

Tests Unitaires en Python

Introduction

Les tests unitaires sont une technique essentielle en développement logiciel permettant de vérifier que les unités de code (fonctions, classes, modules) se comportent comme attendu. Ils aident à détecter les erreurs rapidement et facilitent la maintenance du code en permettant d'apporter des modifications en toute confiance.

Les tests unitaires reposent sur l'idée que chaque composant du code doit être testé de manière indépendante pour garantir son bon fonctionnement. Une suite de tests bien conçue permet non seulement de valider que le code fonctionne comme prévu, mais aussi de prévenir les régressions lors de modifications ultérieures.

Python fournit un module intégré, `unittest`, qui permet d'écrire et d'exécuter des tests unitaires de manière structurée. Nous allons aussi vous parler des mocks qui permettent de tester du code en impliquant des dépendances externes.

On parle chinois ? Ok, lisez ceci :

"Imaginez que vous développez un site e-commerce. Après une mise à jour, un client signale que la fonction de calcul du total de la commande est incorrecte. Vous devez vérifier si la mise à jour a introduit une régression. Comment être sûr que chaque modification du code ne casse pas une autre partie de l'application ? La réponse : les tests unitaires !"

Objectifs des tests unitaires

Les tests unitaires visent plusieurs objectifs :

- **Validation du comportement attendu** : Assurer que chaque unité de code produit le résultat attendu en fonction des entrées fournies.
- **Détection rapide des erreurs** : En exécutant les tests après chaque modification, on identifie immédiatement les problèmes introduits.
- **Facilitation de la maintenance** : Un bon ensemble de tests permet de modifier le code en toute confiance.
- **Amélioration de la qualité du code** : Écrire des tests pousse à mieux structurer son code pour le rendre plus modulaire et testable.

1. Caractéristiques des Tests Unitaires

Les tests unitaires doivent respecter certaines propriétés pour être efficaces :

- **Automatisables** : Ils doivent pouvoir être exécutés sans intervention humaine.
- **Rapides** : Un test unitaire ne doit pas nécessiter de longues minutes pour s'exécuter.
- **Indépendants** : Chaque test doit être isolé des autres afin d'éviter les effets de bord.
- **Répétables** : Un test doit toujours produire le même résultat pour les mêmes entrées.

2. Quand et comment écrire des tests unitaires ?

Les tests unitaires doivent être écrits en parallèle du développement du code ou immédiatement après. Il est conseillé d'adopter une approche **Test-Driven Development (TDD)**, où l'on écrit d'abord les tests avant d'implémenter le code.

Bonnes pratiques pour écrire des tests unitaires

1. **Nommer les tests de manière explicite** : Le nom d'un test doit clairement indiquer ce qu'il vérifie.
2. **Tester un seul comportement par test** : Chaque test doit se concentrer sur un aspect précis du code.
3. **Utiliser des jeux de données pertinents** : Tester avec différentes valeurs d'entrée permet d'identifier d'éventuels problèmes.
4. **Isoler les dépendances avec des mocks** : Si une fonction dépend d'une API externe, d'une base de données ou d'un fichier, utiliser un mock pour ne tester que la logique interne.
5. **Mettre en place des tests couvrant les cas limites** : Vérifier les cas extrêmes et les comportements en cas d'erreur.

Le rôle des tests dans un projet concret

Pourquoi les tests sont-ils essentiels ?

Dans un projet réel, le développement logiciel suit généralement un cycle comprenant plusieurs étapes : conception, développement, tests, déploiement et maintenance. Les tests unitaires jouent un rôle clé dans ce processus en permettant d'assurer la fiabilité et la stabilité du code.

Cas concret : développement d'une application web

Prenons l'exemple d'une application web de gestion de tâches.

Scénario sans tests unitaires

- Un développeur implémente une fonctionnalité permettant d'ajouter des tâches.
- Une mise à jour est effectuée pour améliorer les performances.
- Sans tests unitaires, il n'y a aucun moyen rapide de s'assurer que la mise à jour n'a pas cassé l'ajout de tâches.
- Après le déploiement, les utilisateurs signalent des bugs qui auraient pu être détectés en amont.

