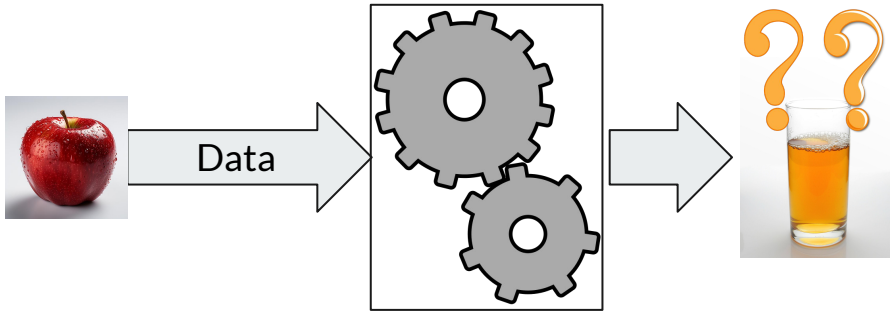
Les tests

Définition

Les tests de logiciels consistent à vérifier si les logiciels répondent aux attentes.

Un test ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de trois éléments : les **données en entrée**, l'**objet à tester** et les **observations attendues**. Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions et, dans l'idéal, être reproduit



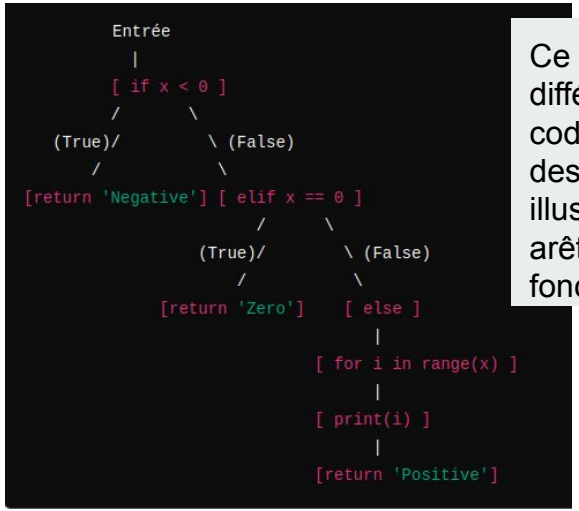
Les tests :

- Tests fonctionnels : valider la conformité d'un logiciel à son cahier des charges (recette informatique).
- Tests unitaires : tester le code éviter les régressions /Intégration continue
- Tests de robustesse : tester les réactions de l'application dans le cadre d'usages limite (sollicitation, volume de données).
- Stratégie de test : quoi tester, quand ?

Les métriques : complexité cyclomatique

La complexité cyclomatique est une mesure de la complexité d'un programme basée sur le nombre de chemins indépendants dans son graphe de contrôle. Elle est utilisée pour évaluer la complexité du code, en particulier pour déterminer la difficulté à tester et maintenir le code.

```python
def example_function(x):
    if x < 0:
        return 'Negative'
    elif x == 0:
        return 'Zero'
    else:
        for i in range(x):
            print(i)
        return 'Positive'
```



Ce graphe montre les différents chemins que le code peut prendre en fonction des valeurs de x, ce qui illustre les nœuds et les arêtes impliqués dans la fonction.

Calcul de la complexité cyclomatique

Complexite´ Cyclomatique=E−N+2P

- E est le nombre d'arêtes (transitions entre les nœuds).
- N est le nombre de nœuds (points de décision).
- P est le nombre de composants connectés (généralement 1 pour une seule fonction).

Interprétation :

- **1-10** : Complexité basse, facile à tester et maintenir.
- **11-20** : Complexité modérée, test et maintenance nécessitent plus d'efforts.
- **21-50** : Complexité élevée, le test et la maintenance sont difficiles.
- **>50** : Complexité très élevée, le code est très difficile à tester et à maintenir.

