éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

MISE AU POINT DE PROGRAMMES TESTÉS

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. Introduction

Écrire un programme, c'est bien. Écrire un programme juste, c'est mieux. La panacée serait de pouvoir *prouver* tous les programmes que l'on écrit, mais outre que ce n'est pas forcément évident (cf le langage COQ [1, 4] qui permet une telle chose mais n'est pas forcément accessible au premier venu), on n'est jamais à l'abri d'une erreur de frappe qui nous fait lire ce que l'on voudrait lire et non ce qui est réellement écrit dans le code [1].
C'est pourquoi il est intéressant (bien que non suffisant en général [2]) de tester les bouts de code que l'on écrit sur des cas particuliers pour lesquels on connait bien le résultat attendu.

2. Développement piloté par les tests

Le développement piloté par les tests ou TDD (pour Test Driven Development) est une méthode d'écriture de programme qui met en avant le fait d'écrire *d'abord* un test pour chaque spécification du programme *puis* écrire le code qui permettra au programme de passer ce test avec succès. Cette manière de procéder permet en général de penser en premier lieu aux spécifications voulues, ce qui améliore la structure générale du code produit et permet de s'assurer que l'on dispose de toute une batterie de tests sous la main [3].
Plus particulièrement, le TDD a été théorisé [2] sous forme de trois lois.

- On doit écrire un test qui échoue *avant* de pouvoir écrire le code permettant de le faire réussir.
- Il ne faut tester qu'un point précis du programme sous forme d'une assertion unique [4].
- Il ne faut écrire que le minimum de code permettant de réussir le test.

1. Donald Knuth lui-même disait « Beware of bugs in the above code; I have only proved it correct, not tried it. », soit « Attention aux bugs dans le code ci-dessus. Je ne l'ai pas testé, j'ai seulement prouvé qu'il était correct. »
2. Edsger Dijkstra quant à lui a dit « Testing shows the presence, not the absence of bugs », soit « Tester un programme démontre la présence de bugs, pas leur absence. »
3. En effet, comme on dépense en général déjà beaucoup d'énergie à écrire un code qui nous semble correct, l'écriture de tests en surplus par la suite (alors qu'on est déjà persuadé que le code est bon) peut sembler superflu et passe généralement à l'as.
4. Oubliez les tests « tout en un ».

Bien qu'il soit exclu à notre niveau d'avoir un environnement de développement complet qui soit TDD-compatible, le fait d'écrire *a priori* des tests qui sont supposés être réussis par les fonctions que l'on veut coder oblige à penser à tous les cas particuliers que l'on peut être amené à croiser dans le problème que l'on tente de résoudre.

3. Exemple pratique : la division euclidienne

3.1. Algorithme à tester

On se propose de tester l'algorithme qui permet de faire la division euclidienne de a par b en soustrayant itérativement b à a et en comptant le nombre de soustractions nécessaires pour que le reste soit compris entre 0 et b (exclu). La fonction est supposée renvoyer un tuple contenant respectivement le quotient q et le reste r de la division euclidienne de a par b. S'il n'est pas possible d'obtenir le résultat demandé par l'algorithme proposé (par exemple si a ou b sont négatifs) ou que les arguments donnés à la fonction ne sont pas entiers, on choisit de renvoyer -1.

3.2. Tests simples : utilisation de `assert`

L'idée la plus simple à mettre en place est d'utiliser la commande `assert` pour vérifier que les résultats voulus correspondent à ce que renvoie la fonction désirée. En particulier :

- la division euclidienne de 10 par 2 doit renvoyer un quotient de 5 pour un reste nul ;
- celle de 2 par 10 doit renvoyer un quotient nul pour un reste qui vaut 2 ;
- celle de 37 par 3 doit renvoyer un quotient de 12 pour un reste de 1 ;
- celles de -10 par 7, de 10 par -7, de 10.3 par 4 ou encore de 11 par 3.5 doivent toutes renvoyer le code -1 « d'erreur » ;
- reste le cas particulier du zéro : une division par zéro doit renvoyer -1 alors que 0 divisé par n'importe quoi (sauf 0) donne un quotient et un reste nuls.

Le fichier `division_euclidienne.py` du module contenant la définition de notre fonction (appelée sans grande originalité `division_euclidienne`) peut alors être écrit sous la forme suivante, où l'on utilise la construction `if __name__ == '__main__':` qui permet de n'exécuter les tests que lorsqu'on appelle Python directement sur le module et non pas quand celui-ci est inclus dans un programme plus large.

```
def division_euclidienne(a,b):
    return None # Il va falloir écrire quelque chose


if __name__ == '__main__':
    assert division_euclidienne(10, 2) == (5, 0)
    assert division_euclidienne(2, 10) == (0, 2)
    assert division_euclidienne(37, 3) == (12, 1)
    assert division_euclidienne(-10, 7) == -1
    assert division_euclidienne(10, -7) == -1
    assert division_euclidienne(10.3, 4) == -1
    assert division_euclidienne(11, 3.5) == -1
    assert division_euclidienne(3, 0) == -1
    assert division_euclidienne(0, 3) == (0, 0)
    assert division_euclidienne(0, 0) == -1
```

À noter que la division de 0 par un flottant ou un entier négatif n'est pas clairement spécifié... Il ne coûte pas grand chose de renvoyer `(0,0)` si a vaut 0 quel que soit b, mais il faudrait savoir exactement ce qu'attend l'utilisateur final de la fonction et utiliser le test adéquat.

