

# Les test unitaires

# Table des matières

<b>I. Introduction</b>	<b>3</b>
<b>II. Exercice : Quiz</b>	<b>6</b>
<b>III. Mise en pratique</b>	<b>7</b>
<b>IV. Exercice : Quiz</b>	<b>16</b>
<b>V. Essentiel</b>	<b>17</b>
<b>VI. Auto-évaluation</b>	<b>17</b>
A. Exercice .....	17
B. Test .....	17
<b>Solutions des exercices</b>	<b>18</b>

# I. Introduction

**Durée :** 1 h 30

**Prérequis :** avoir suivi les cours précédents

**Environnement de travail :** ordinateur connecté à Internet

## Contexte

La notion de test unitaire n'est pas toujours évidente à saisir en règle générale. Après tout, on a écrit soi-même son code, il fonctionne ; on a effectué des *prints* pour déceler la moindre petite erreur, et l'on sait que le code est parfaitement fonctionnel. Néanmoins, les tests unitaires sont une partie essentielle à ne surtout pas négliger, ils permettent de tester automatiquement le code afin de s'assurer qu'il fonctionne comme prévu. On peut très bien tester son code à la main, mais à ce stade de la formation, mais avec des fichiers qui font plusieurs centaines de lignes, réparties sur plusieurs fichiers, cela peut s'avérer fastidieux. Il est important de tester ses fichiers afin de prévenir les erreurs en amont, et avant la mise en production des *scripts*. Ce cours vous montrera pourquoi il est important de tester son code. Il vous apprendra à mettre en place des tests unitaires grâce à différentes bibliothèques, comme *Unittest* ou *Pytest*.

## Pourquoi faut-il tester son code ?

L'intérêt de créer des tests se présente quand on commence à avoir des programmes conséquents.

Gardons bien en tête que si l'on a un petit fichier, voire deux, avec pour chacun une seule classe, on va facilement pouvoir effectuer les tests soi-même, sans soucis, et le code sera parfaitement fonctionnel. Imaginons maintenant qu'il y a plus d'une dizaine de classes réparties dans plusieurs fichiers différents, et qu'il faut les tester à chaque modification du code pour voir si tout fonctionne correctement : tester son code efficacement s'avère indispensable.

## Attention

Les tests ne garantissent pas à 100 % le caractère fonctionnel et infaillible du code. Il peut se présenter un bogue. Nous sommes humains, il est possible de ne pas tester tous les cas de figures possibles qui permettraient de déceler un bogue.

## Où doit-on placer les tests ?

Ici, aucune règle précise pour savoir où placer les fichiers de tests unitaires. Cependant, plusieurs solutions s'offrent à vous.

- Dans le cas d'un projet « *simple* », avec peu de fichiers, il est tout à fait possible de placer tous les tests unitaires au sein d'un seul et même fichier, et au même niveau que les fichiers de votre projet.
- Il est également possible de créer un fichier de test pour chaque fichier de votre projet. Par exemple, si votre projet contient trois fichiers, alors nous créons trois fichiers de tests.
- Enfin, il est possible de créer un dossier que nous appelons « *tests* » et qui va accueillir l'ensemble de nos fichiers de tests.

La dernière option est la plus pratique en termes de cadre de travail. Elle permet de ne pas se perdre dans un dossier de travail contenant beaucoup de fichiers. Il est vivement conseillé de nommer le début de chaque fichier de test par « *test\_* » suivi du nom du fichier. Par exemple : « *test\_main.py* ».

Voyons ensemble les différentes librairies qu'il est possible d'utiliser, avant d'écrire votre premier test.

## La librairie Unittest

Cette librairie est très connue. Ce n'est cependant pas la plus utilisée. Attention à la petite subtilité dans le nom de la librairie, cela se note bien « *unittest* » avec deux T. Cette librairie n'est pas la plus pratique pour réaliser ses tests, mais elle reste très pratique car elle est incluse directement avec Python. Aucune installation supplémentaire n'est requise.

## Doctest

Doctest permet de créer rapidement et facilement des tests unitaires. Avec cette librairie, nous allons créer nos tests directement dans les « *Docstring* ». Pour rappel, une docstring est une documentation sous forme de chaîne de caractères, directement écrite au sein des fichiers de code. Elle s'écrit de cette façon : `"""Je suis un exemple de Docstring"""`.

```
1 import math
2 def factorial(n):
3     """return the factorial of n, an exact integer >= 0.
4     >>> [factorial(n) for n in range(6)]
5     [1, 1, 2, 6, 24, 120]
6     >>> factorial(5)
7     120
8     >>> factorial(10)
9     12345
10    >>> factorial(-1)
11    Traceback (most recent call last):
12    ...
13    ValueError: n must be >= 0
14    """
15    if not n >= 0:
16        raise ValueError("n must be >= 0")
17    if math.floor(n) != n:
18        raise ValueError("n must be exact integer")
19    if n + 1 == n:
20        raise OverflowError("n too large")
21    result = 1
22    factor = 2
23    while factor <= n:
24        result *= factor
25        factor += 1
26    return result
27 if __name__ == "__main__":
28     import doctest
29     doctest.testmod()
```

Au sein de la fonction factorielle, vous allez utiliser `>>>` afin de donner une donnée à tester, tout comme dans un terminal Python. Juste en dessous, vous allez mettre la donnée ou le message d'erreur attendu.

