TP-Introduction au framework de test Pytest

Étape 1: Installation-Pytest

Étape 2: syntaxe Pytest pour l'écriture des tests-

Étape 3: Écrire des tests de base dans pytest

Étape 4: Paramétrage des méthodes de test

Étape 5: Utilisation des fixtures
Étape 6: Marquages de cas de test
Étape 7: Cheat sheet pour exécuter pytest avec différentes options

Démarche :

Étape 1: Installation

pytest prend en charge les deux versions de python 2 et 3.

Pour installer pytest, nous utiliserons la commande pip-

```
python -m venv enA61
envA61\Scripts\activate
python -m pip install --upgrade pip
pip install -U pytest
deactivate
```

Étape 2: syntaxe Pytest pour l'écriture des tests-

- Les noms de fichiers doivent commencer ou se terminer par « **test** », comme par exemple **test_example.py** ou **example_test.py.**

- Si les tests sont définis comme des méthodes sur une classe, le **nom de** la **classe** doit commencer par « **Test** », comme dans **TestExample**. La classe ne doit pas avoir de méthode **__init__.**

- Les noms de méthodes ou noms de fonctions devrait commencer par "**test_**", par exemple **test_example**. Les méthodes dont les noms ne correspondent pas à ce modèle ne seront pas exécutées en tant que tests.

Créons le répertoire du projet- et un fichier python :

demo_tests /

```
- test_example.py
```

Étape 3: Écrire des tests de base dans pytest

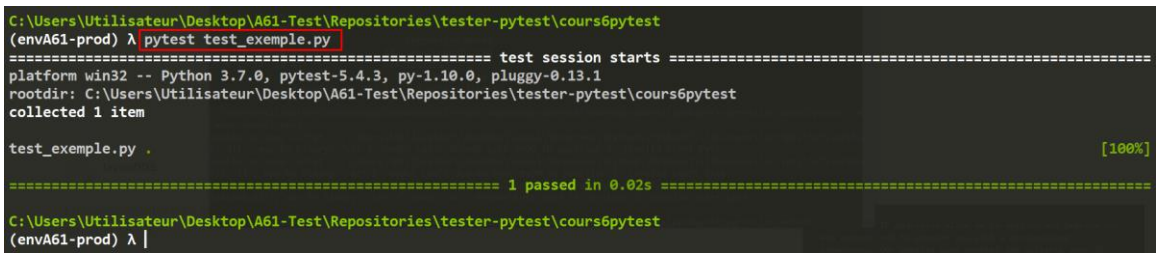
Dans le fichier **test_example.py**, écrivons une fonction très simple **sum** qui prend deux arguments **num1** et **num2** et retourne leur somme.

```
def sum (num1, num2):
    "" "Il renvoie la somme de deux nombres" ""
    return num1 + num2
```

Maintenant, nous allons écrire des tests pour tester notre fonction **sum**

```
import pytest

#assurez-vous que le nom de la fonction commence par test
def test_sum():
  assert sum(1, 2) == 3
```

Après avoir créer votre **env virtuel**, lancer le test, comme suit :



Ici, chaque point représente un cas de test. Comme nous n'avons qu'un seul cas de test pour le moment, nous pouvons voir 1 point et cela passe en 0,02 seconde.

Pour obtenir plus d'informations sur l'exécution du test, utilisez la commande ci-dessus avec l' option -v (verbose).

```
C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
(envA61-prod) λ pytest test_exemple.py -v
============================================= test session starts =============================================
platform win32 -- Python 3.7.0, pytest-5.4.3, py-1.10.0, pluggy-0.13.1 -- c:\users\utilisateur\appdata\local\programs\python\
thon.exe
cachedir: .pytest_cache
rootdir: C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
collected 1 item

test_exemple.py::test_sum PASSED                                                                      [100%]

============================================== 1 passed in 0.02s ==============================================
```

Ajoutez un autre test dans le fichier test_example.py, qui testera le type de sortie que la fonction sum donne ie entier.

```
def test_sum_output_type():
    assert type(sum(1, 2)) is int
```