Scénario avec tests unitaires

- Lors du développement, des tests unitaires sont écrits pour vérifier que la fonctionnalité d'ajout fonctionne correctement.
- Avant chaque mise à jour, les tests sont exécutés automatiquement.
- Si une mise à jour introduit un bug, les tests échouent immédiatement, permettant aux développeurs de corriger le problème avant le déploiement.

Automatisation des tests et intégration continue

Dans les projets professionnels, les tests unitaires sont souvent intégrés dans un processus d'**intégration continue (CI/CD)**.

- À chaque modification du code, une plateforme comme **GitHub Actions**, **GitLab CI/CD** ou **Jenkins** exécute les tests automatiquement.
- Si tous les tests passent, la nouvelle version du code peut être fusionnée et déployée en production.
- En cas d'échec, les développeurs sont alertés immédiatement.

Avantages dans un projet d'équipe

- **Collaboration facilitée** : plusieurs développeurs peuvent travailler sur le même code sans crainte de régressions.
- **Documentation du comportement attendu** : les tests servent aussi de référence pour comprendre comment le code est censé fonctionner.
- **Gain de temps sur le long terme** : détecter les bugs tôt évite des corrections coûteuses en production.

En résumé, intégrer les tests unitaires dans un projet réel permet de gagner en **qualité, fiabilité et efficacité** !

3. Types de Tests Complémentaires

Les tests unitaires font partie d'une pyramide de tests comprenant d'autres niveaux de validation :

- **Tests d'intégration** : Vérifient que plusieurs modules interagissent correctement.
- **Tests système** : S'assurent que l'application fonctionne comme prévu dans un environnement complet.
- **Tests fonctionnels** : Vérifient que l'application répond aux exigences fonctionnelles définies.

4. Limitations des Tests Unitaires

Bien que les tests unitaires soient très utiles, ils présentent certaines limites :

- **Ne garantissent pas l'absence totale de bugs** : Ils valident des unités spécifiques, mais ne couvrent pas nécessairement tous les scénarios d'utilisation.
- **Nécessitent un effort de maintenance** : Lorsque le code évolue, les tests doivent être mis à jour en conséquence.
- **Peuvent donner un faux sentiment de sécurité** : Un code peut réussir tous ses tests unitaires mais échouer en production en raison d'un problème d'intégration.

5. Comprendre les Assertions dans unittest

Les assertions sont des vérifications permettant de comparer une valeur obtenue avec une valeur attendue. Si la comparaison échoue, le test est marqué comme un échec.

5.1. Structure de base d'un test unitaire

Un test unitaire est généralement structuré en trois étapes :

1. **Mise en place (setup)** : initialisation des données nécessaires au test.
2. **Exécution** : appel de la fonction ou de la méthode à tester.
3. **Vérification (assertion)** : comparaison des résultats obtenus avec ceux attendus.

Voici un exemple minimaliste :

```
import unittest

def addition(a, b):
    return a + b

class TestAddition(unittest.TestCase):
    def test_addition(self):
        result = addition(2, 3)
        expected = 5
        self.assertEqual(result, expected, "L'addition de 2 et 3 doit donner
5")

if __name__ == "__main__":
    unittest.main()
```

5.2. Explication détaillée du code

- La fonction addition(a, b) retourne la somme de a et b.
- La classe TestAddition hérite de unittest.TestCase, ce qui permet d'utiliser les méthodes de test intégrées.
- La méthode test_addition exécute addition(2, 3), puis compare le résultat à la valeur attendue 5 grâce à self.assertEqual(result, expected).
- unittest.main() exécute tous les tests présents dans le fichier.

Remarques :

Explanation :

1. __name__ en Python

- Lorsque Python exécute un script, il attribue une valeur à la variable spéciale __name__.
- Si le fichier est exécuté directement, __name__ prend la valeur "__main__".
- Si le fichier est importé en tant que module dans un autre script, __name__ prend le nom du fichier (sans l'extension .py).

2. Pourquoi l'utiliser ?

- Cela permet de séparer le code qui doit s'exécuter uniquement lorsqu'on lance directement le fichier du code qui doit être disponible lorsqu'il est importé.
- Très utile pour structurer le code et éviter des exécutions non désirées lors de l'importation d'un module.

