5_Ecrire un test avec Pytest

Afin de comprendre comment fonctionne Pytest, écrivons le **code source** ci-dessous dans un fichier `source.py` .

```
def reverse_str(initial_string):
    final_string = ''
    index = len(initial_string)
    while index > 0:
        final_string += initial_string[index - 1]
        index = index - 1
    return final_string
```

Ici, nous avons défini une fonction qui, lorsqu'elle sera exécutée, renverra l’inverse de la chaîne de caractères passée en paramètre.

On a ensuite écris une nouvelle fonction, dans le fichier `test.py` , pour tester la fonction `reverse_str(initial_string)` .

Je commence en la nommant `test_should_reverse_string` car il s'agit d'un test qui permettra de vérifier si la fonction renvoie bien l’inverse de la chaîne de caractères. À l'intérieur, on utilise le mot-clé `assert` , puis un espace, et écris ce que je souhaite tester. Ainsi, si l’assertion n’est pas soulevée, le test **passe** !

Dans notre cas, nous souhaitons tester que le résultat de l'exécution de la fonction `reverse_str('abc')` renvoie bien une valeur égale à la chaîne de caractères `"cba"` .

```
from source import reverse_str
```

```python
def test_should_reverse_string():
    assert reverse_str('abc') == 'cba'
```

```
aluboya@PCPW11-PAR-001 MINGW64 ~/Documents/MES COURS/M/Tests Unitaires/TP/1/CM
$ pytest test.py
============================================= test session starts =============================================
platform win32 -- Python 3.10.4, pytest-7.4.4, pluggy-1.3.0
rootdir: C:\Users\aluboya\Documents\MES COURS\M\Tests Unitaires\TP\1\CM
plugins: dash-2.16.1, cov-4.1.0
collected 1 item

test.py .                                                                                          [100%]

============================================== 1 passed in 0.04s ==============================================
```

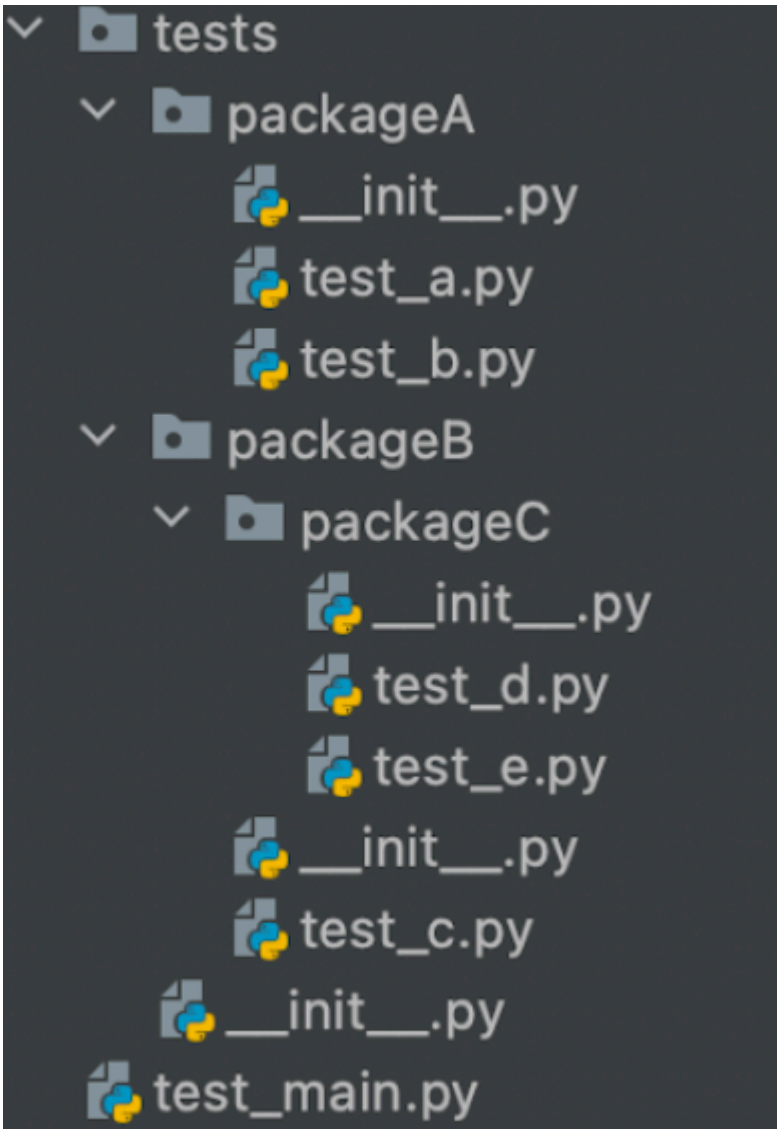
Organisez les fichiers de tests dans un projet

Avant même de commencer à écrire l'ensemble des tests de notre projet, il est important d'organiser l'architecture de notre projet afin de **séparer le code source et les tests**. Néanmoins, il existe quelques méthodes qui permettent tout de même facilement de retrouver le code source associé aux tests.

Nous pouvons tout d'abord **créer une arborescence** dédiée aux tests, cette méthode permettra de regrouper l'ensemble des tests de l'application au même endroit.

Quelques bonnes pratiques :

- Créer une arborescence de tests calquée sur le répertoire des fichiers sources.
- Nommer les fichiers de tests avec le même nom que le fichier source précédé de `test_` .



Pytest va en effet lancer les tests de tous les fichiers qui commencent par `test_` ou qui finissent par `_test`. Avec cette méthode, nous pouvons simplement lancer la commande `pytest` pour lancer tous les tests de notre projet.

Exercices

Vous allez mettre en place l'ensemble des **tests unitaires** du projet [super-calculatrice], qui permettent de vérifier la logique du code. Vous le ferez en utilisant **Pytest** pour vous entraîner.

Votre mission :

- Ajoutez un package de tests qui contiendra l'**arborescence de tests**.

- Créez la suite de tests concernant le module **view** avec Pytest.
- Créez la suite de tests concernant le module **operators** avec Pytest.

> ne toucher pas au module **controller** pour l'instant.

En résumé

- Il sera très utile **d'organiser** ses tests dans un répertoire dédié aux tests et en calquant l'arborescence du code source.
- Pour implémenter un test, il faut définir une **fonction** commençant par le préfixe `test_` ou terminant par le suffixe `_test` .
- Le mot-clé `assert` permet de **vérifier** les éléments que nous souhaitons valider.
- **Lancez** les tests d'un module de tests à l'aide de la commande `pytest nom_du_module.py`