

TDD - python

Sommaire

- Objectifs de formation
- Introduction au TDD
- Tests de traitement Data

Objectifs de formation

Objectifs pédagogiques :

- Comprendre la démarche de Test Driven Development (TDD)
- Utiliser PyTest pour écrire et organiser les tests
- Assurer la qualité logicielle des scripts Python et Spark
- Intégrer les tests dans une chaîne CI/CD

Importance pour le cursus Data Engineer :

- Le TDD renforce la **fiabilité** et la **maintenabilité** du code data
- **PyTest** est la solution de **référence** pour les tests Python dans les **pipelines DevOps**
- Indispensable pour **tester les transformations**, les **modèles**, les **API**, et les **traitements distribués**

Introduction au TDD

Qu'est-ce que le TDD ?

Le **Test Driven Development** (TDD) est une méthode visant à **sécuriser** le développement en s'appuyant sur des **tests automatisés**. En Python, les tests **unitaires** sont les plus courants et peuvent être écrits avec plusieurs frameworks :

- `unittest` (intégré à la bibliothèque standard)
- `pytest` (le plus populaire, simple et puissant)
- `nose2` (héritier de Nose, moins utilisé)

Intérêt

Les tests **unitaires** permettent de valider les **fonctionnalités**.
Exemple : une fonction qui calcule un score de bowling doit **garantir** :

- qu'on ne puisse pas dépasser 10 quilles par lancer,
- qu'un strike soit reconnu correctement,
- qu'un spare soit compté correctement.

Les tests suivent la **convention de nommage** :
`nom_methode_conditions_resultat_attendu()`

Le cycle TDD

1. Red

- Écrire un test avant d'avoir de code implémenté.
- Le test doit échouer (rouge) car la fonctionnalité n'existe pas encore.
- Cet échec confirme que le test est pertinent et détecte bien l'absence de la fonctionnalité.

Le cycle TDD

2. Green

- Écrire le minimum de code nécessaire pour que le test passe.
- L'objectif n'est pas la propreté du code, mais simplement de rendre le test vert (réussi).
- On valide ainsi que le code répond au besoin défini par le test.

Le cycle TDD

3. Refactor

- Réorganiser et améliorer le code pour le rendre plus lisible, maintenable et performant.
- Vérifier que les tests restent verts pendant cette phase.
- Aucune nouvelle fonctionnalité n'est ajoutée, seul le design du code est amélioré.

Types de tests

1. Tests unitaires :

- Testent une unité de code isolée (fonction, méthode, classe) indépendamment du reste du système. Ils sont rapides et précis.

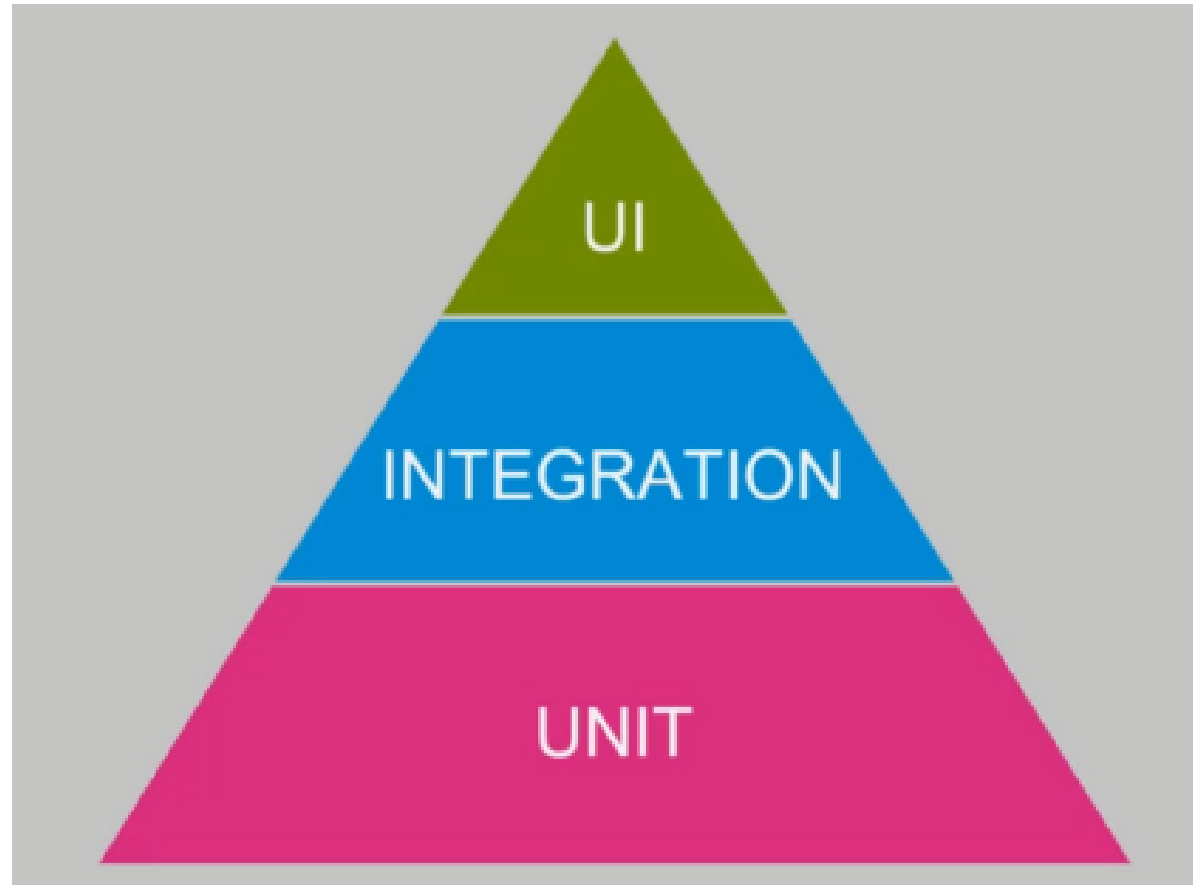
2. Tests d'intégration :