Si vous ne comprenez pas tout ce qui est écrit au sein du code, ne vous inquiétez pas. Prenez votre temps pour analyser le code.

Pour lancer un test unitaire avec Doctest, vous pouvez simplement lancer votre fichier en ligne de commande en ajoutant l'option « `-v` ». Cette option va permettre de donner l'ensemble du processus de test au sein de votre terminal.

```

Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    factorial(10)
Expecting:
    12345
*****
File "C:\Users\user\AppData\Local\Temp\1\doctest_example.py", line 13, in __main__.factorial
Failed example:
    factorial(10)
Expected:
    12345
Got:
    3628800
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
    ...
    ValueError: n must be >= 0
ok
1 items had no tests:
    __main__
*****
1 items had failures:
    1 of 4 in __main__.factorial
4 tests in 2 items.
3 passed and 1 failed.
***Test Failed*** 1 failures.

```

### Exemple de retour du terminal

## Pytest

Pytest est probablement la librairie de test la plus utilisée en général. Cependant, elle n'est pas fournie avec Python par défaut et doit être installée avec pip. Sa syntaxe est plus simple et plus épurée. Pas de solutions miracles, il faut tout de même écrire soi-même les tests, mais cela peut se faire de manière plus simple et, par conséquent, avec un peu plus de rapidité. De nombreux plugins sont disponibles afin d'augmenter les fonctionnalités de cette librairie, sans pour autant alourdir le dossier de tests. L'exécution de tests se fait de manière plus rapide que sur unittest.

Un autre point fort est que Pytest est compatible avec unittest. Cela signifie que si vous commencez à écrire des tests avec unittest et que vous souhaitez migrer votre code vers Pytest, cela pourra se faire de manière simple, sans avoir à réécrire l'ensemble des tests.

## Le « Test Driven Development » ou le TDD

Pour votre premier test, nous allons prendre un exemple très simple :

```

1 def add(a, b):
2     return a + b

```

Dans l'exemple ci-dessus, vous avez une fonction simple qui va retourner la valeur de  $a + b$ . De manière logique, vous savez parfaitement qu'il faut alors insérer deux chiffres (ou nombres) en paramètres de cette fonction, afin de retourner une addition. On pense alors que la fonction fonctionne correctement et ne pourra jamais faire « planter » le script.

Cependant, que se passerait-il si vous passiez par exemple deux chaînes de caractères ou deux booléens ? Dans votre exemple, votre fonction est déjà écrite, déjà présente, et vous allez écrire les tests après.

Il existe cependant une autre façon de développer qui s'appelle le « *Test Driven Development* » ou, en français, le « *développement piloté par les tests* ». Cela signifie tout simplement qu'il faut d'abord écrire les tests, puis écrire le code. Cela peut paraître absurde, mais c'est une méthode terriblement efficace dans le développement. En effet, vous allez d'abord penser à la façon dont la fonction *doit* fonctionner et prévoir les principaux cas de figures qui pourraient faire échouer le code avant de l'écrire.

On pourrait imaginer ce fonctionnement grâce à une métaphore : imaginez que vous souhaitiez organiser une sortie en famille. Vous allez d'abord planifier votre sortie en pensant à ce qui va être fait et au temps qu'il va faire. Vous penserez ensuite aux cas de figures qui pourraient ternir cette journée, et vous allez alors mettre en place des façons de résoudre ces problèmes.

Dans une certaine logique, pour créer votre test, vous allez écrire votre fonction qui ne fonctionne pas et votre objectif sera alors de la faire fonctionner. Dans votre exemple, vous aurez alors tendance à écrire un test pour vérifier si votre fonction vous permet bel et bien d'additionner deux nombres, puis nous écrirons un test afin de pouvoir additionner deux chaînes de caractères, etc. L'idée est d'abord d'écrire le test, puis de valider le test.

#### Définition mock

Les mock sont des objets factices ou des pseudo - objets qui, dans les tests unitaires et le développement piloté par les tests (TDD), font croire au code du programme à tester qu'ils ont certaines fonctionnalités, mais ne les implémentent généralement pas réellement. Ils servent de substitut aux objets réels lors des tests.

## Exercice : Quiz

[solution n°1 p.19]

### Question

Pytest et Unittest sont des modules déjà fournis avec Python.

- ☐ Vrai
- ☐ Faux

### Question 2

Les tests unitaires permettent de vérifier le fait que son code est infaillible en toutes circonstances.

- ☐ Vrai
- ☐ Faux

### Question 3

Le Test Driven Development (TDD) est le fait de tester son code avant même de l'avoir écrit.

- ☐ Vrai
- ☐ Faux

### Question 4

Où doit-on placer ses tests ?

