

Tutoriel : Réalisation De Tests Unitaires en Python avec Pytest

Document fourni par : Mlle Metra Phirielle VANDJY BOUNDZANGA

Etudiante en Master II Programmation à GROUPE SUP'INFO

metraphirielle@gmail.com

Module : Qualité Logicielle / Assurance Qualité

Table des matières

1	Introduction.....	3
1.1	Cible.....	3
1.2	Objectif visé.....	3
1.3	Prérequis	3
2	Plan du cours.....	4
3	Introduction aux tests unitaires.....	5
3.1	Pourquoi utiliser des tests unitaires ?.....	5
4	Présentation de pytest.....	5
5	Structuration d'un projet Python pour les tests unitaires.....	6
5.1	Explications	6
6	Étapes de création des tests avec pytest.....	6
7	Exemple pratique : Projet de Gestionnaire de tâche.....	7
7.1	Code source (src/ <i>todo.py</i>).....	7
	Qu'est-ce que c'est ?	8
	Les parties principales :	8
	Exemple simple :	9
7.2	Tests unitaires (test/<i>test_todo.py</i>)	9
	Qu'est-ce que c'est ?.....	10
	Les parties principales :	10
	Exemple simple :	11
	À retenir :	11
7.3	Exécution des tests	12
8	Bonnes pratiques et astuces.....	13
9	Bibliographie et webographie.....	13
	— Bibliographie :.....	13
	— Webographie :.....	13

1 Introduction

Ce document est un support de cours destiné à enseigner la réalisation de tests unitaires en Python à l'aide du framework pytest. Il s'agit d'un guide pratique pour les développeurs débutants souhaitant apprendre à écrire et exécuter des tests unitaires pour garantir la fiabilité de leur code. En effet, Les tests unitaires permettent de vérifier que chaque composant ou partie d'un programme fonctionne correctement de manière isolée et prévue.

1.1 Cible

Ce cours s'adresse aux étudiants en informatique, aux développeurs Python débutants ou intermédiaires, et à toute personne souhaitant approfondir ses compétences en tests unitaires dans un contexte de développement logiciel.

1.2 Objectif visé

L'objectif est de permettre aux apprenants de :

- Comprendre le rôle et l'importance des tests unitaires dans le cycle de vie du logiciel.
- Maîtriser l'utilisation de pytest pour écrire, organiser et exécuter des tests unitaires.
- Structurer un projet Python avec des tests unitaires.
- Appliquer les bonnes pratiques pour garantir un code robuste et maintenable.
- Rédiger des assertions et des fixtures.

1.3 Prérequis

Pour suivre ce tutoriel, il est requis de :

- Connaître les bases de la programmation Python (fonctions, classes, modules).
- Avoir Python (version 3.8 ou supérieure) installé sur votre machine.
- Installer pytest via la commande `pip install pytest`.
- Disposer d'un éditeur de code (par exemple, VS Code, PyCharm) et d'un terminal.
- Savoir créer des fonctions/modules Python

2 Plan du cours

1. Introduction aux tests unitaires
2. Présentation de pytest
3. Structuration d'un projet Python pour les tests unitaires
4. Étapes de création des tests avec pytest
5. Exemple pratique : Projet de todo liste
6. Bonnes pratiques et astuces
7. Bibliographie et webographie

3 Introduction aux tests unitaires

Les tests unitaires consistent à tester individuellement les unités de code (fonctions, méthodes, classes) de manière isolée pour s'assurer qu'elles fonctionnent comme prévu. Ils permettent de détecter rapidement les erreurs, de faciliter la maintenance et d'améliorer la qualité du code.

3.1 Pourquoi utiliser des tests unitaires ?

- **Fiabilité** : Vérifier que chaque composant fonctionne correctement.
- **Maintenance** : Faciliter la détection des régressions lors de modifications.
- **Documentation** : Les tests servent de documentation vivante du code.

4 Présentation de pytest

Qu'est-ce que pytest ?

Pytest est un framework de test puissant, flexible, simple à utiliser et très populaire dans l'écosystème Python. Il est apprécié pour sa simplicité, sa lisibilité et ses nombreuses fonctionnalités, telles que :

- Syntaxe intuitive pour écrire des tests.
- Détection automatique des fichiers et fonctions de test.
- Support des assertions Python standard sans modules supplémentaires.
- Plugins pour étendre ses fonctionnalités.

