# Bienvenue dans Pytest

**Installation :**

Dans votre environnement python installer la library pytest :

$$\boldsymbol{pip3\ install\ pytest == 8.2.0}$$

**Exercice 1 : Écrire un Test Simple**

Écrivez un test simple en utilisant « pytest » pour vérifier que la fonction « ajouter(a, b) » retourne la somme de deux nombres.

Code Python:

```python
def ajouter(a, b):
    return a + b

def test_ajouter():
    assert ajouter(2, 3) == 5
```

**Notion importante :**

Les **fixtures* de 'pytest' sont une fonctionnalité puissante pour la configuration des tests. Elles permettent de définir et de fournir des données ou des configurations de test de manière modulaire et réutilisable. Grâce aux *fixtures*, vous pouvez préparer l'environnement nécessaire pour exécuter un test, comme initialiser des bases de données, créer des objets de test, ou configurer des états systèmes. Après l'exécution du test, les 'fixtures' peuvent également être utilisées pour nettoyer tout ce qui a été configuré.

Voici comment vous pourriez utiliser une fixture dans 'pytest'. Imaginons que vous avez besoin de tester des fonctions qui nécessitent un objet de configuration complexe. Vous pouvez définir une fixture pour créer cet objet et l'utiliser dans plusieurs tests.

Exemple de code avec pytest fixture :

```python
import pytest

# Définition de la fixture
@pytest.fixture
def config():
    # Imaginons que cet objet de configuration est complexe à construire
    configuration = {'db': 'test_db', 'host': 'localhost', 'port': 5432}
    return configuration
```

```
# Utilisation de la fixture dans un test
def test_database_connection(config):
    # Ici, config est l'objet retourné par la fixture
    assert config['db'] == 'test_db'
    assert config['host'] == 'localhost'
    assert config['port'] == 5432
    # Vous pourriez ici initier une connexion à la base de données en utilisant ces paramètres

# Un autre test utilisant la même fixture
def test_database_port(config):
    # Test de vérification du port
    assert config['port'] == 5432
```

Dans cet exemple, ‘config’ est une fixture qui crée un dictionnaire simulant une configuration. Les fonctions de test ‘test_database_connection’ et ‘test_database_port’ utilisent cette fixture en l'indiquant en paramètre. ‘pytest’ s'occupe de passer l'objet ‘config’ aux tests automatiquement.

**Exercice 2 : Utiliser des Fixtures**

Utilisez une fixture pour initialiser une liste avant chaque test. Vérifiez si la liste est vide au début de chaque test.

Code Python:
```python
import pytest

@pytest.fixture
def liste_vide():
    return []

def test_liste_vide_initialement(liste_vide):
    assert len(liste_vide) == 0
```

**Exercice 3 : Paramétrisation de Tests**

Paramétrez un test pour vérifier la fonction ‘ajouter(a, b)’ avec plusieurs paires de nombres.

Code Python:
```python
import pytest

@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5),
    (1, 1, 2),
```

```
    (0, 0, 0),
])
def test_ajouter_parametre(a, b, expected):
    assert ajouter(a, b) == expected
```

**Exercice 4 : Tester des Exceptions**

Écrivez un test pour vérifier qu'une exception est levée lorsque vous divisez par zéro dans la fonction `diviser(a, b)`.

Code Python:
```python
def diviser(a, b):
    if b == 0:
        raise ValueError("Division par zéro")
    return a / b

def test_diviser_par_zero():
    with pytest.raises(ValueError):
        diviser(10, 0)
```

**Exercice 5 : Utilisation de Plusieurs Assertions**

Créez un test avec plusieurs assertions pour vérifier différentes opérations arithmétiques dans une fonction ‘calculer(a, b)’ qui retourne la somme, la différence et le produit.

Code Python:
```python
def calculer(a, b):
    return a + b, a - b, a * b

def test_calculer():
    resultat_attendu = (5, -1, 6)
    assert calculer(3, 2) == resultat_attendu
```