Des outils comme radon permette de calculer la complexité cyclomatique simplement :

```
# Installer radon
pip install radon

# Analyser un fichier Python pour la complexité cyclomatique
radon cc path/to/your/file.py -a
```

Densité de Défauts

La densité de défauts est une mesure du nombre de défauts trouvés dans une unité de taille de code (par exemple, par millier de lignes de code - KLOC). Elle permet d'évaluer la qualité du code en fonction du nombre de bugs ou de défauts signalés.

$$\text{Densité de Défauts} = \frac{\text{Nombre de Défauts}}{\text{Taille du Code en KLOC}}$$

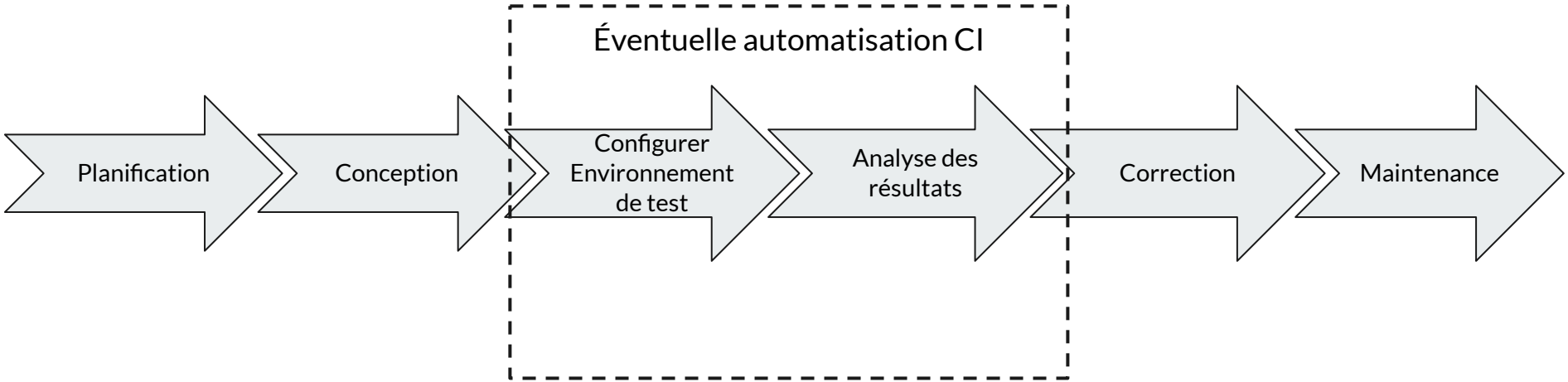
Utilisation des Métriques dans le Cycle de Vie du Projet

Revue de Code : Les métriques aident à identifier les parties du code qui nécessitent une révision plus approfondie.

Tests : La complexité cyclomatique peut aider à déterminer le nombre minimum de cas de test nécessaires pour obtenir une couverture complète.

Maintenance : Les métriques de qualité logicielle aident à planifier les efforts de maintenance en identifiant les zones du code susceptibles de contenir des défauts ou de nécessiter des améliorations.

Processus de test dans le cycle de vie d'un projet



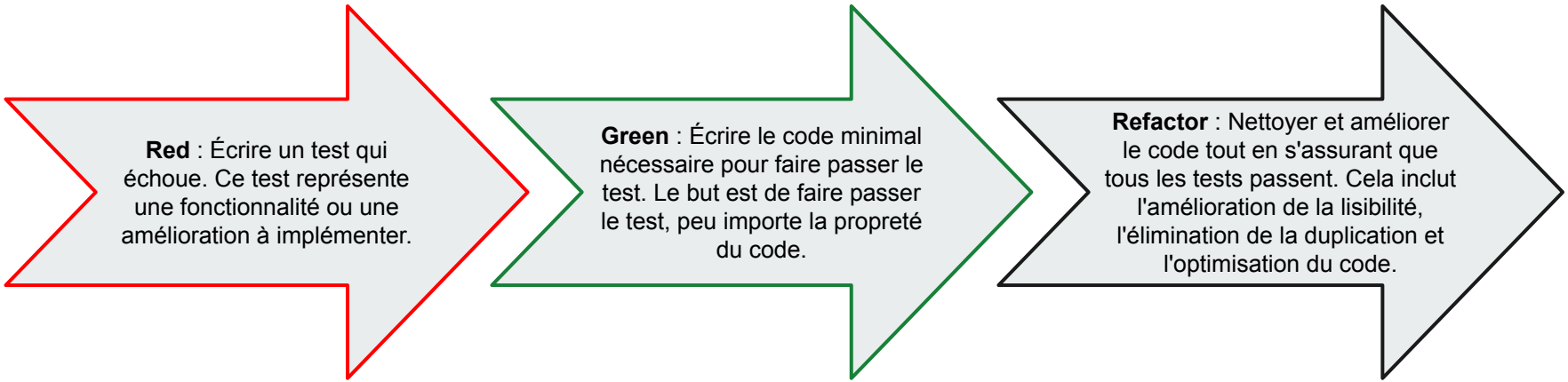
eXtreme Programming et TDD (test driven development)

L'eXtreme Programming (XP) est une méthodologie de développement agile qui met l'accent sur la flexibilité, la réactivité aux changements, la communication et la qualité du code. Elle se distingue par des pratiques telles que le développement piloté par les tests (TDD), le pair programming et l'intégration continue.

