CHAPITRE 6

Prototypage d'une fonction

Il est plus facile de changer la spécification pour qu'elle corresponde au programme que le contraire.

Alan Jay Perlis

Thème : Langage et programmation

Contenus	**Capacités attendues**
Spécification	Prototyper une fonction. Décrire les préconditions sur les arguments. Décrire des postconditions sur les résultats.

Activité 1.
On s'intéresse à la fonction suivante :

```
1  def fonction(a,b):
2      r = a
3      q = 0
4      while r >= b:
5          r = r - b
6          q = q + 1
7      return q, r
```

1. De quel type sont les arguments a et b ?
2. Quelle opération mathématique fait cette fonction ?
3. Combien de temps avez vous passé à répondre à ces deux questions ?

I. Documenter une fonction

Une spécification permet d'informer les utilisateurs de la tâche effectuée par la fonction, de préciser les contraintes imposées pour les paramètres d'entrée et ce qui est 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 quotes.

Déf.1 La "docstring" est une chaîne de caractères que l'on n'assigne pas, et qui est placée à un endroit spécifique du code pour décrire ce dernier.

I. 1. Placement de la docstring

La docstring la plus courante est placée sous une fonction.

Exemple 1 (Voici une fonction SANS docstring)

```
1 def ajouter(a, b):
2     return a + b
```

Exemple 2 (Et voici une fonction AVEC docstring)

```
1 def ajouter(a, b):
2     """Ajoute deux nombres l'un à l'autre et retourne le résultat."""
3     resultat = a + b
4     return resultat
```

I. 2. Avantages d'une docstring

Écrire des docstrings offre un avantage important. La fonction `help()` affiche cette documentation dans un shell.

Exemple 3 (Utilisation de la fonction help() dans la console Python)

```
>>> help(ajouter)
Help on function ajouter in module __main__:

ajouter(a, b)
Ajoute deux nombres l'un à l'autre et retourne le résultat.
```

I. 3. Que mettre dans une docstring ?

La spécification d'une fonction est écrite à l'intention des utilisateurs. Ce sont eux qui liront le code, plus souvent que l'auteur. L'objectif est de les éclairer, de les aider à saisir rapidement le rôle d'une ou plusieurs instructions.

Il convient donc d'écrire des phrases (sujet, verbe, complément).

Les informations que l'on peut inclure sont précisées dans une norme [1]. Aucun champ n'est obligatoire, et aucun champ ne dépend d'un autre.

Les informations sont présentées ainsi :

- ▷ la description de ce que fait la fonction ;
- ▷ chaque argument d'entrée est précisé : que représente-t-il ? Quel est son type ?
- ▷ chaque argument de sortie est précisé : que représente-t-il ? Quel est son type ?

1. PEP 257 : `https://www.python.org/dev/peps/pep-0257/`

▷ quelques exemples de ce que doit retourner la fonction. On pensera en particulier aux cas "extrêmes" ou qui peuvent poser problème (bornes d'une boucle, division par 0).

I. 4. Les règles de bonne pratique

Les docstrings sur plusieurs lignes sont constituées :

▷ d'une première ligne résumant brièvement la fonction ;

▷ suivie d'un saut de ligne ;

▷ suivi d'une description plus longue ;

▷ les triples quotes de fin se trouvent sur une ligne unique.

La première ligne de la docstring peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous.

Dans tous les cas, le reste de la docstring doit être indentée au même niveau que la première ligne :

Exemple 4 (La docstring complète)

```
def add(a, b):
    """ Additionne deux nombres et retourne le résultat.

    Utilisez cette fonction à la place de l'opérateur `+`.

    Arguments d'entrée :
        a : le premier nombre à additionner
            type : un entier
        b : le deuxième nombre à additionner
            type : un entier

    Sortie
        resultat : le résultat de l'addition
            type : un entier

    Exemple :

    >>> add(1, 1)
    2
    """

    # écriture du code
    resultat = a + b
    return resultat
```

Activité 2 (Écriture d'une docstring).
On se propose de réaliser une fonction *division_euclidienne*(a, b) permettant de faire la division euclidienne de a par b. La fonction est supposée renvoyer un couple de valeurs contenant respectivement le quotient q et le reste r de la division euclidienne de a par b.

Écrire la documentation de cette fonction. On donnera successivement :

— la description de la fonction,

— les informations nécessaires sur les arguments d'entrée et les résultats,

— un exemple d'exécution.

On rappelle qu'en Python, le quotient de la division euclidienne s'écrit `//` et le reste s'écrit `%`.

