UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Pytest

Qu'est-ce que Pytest ?

- **Pytest** : Un framework de tests Python simple et puissant.
- Permet de tester :
  - **Tests unitaires** : Vérification de petites unités de code.
  - **Tests fonctionnels** : Vérification des fonctionnalités globales.
- Avantages :
  - Syntaxe claire et concise.
  - Compatible avec les tests `unittest`.
  - Tests paramétriques et riches rapports.

Installation

Installez Pytest :

```
pip install pytest
```

Vérifiez l'installation :

```
pytest --version
```

Premier test avec Pytest

1. Créez un fichier `test_example.py` :

```
def test_addition():
    assert 1 + 1 == 2

def test_subtraction():
    assert 5 - 3 == 2
```

2. Exécutez les tests dans le terminal :

```
pytest
```

Rapport d'échec

Si un test échoue, Pytest donne un rapport clair. Exemple :

```
def test_failure():
    assert 1 + 1 == 3
```

Rapport :

```
test_example.py F

test_failure
    assert 2 == 3
```

Fixtures

Une fixture est une fonction qui permet de préparer un environnement de test stable et réutilisable.

```
import pytest

@pytest.fixture
def sample_data():
    return {"name": "Alice", "age": 30}

def test_fixture_usage(sample_data):
    assert sample_data["name"] == "Alice"
```

Paramétrisation

Testez plusieurs cas avec un seul test :

```python
import pytest

@pytest.mark.parametrize("a, b, result", [
    (1, 2, 3),
    (2, 3, 5),
    (3, 5, 8),
])
def test_addition(a, b, result):
    assert a + b == result
```

Exécution : Le test est exécuté 3 fois avec des valeurs différentes.

Organisation avec des classes

Organisez vos tests (optionnel) :

```
class TestMathOperations:
    def test_multiply(self):
        assert 2 * 3 == 6

    def test_divide(self):
        assert 6 / 2 == 3
```

Commandes utiles

- **Exécuter un test spécifique** :

```
pytest test_example.py::test_addition
```

- **Augmenter la verbosité** :

```
pytest -v
```

Commandes utiles

- **Relancer les tests échoués** :

```
pytest --lf
```

- **Générer un rapport HTML** :

```
pip install pytest-html
pytest --html=rapport.html
```

Tests avancés

Vérifier une exception

```
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

Introduction au Mocking avec Pytest

Le **mocking** est une technique pour simuler le comportement d'une dépendance externe.
Cela permet de :

- Tester une unité de code sans exécuter réellement ses dépendances.
- Éviter des appels coûteux (API, bases de données, etc.).
- Contrôler le comportement des dépendances en configurant leurs résultats.

Cas d'utilisation :

- Mock d'une **API externe** pour éviter d'effectuer de vrais appels réseau.
- Simulation de réponses provenant d'une **base de données**.
- Test de méthodes dépendant de l'horloge système.

Installation de pytest-mock

1. Installez le package `pytest-mock` :

```
pip install pytest-mock
```

2. `pytest-mock` fournit une fixture intégrée appelée `mocker`, qui est un wrapper autour de `unittest.mock`.

Mock d'une fonction

Supposons que vous avez une fonction qui dépend d'une autre fonction externe :

```
# module.py
def fetch_data():
    return {"name": "Alice"}

def process_data():
    data = fetch_data()
    return data["name"].upper()
```

Mock d'une fonction

Dans ce cas, vous pouvez **mock** la fonction `fetch_data` pour contrôler sa sortie :

```
from module import process_data

def test_process_data(mocker):
    mock = mocker.patch("module.fetch_data", return_value={"name": "Bob"})

    result = process_data()

    assert result == "BOB"
    mock.assert_called_once()
```

Mock d'une méthode d'objet

Supposons que vous avez une classe avec une méthode :

```
# service.py
class UserService:
    def get_user(self):
        return {"id": 1, "name": "Alice"}
```

Mock d'une méthode d'objet

Vous pouvez mocker la méthode `get_user` dans vos tests :

```python
# test_service.py
from service import UserService

def test_get_user(mocker):
    user_service = UserService()

    # Mock de la méthode d'instance `get_user`
    mock = mocker.patch.object(user_service, "get_user", return_value={"id": 2, "name": "Bob"})

    # Appel de la méthode mockée
    user = user_service.get_user()

    # Assertions
    assert user["name"] == "Bob"
    mock.assert_called_once()
```

Mock d'une fonction tierce

Vous pouvez également mocker une fonction provenant d'une bibliothèque tierce :

```python
import requests

def fetch_from_api():
    response = requests.get("https://example.com/api")
    return response.json()

def test_fetch_from_api(mocker):
    mock = mocker.patch("requests.get")
    mock.return_value.json.return_value = {"status": "success"}
    result = fetch_from_api()
    assert result["status"] == "success"
    mock.assert_called_once_with("https://example.com/api")
```

Mock avec des effets secondaires

Les mocks peuvent également déclencher des **exceptions** ou des comportements spécifiques :

```
def test_fetch_data_failure(mocker):
    # Mock qui lève une exception
    mocker.patch("module.fetch_data", side_effect=ValueError("Erreur simulée"))

    # Test que l'exception est levée
    with pytest.raises(ValueError, match="Erreur simulée"):
        process_data()
```

Bonnes pratiques pour le Mocking

1. **Ne mockez que ce qui est nécessaire** : Mockez uniquement les dépendances externes.
2. **Vérifiez les appels** : Utilisez `mock.assert_called_once()` ou `mock.call_args` pour valider les interactions.
3. **Organisez les imports** : Assurez-vous de patcher le chemin correct (`module.fetch_data`).
4. **Utilisez `pytest-mock` pour simplifier** : Il offre une API plus intuitive que `unittest.mock`.

Bonnes pratiques de testing

1. **Nommez clairement vos tests.**
2. **Évitez la duplication** avec des fixtures.
3. **Paramétrez vos tests** pour couvrir plusieurs cas.
4. **Exécutez fréquemment vos tests.**