L'inconvénient majeur de cette méthode (même si cela peut en fait être un avantage) est que l'on doit traiter une assertion après l'autre car le programme s'arrête dès le premier test qui échoue. En cas de réussite, il ne se passe simplement rien.

3.3. Tests dans la docstring : le module `doctest`

Le module `doctest` [3] permet d'inclure les tests dans la docstring descriptive de la fonction écrite. On présente dans la docstring, en plus des explications d'usage, des exemples d'utilisation de la fonction tels qu'ils pourraient être tapés directement dans la console Python. 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 *in situ* les tests et leurs raison d'être et avoir une sortie détaillée et globale des tests qui ont réussi ou raté.

```
def division_euclidienne(a,b):
    """(int,int) -> tuple(int,int)
    Preconditions: a >= 0 et b > 0 avec a et b entiers.
    Si les préconditions ne sont pas respectées, doit renvoyer -1.
    Fonction effectuant la division euclidienne de a par b en utilisant
    l'algorithme par soustraction. Renvoie un doublet (quotient,reste), de
    sorte que a = quotient*b + reste avec 0 <= reste < b.

    Cas "normaux" de division euclidienne
    >>> division_euclidienne(10, 2)
    (5, 0)
    >>> division_euclidienne(2, 10)
    (0, 2)
    >>> division_euclidienne(37, 3)
    (12, 1)

    Si les arguments sont négatifs -> Erreur
    >>> division_euclidienne(-10, 7)
    -1
    >>> division_euclidienne(10, -7)
    -1

    Si les arguments ne sont pas entiers -> Erreur
    >>> division_euclidienne(10.3, 4)
    -1
    >>> division_euclidienne(11, 3.5)
    -1

    Division de 0
    >>> division_euclidienne(0, 3)
    (0, 0)

    Division par 0
    >>> division_euclidienne(3, 0)
    -1
    >>> division_euclidienne(0, 0)
    -1
    """
    return None # Il va falloir écrire quelque chose


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

Lorsqu'on exécute Python sur le fichier précédent, on obtient

```
[...]
File "exemple_doctest.py", line 36, in __main__.division_euclidienne
Failed example:
    division_euclidienne(0,0)
Expected:
    -1
Got nothing
**********************************************************************
1 items had failures:
  10 of  10 in __main__.division_euclidienne
***Test Failed*** 10 failures.
```

Alors que si la fonction est correctement remplie, on obtient, en ayant pris soin d'exécuter Python en mode verbeux (option `-v`)

```
[...]
Trying:
    division_euclidienne(0, 0)
Expecting:
    -1
ok
1 items had no tests:
    __main__
1 items passed all tests:
  10 tests in __main__.division_euclidienne
10 tests in 2 items.
10 passed and 0 failed.
Test passed.
```

Si le mode verbeux n'est pas activé, une exécution correcte de tous les tests n'affiche simplement rien du tout.
L'avantage de cette méthode est que tout est directement documenté (on en a même profité ici pour donner des spécifications sur les types attendus en entrée et en sortie). L'inconvénient étant que la docstring peut rapidement devenir énorme et qu'il faut parfaitement connaître le format de sortie de la console Python (renvoyer `(0,0)` n'est pas la même chose que `(0, 0)` par exemple alors que `assert division_euclidienne(0, 3) == (0, 0)` aura le même effet que `assert division_euclidienne(0,3)`

4. Conclusion

Même si l'écriture de tests ne garantit en rien qu'un programme est correct, le fait de réfléchir aux cas limites que l'on peut rencontrer et en écrire une vérification explicite permet d'assainir l'environnement de développement. En particulier dans le cadre d'une application réelle où les demandes sont souvent amenées à être plus spécifiques à mesure que le temps avance, avoir une batterie de tests sous la main permet d'étendre le code existant sans avoir (trop) peur que la modification ne casse complètement ce qui existait déjà. Tout du moins, on s'en rend compte rapidement.
D'un point de vue pédagogique, réfléchir *a priori* aux cas qui pourront se présenter en théorie et en pratique permet souvent une meilleure conception des fonctions demandées avec un soin particulier porté aux spécifications des cas particuliers qui doivent souvent être traités avant le cas général et ne pas être oubliés dans la bataille.

Références

[1] INRIA. The coq proof assistant. https://coq.inria.fr/about-coq.

[2] Robert C. Martin. The cycles of tdd. http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html.

[3] Documentation Python. doctest : test interactive python examples. https://docs.python.org/3.7/library/doctest.html.

[4] C. Tasson. Cours, td et tp de preuve de programme. https://www.irif.fr/~tasson/doc/cours/cours_pp.pdf.