5.3. Principales méthodes d'assertion

Les assertions sont des fonctions fournies par `unittest.TestCase` permettant de vérifier divers aspects des résultats attendus. Voici les principales :

Méthode	Description
<code>assertEqual(a, b)</code>	Vérifie que <code>a == b</code>
<code>assertNotEqual(a, b)</code>	Vérifie que <code>a != b</code>
<code>assertTrue(x)</code>	Vérifie que <code>x</code> est <code>True</code>
<code>assertFalse(x)</code>	Vérifie que <code>x</code> est <code>False</code>
<code>assertIs(a, b)</code>	Vérifie que <code>a</code> est <code>b</code> (même objet)
<code>assertIsNot(a, b)</code>	Vérifie que <code>a</code> n'est pas <code>b</code>
<code>assertIsNone(x)</code>	Vérifie que <code>x</code> est <code>None</code>
<code>assert IsNotNone(x)</code>	Vérifie que <code>x</code> n'est pas <code>None</code>
<code>assertIn(a, b)</code>	Vérifie que <code>a</code> est dans <code>b</code>
<code>assertNotIn(a, b)</code>	Vérifie que <code>a</code> n'est pas dans <code>b</code>
<code>assertRaises(exception, fonction, *args)</code>	Vérifie que <code>fonction(*args)</code> lève une exception

6 Comparatif `unittest` vs `pytest`

Critère	<code>unittest</code>	<code>pytest</code>
Simplicité	Plus verbeux	Syntaxe plus concise
Exécution	<code>unittest.main()</code>	<code>pytest</code> sans besoin de classe
Assertions	<code>self.assertEqual(a, b)</code>	<code>assert a == b</code>
Extensions	Standard	Très extensible avec plugins
Découverte des tests	Nécessite d'hériter de <code>unittest.TestCase</code>	Recherche automatique des fichiers <code>test_*.py</code>

Exemple de test avec `pytest` :

```
import pytest
from mon_module import addition

def test_addition():
    assert addition(2, 3) == 5
```

7. Explication des Mocks en Python avec unittest.mock

Introduction

Lors de l'écriture de tests unitaires, il est courant de rencontrer des fonctions ou des classes qui dépendent d'éléments externes comme des bases de données, des API, des fichiers système, ou même d'autres modules internes. Tester directement ces dépendances peut être problématique, car :

- Elles peuvent ralentir les tests (par exemple, si une requête API prend plusieurs secondes).
- Elles peuvent introduire des erreurs indépendantes du code testé (comme une panne du serveur API).
- Elles rendent les tests plus difficiles à exécuter de manière isolée.

C'est là qu'interviennent les **Mocks** ! Un **mock** est un objet simulé qui imite le comportement d'un objet réel tout en offrant un contrôle total sur ses interactions. En Python, on utilise le module `unittest.mock` pour créer et manipuler ces objets factices.

Pourquoi utiliser des Mocks ?

L'utilisation des mocks en tests unitaires présente plusieurs avantages majeurs :

1. **Isolation** : On peut tester un module sans dépendre des interactions avec des composants externes.
2. **Contrôle des résultats** : On peut forcer les retours des fonctions simulées pour tester des scénarios spécifiques.
3. **Vérification des appels** : On peut vérifier si une méthode a été appelée, combien de fois et avec quels arguments.
4. **Simulation d'erreurs** : On peut simuler des comportements d'échec pour tester la robustesse du code.
5. **Amélioration de la rapidité des tests** : Éviter d'attendre des opérations réseau ou des accès disque accélère les tests.

Fonctionnement de unittest.mock

Identifier les dépendances externes

Avant de commencer à écrire un test avec un mock, il faut identifier les parties du code qui interagissent avec des services externes. Ces dépendances peuvent être :

- Des appels à une API (ex: `requests.get`)
- Des accès à une base de données
- Des fichiers lus ou écrits
- Des classes ou fonctions internes qui doivent être isolées

Création d'un Mock Basique

Un **mock** est un objet que l'on peut paramétrer pour retourner une valeur spécifique lorsqu'on l'appelle :

```
from unittest.mock import Mock

# Création d'un mock
mock_obj = Mock()

# Définition d'un retour spécifique
mock_obj.method.return_value = "Valeur simulée"

# Appel de la méthode mockée
print(mock_obj.method()) # Affiche : Valeur simulée
```

Simuler une fonction ou une méthode avec patch

Le décorateur `patch` permet de remplacer temporairement une fonction ou une classe par un mock pendant l'exécution d'un test.