TDD

Le Test-Driven Development (TDD) est une pratique clé de l'eXtreme Programming (XP) et des méthodologies agiles en général. Il met l'accent sur l'écriture des tests avant le code de production. TDD est un processus itératif et consiste à écrire un test pour une nouvelle fonctionnalité, écrire le code nécessaire pour faire passer ce test, puis refactoriser le code.

Red -> Green -> Refactor



Styles de TDD

Classicist TDD (ou Chicago School)

- **Approche :** Met l'accent sur l'écriture de tests unitaires de bas niveau pour chaque composant du système. Les tests sont généralement isolés les uns des autres.
- **Objectif :** Garantir que chaque unité de code fonctionne correctement de manière isolée.
- **Exemple :** Test d'une méthode ou d'une fonction spécifique sans dépendre d'autres parties du système.

Mockist TDD (ou London School)

- **Approche :** Utilise des mocks et des stubs pour isoler les tests des dépendances. Met l'accent sur les interactions entre les objets.
- **Objectif :** Vérifier non seulement le comportement, mais aussi les interactions entre les composants du système.
- **Exemple :** Test d'un service en isolant ses dépendances avec des mocks.

Outside-In TDD (ou Behavior-Driven Development, BDD)

- **Approche :** Commence par écrire des tests au niveau des fonctionnalités (tests d'acceptation) puis descend progressivement vers des tests unitaires.
- **Objectif :** Se concentrer sur le comportement du système du point de vue de l'utilisateur.
- **Exemple :** Test d'une fonctionnalité complète avant de tester les composants individuels.

Refactoring

Le refactoring est le processus de restructuration du code existant sans en modifier le comportement externe. L'objectif est d'améliorer la lisibilité, la maintenabilité, et la qualité du code.

Les objectifs :

- Prévention de la Dérive Technologique
- Amélioration de la Lisibilité
- Réduction de la Complexité
- Augmentation de la Maintenabilité
- Optimisation des Performances

Les pratiques :

Compréhension du code.

Renommer les Variables et Méthodes.

Améliorer la lisibilité et la réutilisabilité.

Diviser des méthodes longues ou complexes en méthodes plus petites et plus spécifiques

Respect des bonnes pratiques (SOLID) et introduction des Design Patterns

Réorganiser les classes

Élimination du Code Dupliqué

Supprimer les répétitions de code en créant des méthodes ou des classes réutilisables.

Simplification des Conditions

Réduire la complexité des structures conditionnelles pour rendre le flux du programme plus évident.

Vocabulaire : fixture

Une **fixture** est un composant utilisé en tests logiciels pour configurer l'environnement de test avant d'exécuter les tests. Les fixtures sont couramment utilisées pour :

1. **Initialiser l'état** : Préparer les conditions nécessaires pour les tests, telles que la création d'objets, l'établissement de connexions à des bases de données ou la configuration d'autres ressources externes.
2. **Nettoyer après les tests** : Réaliser des opérations de nettoyage après l'exécution des tests, telles que la suppression d'objets, la fermeture de connexions ou la réinitialisation de l'état.

Vocabulaire : couverture de test

La **couverture de test** est une mesure qui évalue dans quelle mesure le code source d'un programme est testé par les tests automatisés. Elle permet de déterminer quelles parties du code ont été exécutées (et testées) et lesquelles ne l'ont pas été.

Types de couverture de test :

1. **Couverture de lignes** : Mesure le pourcentage de lignes de code qui ont été exécutées par les tests.
2. **Couverture de branches** : Évalue le pourcentage de chemins de décision (comme les conditions if/else) qui ont été pris par les tests.
3. **Couverture de fonctions** : Indique le pourcentage de fonctions qui ont été appelées pendant les tests.
4. **Couverture de conditions** : Vérifie que chaque condition dans les expressions booléennes a été évaluée à vrai et à faux.

