Spécification de programmes

Spécification

Tout programme a un but. Ce but peut paraître évident pour son concepteur au moment où il l'écrit, mais cela peut ne plus être le cas quelques mois ou années plus tard. Et pour quelqu'un d'autre, le but de ce programme peut sembler totalement obscure. C'est pourquoi il est si important de commenter et de documenter ses programmes.

```
def division(a, b):
    """ Renvoie le quotient et le reste de la division de a par b.
        a est un entier positif et b un entier positif non nul."""
    q = 0
    r = a
    while r >= b:
        r = r - b
        q = q + 1
    return q, r
```

On considère la fonction ci-dessus. Le commentaire en dessous de la première ligne s'appelle la **spécification**. Elle indique ce que fait la fonction et le lien qu'il y a entre les paramètres et le résultat. Elle indique également les conditions qui s'appliquent sur les paramètres. On les appelle les **pré-conditions**. De la même manière, on pourrait poser des **post-conditions** sur le résultat en précisant que `0 <= r < b` et `a = q*b + r`.
Ce commentaire juste après la définition de la fonction s'appelle un **docstring**. Il est reconnu par Python et peut être affiché à l'aide de la fonction **help** :

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

division(a, b)
    Renvoie le quotient et le reste de la division de a par b.
    a est un entier positif et b un entier positif non nul.
```

Lors de l'élaboration des pré-conditions, il faut bien s'assurer qu'à l'utilisation, on n'appellera pas la fonction avec des valeurs inapproriées. Si c'est le cas, aucune garantie n'est apportée sur le bon déroulement de la fonction ou sur la validité du résultat.
Afin de s'assurer que les pré-conditions sont satisfaites, on rajoute des tests dans le programme. Python possède la commande **assert** qui est prévue à cet effet. On peut ainsi rajouter deux lignes au programme.

```
def division(a, b):
    """ Renvoie le quotient et le reste de la division de a par b.
        a est un entier positif et b un entier positif non nul."""
    assert (a >= 0), "a est négatif"
    assert (b > 0), "b est négatif ou nul"
    q = 0
    ...
```

Si on appelle la fonction avec des paramètres ne vérifiant pas les pré-conditions, le programme sera interrompu et un message sera affiché :

```
>>> division(5, 0)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File ".../specification.py", line 28, in division
    assert (b > 0), "b est négatif ou nul"
AssertionError: b est négatif ou nul
```

On parle de **programmation offensive** le fait de provoquer des erreurs lorsque le programme ne se comporte pas comme prévu. Au contraire, en **programmation défensive**, on rajoute des valeurs par défaut ou des cas spéciaux afin de garantir que le programme continue de fonctionner quoi qu'il arrive. Typiquement dans une étude de cas, on rajoutera un **else** qui lève une erreur en programmation offensive et une valeur par défaut dans la programmation défensive.

Jeu de tests

Une fois que les spécifications de la fonction, ou du programme, sont définies, il faut s'assurer que le code obtenu fait bien ce qui est attendu. Il ne suffit pas de vérifier que la fonction donne le bon résultat dans un seul cas. Mais il n'est généralement pas possible de tester toutes les valeurs possibles pour les paramètres, puisqu'il y en a le plus souvent une infinité.
Il faut alors définir un ensemble de cas, appelé **jeu de tests**, qui permettra, si possible, d'explorer tous les comportements attendus. Par exemple si la fonction contient des instructions conditionnelles, on fera en sorte de produire des tests qui correspondront à chacun des cas possibles. Il faut porter une attention particulières aux cas limites : liste vide, valeurs nulles… Dans le cas où la fonction renvoie une valeur spéciale si les pré-conditions ne sont pas vérifiées, il faut également le vérifier. Autrement, on ne teste pas le comportement de la fonction en dehors de ses spécifications. Pour `division`, on ne testera pas ce qui se passe si les paramètres ne sont pas entiers.
On peut également générer des tests de façon aléatoire et s'assurer que la fonction est valide pour chacun de ces tests. La difficulté est alors de déterminer le résultat attendu. On peut parfois écrire un **oracle** qui vérifie que les post-conditions sont vérifiées. Par exemple, pour un algorithme de tri, on pourra écrire un oracle qui teste si le tableau obtenu est bien trié. Bien entendu, il faut également que l'oracle soit correct. Mais généralement, son code est plus court et plus simple à vérifier que celui de la fonction que l'on souhaite tester. On peut également partir du résultat attendu pour trouver les paramètres à donner à la fonction pour l'obtenir.
C'est généralement une bonne idée de confier l'écriture de la fonction et celui des tests à deux personnes, ou équipes, différentes. Ainsi, on minimise le risque que l'oubli d'un cas se retrouve dans la fonction et dans les tests.
En Python, ces tests peuvent être faits à l'aide de **assert**. Par exemple, on peut rajouter les tests suivants après la définition de `division`.

```
assert division(0, 7) == (0, 0)
assert division(15, 5) == (3, 0)
assert division(9, 9) == (1, 0)
assert division(3, 11) == (0, 3)
assert division(159, 10) == (15, 9)
```

Ces tests sont exécutés à chaque fois que le script est rechargé. Ils permettent de s'assurer qu'aucune **régression** n'est introduite lors des modifications du programme.