- ☐ En dehors du projet
- ☐ À la racine du projet
- ☐ Dans un dossier qui s'appelle « tests »

### Question 5

Parmi la liste suivante, quels modules avez-vous abordés pour effectuer des tests ?

- ☐ Doctest
- ☐ Unittest
- ☐ Pythontest
- ☐ Codetest
- ☐ Pytest

### III. Mise en pratique

#### Méthode Premier test

Passez à l'écriture de votre premier test. Dans un fichier `main.py`, vous allez écrire votre fonction `add()`.

Puis, dans un second fichier `test.py`, vous allez écrire votre test.

Vous allez utiliser la class `TestCase` dans `Unittest`. Pour cela, ajoutez tout en haut de votre fichier `test.py` « `from unittest import TestCase` ».

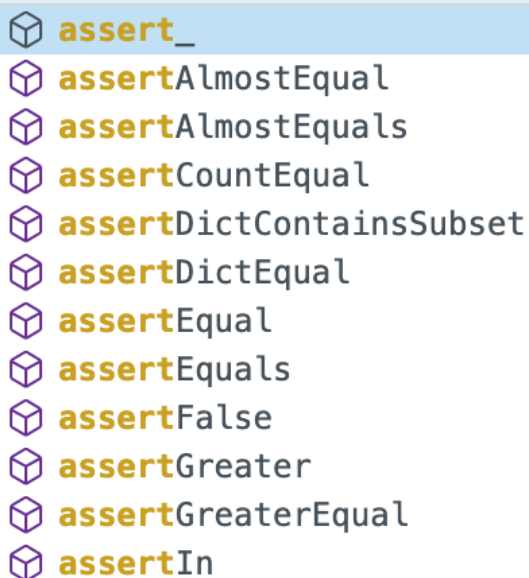
Créez une classe dans le fichier que l'on nommera `TestCalculatrice`. Cette classe doit hériter de `TestCase`.

Ajoutez une fonction qui commence forcément par le préfixe « `test-` ». Autrement, la fonction ne sera pas considérée comme une fonction de test et ne sera donc pas exécutée par le programme de test.

Il faut toujours être le plus précis possible dans le nom de ses fonctions de tests. Il n'est pas rare de voir des fonctions avec un nom très long, mais c'est pour permettre d'être le plus précis possible au sujet du test que l'on souhaite faire passer. En effet, lorsqu'une fonction n'est pas correctement exécutée ou lorsqu'elle retourne une erreur, c'est le nom de la fonction qui est retourné dans le terminal. C'est pour cette raison qu'il faut être le **plus précis possible**.

Nommez votre fonction « `test-add-two-numbers` ». Utilisez l'attribut `self`, car vous souhaitez que la fonction se réfère à tout ce qui se trouve au sein de la classe.

Pour le premier attribut, vous allez utiliser « `self.assertEqual()` ». Vous pouvez le voir ci-dessous, lorsque l'on commence à écrire `assert`, il y a de nombreux asserts disponibles. Ces derniers sont assez explicites.



```

assert_
assertAlmostEqual
assertAlmostEquals
assertCountEqual
assertDictContainsSubset
assertDictEqual
assertEquals
assertEquals
assertFalse
assertGreater
assertGreaterEqual
assertIn
  
```

### Fenêtre retournée par l'IDE

Utilisez comme précisé juste au-dessus, `assertEqual`.

Importez la fonction au sein du fichier de test : pour ce faire, il faut ajouter, en dessous de la ligne d'import d'`unittest`, la ligne suivante : « `from main import add` ».

De retour dans votre `assertEqual`, vous allez donc vérifier :

```
1 from main import add
2 from unittest import TestCase
3 class TestCalculatrice(TestCase):
4     def test_add_two_numbers(self):
5         self.assertEqual(add(5, 10), 15)
```

Vérifiez que le résultat de votre fonction `add`, dont les paramètres sont 5 et 10, est bien égal à 15.

Vous avez écrit votre premier test.

### Méthode Exécuter des tests unitaires

Pour exécuter des tests unitaires, la méthode la plus basique est simplement d'importer dans son fichier de test, en plus de `TestCase`, la fonction `main`, qui provient également d'`unittest` :

```
1 from main import add
2 from unittest import TestCase, main
3 class TestCalculatrice(TestCase):
4     def test_add_two_numbers(self):
5         self.assertEqual(add(5, 10), 15)
6 main()
```

Vous pouvez maintenant exécuter le fichier de test depuis le terminal.

```

.
-----
Ran 1 test in 0.000s
OK

```

Votre terminal vous confirme alors que le test a été correctement exécuté et qu'il est bon. Ici, vous n'avez aucun détail, mais vous pouvez obtenir une version un peu plus explicite dans votre terminal, en ajoutant l'option `-v` :

```

test_add_two_numbers (__main__.TestCalculatrice) ... ok
-----
Ran 1 test in 0.000s
OK

```

### Retour du terminal

Vous avez alors le nom de la fonction qui a été exécutée et vous savez si elle fonctionne ou non.