II. Préconditions sur les arguments

Activité 3.
On s'intéresse à la fonction *division* définie par :

```
def division(a, b):
    """fonction qui effectue la division de a par b"""
    resultat = a / b
    return resultat
```

1. Quel sera le résultat de l'appel `division(5, 0)` ?

2. Quel sera le résultat de l'appel `division("toto", 3)` ?

Déf.2 Une précondition est une condition que DOIVENT respecter les arguments d'entrée d'une fonction.

II. 1. Les exceptions

Une exception est une erreur que peut rencontrer Python en exécutant une fonction. Si Python renvoie une erreur, c'est qu'il y a une raison ! Par exemple, on peut rencontrer :

Exemple 5

```
>>> variable = 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

ZeroDivisionError est le type de l'exception. Il y en a d'autres :

- ▷ **NameError** : l'une des variables n'a pas été définie.
- ▷ **TypeError** : l'instruction n'est pas compatible avec le type de la variable (par exemple, il est aberrant de diviser par un caractère.
- ▷ **ValueError** : le type de la variable est bon, mais la valeur est incorrecte.
- ▷ **IndexError** : l'index utilisé pour parcourir un objet est trop grand pour l'objet (par exemple, pour la chaîne de caractères chaine="taratata" qui contient 8 caractères numérotés de 0 à 7, chaine[8] lèvera une exception IndexError.

La liste complète des exceptions est disponible sur le site officiel [2].

2. https://docs.python.org/fr/3.5/library/exceptions.html

II. 2. Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. La syntaxe est la suivante :

Déf.3

```
assert test, message_si_faux
```

Si le test renvoie True, l'exécution se poursuit normalement. Sinon, une exception `AssertionError` est levée, et le message est affiché.

Exemple : dans la fonction `add(a, b)`, a et b doivent être de type entier. Pour tester cette condition, on pourra utiliser le code suivant :

Exemple 6 (Deux exemples d'assertion)

```
assert type(a) == int, "le premier argument doit être un entier"
assert type(b) == int, "le deuxième argument doit être un entier"
```

Ce morceau de code est à placer dans la fonction `add(a, b)`, juste avant d'exécuter la première instruction :

Exemple 7 (Observez où sont placées les assertions - lignes 22 et 23)

```
def add(a, b):
    """ Additionne deux nombres et retourne le résultat.

    Utilisez cette fonction à la place de l'oérateur `+`.

    Arguments d'entrée :
        a : le premier nombre à additionner
            type : un entier
        b : le deuxième nombre à additionner
            type : un entier

    Sortie :
        resultat : le résultat de l'addition
            type : un entier

    Exemple :

    >>> add(1, 1)
    2
    """
    # préconditions
    assert type(a) == int, "le premier argument doit être un entier"
    assert type(b) == int, "le deuxième argument doit être un entier"

    # écriture du code
    resultat = a + b
    return resultat
```

Exemple 8 (Une exécution réussie)

```
>>> add(2, 3)
5
```

Exemple 9 (Une exécution qui lève une exception)

```
>>> add(2, 3.2)
5
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
File "/home/jose/Documents/toto.py", line 22 in add
assert type(b) == int, "le deuxième paramètre doit être un entier"
AssertionError : le deuxième paramètre doit être un entier
```

II. 3. Usage des assertions dans des préconditions

Les assertions sont utilisées dans les préconditions pour s'assurer que l'utilisateur entre des arguments qui répondent bien au cahier des charges.

- ▷ le nombre de bits est-il bien égal à 8 ?
- ▷ l'octet contient-il bien uniquement des 0 et des 1 ?
- ▷ l'entrée correspond-elle bien au type attendu ?

Activité 4.
Complétons l'activité vue à la partie 2 à la page 3.

1. Quelles sont les restrictions pouvant s'appliquer aux entrées en terme :

a. de type ?

b. de valeur ?

c. de signe ?

2. Écrire les assertions qui s'y rapportent. Inclure les commentaires pour chaque assertion.

III. Postconditions sur les résultats

III. 1. Résultats attendus

On a vu dans la section I. 3. page 2, que la docstring contient quelques exemples. On les choisis de telle façon qu'ils permettent de tester le fonctionnement correct d'une fonction. Il s'agit

d'indiquer quel doit être le résultat attendu de la fonction. Il faut donc préparer ce jeu de tests **AVANT** de coder la fonction.

On peut créer autant de tests que l'on souhaite. Cependant, il vaut mieux 4 ou 5 tests bien choisis plutôt qu'une multitude de tests mal choisis.

Les conditions à tester prioritairement sont :

- ▷ les limites des boucles (limite inférieure ET supérieure)
- ▷ les nombres positifs OU négatifs
- ▷ le zéro ET le un
- ▷ les ensembles vides
- ▷ tout autre cas qui semblerait poser problème.

Voici un exemple : pour la fonction `add(a, b)`, on pourrait s'intéresser aux cas suivants :

- ▷ Si a prend la valeur 0, le résultat attendu est b. Il en est de même si b prend la valeur a.