Liens utiles :

- Documentation officielle:
 - <https://docs.pytest.org/>
 - <https://pypi.org/project/pytest/>
 - [Effective Python Testing With pytest – Real Python](#)
- Installation : <https://docs.pytest.org/en/stable/getting-started.html> ou [pytest documentation](#)

5 Structuration d'un projet Python pour les tests unitaires

Un projet Python bien structuré facilite l'écriture et l'exécution des tests. Voici une structure typique :

```
1 mon_projet/ src/
2     calculatrice .py  # Code source
3 tests/
4     test_calculatrice .py  # Tests unitaires
5
6 requirements.txt      # Dépendances
README .md # Documentation
```

5.1 Explications

- src/ : Contient le code source de l'application.
- tests/ : Contient les fichiers de test.

6 Étapes de création des tests avec pytest

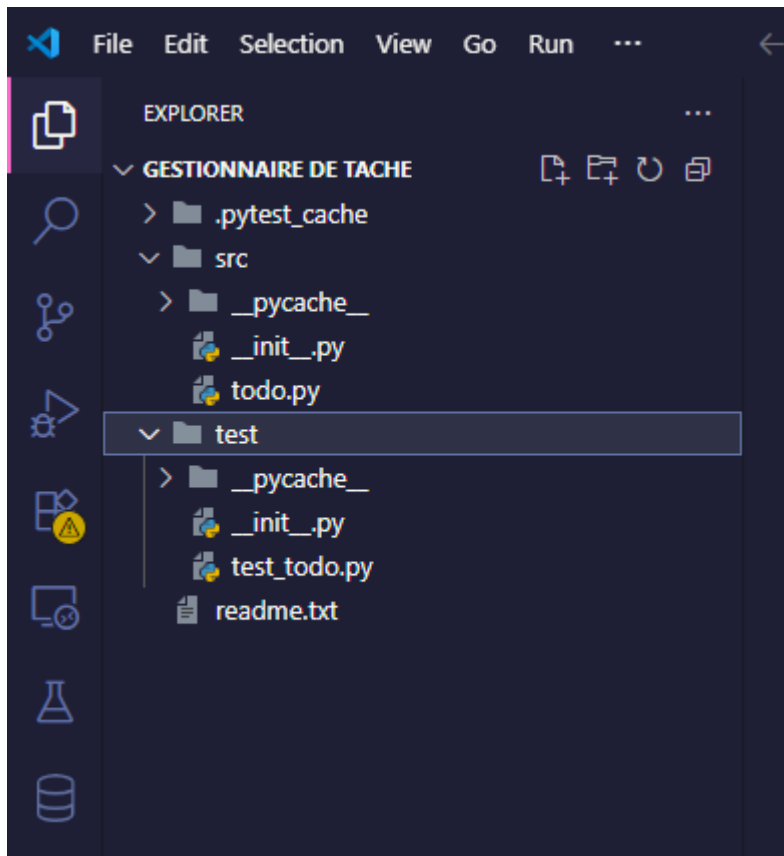
Voici les étapes pour créer et exécuter des tests avec pytest :

1. **Installer pytest** : Exécutez `pip install pytest` dans votre environnement ou terminal.
2. **Créer un module à tester** : Écrivez le code de votre application dans un dossier src/.
3. **Écrire les tests** : Créez des fichiers de test dans tests/ avec le préfixe test. **Exécuter les tests** :

Depuis la racine du projet, exécutez : `pytest` ou `pytest -v` pour une affichage détaillé.

- 4.. **Analyser les résultats** : Pytest affiche les tests réussis (.), échoués (F), ou ignorés (s).

Voici un exemple pratique pratique :



Comme vous pouvez le voir sur la capture d'écran on a le nom du projet nommé « **GESTIONNAIRE DE TACHE** »

Ensuite on a le dossier « **src** » et le dossier « **test** »

On a aussi le fichier « **todo.py** » dans le dossier « **src** »

Et le fichier « **test_todo** » dans le dossier « **test** »

Pour que les fichier puisse s'exécuter, il faut créer dans les dossier « **src** » et « **test** », un fichier « **__init__.py** »

7 Exemple pratique : Projet de Gestionnaire de tache

Voici un exemple concret d'implémentation simple d'un Gestionnaire de tache avec des tests unitaires.

7.1 Code source (src/*todo.py*)

Voici une explication claire et simple de la capture d'écran:

Qu'est-ce que c'est ?

- La classe **GestionnaireTaches** est comme un outil qui aide à organiser tes tâches. Elle te permet d'ajouter, de compléter ou de supprimer des tâches, et de voir la liste.