Modifiez votre fonction de test et que vous souhaitez vérifier si 5 et 10 retournent 20, il faut modifier votre méthode par « `assertEqual(add(5,10), 20)` ».



```

F
=====
FAIL: test_add_two_numbers (__main__.TestCalculatrice)
Traceback (most recent call last):
  File "/Tests Unitaires/test.py", line 8, in test_add_two_numbers
    self.assertEqual(add(5, 10), 20)
AssertionError: 15 != 20
=====
Ran 1 test in 0.000s
FAILED (failures=1)

```

### Retour du terminal

Observez ici le fait que le terminal vous renvoie une erreur et qu'il vous décrit exactement l'origine de l'erreur. Ici, il y a une différence entre 15 et 20, ce qui fait que votre test a échoué.

Une autre façon de lancer des tests de son code est de lancer directement le test depuis le terminal, sans rien importer de plus dans le fichier de test.

Revenez à l'étape précédant l'importation de `main` de votre module `Unittest`. Supprimez donc cet import et l'appel de la fonction `main()`.

```

1 from main import add
2 from unittest import TestCase
3 class TestCalculatrice(TestCase):
4     def test_add_two_numbers(self):
5         self.assertEqual(add(5, 10), 15)

```

Maintenant, dans le terminal, rentrons la ligne de commande suivante :

```
python -m unittest
```

Ici, plusieurs options s'offrent à vous. Dans un premier temps, l'ajout de l'option `-m` signifie que vous souhaitez utiliser un module. Ensuite, vous précisez que ce module est simplement `unittest`. Nul besoin de préciser le fichier que nous souhaitons exécuter, `Unittest` sait reconnaître tous les fichiers commençant par « `test` ». Si vous voulez tester un fichier en particulier, vous pouvez alors renseigner le fichier en l'ajoutant à la suite de la ligne de commande, comme `python -m unittest test.py`. Sans cette précision, `Unittest` va lancer tous les fichiers de test présents dans le dossier de travail.

## Compléter les tests

Jusqu'ici, nous avons testé la fonction uniquement avec deux nombres. Nous allons maintenant tester la fonction avec d'autres arguments.

Testons d'abord votre fonction avec des lettres :

```

1 def test_add_two_strings(self):
2     self.assertEqual(add("a", "b"), "ab")

```

Vous pouvez également tester avec des booléens. Par défaut, il faut que chaque fonction de test ne puisse tester qu'une seule et unique chose. Dans votre cas, vous allez ajouter plusieurs tests. En effet, il y a deux types de booléens et donc, trois cas de figure possibles. Dans les booléens, `True` est égal à 1, et `False` à 0. Nous allons donc prendre en compte ces données pour ajuster nos tests :

```

1 def test_add_two_booleans(self):
2     self.assertEqual(add(True, False), 1)
3     self.assertEqual(add(True, True), 2)
4     self.assertEqual(add(False, False), 0)

```

Ici, si vous lancez votre test, tout se passera sans encombre, et vous le savez très bien.

Abordons ensemble un autre cas de figure : nous souhaitons que votre fonction retourne une erreur, nous voulons qu'un cas de figure précis vous renvoie une erreur et ne puisse pas fonctionner. Prenons un exemple avec votre fonction d'addition. Si nous essayons d'additionner deux `None`, votre fonction vous retourne forcément une erreur :

```
Traceback (most recent call last):
  File "Tests Unitaires/main.py", line 5, in <module>
    add(None, None)
  File "Tests Unitaires/main.py", line 2, in add
    return a + b
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Retour du terminal

Mais dans ce cas, comment écrire ce test, en faisant en sorte qu'il puisse signaler le fait que la fonction ne fonctionne pas ?

Pour ce faire, nous allons utiliser une déclaration différente dans votre test :

```
1 def test_add_with_two_none(self):
2     with self.assertRaises(TypeError):
3         add(None, None)
```

Vous allez utiliser l'argument `with` qui est un *context manager* (manager de contexte). Il va vérifier le fait que votre `assertRaises` est bien une erreur de type `TypeError`, dans votre fonction `add`, avec les arguments `None`. Ainsi, votre terminal vous signale désormais que votre test a bien été exécuté, et il retourne bien une erreur au lancement de la fonction. Le test vérifie alors que la fonction ne marche pas. C'est un peu paradoxal, il faut en convenir, mais vous pouvez avoir une fonction qui ne fonctionne pas, mais un test qui, lui, fonctionne.