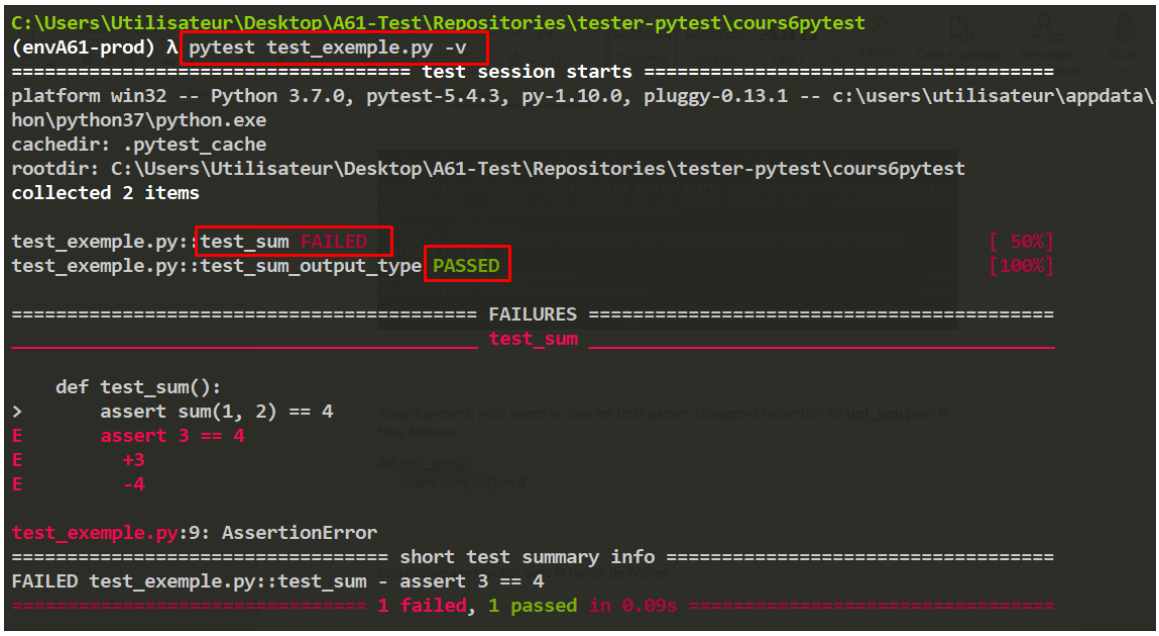
Maintenant, si nous exécutons à nouveau les tests, nous obtiendrons la sortie de deux tests et test_sum et test_sum_output_type

```
C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
(envA61-prod) λ pytest test_exemple.py -v
=================================== test session starts ===================================
platform win32 -- Python 3.7.0, pytest-5.4.3, py-1.10.0, pluggy-0.13.1 -- c:\users\utilisateur\
appdata\local\programs\python\python37\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
collected 2 items

test_exemple.py::test_sum PASSED                                                      [ 50%]
test_exemple.py::test_sum_output_type PASSED                                          [100%]

==================================== 2 passed in 0.04s ====================================
```

Jusqu'à présent, nous avons vu tous les tests passer, changeons l'assertion de **test_sum** pour la faire échouer

```
def test_sum():
    assert sum(1, 2) == 4
```

Cela donnera le résultat avec la raison de l'échec



Cela couvre les **bases** de **pytest**. Nous allons maintenant plonger dans quelques concepts avancés qui rendent l'écriture de test dans pytest plus puissante.

Étape 4: Paramétrage des méthodes de test

Si vous regardez notre fonction **test_sum**, elle teste la fonction **sum** avec un seul ensemble d'entrées (1, 2) et le test est **codé en dur** avec cette valeur.

Une meilleure approche pour couvrir plus de scénarios serait de transmettre les données de test en tant que paramètres à notre fonction de test, puis d'affirmer le résultat attendu.