```
>>>add(0, 5)
5
>>>add(5, 0)
5
```

- ▷ On pratique de même si a prend la valeur 1 ou si b prend la valeur 1.

```
>>>add(1, 5)
6
>>>add(5, 1)
6
```

- ▷ On teste aussi le cas où a et b sont de signe différent, avec un résultat qui sera positif ou négatif :

```
>>>add(1, -5)
-4
>>>add(5, -1)
4
```

- ▷ On teste le cas où a et b sont de même signe, avec un résultat positif ou négatif :

```
>>>add(3, 5)
8
>>>add(-3, -5)
-8
```

III. 2. Insertion dans la doctype

La doctype complète de la fonction `add(a, b)` devient alors :

```
def add(a, b):
    """ Additionne deux nombres et retourne le résultat.

    Utilisez cette fonction à la place de l'opérateur `+`.

    Arguments d'entrée :
    a : le premier nombre à additionner
        type : un entier
    b : le deuxième nombre à additionner
        type : un entier

    Sortie
    resultat : le résultat de l'addition
        type : un entier

```

```
    Exemples :
    >>> add(0, 5)
    5
    >>> add(5, 0)
    5
    >>> add(1, 5)
    6
    >>> add(5, 1)
    6
    >>> add(1, -5)
    -4
    >>> add(5, -1)
    4
    >>> add(3, 5)
    8
    >>> add(-3, -5)
    -8
    """

    # préconditions
    assert type(a) == int, "le premier paramètre n'est pas un entier"
    assert type(b) == int, "le deuxième paramètre n'est pas un entier"

    # écriture du code
    resultat = a + b
    return resultat
```

Activité 5.
Complétons l'activité vue à la partie 2 à la page 3. On cherche à écrire des postconditions que doivent vérifier les résultats. Pour cela, on cherche quels sont les valeurs de a et b qui pourraient poser problème.

1. Quelles sont les valeurs de a et b qui pourraient poser problème ?

2. À quelle condition sur a et b obtient-on :

a. une valeur de q nulle ?

b. une valeur de r nulle ?

3. Écrire quelques exemples d'exécution qui s'y rapportent.

III. 3. Le module doctest

Le module doctest [3] permet d'inclure les tests dans la docstring descriptive de la fonction écrite. Le module doctest(via l'appel à `doctest.testmod()`) reconnaît les bouts de code correspondant à ces exemples et les exécute pour les comparer à la sortie demandée. On peut alors commenter

3. `https://docs.python.org/3.7/library/doctest.html`

in situ les tests et leurs raison d'être et avoir une sortie détaillée et globale des tests qui ont réussi ou raté.

Définition 4

Pour utiliser le module doctest, il suffit d'inclure **APRÈS LA FONCTION**, le morceau de code suivant :

```
if __name__== '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

L'option `verbose = True` (on parle de mode verbeux) permet d'afficher le résultat des tests quelqu'en soit le résultat.
Lorsqu'on exécute Python sur le fichier précédent, on obtient :

▷ aucun message d'erreur si la fonction satisfait aux tests de postcondition.

```
>>>runfile('C:/Users/José/Desktop/sanstitre2.py', wdir='C:/Users/José/Desktop')
Trying:
add(0, 5)
Expecting:
5
ok
1 items had no tests:
__main__
1 items passed all tests:
1 tests in __main__.add
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

▷ un message d'erreur si la fonction ne satisfait aux tests de postcondition.

```
>>>runfile('C:/Users/José/Desktop/sanstitre2.py', wdir='C:/Users/José/Desktop')
**********************************************************************
File "C:/Users/José/Desktop/sanstitre2.py", line 24, in __main__.add
Failed example:
add(0, 5)
Expected:
4
Got:
5
**********************************************************************
1 items had failures:
1 of   8 in __main__.add
***Test Failed*** 1 failures.
```

L'appel `doctest.testmod()` teste toutes les postconditions présentes dans toutes les fonctions d'un fichier python (.py). Pour tester uniquement les postconditions d'une seule fonction, on pourra utiliser **dans la console** l'appel
`doctest.run_docstring_examples(ma_fonction, globals(), verbose=True)`, où *ma_fonction* est le nom de la fonction à tester.

