# Spécification et vérification 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
```

Il convient alors de s'assurer que ces conditions sont vérifiées avant d'appeler la fonction. On appelle cela la **programmation défensive**. Il est aussi possible de renvoyer une valeur spéciale comme `None` ou `-1` pour indiquer que les pré-conditions ne sont pas satisfaites. Dans ce cas, il faut tester la validité de la réponse obtenue avant de l'utiliser dans la suite du programme. On parle parfois de **programmation offensive**.

## *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.

## *Vérification des boucles*

Les tests permettent de s'assurer que le programme se comporte comme attendu dans un certain nombre de cas, mais cela ne correspond pas à une preuve de correction. Il se peut qu'on ait oublié certains cas et que le programme comporte encore des bugs non détectés. Il existe d'autres techniques de vérification, qu'on appelle **méthodes formelles** qui ont pour but de faire des preuves "mathématiques" de correction des programmes. Sans rentrer dans ce genre de vérifications, il est néanmoins possible de s'assurer que certains bouts de programmes sont corrects.
En général, les boucles sont les parties les plus critiques des algorithmes, et plus particulièrement les boucles non-bornées. Il y a deux choses à vérifier : "est-ce que la boucle se termine ?" et "est-ce que la boucle fait bien ce qu'on attend ?". Pour le premier cas, on parle de preuve de **terminaison** et dans le deuxième de preuve de **correction**.
Ces preuves sont généralement faites au niveau de l'algorithme et il faut ensuite s'assurer que le programme écrit correspond bien à l'algorithme.

## *Preuve de terminaison*

Pour les boucles `for`, la terminaison est assurée. Ce n'est pas le cas des boucles `while`. Afin de démontrer qu'elles se terminent, on utilise un **variant de boucle**. C'est une expression dont la valeur est positive et entière, et qui décroit à chaque tour de boucle. Dans le cas de la fonction `division`, il suffit de prendre `r`. Sa valeur diminue à chaque itération et la pré-condition sur `a` assure que sa valeur initiale est positive. Et puisque $r \geq b$ au début de la boucle, à la fin, on aura $r - b \geq 0$. La valeur de `r` à la fin de la boucle est donc toujours positive et strictement inférieure à la valeur précédente.

## *Preuve de correction*

Pour montrer qu'une boucle fait bien ce qu'on attend, on utilise un **invariant de boucle**. Cette fois, il faut trouver une expression dont la valeur reste la même à chaque itération. Trouver cet invariant peut devenir très complexe. Dans le cas de la fonction `division`, on veut obtenir $q \times b + r = a$ à la fin de l'exécution de la fonction. Or, on peut remarquer que cette égalité est toujours vraie au début de chaque tour de boucle. L'expression $q \times b + r$ est donc notre invariant.
À la fin d'une itération, on a `r` vaut $r - b$ et `q` vaut $q + 1$. Or :

$$\begin{aligned}(q+1) \times b + (r-b) &= q \times b + b + r - b \\ &= q \times b + r\end{aligned}$$

L'invariant et donc bien préservé à chaque itération. Il faut également s'assurer que lors de la première itération, la valeur de l'invariant est la bonne. Or, à ce moment là, on a `r` qui vaut `a` et `q` qui vaut 0. Il est donc facile de vérifier que l'expression vaut bien `a`.
On a ainsi la garantie qu'à la fin de la boucle, la post-condition est bien vérifiée. Il est d'ailleurs possible de rajouter `assert q*b+r == a` au début de la boucle pour être sûr que l'invariant est bien vérifié à chaque itération.

## *Un exemple concret*

Afin d'illuster ces techniques de vérification, nous allons prendre un autre exemple, donné par la fonction ci-contre.
Cette technique de multiplication est très efficace sur ordinateur puisque les seules opérations utilisées sont des additions et des multiplications ou divisions par 2, ce qui revient à décaler d'un chiffre en binaire.

```
1  def multiplication_russe(a, b):
2      """ Renvoie le produit de a par b
3          a est un entier positif."""
4      r = 0
5      while a > 0:
6          if (a%2 == 1):
7              r = r + b
8              a = a - 1
9          a = a // 2
10         b = b * 2
11     return r
```

**Exercice 1 :**

1) Complétez le tableau de valeurs ci-contre en indiquant à chaque fois les valeurs de `a`, `b` et `r` à la ligne 8 si le test est vérifié et à la ligne 10.
2) Donnez des valeurs pouvant être utilisées pour réaliser des tests à l'aide de la commande `assert multiplication_russe(..., ...) == ...`.
3) Donnez un `assert` pouvant être rajouté avant la ligne 4.
4) Donnez un `assert` pouvant être mis avant la ligne 9.

| ligne | a | b | r |
|---|---|---|---|
| 4 | 35 | 27 | 0 |
| 8 | 34 | 27 | 27 |
| 10 | 17 | 54 | 27 |

**Exercice 2 : (Preuve de terminaison)**

1) Justifiez pourquoi `a` est forcément pair lors de la division par 2.
2) Expliquez pourquoi `a` peut servir de variant de boucle.

On peut remarquer que `a` est divisé par 2 à chaque itération. Il diminue donc très vite. En fait si on fait une itération pour chaque chiffre de l'écriture binaire de `a`, en allant de droite à gauche, et on ajoute `b` à `r` chaque fois qu'il y a un 1. Le nombre d'itérations est donc l'entier $n$ tel que $2^{n-1} \leq a < 2^n$. Non seulement, on vient de montrer que cette boucle se termine toujours, mais en plus on sait en combien d'étapes.

**Exercice 3 : (Preuve de correction)**

Pour cet exercice, nous noterons $a_0$ et $b_0$ les valeurs initiales de `a` et `b`.
Nous allons utiliser l'invariant de boucle : $a \times b + r$.

1) Expliquez pourquoi l'invariant est préservé lors d'une itération si `a` est pair.
2) Expliquez pourquoi l'invariant est préservé lors d'une itération si `a` est impair.
3) En déduire que $a \times b + r$ est bien un invariant de cette boucle.
4) Exprimez la valeur de l'invariant en fonction de $a_0$ et $b_0$ avant le premier tour de boucle.
5) Expliquez pourquoi $r = a_0 \times b_0$ à la sortie de la fonction.

## *Pour aller plus loin*

On se doute bien que pour une fonction très complexe, il est très difficile de pouvoir expliciter un invariant de boucle. Mais ce n'est pas parce qu'une boucle est "simple" que cette tâche est plus aisée. Dans le cas de la suite de Syracuse, personne n'a jamais réussi à prouver que la boucle se terminait pour n'importe quel nombre entier positif.

```
def syracuse(n):
    while n > 1:
        if (n%2 == 1):
            n = 3*n + 1
        n = n//2
    return n
```