Utilisation de patch comme décorateur :

```
from unittest.mock import patch
import unittest

def get_data():
    return "Données réelles"

def process_data():
    data = get_data()
    return data.upper()

class TestMock(unittest.TestCase):
    @patch("__main__.get_data", return_value="Données simulées")
    def test_process_data(self, mock_get_data):
        result = process_data()
        self.assertEqual(result, "DONNÉES SIMULÉES")
```

Explication :

- `@patch("__main__.get_data", return_value="Données simulées")` remplace temporairement `get_data` par une version qui retourne toujours "Données simulées".
- Le test exécute `process_data()`, qui utilise le mock au lieu de la fonction réelle.
- On vérifie que le résultat final correspond à ce qu'on attend.

Explication détaillée

1 Importation des modules

```
from unittest.mock import patch  
import unittest
```

- `unittest.mock.patch` : permet de **remplacer temporairement une fonction** par une version simulée.
- `unittest` : module Python pour **écrire et exécuter des tests unitaires**.

2 Définition des fonctions

```
def get_data():  
  
    return "Données réelles"
```

- **Fonction `get_data()` :**
 - Cette fonction retourne la chaîne "Données réelles".
 - Dans une vraie application, elle pourrait récupérer des données d'une **API**, d'un **fichier** ou d'une **base de données**.

```
def process_data():  
  
    data = get_data()  
  
    return data.upper()  
  
• Fonction process_data() :

- Elle appelle get_data() pour récupérer des données.
- Elle convertit ces données en majuscules (data.upper()).
- Retour attendu : "Données réelles" devient "DONNÉES RÉELLES".

```

3 Classe de test avec mock

```
class TestMock(unittest.TestCase):  
  
    • Définition d'une classe de test qui hérite de unittest.TestCase.  
  
    @patch("__main__.get_data", return_value="Données simulées")  
    • Décorateur @patch("__main__.get_data", return_value="Données simulées") :

- Remplace temporairement get_data() par une version simulée qui retourne "Données simulées".
- "__main__.get_data" :
  - __main__ est le nom du module courant.
  - get_data est la fonction à remplacer.

```

```

def test_process_data(self, mock_get_data):
    • La fonction test_process_data(self, mock_get_data) :
        ○ mock_get_data est un argument qui reçoit le mock appliqué à get_data().
        ○ Ce paramètre n'est pas utilisé directement dans le test, car patch s'applique déjà.

result = process_data()
    • Exécution de process_data() :
        ○ Normalement, process_data() appelle get_data(), qui retourne "Données réelles".
        ○ Grâce au mock, get_data() est remplacée et retourne "Données simulées".
            self.assertEqual(result, "DONNÉES SIMULÉES")
    • Vérification du résultat :
        ○ process_data() doit retourner "DONNÉES SIMULÉES" en majuscules.
        ○ Si le mock fonctionne correctement, self.assertEqual(result, "DONNÉES SIMULÉES") doit réussir.

```

Comportement du mock dans ce test

- Sans mock :
 - `get_data()` retourne "Données réelles", donc `process_data()` retourne "DONNÉES RÉELLES".
- Avec mock :
 - `get_data()` **est remplacée** et retourne "Données simulées".
 - `process_data()` retourne donc "DONNÉES SIMULÉES".

Résumé des rôles

Élément	Rôle
<code>get_data()</code>	Fonction qui retourne des données réelles.
<code>process_data()</code>	Appelle <code>get_data()</code> et met les données en majuscules.
<code>@patch("__main__.get_data", return_value="Données simulées")</code>	Mock <code>get_data()</code> pour retourner "Données simulées" au lieu de "Données réelles".
<code>test_process_data(self, mock_get_data)</code>	Vérifie que <code>process_data()</code> fonctionne avec les données simulées.

