Chapitre 3
Les tests unitaires en Python

1. Pourquoi les tests unitaires ?

Le lecteur peut être surpris de commencer son apprentissage par les tests unitaires avant de s'exercer aux tests fonctionnels. Malgré la différenciation de ces deux types de tests, ils n'en restent pas moins des tests. Ils se basent donc sur la même logique et les mêmes éléments de syntaxe.

Effectivement, en Python, un test fonctionnel s'écrit à travers la structure d'un test unitaire : quel que soit l'élément que nous testons, une fonction en test unitaire ou un comportement en test fonctionnel, nous posons une assertion à propos de cet élément et nous en vérifions la validité par la suite. Vous pouvez voir le principe d'assertion comme la condition d'une instruction conditionnelle, dans le sens où elle est soit vraie, i.e. le test est accepté, ou fausse, i.e. le test est rejeté.

2. Assertions

Nous pourrions penser qu'un test est une simple instruction conditionnelle `if`, car son résultat est un booléen, le test est validé ou non. Avec cette logique, l'écriture des tests serait rébarbative, non différenciée de l'écriture du code, et nécessiterait une convention interne de code pour bien organiser le code et les tests ainsi que leur validation. Il nous faudrait indiquer ce que nous devons afficher lorsqu'un test réussit, lorsqu'il échoue, indiquer un nouveau fichier principal pour lancer les tests, etc. Un réel bazar qui ne serait pas pratique donc pas maintenable.

Python intègre une instruction spécifique pour valider un test : `assert`.

```
assert test_conditionnel, [message]
```

L'instruction `assert` est directement suivie par le test et peut contenir un message à afficher lorsque le test est rejeté. L'instruction représentant le test doit toujours pouvoir être interprétée comme un booléen.

Si le test est valide, i.e. une instruction retournant `True`, l'instruction `assert` laisse l'interpréteur Python continuer l'exécution du test. Dans le cas contraire, elle renvoie une exception `AssertionError` qui stoppe l'exécution du reste du script.

```
def divide(a, b):
    assert b > 0, "Division par zéro impossible !!!"
    return a / b

if __name__ == '__main__':
    print("1 divisé par 2 :", divide(1, 2))
    print("1 divisé par 0 :", divide(1, 0))
    print("3 divisé par 2 :", divide(3, 2))
```

L'exemple précédent permet de tester qu'une division n'a pas de diviseur nul, ce qui n'est pas autorisé en mathématiques. Le premier appel à la fonction `divide` contenant l'assertion se déroule correctement, mais le deuxième appel lance une exception et stoppe l'exécution du script avec l'affichage suivant :

```
1 divisé par 2 0.5
Traceback (most recent call last):
  File "codeChapitre3.py", line 7, in <module>
```

```
    print("1 divisé par 0", divide(1,0))
                             ^^^^^^^^^^^
  File "codeChapitre3.py", line 2, in divide
    assert b > 0, "Division par zéro impossible !!!"
           ^^^^^
AssertionError: Division par zéro impossible !!!
```

Nous retrouvons bien le message que nous avons défini en argument de l'assertion dans la *traceback* de l'interpréteur Python. Si nous n'avions pas complété l'assertion avec ce message, nous n'aurions eu aucune information sur la raison du rejet du test, nous aurions simplement eu la ligne du script ayant soulevé l'exception, comme le montrent le code suivant et son affichage console :

```
def divide(a, b):
    assert b > 0
    return a / b

if __name__ == '__main__':
    print("1 divisé par 2", divide(1,2))
    print("1 divisé par 0", divide(1,0))
    print("3 divisé par 2", divide(3,2))

# Sortie console
1 divisé par 2 0.5
Traceback (most recent call last):
  File "codeChapitre3.py", line 7, in <module>
    print("1 divisé par 0", divide(1,0))
                             ^^^^^^^^^^^
  File "codeChapitre3.py", line 2, in divide
    assert b > 0
           ^^^^^
AssertionError
```

Remarque

L'argument représentant le message à afficher dans l'instruction `assert` de Python a une importance capitale pour des tests maintenables et clairs pour les développeurs.

3. Léger tour d'horizon des tests unitaires

Python étant un langage libre, ouvert et surtout avec une grande communauté très active, il existe une multitude de librairies de tests pour écrire des tests unitaires. Dans cette section, nous ne présenterons que les plus connues et/ou utilisées pour la rédaction de tests unitaires.

3.1 Script utilisé pour le comparatif

Afin de présenter les différentes librairies de tests en Python, nous allons implémenter un script représentant deux fonctions simples d'une calculette :

– une division entre deux numériques ;

– une soustraction entre deux numériques.

```
# Fonction de la division
def divide(a, b):
    return a / b

# Fonction de la soustraction
def substract(a, b):
    return a - b
```

3.2 Librairie Unittest

La librairie Unittest est historiquement la première librairie en Python permettant d'automatiser les tests unitaires.

Remarque

Pour le lecteur connaissant le langage Java, cette librairie s'inspire de l'API JUnit.

La librairie Unittest est installée avec l'installation de Python, nous pouvons donc la voir comme la version native des tests unitaires de Python.