Les parties principales :

1. **Début (__init__) :**
 - Quand tu crées un **GestionnaireTaches**, il commence avec deux listes vides :
 - **self.taches** : pour les tâches que tu dois encore faire.
 - **self.taches_completees** : pour les tâches que tu as terminées.
2. **Ajouter une tâche (ajouter_tache) :**
 - Cette fonction ajoute une nouvelle tâche (comme "Faire les courses") à la liste taches.
 - Si tu essaies d'ajouter une description vide, elle te dit "non, ça ne marche pas" avec une erreur.
3. **Compléter une tâche (completer_tache) :**
 - Tu donnes un numéro (index) pour choisir une tâche dans taches.
 - Si le numéro est incorrect, tu reçois une erreur.
 - Sinon, elle enlève la tâche de taches et la met dans **taches_completees**.

4. **Supprimer une tâche (supprimer_tache) :**
 - Tu donnes un numéro pour choisir une tâche dans tâches.
 - Si le numéro est incorrect, erreur. Sinon, elle supprime cette tâche de la liste.
5. **Voir les tâches (lister_taches et lister_taches_completees) :**
 - Ces fonctions te donnent une copie de la liste des tâches actives ou terminées pour que tu puisses les voir sans les modifier.
6. **Compter les tâches (nombre_total_taches) :**
 - Cette fonction additionne le nombre de tâches actives et terminées pour te donner le total.

Exemple simple :

- Tu crées un gestionnaire : `mon_gestionnaire = GestionnaireTaches()`.
- Tu ajoutes "Lire un livre" : `mon_gestionnaire.ajouter_tache("Lire un livre")`.
- Tu la marques comme terminée : `mon_gestionnaire.completer_tache(0)`.
- Tu vois les tâches terminées : `mon_gestionnaire.lister_taches_completees()` te donnera ["Lire un livre"].

C'est un outil basique mais utile pour suivre tes tâches.

