring** » en début de fonction. C'est une chaîne de caractères qui est affichée lors de l'instruction `help(fonction)`.
On peut contrôler les préconditions ou postconditions à l'aide d'une **assertion**.

Une assertion est une expression logique (s'évaluant à `True` ou `False`) introduite par le mot clef `assert`. On peut ajouter à la suite de cette instruction un message d'information complémentaire.
Si l'expression vaut `True`, rien de particulier ne se passe, l'interpréteur Python passe sous silence l'instruction correspondante et le programme continue normalement. En revanche, si l'expression vaut `False`, une erreur d'assertion est levée (`AssertionError`), et le programme s'arrête. Si un message d'information optionnel est présent, celui est écrit à la suite de `AssertionError` par l'interpréteur Python.

Exemple : la fonction `rapport` doit renvoyer le rapport de 2 nombres.
— précondition : les entrées sont 2 nombres a et b. Et le nombre b n'est pas nul.

```
def rapport(a, b):
    """ Renvoie le rapport a/b.
    - entrées : a et b : float, float
    - sortie : a/b : float
    - précondition : b non nul
    """
    assert b != 0, "b ne doit pas être nul (division par 0)."
    return a / b
```

3 Utilité et limite des tests

Effectuer des tests est très important pour **essayer de contrôler le bon fonctionnement d'un programme**, mais il est **impossible de prouver la correction d'un programme seulement avec des tests**.

En revanche, si un test échoue, cela assure que le programme a un problème !

La qualité d'un jeu de tests est importante pour essayer de prévoir tous les cas possibles qui pourraient intervenir dans l'utilisation d'un programme.
Il faut en particulier penser à contrôler tous les cas limites.

Ex : cas d'un tableau vide, bien vérifier les bornes des boucles, un signe $<$ ou $<=$? ...

Remarque : **on n'effectue jamais des tests d'égalité sur les nombres flottants** (voir chap. P1-3 pour les explications). On préférera toujours vérifier qu'un flottant est très proche d'un nombre.

Exemple : Voici comment tester que (0.1 + 0.2) vaut bien 0.3 :

```
>>> precision = 0.000001                # on peut choisir la précision désirée
>>> assert 0.1 + 0.2 - 0.3 < precision
```