```
test_add_two_booleans (test.TestCalculatrice) ... ok
test_add_two_numbers (test.TestCalculatrice) ... ok
test_add_two_strings (test.TestCalculatrice) ... ok
test_add_with_two_none (test.TestCalculatrice) ... ok
```

Ran 4 tests in 0.000s

OK

Retour du terminal

Si, au contraire, vous souhaitez que votre fonction d'addition fonctionne avec des arguments `None`, il faut alors gérer ce cas de figure et modifier votre fonction dans le fichier `main` directement. Par exemple, vous pouvez souhaiter que la fonction `add`, avec deux arguments `None`, retourne `-1`.

### Méthode Afficher la couverture des tests

Pour afficher la couverture de test, vous allez installer `coverage` avec `pip`. Pour ce faire, rendez-vous dans le terminal et entrons la commande `pip install coverage`.

Maintenant que vous avez installé `coverage`, vous pouvez lancer la commande `coverage run -m unittest`. Vous pouvez observer qu'un fichier `.coverage` a été créé dans le dossier de travail.

Vous pouvez désormais lancer la commande `coverage html`, afin de créer un dossier `htmlcov`, dans lequel vous trouverez un fichier `index.html`. Celui-ci va venir vous montrer le `coverage` réalisé par nos tests.

Ouvrons ce fichier dans un navigateur. Pour ce faire, rien de plus simple, on peut faire glisser le fichier directement depuis le dossier dans votre navigateur.

Module	statements	missing	excluded	coverage ↑
main.py	2	0	0	100%
test.py	14	0	0	100%
<b>Total</b>	<b>16</b>	<b>0</b>	<b>0</b>	<b>100%</b>

Vous pouvez voir que l'ensemble du code a été couvert par les tests. Vous pouvez même cliquer sur les modules afin d'afficher des détails supplémentaires.

## Coverage for test.py: 100%

14 statements

14 run

0 missing

0 excluded

```
1
2 from main import add
3 from unittest import TestCase
4
5
6 class TestCalculatrice(TestCase):
7     def test_add_two_numbers(self):
8         self.assertEqual(add(5, 10), 15)
9
10    def test_add_two_strings(self):
11        self.assertEqual(add("a", "b"), "ab")
12
13    def test_add_two_booleans(self):
14        self.assertEqual(add(True, False), 1)
15        self.assertEqual(add(True, True), 2)
16        self.assertEqual(add(False, False), 0)
17
18    def test_add_with_two_none(self):
19        with self.assertRaises(TypeError):
20            add(None, None)
```

### Attention

Avoir un code couvert à 100 % ne veut pas dire qu'il sera infaillible en toutes circonstances. Encore une fois, cela ne représente pas la qualité du code. Nous pouvons très bien avoir un code testé à 100 %, mais ne pas avoir pensé à tous les cas de figures possibles. On découvrira donc une faille par la suite.

Si vous ajoutez du code dans votre fichier `main` et que vous relancez les commandes `coverage`, vous allez forcément avoir une baisse du pourcentage de votre coverage. En effet, la fonction n'aura pas été testée, et par conséquent, ne sera pas exécutée. Les chiffres sont donc à prendre avec des pincettes. Il faut toujours essayer de penser à chaque cas de figure possible lorsque l'on écrit un test.

### Fondamental

Résumé des commandes :

- `Coverage run -m unittest` : lance les tests du dossier de travail et crée un fichier `.coverage`.
- `Coverage html` : crée un dossier `htmlcov` dans lequel vous pouvez retrouver le fichier `index.html`.

### Méthode Installation et exploitation de PyTest

**Étape 1** : pour installer PyTest, passez par le terminal avec `pip`, en lançant la commande `pip install pytest`.

**Étape 2** : pour lancer les tests Unittest avec Pytest, utilisez simplement la commande `pytest` en précisant le nom du fichier et son extension : « `pytest test.py` ».

```
===== test session starts =====
platform darwin -- Python 3.9.10, pytest-7.0.1, pluggy-1.0.0
rootdir: [redacted] /Tests Unitaires
collected 4 items

test.py .... [100%]

===== 4 passed in 0.01s =====
```

Pour rappel, les quatre tests passés sont les tests qui vous ont été présentés.

**Étape 3** : faites-en sorte d'échouer un test afin de voir. Mettez par exemple 0 en résultat attendu pour votre fonction `test-add-two-numbers` :

```
===== test session starts =====
platform darwin -- Python 3.9.10, pytest-7.0.1, pluggy-1.0.0
rootdir: [redacted] /Tests Unitaires
collected 4 items

test.py .F.. [100%]

===== FAILURES =====
_____ TestCalculatrice.test_add_two_numbers _____

self = <test.TestCalculatrice testMethod=test_add_two_numbers>

    def test_add_two_numbers(self):
>       self.assertEqual(add(5, 10), 0)
E       AssertionError: 15 != 0

test.py:8: AssertionError
===== short test summary info =====
FAILED test.py::TestCalculatrice::test_add_two_numbers - AssertionError: 15 != 0
===== 1 failed, 3 passed in 0.06s =====
```

Vous avez bien une erreur qui stipule que quelque chose a échoué dans votre fichier `test.py`, mais vous n'avez pas beaucoup plus d'informations à ce sujet.

**Étape 4 :** ajoutez l'option -v.