Qu'est-ce que la couverture ? Atlassian.com

Gherkin

Gherkin est un langage de spécification de tests lisible par les non spécialiste qui permet de décrire les comportements attendus d'une application en utilisant une syntaxe simple et structurée. Il est structuré ainsi : contexte, événement, conséquence attendue.

Feature : Décrit la fonctionnalité testée.

Scenario : Décrit un cas de test spécifique.

Given : Décrit le contexte initial de l'application avant l'action.

When : Décrit l'action ou l'événement déclencheur.

Then : Décrit le résultat attendu ou la sortie.

```
Feature: Shopping Cart

  Scenario: Add a new item to the shopping cart
    Given the user is on the product page
    When the user selects a product
    And clicks the "Add to Cart" button
    Then the product should be added to the shopping cart
    And the shopping cart count should be incremented by 1

  Scenario: Remove an item from the shopping cart
    Given the user has items in the shopping cart
    When the user clicks the "Remove" button for a specific item
    Then the item should be removed from the shopping cart
    And the shopping cart count should be decremented by 1
```

Unittest

Unittest est le module de test intégré de Python, inspiré par le framework JUnit pour Java. Il permet de créer et d'exécuter des tests unitaires pour vérifier que les différentes parties du code fonctionnent comme prévu. Voici un aperçu des principales fonctionnalités de unittest :

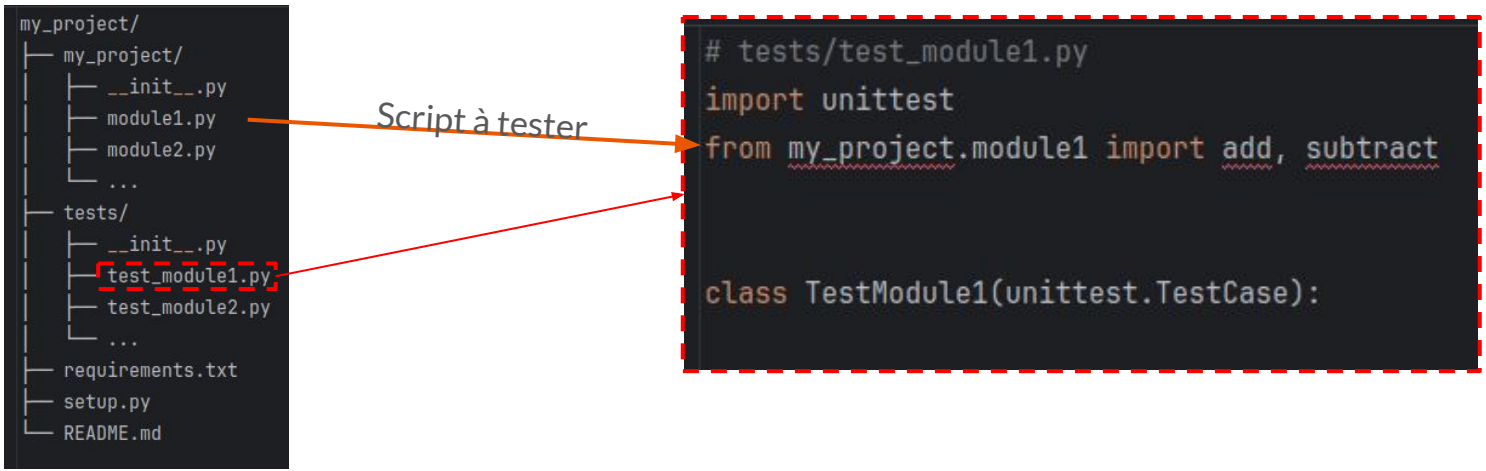
Caractéristiques principales

1. **Organisation des tests** : Les tests sont organisés en classes et méthodes.
2. **Assertions** : Fournit une variété de méthodes d'assertion pour vérifier les conditions.
3. **Fixtures** : Permet de définir des méthodes setUp et tearDown pour préparer l'environnement de test avant chaque test et le nettoyer après.
4. **Exécution des tests** : Les tests peuvent être exécutés via la ligne de commande ou intégrés dans des outils d'intégration continue.

Placement des tests dans un projet Python

Structure recommandée

1. **Dossier de tests séparé** : Placez tous les tests dans un dossier séparé à la racine du projet.
2. **Correspondance avec la structure du code source** : La structure des tests devrait refléter celle du code source pour une organisation cohérente et facile à naviguer.
3. **Nommer les fichiers de test** : Utilisez des noms de fichiers descriptifs qui indiquent clairement ce qui est testé. Préfixez souvent les noms de fichiers de test par test_.