Pour écrire un test avec cette librairie, nous devons l'importer dans le script, puis créer une nouvelle classe héritant de `TestCase` et représentant tous les tests que nous voulons effectuer. La convention est de déclarer une méthode par test : chaque méthode n'aura donc qu'une seule instruction d'assertion. Cette convention nous permet de mieux comprendre les tests qui sont rejetés et donc de corriger notre code plus rapidement et surtout de manière plus cohérente.

La librairie Unittest utilise ses propres instructions d'assertion, dont :

– La méthode `assertEqual(paramètre1, paramètre2)` testant que le premier paramètre retourne bien la valeur du deuxième paramètre.
– La méthode `assertTrue(condition)` qui lance une exception lorsque la condition est fausse.
– La méthode `assertFalse(condition)` qui est l'inverse de la méthode `assertTrue`.

Le lancement des tests de la classe se fait par l'appel de l'instruction `unittest.main()`.

```
import unittest
import calculette_fonctions

# Déclaration de la classe contenant tous les tests
class Test(unittest.TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
          self.assertEqual(calculette_fonctions
.divide(5, 5), 1)

    def testSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    unittest.main()
```

Nos deux tests sont valides et nous obtenons l'affichage console suivant :

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Ajoutons maintenant un test qui sera rejeté en déclarant que *5 – 5* doit retourner 10.

```
import unittest
import calculette_fonctions


# Déclaration de la classe contenant tous les tests
class Test(unittest.TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
          self.assertEqual(calculette_fonctions
.divide(5, 5), 1)

    def testRejectedSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 10)

    def testSubstract(self):
          self.assertEqual(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    unittest.main()
```

L'un des avantages de la librairie Unittest, comparativement à la simple instruction `assert` de Python, vient du fait qu'un test rejeté n'interrompt pas les autres tests. Il est juste indiqué sur la sortie console que l'assertion est fausse.

```
.F.
======================================================================
FAIL: testRejectedSubstract (__main__.Test.testRejectedSubstract)

----------------------------------------------------------------------
Traceback (most recent call last):
  File "/ unittest-exemple.py", line 13, in testRejectedSubstract
    self.assertEqual(calculette.substract(5,5),10)
AssertionError: 0 != 10

----------------------------------------------------------------------
Ran 3 tests in 0.001s
```

3.3 Librairie DocTest

La librairie DocTest est également intégrée à la distribution officielle de Python, il n'y a donc pas d'installation à faire.

Elle se base sur la documentation des scripts avec les docstrings. En plus de donner une description de la fonction (paramètres et retour) dans la docstring, nous devons fournir des exemples d'interprétation par Python.

```
# Fonction de la division
def divide(a, b):
    """
          paramètre a, b : number
          retour : number
          Exemples :
          >>> divide(1, 2)
          0.5
          >>> divide(2, 1)
          1
    """
    return a / b

# Fonction de la soustraction
def substract(a, b):
```

```
    """
          paramètre a, b : number
          retour : number
          Exemples :
          >>> substract(1, 2)
          -1
          >>> substract(2, 1)
          1
    """
    return a - b

if __name__ == '__main__':
    # Lancement des tests décrits dans les docstrings
    import doctest
    doctest.testmod()
```

Nous indiquons les tests à exécuter ainsi que le résultat attendu dans la docstring grâce aux triples chevrons représentant l'appel de l'interpréteur Python et le saut de ligne pour indiquer l'affichage console attendu lors du test.

Remarque

Lors de l'exécution des tests unitaires avec DocTest, l'interpréteur Python compare, non pas la valeur, mais ce qui est affiché, ce qui peut être déroutant.

Pour lancer les tests unitaires avec DocTest, nous devons simplement exécuter le script avec l'interpréteur Python.

```
**********************************************************************
File "doctest-exemple.py", line 9, in __main__.divide
Failed example:
    divide(2,1)
Expected:
    1
Got:
    2.0
**********************************************************************
1 items had failures:
   1 of   2 in __main__.divide
***Test Failed*** 1 failures.
```

Comme avec la librairie Unittest, le test rejeté n'est pas bloquant, tous les tests sont bien exécutés. Nous remarquons également que, pour un test qui a échoué, DocTest affiche l'instruction, le résultat attendu et le résultat obtenu.

3.4 Librairie Testify

La librairie Testify est le successeur de la librairie Unittest. Les tests sont écrits dans les méthodes d'une classe héritant de `TestCase`. L'avantage majeur de Testify est de pouvoir lancer des méthodes automatiques selon le cycle de vie de la classe, par exemple avant ou après tous les tests.

Pour installer Testify, nous devons utiliser la ligne de commande pip :

```
pip install testify
```

Pour lancer les tests, nous utilisons l'instruction `run()`.

```
from testify import *
import calculette_fonctions

# Déclaration de la classe contenant tous les tests
class Test(TestCase):

    # une méthode par test
    # => un seul assert par méthode
    def testDivide(self):
           assert_equal(calculette_fonctions
.divide(5, 5), 1)

    def testRejectedSubstract(self):
          assert_equal(calculette_fonctions
.substract(5, 5), 10)

    def testSubstract(self):
          assert_equal(calculette_fonctions
.substract(5, 5), 0)

if __name__ == '__main__':
    # Lancement de tous les tests de la classe Test
    run()
```