```

===== test session starts =====
platform darwin -- Python 3.9.10, pytest-7.0.1, pluggy-1.0.0 -- /opt/homebrew/opt/python@3.9/bin/python3.9
cachedir: .pytest_cache
rootdir: [REDACTED]/Tests Unitaires
collected 4 items

test.py::TestCalculatrice::test_add_two_booleans PASSED [ 25%]
test.py::TestCalculatrice::test_add_two_numbers FAILED [ 50%]
test.py::TestCalculatrice::test_add_two_strings PASSED [ 75%]
test.py::TestCalculatrice::test_add_with_two_none PASSED [100%]

===== FAILURES =====
_____ TestCalculatrice.test_add_two_numbers _____

self = <test.TestCalculatrice testMethod=test_add_two_numbers>

    def test_add_two_numbers(self):
>     self.assertEqual(add(5, 10), 0)
E     AssertionError: 15 != 0

test.py:8: AssertionError
===== short test summary info =====
FAILED test.py::TestCalculatrice::test_add_two_numbers - AssertionError: 15 != 0
===== 1 failed, 3 passed in 0.05s =====

```

Vous avez alors un peu plus d'informations sur la fonction qui a échoué. Vous savez désormais quelle est la fonction qui échoue et vous pouvez alors corriger votre erreur.

**Méthode**

Comme vous avez pu le voir, les tests écrits avec unittest peuvent être lancés directement dans Pytest. Voyons désormais comment écrire nos tests avec Pytest.

Pour écrire des tests unitaires avec Pytest, vous allez créer un nouveau fichier afin de pouvoir comparer les deux façons d'écrire des tests. Créons un fichier `test-pytest.py`.

Afin de déclarer des tests, nul besoin de créer une classe. Vous allez pouvoir directement écrire nos fonctions de test au sein de votre fichier.

Pour ce qui est des `assertEqual`, vous allez voir de quelle façon vous pouvez écrire cela avec Pytest.

```

1 from main import add
2 def test_add_two_numbers():
3     assert add(5, 10) == 15
4 def test_add_two_strings():
5     assert add("a", "b") == "ab"
6 def test_add_two_booleans():
7     assert add(True, False) == 0
8     assert add(True, True) == 2
9     assert add(False, False) == 0

```

Vous allez simplement écrire `assert`, suivi de votre fonction à tester. Il faut insérer des arguments et comparer avec le résultat attendu. Si vous lancez votre fichier dans le terminal, vous obtenez ceci :

```
===== test session starts =====
platform darwin -- Python 3.9.10, pytest-7.0.1, pluggy-1.0.0 -- /opt/homebrew/opt/python@3.9/bin/python3.9
cachedir: .pytest_cache
rootdir: /Users/jean-louis/Code/Python/Pytest/Tests Unitaires
collected 3 items

test_pytest.py::test_add_two_numbers PASSED [ 33%]
test_pytest.py::test_add_two_strings PASSED [ 66%]
test_pytest.py::test_add_two_booleans PASSED [100%]

===== 3 passed in 0.01s =====
```

Pour ce qui concerne votre fonction de tests avec `None`, voici la marche à suivre :