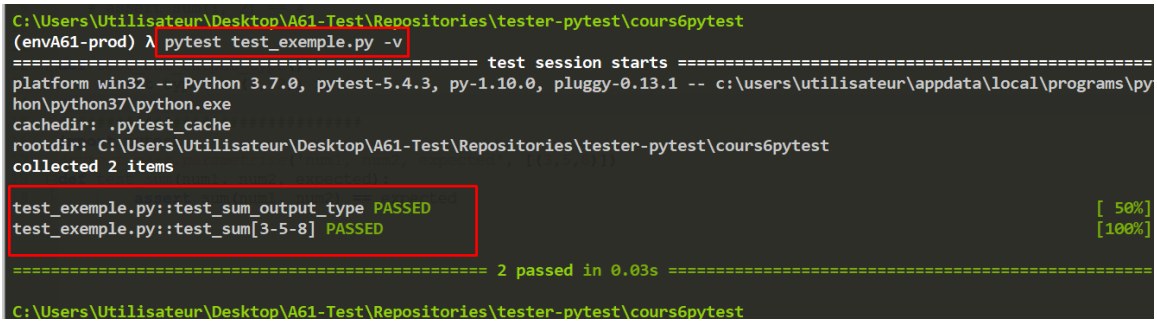
Modifions notre fonction **test_sum** pour utiliser des paramètres.

argnames - une chaîne séparée par des virgules indiquant un ou plusieurs noms d'argument, ou une liste / un tuple de chaînes d'arguments. Ici, nous avons passé **num1**, **num2** et **expected** comme 1ère entrée, 2ème entrée et somme attendue respectivement.

argvalues - La liste des argvalues détermine la fréquence à laquelle un test est appelé avec différentes valeurs d'argument. Si un seul nom d'argument a été spécifié, argvalues est une liste de valeurs. Si N noms d'arguments ont été spécifiés, **argvalues** doit être une liste de N-tuples, où chaque élément de tuple spécifie une valeur pour son nom d'argument respectif. Ici, nous avons passé un tuple de (3,5,8) dans une liste où 3 est **num1**, 5 est **num2** et 8 est **expected** sum.

```
import pytest

@pytest.mark.parametrize('num1, num2, expected', [(3,5,8)])

def test_sum(num1, num2, expected):

    assert sum(num1, num2) == expected
```

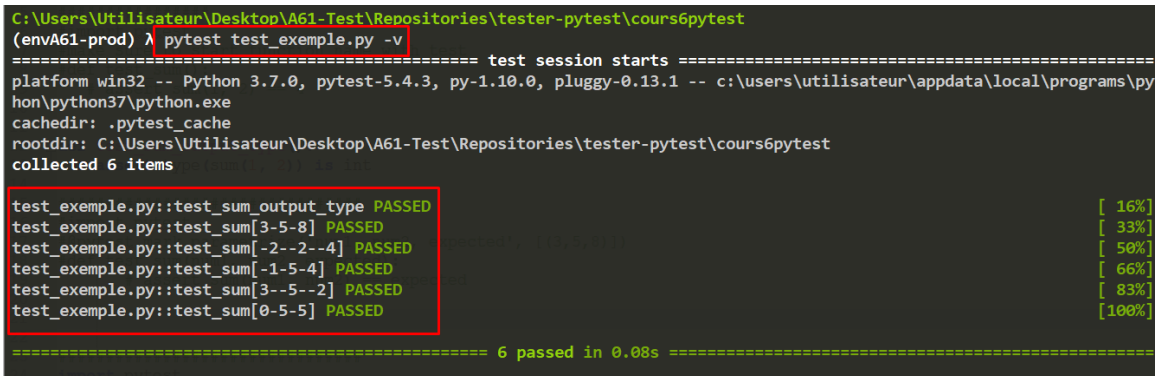
Ici, le décorateur **@parametrize** définit un tuple (num1, num2, expected) pour que la fonction **test_sum** s'exécute une fois.