Les assertions

Les assertions sont des instructions dans les tests qui vérifient si une condition est vraie. Si la condition est fausse, le test échoue.

```python
import unittest


4 usages
def subtract(a, b):
    return a - b


class TestMathOperations(unittest.TestCase):

    def test_subtract(self):
        self.assertEqual(subtract( a: 5,  b: 3),  second: 2)
        self.assertNotEqual(subtract( a: 5,  b: 3),  second: 1)
        self.assertTrue(subtract( a: 5,  b: 3) > 0)
        self.assertFalse(subtract( a: 5,  b: 3) < 0)
```

Principales assertions :

- assertEqual(a, b) : Vérifie que a == b
- assertNotEqual(a, b) : Vérifie que a != b
- assertTrue(x) : Vérifie que x est vrai
- assertFalse(x) : Vérifie que x est faux
- assertIsNone(x) : Vérifie que x est None
- assertIsNotNone(x) : Vérifie que x n'est pas None

Effets de bord et test

- **Définition** : Un effet de bord se produit lorsque l'exécution d'une fonction modifie l'état externe de l'application ou interagit avec des ressources externes, telles que des fichiers, des bases de données, ou des API.
- **Problème** : Les effets de bord rendent les tests unitaires difficiles car ils introduisent des dépendances qui ne sont pas contrôlées par le test.