Utilisation de patch avec with

Plutôt qu'un décorateur, on peut utiliser patch sous forme de gestionnaire de contexte :

```
with patch("__main__.get_data", return_value="Données simulées"):
    print(get_data()) # Affiche "Données simulées"
```

Cela permet d'appliquer le mock uniquement dans un bloc précis du code.

Explication complète du with et différence

1 Même code avec with patch()

```
from unittest.mock import patch
import unittest
def get_data():
    return "Données réelles"
def process_data():
    data = get_data()
    return data.upper()
class TestMock(unittest.TestCase):
    def test_process_data(self):
        with patch("__main__.get_data", return_value="Données simulées") as
mock_get_data:
            result = process_data()
            self.assertEqual(result, "DONNÉES SIMULÉES")
```

2 Explication détaillée

A) Fonctionnement de with patch()

```
with patch("__main__.get_data", return_value="Données simulées") as
mock_get_data:
```

1. **patch("__main__.get_data", return_value="Données simulées")**
 - Remplace temporairement get_data() par un **mock** qui **retourne "Données simulées"**.
 - **Se fait uniquement à l'intérieur du bloc with.**
2. **as mock_get_data**
 - Permet d'accéder au mock si on veut l'inspecter (mock_get_data).
3. **Tout ce qui est dans le bloc with**
 - **Utilisera la version mockée** de get_data().
 - Quand on appelle process_data(), celui-ci exécute get_data(), qui est maintenant **mocké**.
4. **En sortie du with**
 - **patch() rétablit automatiquement get_data()** dans son état initial.
 - get_data() redevient "Données réelles".

B) Différences avec @patch()

Version avec @patch() (décorateur)

```
from unittest.mock import patch
import unittest

class TestMock(unittest.TestCase):
    @patch("__main__.get_data", return_value="Données simulées")
    def test_process_data(self, mock_get_data):
        result = process_data()
        self.assertEqual(result, "DONNÉES SIMULÉES")
```

Comparaison des deux approches

Critère	@patch() (décorateur)	with patch()
Portée du mock	Toute la fonction de test	Seulement à l'intérieur du with
Injection du mock	Automatique en argument du test	Assigné manuellement avec <code>as mock_variable</code>
Facilité de lecture	Simple si on mocke une seule fonction	Plus clair quand on mocke plusieurs éléments
Réversibilité	Le mock est annulé après la fin du test	Le mock est annulé dès la sortie du bloc with
Flexibilité	Moins flexible (gère un seul mock)	Permet de mocker plusieurs fonctions proprement
Bonne pratique	Tests simples, une seule fonction à mocker	Cas complexes, plusieurs mocks, besoin de rétablir rapidement une fonction

3 Exemples concrets pour voir la différence

Cas où @patch() est plus simple

Si on veut **mockeer une seule fonction**, @patch() est **plus rapide et lisible** :

```
class TestMock(unittest.TestCase):
    @patch("__main__.get_data", return_value="Données simulées")
    def test_process_data(self, mock_get_data):
        result = process_data()
        self.assertEqual(result, "DONNÉES SIMULÉES")
```

Avantage :

- Pas besoin de with, le mock est **géré automatiquement** par unittest.
- **Simple et efficace** pour **un seul mock**.

Cas où with patch() est mieux

Si on veut **mocker plusieurs fonctions en même temps**, with patch() est plus adapté :

```
class TestMock(unittest.TestCase):
    def test_process_data(self):
        with patch("__main__.get_data", return_value="Données simulées"), \
            patch("__main__.another_function", return_value=42):

            result = process_data()
            self.assertEqual(result, "DONNÉES SIMULÉES")

            # Tester another_function()
            self.assertEqual(another_function(), 42)
```

Avantages :

- Permet de **mocker plusieurs fonctions en même temps**.
- Les mocks **n'affectent que le bloc with**, donc la suite du code reste normale.
- **Plus propre** si on veut éviter d'injecter trop d'arguments dans la fonction de test.

4 Quelle approche choisir ?

Cas d'usage	Recommandation
Mocker une seule fonction	Utiliser @patch() (plus simple et lisible)
Mocker plusieurs fonctions	Utiliser with patch() (plus propre)
Besoin d'un mock temporaire dans un test	Utiliser with patch()
Besoin d'un mock sur tout le test	Utiliser @patch()

Conclusion

@patch() est plus rapide et plus lisible quand on veut **mocker une seule fonction**.

with patch() est plus flexible et propre quand on veut **mocker plusieurs fonctions** ou limiter l'effet du mock à une partie du test.