Nous pouvons ajouter plusieurs tuples de **(num1, num2, expected)** dans la liste passée comme 2ème argument dans l'exemple ci-dessus.

```
import pytest

@pytest.mark.parametrize('num1, num2, expected',[(3,5,8),
(-2,-2,-4), (-1,5,4), (3,-5,-2), (0,5,5)])
def test_sum(num1, num2, expected):
        assert sum(num1, num2) == expected
```

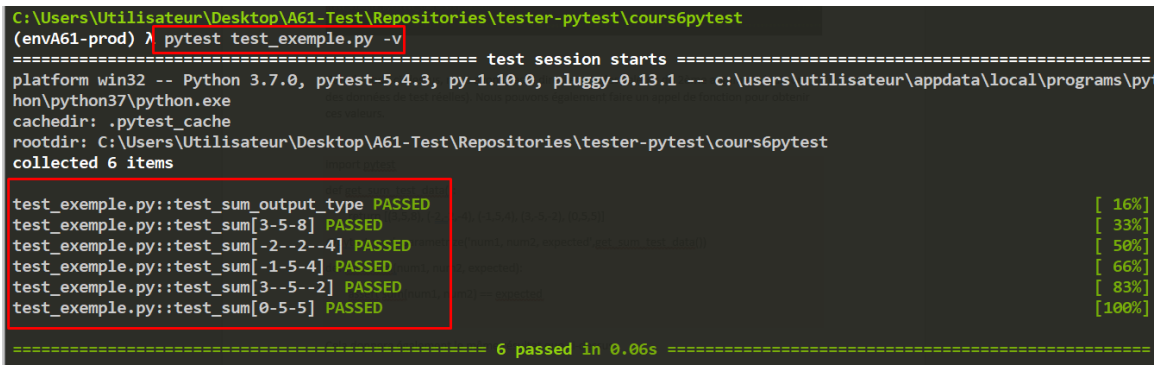
Ce test **test_sum** sera exécuté 5 fois pour les paramètres ci-dessus :



Dans le code ci-dessus, nous avons passé directement les valeurs du 2ème argument (qui sont des données de test réelles). Nous pouvons également faire un appel de fonction pour obtenir ces valeurs.

```
import pytest

def get_sum_test_data():
    return [(3,5,8), (-2,-2,-4), (-1,5,4), (3,-5,-2), (0,5,5)]

@pytest.mark.parametrize('num1, num2, expected',get_sum_test_data())

def test_sum(num1, num2, expected):
    assert sum(num1, num2) == expected
```

Cela donnera également le même résultat que ci-dessus.



Étape 5: Utilisation des fixtures

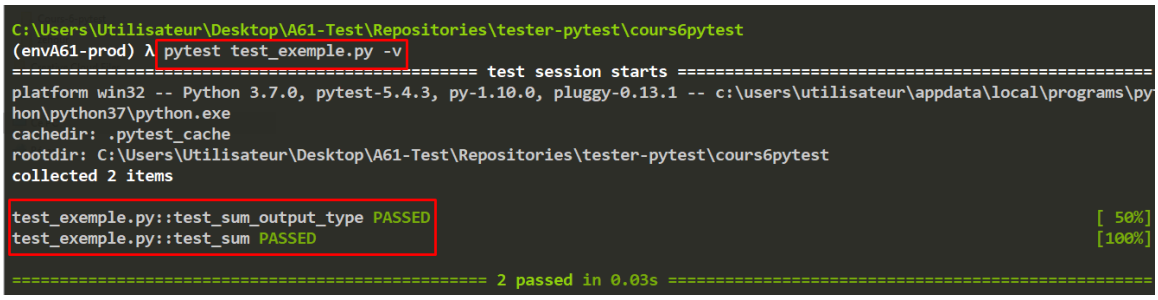
Selon la documentation officielle de pytest: **le [but des tests fixtures] est de fournir une base de référence fixe sur laquelle les tests peuvent être exécutés de manière fiable et répétée.**

Les fixtures peuvent être utilisées pour partager les données de test entre les tests, exécuter les méthodes de configuration et de démontage avant et après les exécutions de test respectivement.