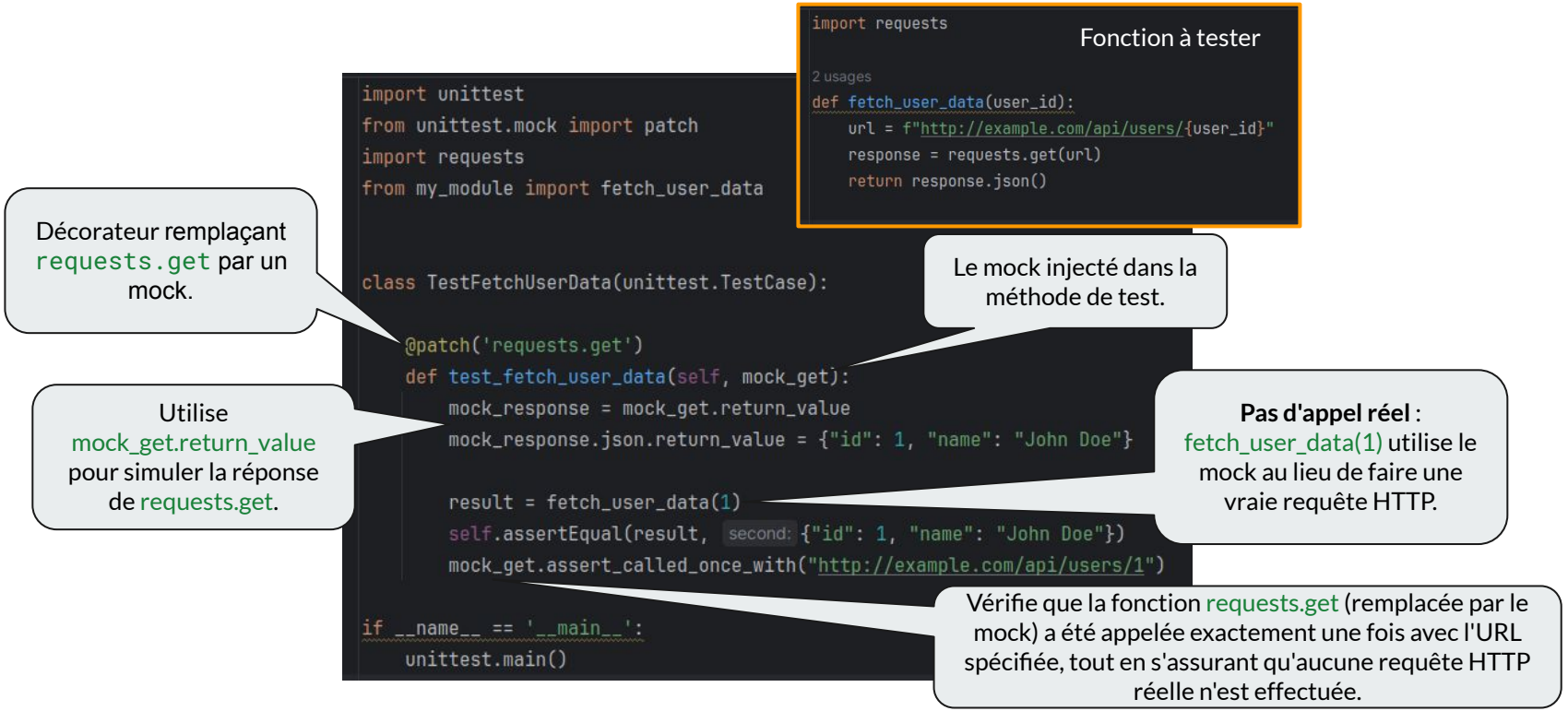
unittest.mock

Définition : unittest.mock est une bibliothèque de Python utilisée pour créer des objets mock (simulacres) qui imitent le comportement des objets réels. Il permet de simuler et de contrôler les interactions avec les dépendances externes.

Composants Clés :

- **Mock** : Un objet mock générique qui peut simuler n'importe quel objet.
- **patch** : Un décorateur/context manager pour remplacer des objets dans le code sous test par des mocks.

unittest.mock



setUp et tearDown

- **setUp** est une méthode qui est exécutée avant chaque méthode de test. Elle est utilisée pour initialiser des objets ou des ressources nécessaires aux tests.
- **tearDown** est une méthode qui est exécutée après chaque méthode de test. Elle est utilisée pour nettoyer ou réinitialiser les objets ou les ressources utilisés pendant les tests.

setUp et tearDown

```python
import unittest
from database_demo import Database  # Assurez-vous que ce chemin est correct

class TestDatabase(unittest.TestCase):

    def setUp(self):
        """Initialisation avant chaque test."""
        self.db = Database()
        print("setUp: Initialisation de la base de données")

    def tearDown(self):
        """Nettoyage après chaque test."""
        self.db.close()
        print("tearDown: Fermeture de la base de données")

    def test_add_user(self):
        """Test de l'ajout d'un utilisateur."""
        user_id = self.db.add_user('John Doe')
        self.assertIsNotNone(user_id)
        print(f"test_add_user: Utilisateur ajouté avec l'ID {user_id}")

    # etc...
```

Ouvre la communication avec la base de données avant le début du test.

Ferme la communication avec la base de données après la fin du test.

Tests d'IHM web

Les tests d'IHM web permettent de vérifier que l'interface utilisateur d'une application web fonctionne correctement. Ils simulent les interactions de l'utilisateur avec l'application, comme la navigation, la saisie de données, et la soumission de formulaires.

Frameworks de tests fonctionnels

Selenium est un framework open-source permettant d'automatiser les tests des applications web. Il permet de contrôler un navigateur web en utilisant un script. Il supporte de nombreux navigateur (Chrome, Firefox, Safari, etc.) et s'intègre avec divers langages de programmation (Python, Java, C#, etc.).

Autres Frameworks

- **Cypress** : Moderne, rapide et facile à configurer, mais limité à JavaScript.
- **Playwright** : Développé par Microsoft, similaire à Cypress avec un support multi-langages.
- **Puppeteer** : Contrôle de Chrome/Chromium, principalement utilisé avec Node.js.

Configuration de Selenium

Afin de pouvoir contrôler le navigateur, Selenium a besoin d'un driver compatible avec la version du navigateur :

- Le driver pour Chrome est disponible ici : https://googlechromelabs.github.io/chrome-for-testing/
- Le driver pour FireFox est disponible ici : https://github.com/mozilla/geckodriver/releases

Installation du web driver :

- Sur Linux : ce tutoriel explique les étapes de l'installation (pour éviter les incompatibilités entre navigateur et web driver, téléchargez directement le driver sur le lien indiqué plus haut, sans passer par l'étape wget du tutoriel). https://skolo.online/documents/webscrapping/#step-2-install-chromedriver
- Pour windows (et autres) : https://www.makeuseof.com/how-to-install-selenium-webdriver-on-any-computer-with-python

Atelier d’application

Cette page d'entraînement permet de s’exercer avec Selenium :
https://practicetestautomation.com/practice-test-login/

La correction du premier exercice vous est donné. Tenter de trouver le code de test adapté pour les cas 2 et 3.