```
>>> import doctest
>>> doctest.run_docstring_examples(add, globals(), verbose=True)
Finding tests in NoName
Trying:
add(0, 5)
Expecting:
5
ok
```

IV. Écriture complète d'une fonction

Définition 5

1 Une fonction commence par le mot-clé `def` suivi des arguments d'entrée. On ajoute un double point à la fin.

2 On ajoute ensuite la doctype, encadré par des triples quotes.

- ▷ une description de ce que fait la fonction
- ▷ les arguments d'entrée et de sortie
- ▷ les types des arguments d'entrée et de sortie
- ▷ les exemples d'exécution qui vont vérifier les postconditions

3 On ajoute les assertions qui vont vérifier les préconditions

4 On écrit le code de la fonction.

Exemple 10 (La fonction add(a, b) complète)

```
1  def add(a, b):
2      """ Additionne deux nombres et retourne le résultat.
3
4      Utilisez cette fonction à la place de l'opérateur `+`.
5
6      Arguments d'entrée :
7      a : le premier nombre à additionner
8          type : un entier
9      b : le deuxième nombre à additionner
10         type : un entier
11
12     Sortie
13     resultat : le résultat de l'addition
14         type : un entier
15
16     Exemples :
17     >>> add(0, 5)
18     5
19     >>> add(5, 0)
20     5
21     >>> add(1, 5)
22     6
23     >>> add(5, 1)
24     6
25     >>> add(1, -5)
26     -4
27     >>> add(5, -1)
28     4
29     >>> add(3, 5)
30     8
31     >>> add(-3, -5)
32     -8
33     """
34
35     # écriture du code
36     resultat = a + b
37     return resultat
```

Activité 6.
Complétez l'activité vue à la partie 2 à la page 3. Écrivez la fonction complète, avec l'intégralité de sa documentation.

Je retiens

- La documentation d'une fonction s'écrit dans la docstring, et doit être placée entre des triples quotes ;
- elle donne une description utile de l'usage de cette fonction ;
- il est d'usage de faire figurer une brève eescription, les arguments d'entrée avec leur type, les sorties avec leur type, et quelques exemples d'exécution ;
- les assertions permettent de vérifier des préconditions sur les arguments. Il s'expriment par :
  ```
  assert precondition, texte affiche si la precondition n'est pas respecte
  ```
- pour tester la validité d'une fonction, on utilise les postconditions. Elles indiquent le résultat attendu par un appel :

  ```
  >>> appel(arguments)
  résultat attendu
  ```

V. Exercices

Exercice 1.
On cherche à coder une fonction *est_pair* avec un argument a qui renvoie `True` ou `False` en fonction de la valeur du paramètre d'entrée.

1. Quelles sont les préconditions sur l'argument ?
2. Écrivez les assertions correspondantes.
3. Écrivez les postconditions sur le résultat.
4. Comment peut-on savoir si un entier est pair ? Traduire cela à l'aide **d'une seule** instruction Python.
5. Écrivez le code Python de cette fonction, avec sa docstring complète.

Exercice 2.
On cherche à coder une fonction *quel_triangle* avec trois arguments (les longueurs des trois côtés d'un triangle) qui renvoie la nature du triangle parmi les termes "quelconque, équilatéral et isocèle".

1. Quelles sont les préconditions sur les arguments ?
2. Comment peut-on savoir si les trois arguments sont cohérents entre eux ?
3. Écrivez les assertions correspondantes.
4. Écrivezles postconditions sur le résultat.
5. Écrivez le code Python de cette fonction, avec sa docstring complète.

Exercice 3.
On cherche à coder une fonction *monome* permettant de calculer $2 \times \sqrt{x} + 2$.

1. Identifiez les arguments et le résultat de cette fonction. Quels sont leur type ?
2. Quelles sont les préconditions sur l'argument ?
3. Écrire les assertions correspondantes.
4. Écrire les postconditions sur le résultat.
5. Écrire le code Python de cette fonction, avec sa docstring complète. (La fonction $\sqrt{x}$ s'écrit en Python `math.sqrt(x)`, après avoir écrit au préalable `import math`.)

Exercice 4.
Proposez une fonction *prix_solde* permettant, à partir d'un prix et d'une remise exprimée en pourcentage, de renvoyer le prix de l'objet soldé en ajoutant la docstring complète.

Exercice 5.
Proposez une fonction *heure* permettant, à partir du nombre de secondes écoulées depuis le début de la journée, de renvoyer le nombre d'heures, de minutes et de secondes écoulées (h,min,sec), en ajoutant la docstring complète.