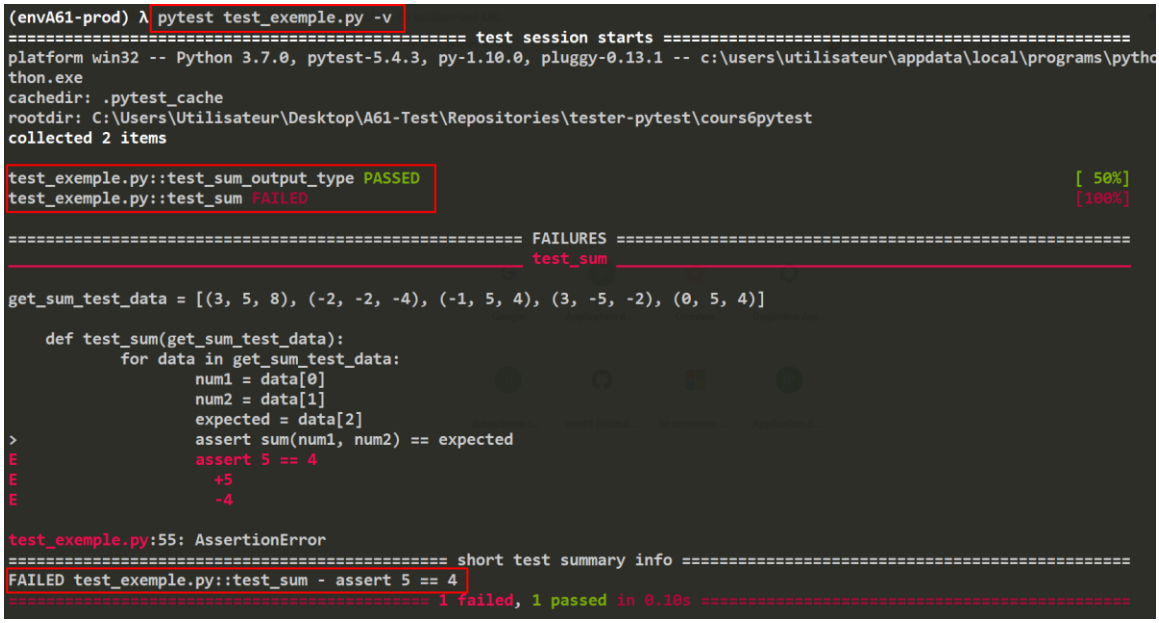
Pour comprendre les fixtures, nous réécrirons la fonction **test_sum** ci-dessus et utiliserons les fixtures pour obtenir des données de test.

```
import pytest
@pytest.fixture
def get_sum_test_data():
        return [(3,5,8), (-2,-2,-4), (-1,5,4), (3,-5,-2),
(0,5,5)]
def test_sum(get_sum_test_data):
        for data in get_sum_test_data:
                num1 = data[0]
                num2 = data[1]
                expected = data[2]
                assert sum(num1, num2) == expected
```

Cela donnera la sortie



Si vous regardez le résultat du test ci-dessus, vous pourriez avoir un doute sur l'exécution du test pour toutes les valeurs, car il n'est pas clair lors de l'exécution du test. Modifions donc l'une des données de test pour inclure des erreurs et réexécutons le test.



Scope of fixture - Scope contrôle la fréquence à laquelle une fixture est appelé. La valeur par défaut est function.

Voici les options pour scope:

Function: Exécuter une fois par test

Class: Exécuter une fois par classe de tests

Module: Exécuter une fois par module

Session: exécuter une fois par session

Les portées possibles, de la zone la plus basse à la plus haute sont:

function <classe <module <session.

Fixture peut être marqué comme **autouse=True**, ce qui permettra à chaque test de votre suite de l'utiliser par défaut.

Maintenant, nous allons utiliser des fixtures pour écrire la fonction **setup_and_teardown**. La fonction **setup** lit certaines données de la base de données avant le début du test et la fonction de **teardown** écrit les données du test dans la base de données une fois le test terminé. Pour plus de simplicité, nos fonctions **setup_and_teardown** imprimeront simplement un message.