```
1 import pytest
2 def test_add_two_nones():
3     with pytest.raises(TypeError):
4         add(None, None)
```

Vous allez importer `pytest` tout en haut de votre fichier, avec la ligne « `import pytest` ». Vous allez utiliser la fonction « `pytest.raises` », afin de vérifier que votre test relève bien une erreur `TypeError`.

Pour créer un rapport HTML avec Pytest, vous allez installer un module permettant de le générer pour vous. Il ne s'agit pas d'un rapport de coverage, mais simplement d'un rapport des tests ayant réussi ou échoué sous format HTML.

Vous allez installer ce package avec la commande `pip install pytest-html`. Une fois cela fait, vous allez pouvoir à nouveau lancer votre test avec la commande, mais en y ajoutant un argument supplémentaire : « `pytest test.py -v --html=index.html` ». Cette commande signifie que vous souhaitez effectuer nos tests et les sauvegarder dans un fichier `index.html`, à la racine de votre projet. Si vous souhaitez le stocker dans un endroit précis, il faut alors remplacer « `--html=index.html` » par le chemin où vous souhaitez le stocker.



Comme tout fichier html, nous pouvons l'ouvrir dans un navigateur web :

## index.html

Report generated on 18-Feb-2022 at 17:03:12 by `pytest-html` v3.1.1

### Environment

Packages	{"pluggy": "1.0.0", "py": "1.11.0", "pytest": "7.0.1"}
Platform	macOS-12.2.1-arm64-arm-64bit
Plugins	{"html": "3.1.1", "metadata": "1.11.0"}
Python	3.9.10

### Summary

4 tests ran in 0.01 seconds.

(Un)check the boxes to filter the results.

☒ 4 passed, ☒ 0 skipped, ☒ 0 failed, ☒ 0 errors, ☒ 0 expected failures, ☒ 0 unexpected passes

### Results

[Show all details](#) / [Hide all details](#)

▲ Result	▼ Test	▼ Duration	▼ Links
Passed <a href="#">(show details)</a>	test_pytest.py::test_add_two_numbers	0.00	
Passed <a href="#">(show details)</a>	test_pytest.py::test_add_two_strings	0.00	
Passed <a href="#">(show details)</a>	test_pytest.py::test_add_two_booleans	0.00	
Passed <a href="#">(show details)</a>	test_pytest.py::test_add_two_nones	0.00	

Vous pouvez alors voir que Pytest génère un fichier vraiment complet qui permet de savoir les éléments suivants : l'heure, le jour, la version de Python, la version de Pytest, l'environnement sur lequel Pytest a été généré (ici sur mac-OS), et le détail des fonctions que vous allez écrire avant de lancer le test.

Lancez ce même test, mais avec un test qui a échoué :

## Summary

4 tests ran in 0.02 seconds.

(Un)check the boxes to filter the results.

☒ 3 passed, ☒ 0 skipped, ☒ 1 failed, ☒ 0 errors, ☒ 0 expected failures, ☒ 0 unexpected passes

## Results

[Show all details](#) / [Hide all details](#)

▲ Result	▼ Test
<b>Failed</b> (hide details)	test_pytest.py::test_add_two_bo
<pre>def test_add_two_booleans(): &gt;     assert add(True, False) == 0 E       assert 1 == 0 E         +1 E         -0  test_pytest.py:14: AssertionError</pre>	
<b>Passed</b> (show details)	test_pytest.py::test_add_two_nu
<b>Passed</b> (show details)	test_pytest.py::test_add_two_stri
<b>Passed</b> (show details)	test_pytest.py::test_add_two_noi

Observez que Pytest vous renvoie une erreur précise sur ce qui n'a pas fonctionné. Ici, la fonction `add` avec `True` et `False` devait renvoyer 1. Or, vous attendiez 0, donc une erreur est générée.

## Exercice : Quiz

[solution n°2 p.20]

### Question 1

Parmi les noms de fonctions suivantes, quelles sont les fonctions qui seront exécutées par Pytest ?

- ☐ `check-this-function`
- ☐ `my-function-test`
- ☐ `test-my-function`

### Question 2

Un test unitaire doit être indépendant de son projet.

- ☐ Vrai
- ☐ Faux

### Question 3



Une couverture de test est :

- ☐ Un pourcentage qui représente la proportion du code qui a été testé
- ☐ Une couverture réseau afin de vérifier si le projet est bien en ligne
- ☐ Une suite de tests générée automatiquement qui permet de vérifier l'ensemble du code

Question 4

Il faut constamment tester les méthodes privées d'une classe.

- ☐ Vrai
- ☐ Faux

Question 5

Quelle commande permet d'installer Pytest ?

- ☐ pip npm install pytest
- ☐ npm install pytest
- ☐ pip install pytest
- ☐ Pytest est déjà fourni avec Python, nul besoin de l'installer

## V. Essentiel

Nous avons vu ensemble comment écrire nos premiers tests avec des exemples simples. Nous avons également vu qu'il fallait parfois pousser la réflexion assez loin pour essayer de trouver chaque cas de figure qui nécessitait de tester votre fonction. Nous avons abordé ensemble différentes façons d'écrire des tests, notamment avec les modules `doctest`, `unittest` et `pytest`. Pytest est l'un des modules les plus utilisés en général.

Les tests unitaires peuvent être une notion complète et complexe, mais savoir coder des tests unitaires est une compétence essentielle pour tout développeur souhaitant progresser dans son métier. Il s'agit d'un élément essentiel à tout code source pour s'assurer que l'application fonctionne toujours comme cela a été prévu, malgré des évolutions dans le code. De plus, les tests unitaires sont également à la base de bonnes pratiques de l'ingénierie logicielle, telles que le Test Driven Development (TDD) ou l'intégration continue dans une boucle DevOps.

## VI. Auto-évaluation

### A. Exercice

Vous intégrez une équipe de développeurs dans un tout nouveau projet. Vous êtes sollicité afin de mettre en place une méthode de développement.

#### Question 1

[solution n°3 p.21]

Pourquoi choisissez-vous de mettre en place le Test Driven Development (TDD) ?

#### Question 2

[solution n°4 p.21]

Justifiez votre choix.

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.21]

Question 1

Il est possible d'effectuer des tests unitaires au sein d'une docstring.

- ☐ Vrai
- ☐ Faux

#### Question 2

Un *mock* évalue en priorité un objet que nous voulons tester.

- ☐ Vrai
