# Informatique PCSI
# Écriture d'un programme (suite)

## 1 Spécification, annotation

### 1.1 Spécification d'une fonction

Une spécification permet d'informer les utilisateurs de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs affichés en cas de mauvaise utilisation. Elle est résumée dans la *docstring*, inscrite au début du corps de la fonction entre des triples guillemets.

```
def nom_de_la_fonction(arguments):
    """ informations sur la fonction, non obligatoires
        mais recommandées """
    corps de la fonction
```

La spécification est destinée à l'utilisateur. Elle n'est pas utilisée par l'interpréteur Python.

Par exemple, si nous souhaitons obtenir des informations sur la fonction nommée `divmod`, nous utilisons la fonction `help` dans l'interpréteur :

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

On peut effectuer un test avec un nombre décimal :

```
>>> x = 7.2
>>> y = 3
>>> div, mod = divmod(x, y)
>>> div
2.0
>>> mod
1.2000000000000002
>>> div*y + mod == x
True
```

Il ne faut pas être surpris de la valeur obtenue pour `mod`. Les calculs avec des flottants sont des calculs approchés. Ceci est lié à la représentation des nombres en machine.

La spécification d'une fonction est écrite, comme l'est un commentaire dans un programme, à l'attention des utilisateurs qui ont besoin de savoir comment l'utiliser. L'objectif est de les éclairer, de les aider à saisir rapidement le rôle d'une ou plusieurs instructions. Un choix pertinent dans les noms de variables et de fonctions participent aussi à la compréhension d'un code.

Voici comment définir une fonction avec sa spécification et quelques annotations :

```
def permute(liste):
    """ liste est de type list
    la fonction permute le premier et le dernier élément
    et renvoie une nouvelle liste
    permute([1, 2, 3, 4]) renvoie [4, 2, 3, 1] """
    n = len(liste)
    copie = liste[:] # une copie superficielle de la liste
    copie[0], copie[n-1] = copie[n-1], copie[0] # permutation des valeurs
    return copie
```

Le texte entre les triples guillemets fournit ici une description de l'entrée, du traitement et du résultat. Ceci constitue, avec le nom de la fonction, sa *signature*.

Avec cette fonction `permute`, une erreur sur le paramètre provoque l'arrêt de l'exécution de la fonction. Cela se produit si la fonction est appelée avec un nombre ou une chaîne de caractères en paramètre. Mais une fonction peut s'exécuter sans problème même si les paramètres ne sont pas ceux attendus. Les problèmes peuvent survenir plus loin dans le programme et la correction de l'erreur est alors plus difficile.

Prenons un exemple très simple :

```
def calcul(x, y):
    return 2 * (x + y)
```

Cette fonction est programmée pour effectuer un calcul avec deux nombres.

```
>>> calcul(1, 3)
8
```

Si ce résultat doit être utilisé ensuite dans le programme pour un autre calcul, il est capital que ce soit bien un nombre. Mais ici, si les deux paramètres sont de type str, la fonction s'exécute sans aucun souci et produit un résultat qui n'est pas du tout du type attendu.

```
>>> calcul('1', '3')
'1313'
```

Il est donc important que les conditions restrictives sur les paramètres soient bien précisées dans la docstring pour que la fonction soit utilisée correctement.

```
def calcul(x, y):
    """ x et y sont du type int ou float """
    return 2 * (x + y)
```

Les exemples sont simples mais il faut bien comprendre qu'une spécification est une sorte de contrat entre l'auteur d'un code et l'utilisateur. L'auteur garantit un résultat sous réserve d'une utilisation correcte qui est précisée.

### 1.2 Annotations, commentaires

Un programme doit pouvoir être lu et relu facilement par l'auteur mais aussi par quelqu'un qui découvre le programme. Il est important pour cela d'annoter certaines lignes de code ou des blocs d'instructions afin de préciser leur rôle.

On utilise pour cela un commentaire qui est une ligne de texte précédée du signe `#`. Comme c'est le cas pour une spécification, un commentaire n'est pas utilisé par l'interpréteur Python.

Un choix de structure ou de méthode par exemple peut aussi être précisé et expliqué à l'aide de commentaires. Pourquoi choisir un dictionnaire plutôt qu'une liste, pourquoi une fonction récursive plutôt qu'itérative, etc.

```
def mafonction(...):
    ...
    # utilisation d'un dictionnaire pour stocker les données
    dico = {}
    # on teste si la clé appartient au dictionnaire
    if cle in dico:
        ...
    # si la clé n'appartient pas au dictionnaire, on la crée
    else:
        dico[cle] = val
    ...
```

On peut découper un programme en plusieurs parties en précisant l'utilité de chaque partie.

```
"""description du contenu,
objectifs du programme,
choix de conception"""

#importation de modules et de fonctions
...

#définitions de constantes, de variables
...

#définitions de fonctions annexes
...

#définition de la fonction principale
...

#exemples d'utilisation et tests
...
```

## 2 Assertion

Une spécification permet d'éclairer sur les données en entrées, le type des valeurs autorisées, la plage de valeurs acceptées. On peut ajouter des instructions qui vont arrêter le programme en cas de mauvaises utilisation.

Une assertion est l'affirmation qu'une propriété est vraie. Elle est composée du mot `assert` suivi d'une expression dont la valeur est interprétée comme une valeur booléenne. Si l'expression a la valeur `True` il ne se passe rien. Sinon le programme est interrompu et un message d'erreur s'affiche terminé par `AssertionErrror`.

Voici un exemple simple :

```
def inverse(x):
    """x est un nombre non nul de type int ou float
    renvoie l'inverse de x
    """
    assert x != 0
    return 1 / x
```

Utilisation de la fonction :

```
>>> inverse(-3)
-0.3333333333333333
>>> inverse(0)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    inverse(0)
  File "C:/Documents/progs/test.py", line 5, in inverse
    assert x != 0
AssertionError
```

Un exemple avec une fonction qui prend en paramètres un dictionnaire et une clé :

```
def fonction(dico, k):
    """dico est un dictionnaire, k est une clé de ce dictionnaire
    """
    assert k in dico
    ...
```

On contrôle si la valeur passée en paramètre pour exécuter le fonction est bien une clé du dictionnaire. Si ce n'est pas le cas, le programme s'interrompt.

De manière générale, on utilise `assert` pour contrôler une valeur.