Remarquez **yield** dans **setup_and_teardown**. Tout ce qui est écrit après **yield** est exécuté une fois les tests terminés.

```
@pytest.fixture(scope='session')
def get_sum_test_data():
        return [(3,5,8), (-2,-2,-4), (-1,5,4), (3,-5,-2),
(0,5,5)]
@pytest.fixture(autouse=True)
def setup_and_teardown():
        print('\nFetching data from db')
        yield
        print('\nSaving test run data in db')
def test_sum(get_sum_test_data):
        for data in get_sum_test_data:
                num1 = data[0]
                num2 = data[1]
                expected = data[2]
                assert sum(num1, num2) == expected
```

Dans le code ci-dessus, notez également que nous n'avons pas passé **setup_and_teardown** en paramètre à notre fonction **test_sum** car les deux sont définis **autouse=True**. Ainsi, ils seront automatiquement appelés avant et après chaque essai. Nous devons lancer le test en utilisant l'option **s** - maintenant pour imprimer sur stdout.

```
pytest test_example.py -v -s
```

```
C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
(envA61-prod) λ pytest test_exemple.py -v -s
============================================= test session starts ==================
platform win32 -- Python 3.7.0, pytest-5.4.3, py-1.10.0, pluggy-0.13.1 -- c:\users\utilis
on\python37\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Utilisateur\Desktop\A61-Test\Repositories\tester-pytest\cours6pytest
collected 2 items

test_exemple.py::test_sum_output_type
Fetching data from db
PASSED
Saving test run data in db

test_exemple.py::test_sum
Fetching data from db
PASSED
Saving test run data in db


============================================== 2 passed in 0.05s ===================
```

Étape 6: Marquages de cas de test

Pytest permet de marquer les tests puis de les exécuter de manière sélective.
En utilisant l'assistant **pytest.mark**, vous pouvez facilement définir des métadonnées sur vos fonctions de test. Il existe des marqueurs intégrés, par exemple:
skip - ignore toujours une fonction de test
skipif - saute une fonction de test si certaine condition est remplie
xfail - produit un résultat **expected failure** si certaine condition est remplie
parametrize pour effectuer plusieurs appels à la même fonction de test. (Nous en avons déjà discuté ci-dessus.)
Dans les exemples ci-dessus, nous pouvons marquer la première fonction de test comme lente et exécuter uniquement cette fonction.

```
@pytest.mark.slow
def test_sum():
    assert sum(1, 2) == 3
def test_sum_output_type():
    assert type(sum(1, 2)) is int
```

Maintenant, pour exécuter uniquement des tests marqués lents dans le fichier **test_exemple.py**, nous allons utiliser-

```
pytest test_exemple.py -m slow
```

Étape 7: Cheatsheet pour exécuter pytest avec différentes options

```
# expressions de mots clés
# Exécutez tous les tests avec une chaîne 'validate' dans le nom

pytest -k "validate"

# Exclure les tests avec 'db' dans le nom mais inclure 'validate'

pytest -k "validate and not db"

# Exécuter tous les fichiers de test dans un dossier demo_tests

pytest demo_tests/

# Exécuter une seule méthode test_method d'une classe de test TestClassDemo
```

```
pytest demo_tests/test_exemple.py::TestClassDemo::test_method


# Exécutez une seule classe de test nommée TestClassDemo

pytest demo_tests/test_exemple.py::TestClassDemo


# Exécutez une seule fonction de test nommée test_sum
pytest demo_tests/test_exemple.py::test_sum


# Exécuter les tests en mode détaillé:
pytest -v demo_tests/


# Exécuter des tests, y compris des instructions print:
pytest -s demo_tests/


# Exécuter uniquement les tests qui ont échoué lors du dernier
test

pytest — lf
```

Documentation:

https://docs.pytest.org/en/stable/getting-started.html

https://docs.pytest.org/en/latest/parametrize.html#parametrize mark