- Vérifient que plusieurs composants fonctionnent correctement ensemble (base de données, API, services externes).

3. Tests fonctionnels (ou end-to-end) :

- Testent l'application complète du point de vue utilisateur, simulant des scénarios réels.

Pyramide des tests



Organisation des projets

- Un projet python est souvent structuré ainsi :

```
mon_projet/  
  src/  
    mon_module.py  
  tests/  
    test_mon_module.py
```

- Les tests sont écrits en parallèle du code applicatif.
- Installation de PyTest :

```
pip install pytest pytest-cov
```

pytest : Les Assertions

Rôle

- Vérifier qu'une condition est vraie.
- Un test sans `assert` n'a pas de sens.
- En cas d'échec, pytest affiche directement la valeur attendue et la valeur obtenue.
- Pas besoin d'ajouter de message personnalisé
- pytest génère un message d'erreur détaillé

Comparaisons fréquentes

- `assert x == y` => égalité
- `assert x != y` => différence
- `assert x is None` => nullité
- `assert isinstance(x, int)` => type
- `assert "abc" in texte` => inclusion

Structure de base d'un test

```
def test_nom_descriptif():  
    # Arrange (Préparation)  
    valeur = 5  
  
    # Act (Action)  
    resultat = ma_fonction(valeur)  
  
    # Assert (Vérification)  
    assert resultat == 10
```

Principes des bons tests unitaires

- Un bon test unitaire doit être **FIRST** :
 - **F**ast : s'exécute en millisecondes
 - **I**ndependent : ne dépend pas d'autres tests
 - **R**epeatable : donne toujours le même résultat
 - **S**elf-validating (Auto-validant) : passe ou échoue clairement
 - **T**imely : écrit avant ou avec le code

Tests paramétrés

- Les tests paramétrés permettent d'exécuter le même test avec différentes entrées.

```
import pytest

@pytest.mark.parametrize("entree, attendu", [
    (5, 25),
    (3, 9),
    (0, 0),
    (-2, 4),
])

def test_carre(entree, attendu):
    assert carre(entree) == attendu
```

Tests paramétrés

- Paramétrage multiple :

```
@pytest.mark.parametrize("a,b,resultat", [  
    (1, 1, 2),  
    (2, 3, 5),  
    (10, -5, 5),  
    (0, 0, 0),  
)  
def test_addition(a, b, resultat):  
    assert additionner(a, b) == resultat
```

Tests paramétrés

- Avec des IDs personnalisés :

```
@pytest.mark.parametrize("email, valide", [
    ("user@domain.com", True),
    ("invalid.email", False),
    ("user@domain", False),
], ids=["email_valide", "sans_arobase", "sans_extension"])
def test_validation_email(email, valide):
    assert valider_email(email) == valide
```

Fixtures

Les fixtures préparent et nettoient l'environnement de test. Elles permettent de réutiliser du code de setup.

```
@pytest.fixture
def exemple_liste():
    """Fournit une liste d'exemple pour les tests."""
    return [1, 2, 3, 4, 5]

def test_somme(exemple_liste):
    assert sum(exemple_liste) == 15

def test_longueur(exemple_liste):
    assert len(exemple_liste) == 5
```

Fixture avec setup et teardown

```
@pytest.fixture
def fichier_temporaire():
    # Setup
    nom_fichier = "temp_test.txt"
    with open(nom_fichier, 'w') as f:
        f.write("Contenu de test")
    # Le test utilise la fixture
    yield nom_fichier
    # Teardown (nettoyage)
    import os
    if os.path.exists(nom_fichier):
        os.remove(nom_fichier)

def test_lecture_fichier(fichier_temporaire):
    with open(fichier_temporaire, 'r') as f:
        contenu = f.read()
    assert contenu == "Contenu de test"
```

Couverture de code avec pytest-cov

- La couverture de code mesure le pourcentage de code exécuté par les tests.
- Installation :

```
pip install pytest-cov
```

- Afficher le rapport dans le terminal :

```
# Afficher le rapport dans le terminal  
pytest --cov=mon_module
```

Interprétation

- **80-100%** : Excellente couverture
- **60-80%** : Bonne couverture
- **40-60%** : Couverture moyenne
- **< 40%** : Couverture insuffisante

Important : Une couverture de 100% ne garantit pas l'absence de bugs, mais une faible couverture garantit des zones non testées.

Merci pour votre attention

Des questions ?