- ☐ Faux

#### Question 3

Lors du Test Driven Development (TDD), il faut écrire tous les tests en même temps, et une fois tous les tests validés, écrire le code.

- ☐ Vrai
- ☐ Faux

#### Question 4

Il est possible de transmettre le rendu de coverage directement via un fichier html.

- ☐ Vrai
- ☐ Faux

#### Question 5

Il est possible de lancer des tests écrits avec Unittest via le module Pytest.

- ☐ Vrai
- ☐ Faux


## Solutions des exercices

**Exercice p. 6 Solution n°1****Question**

Pytest et Unittest sont des modules déjà fournis avec Python.

☐ Vrai

☒ Faux


 Il faut installer Pytest avec pip.

**Question 2**

Les tests unitaires permettent de vérifier le fait que son code est infaillible en toutes circonstances.

☐ Vrai

☒ Faux


 Les tests unitaires ne permettent pas de rendre un code infaillible. Ils permettent seulement de vérifier que sa fonction tourne correctement, selon les cas de figures auxquels on a pensé.

**Question 3**

Le Test Driven Development (TDD) est le fait de tester son code avant même de l'avoir écrit.

☒ Vrai

☐ Faux

 Le concept du TDD réside bel et bien dans le fait de tester une fonction avant même de l'intégrer dans son code.


**Question 4**

Où doit-on placer ses tests ?

☐ En dehors du projet

☒ À la racine du projet

☒ Dans un dossier qui s'appelle « tests »

 Il est possible de placer ses fichiers de test à la racine de son projet. Cependant, dans un souci de lisibilité, il est préférable de les placer dans un dossier nommé « tests ».

**Question 5**

Parmi la liste suivante, quels modules avez-vous abordés pour effectuer des tests ?


☒ Doctest

☒ Unittest

☐ Pythontest

☐ Codetest

☒ Pytest


 Nous avons abordé dans le cours les modules Doctest, Unittest et Pytest.

## Exercice p. 16 Solution n°2

### Question 1

Parmi les noms de fonctions suivantes, quelles sont les fonctions qui seront exécutées par Pytest ?


- ☐ check-this-function
- ☒ my-function-test
- ☒ test-my-function

 En effet, peu importe si le mot clé « test » est mis en début ou fin de nom de fonction, Pytest est capable de reconnaître une fonction de test.

### Question 2


Un test unitaire doit être indépendant de son projet.

- ☒ Vrai
- ☐ Faux

 Il existe en effet plusieurs types de tests : les tests unitaires, les tests d'intégration, les tests de validation, et la phase de recette. Chaque test est à effectuer indépendamment l'un de l'autre.

### Question 3


Une couverture de test est :

- ☒ Un pourcentage qui représente la proportion du code qui a été testé
  - ☐ Une couverture réseau afin de vérifier si le projet est bien en ligne
  - ☐ Une suite de tests générée automatiquement qui permet de vérifier l'ensemble du code
-  Une couverture de test représente bien la proportion du code qui a été testé.

### Question 4


Il faut constamment tester les méthodes privées d'une classe.

- ☐ Vrai
- ☒ Faux

 Une bonne pratique consiste à tester uniquement les méthodes publiques, et non les méthodes privées. Les informations privées ne doivent pas être accessibles en dehors de la classe dans laquelle ces informations existent.

### Question 5

Quelle commande permet d'installer Pytest ?

- ☐ pip npm install pytest
- ☐ npm install pytest
- ☒ pip install pytest
- ☐ Pytest est déjà fourni avec Python, nul besoin de l'installer
-  La commande permettant d'installer Pytest est bien la commande `pip install pytest`.

#### p. 17 Solution n°3

Mettre en place le Test Driven Development va permettre dans un premier temps de réduire le temps de débogage. En effet, vous passerez moins de temps à buter sur un problème car votre code aura préalablement été testé dans les détails.

#### p. 17 Solution n°4

Un des avantages principaux du TDD est la documentation de votre code. Les tests que nous effectuons deviennent votre documentation. Il sera alors plus facile pour un autre développeur qui souhaite travailler avec vous (ou qui doit reprendre le projet) de comprendre ce que doit faire votre code.

Un autre avantage du Test Driven Development est de permettre le test d'une fonction de plusieurs façons. Vous pourrez alors être parés à toute éventualité de bogues. Évidemment, cela n'empêche pas de rencontrer des soucis par la suite, mais cela pourra être corrigé plus rapidement.

Pour finir, il est également possible d'effectuer des revues de code bien plus facilement avec votre équipe grâce à ce test. Cela rendra le code plus présentable et conservera une homogénéité dans le style de codage. Ainsi, il est possible d'améliorer les interactions avec l'équipe.


#### Exercice p. 17 Solution n°5

##### Question 1

---

Il est possible d'effectuer des tests unitaires au sein d'une docstring.

- ☒ Vrai
- ☐ Faux


 Avec Doctype, il est tout à fait possible de créer ses tests unitaires dans des docstrings directement dans le fichier projet.

##### Question 2

---

Un *mock* évalue en priorité un objet que nous voulons tester.

- ☐ Vrai
- ☒ Faux


 Un *mock* va imiter le comportement d'un autre objet que nous voulons tester.

**Question 3**

Lors du Test Driven Development (TDD), il faut écrire tous les tests en même temps, et une fois tous les tests validés, écrire le code.

☐ Vrai

☒ Faux


 L'intérêt majeur du TDD consiste à se concentrer sur un test à la fois.

**Question 4**

Il est possible de transmettre le rendu de coverage directement via un fichier html.

☒ Vrai

☐ Faux


 Il faut au préalable installer le coverage avec la commande `pip install coverage` ou `pip install pytest-html`, selon le module que nous souhaitons utiliser pour effectuer les tests.

**Question 5**

Il est possible de lancer des tests écrits avec Unittest via le module Pytest.

☒ Vrai

☐ Faux

 Pytest est capable d'exécuter les tests écrits avec Unittest, mais la convention d'écriture de Pytest et celle d'Unittest ne sont pas tout à fait similaires.