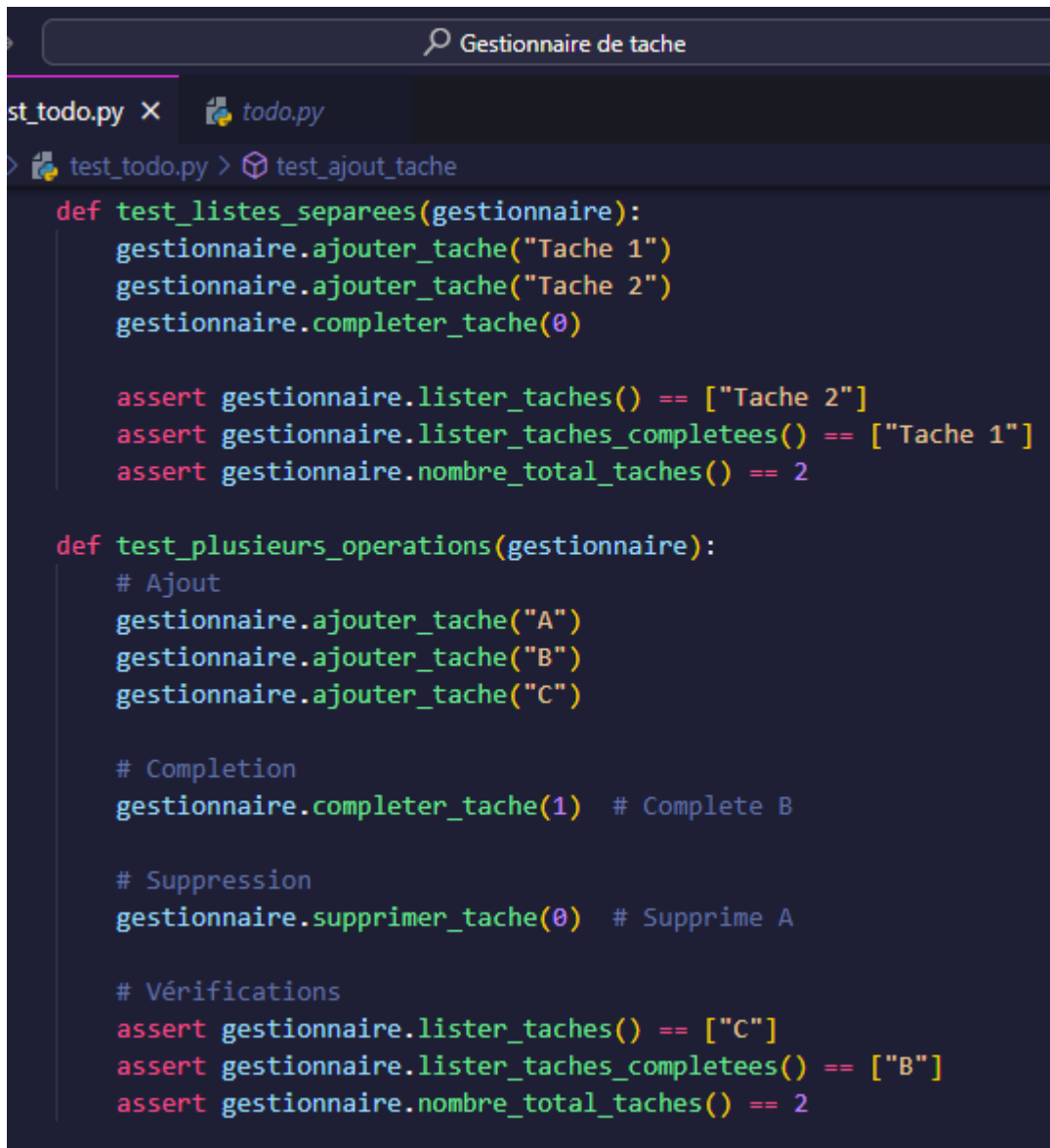
7.2 Tests unitaires (test/test_todo.py)

```

1  import pytest
2  from src.todo import GestionnaireTaches
3
4  @pytest.fixture
5  def gestionnaire():
6      return GestionnaireTaches()
7
8  def test_ajout_tache(gestionnaire):
9      gestionnaire.ajouter_tache("Faire les courses")
10     assert "Faire les courses" in gestionnaire.lister_taches()
11     assert gestionnaire.nombre_total_taches() == 1
12
13  def test_ajout_tache_vide(gestionnaire):
14     with pytest.raises(ValueError, match="La description ne peut pas être vide"):
15         gestionnaire.ajouter_tache("")
16
17  def test_completer_tache(gestionnaire):
18     gestionnaire.ajouter_tache("Faire du sport")
19     gestionnaire.completer_tache(0)
20     assert len(gestionnaire.lister_taches()) == 0
21     assert "Faire du sport" in gestionnaire.lister_taches_completees()
22
23  def test_completer_tache_invalide(gestionnaire):
24     with pytest.raises(IndexError, match="Index de tâche invalide"):
25         gestionnaire.completer_tache(0)
26
27  def test_supprimer_tache(gestionnaire):
28     gestionnaire.ajouter_tache("Appeler maman")
29     gestionnaire.supprimer_tache(0)
30     assert len(gestionnaire.lister_taches()) == 0
31     assert gestionnaire.nombre_total_taches() == 0
32

```

Suite du code :



The screenshot shows a code editor with a search bar at the top containing the text "Gestionnaire de tache". Below the search bar, there are two tabs: "st_todo.py" (closed) and "todo.py". The active tab is "test_todo.py", which contains the following Python code:

```
def test_listes_separees(gestionnaire):
    gestionnaire.ajouter_tache("Tache 1")
    gestionnaire.ajouter_tache("Tache 2")
    gestionnaire.completer_tache(0)

    assert gestionnaire.lister_taches() == ["Tache 2"]
    assert gestionnaire.lister_taches_completees() == ["Tache 1"]
    assert gestionnaire.nombre_total_taches() == 2

def test_plusieurs_operations(gestionnaire):
    # Ajout
    gestionnaire.ajouter_tache("A")
    gestionnaire.ajouter_tache("B")
    gestionnaire.ajouter_tache("C")

    # Completion
    gestionnaire.completer_tache(1) # Complete B

    # Suppression
    gestionnaire.supprimer_tache(0) # Supprime A

    # Vérifications
    assert gestionnaire.lister_taches() == ["C"]
    assert gestionnaire.lister_taches_completees() == ["B"]
    assert gestionnaire.nombre_total_taches() == 2
```

Voici une explication claire et simple de la capture d'écran:

Qu'est-ce que c'est ?

- Ce fichier (**test_todo.py**) contient des "tests" qui vérifient si le code de **GestionnaireTaches** (dans **src/todo.py**) fonctionne bien.
- Chaque fonction commençant par **test_** essaie une action (comme ajouter une tâche) et vérifie si le résultat est correct.

Les parties principales :

1. Imports :

- `import pytest` : Utilise l'outil pytest pour faire les tests.
- `from src.todo import GestionnaireTaches` : Prend la classe `GestionnaireTaches` depuis un fichier `todo.py` dans un dossier `src`.

2. Fixture (gestionnaire) :

- **@pytest.fixture** crée un **GestionnaireTaches** neuf pour chaque test. C'est comme un bloc-notes vide à chaque fois.

3. Tests individuels :

- Chaque fonction test_... vérifie une partie du travail :
 - **test_ajout_tache** : Ajoute "Faire les courses" et vérifie qu'elle apparaît dans la liste et que le total est 1.
 - **test_ajout_tache_vide** : Essaie d'ajouter une tâche vide et vérifie qu'une erreur apparaît.
 - **test_completer_tache** : Ajoute "Faire du sport", la marque comme terminée, et vérifie qu'elle n'est plus dans les tâches actives mais dans les terminées.
 - **test_completer_tache_invalide** : Essaie de compléter une tâche qui n'existe pas et vérifie qu'une erreur arrive.
 - **test_supprimer_tache** : Ajoute "Appeler maman", la supprime, et vérifie que la liste est vide.
 - **test_listes_separees** : Ajoute deux tâches, complète la première, et vérifie que les listes sont bien séparées.
 - **test_plusieurs_operations** : Ajoute trois tâches ("A", "B", "C"), complète "B", supprime "A", et vérifie que seule "C" reste dans les tâches actives.

4. Comment ça marche ?

- **assert** : Vérifie si quelque chose est vrai. Si c'est faux, le test échoue.
- **with pytest.raises** : Vérifie si une erreur spécifique (comme ValueError) se produit.

Exemple simple :

- Dans **test_ajout_tache**, on ajoute "Faire les courses", puis on regarde si elle est dans la liste et si le total est 1. Si oui, le test passe !

À retenir :

- Ces tests assurent que GestionnaireTaches fait ce qu'on attend (ajouter, compléter, supprimer, etc.).
- Quand tu lances pytest, il exécute tous ces tests et te dit si tout va bien ou non.

7.3 Exécution des tests

Pour exécuter les tests, placez-vous à la racine sur le terminal du projet et exécutez :

Résultat attendu :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\hp\Desktop\Gestionnaire de tache> pytest -v
===== test session starts =====
platform win32 -- Python 3.13.2, pytest-8.4.0, pluggy-1.6.0 -- C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\hp\Desktop\Gestionnaire de tache
collected 7 items

test/test_todo.py::test_ajout_tache PASSED [ 14%]
test/test_todo.py::test_ajout_tache_vide PASSED [ 28%]
test/test_todo.py::test_completer_tache PASSED [ 42%]
test/test_todo.py::test_completer_tache_invalide PASSED [ 57%]
test/test_todo.py::test_supprimer_tache PASSED [ 71%]
test/test_todo.py::test_listes_separees PASSED [ 85%]
test/test_todo.py::test_plusieurs_operations PASSED [100%]

===== 7 passed in 0.02s =====
```

Ici ma racine dans le terminal comme sur la capture d'écran c'est « PS

C:\Users\hp\Desktop\Gestionnaire de tache> » et le code pour tester le **test_todo.py** on exécute « **pytest -v** ou **pytest src\test_todo.py** » comme sur le capture d'écran

Résultat d'explication des tests :

- Il y avait eu 7 tests, et ils portent des noms comme `test_ajout_tache`, `test_completer_tache`, etc...
- Chaque test est marqué **PASSED** (passé), ce qui signifie que le code fonctionne comme prévu pour ces cas.
- Exemples de tests :
 - `test_ajout_tache` : Vérifie qu'on peut ajouter une tâche.
 - `test_completer_tache` : Vérifie qu'on peut marquer une tâche comme terminée.
 - `test_listes_separees` : Vérifie que les tâches actives et terminées sont bien séparées.

8 Bonnes pratiques et astuces

- Toujours créer un dossier **test/**
- Nommer les fichiers de tests avec le préfixe **test_**
- Gardez les tests indépendants et rapides
- Éviter les dépendances inutiles dans les tests
- Automatiser les tests avec un CI/CD (ex: GitHub Actions)
- Utiliser des **fixtures** pour éviter la duplication de code.

9 Bibliographie et webographie

- Bibliographie :
 - Hunter, B. (2021). *Python Testing with pytest*. Pragmatic Bookshelf.
- Webographie :
 - Documentation officielle de pytest : : <https://docs.pytest.org/>
 - Tutoriel pytest : [Effective Python Testing With pytest – Real Python](#)
 - Grok, ChatGpt et Deeseek