Meilleure approche :

- **Si tu débutes**, commence par @patch(), c'est plus simple.
- **Si tu as plusieurs mocks, prends with patch()** pour un code plus clair.

Vérification des appels à un Mock

L'un des principaux intérêts des mocks est la possibilité de vérifier comment ils ont été utilisés dans le test. Les mocks permettent aussi de s'assurer qu'une fonction a bien été appelée, et avec quels arguments.

```
mock_obj = Mock()
mock_obj.method("argument1", key="valeur")

# Vérification des appels
mock_obj.method.assert_called() # Vérifie que la méthode a été appelée
mock_obj.method.assert_called_with("argument1", key="valeur") # Vérifie les arguments
mock_obj.method.assert_called_once() # Vérifie que la méthode a été appelée une seule fois
mock_obj.method.assert_called_once_with("argument1", key="valeur") # Vérifie les arguments et le nombre d'appels
mock_obj.method.assert_called_with("argument1") # Échoue car l'argument "key" est manquant
mock_obj.method.assert_called_with("argument1", key="valeur",
key2="valeur2") # Échoue car l'argument "key2" est en trop
```

Simulation d'une classe complète avec Mock

On peut également simuler une classe entière avec un mock pour tester un module de manière isolée.

```
from unittest.mock import Mock

# Classe réelle qui interagit avec une source externe
class Service:
    def fetch_data(self):
        """Simule l'accès à une base de données ou une API externe."""
        return "Données réelles"

# Classe qui dépend de Service
class Consumer:
    def __init__(self, service):
        self.service = service

    def process(self):
        """Transforme les données obtenues en minuscules."""
        data = self.service.fetch_data()
        return data.lower()
```

```

# Test de Consumer en utilisant un Mock pour remplacer Service
class TestConsumer(unittest.TestCase):
    def test_process(self):
        # Création d'un mock pour la classe Service
        mock_service = Mock()
        mock_service.fetch_data.return_value = "Données simulées"

        # Injection du mock dans Consumer
        consumer = Consumer(mock_service)

        # Vérification du comportement attendu
        self.assertEqual(consumer.process(), "données simulées")

        # Vérification que la méthode fetch_data a bien été appelée
        mock_service.fetch_data.assert_called_once()

```

Comment vous pourriez appliquer ces concepts dans tous les cas ?

1. **Analyser le code** : Repérer les parties dépendantes d'éléments externes (API, base de données, fichiers, etc.).
2. **Déterminer ce qui doit être mocké** : Identifier les fonctions et classes qu'il est pertinent de remplacer par un mock.
3. **Créer un mock adapté** : Utiliser Mock() pour simuler une classe ou une fonction.
4. **Remplacer dynamiquement avec patch** : Si nécessaire, utiliser patch pour intercepter les appels réels et les rediriger vers le mock.
5. **Vérifier le comportement attendu** : Toujours s'assurer que les fonctions ont été appelées correctement en utilisant assert_called() et assert_called_with().
6. **Tester différents scénarios** : Penser aux cas normaux, aux erreurs, et aux cas limites.

Bonnes Pratiques pour Utiliser les Mocks

1. **Mocker uniquement ce qui est nécessaire** : Évitez de mockez des fonctions internes de votre module si elles ne dépendent pas de services externes.
2. **Vérifier les appels des mocks** : Toujours valider que le mock a bien été utilisé comme prévu (ex. assert_called_with).
3. **Utiliser des noms explicites** : Les noms des mocks doivent clairement indiquer leur rôle dans le test.
4. **Utiliser patch avec discernement** : Évitez de patcher toute une classe si seule une méthode spécifique doit être remplacée.
5. **Simuler différents scénarios** : Pensez à tester les retours normaux mais aussi les erreurs (ex. lever une exception avec side_effect).

Erreurs courantes et comment les éviter

1. Ne pas isoler les tests

- Mauvais : Tester une fonction qui accède directement à une API
- Solution : Utiliser unittest.mock

2. Écrire des tests trop généraux

- Mauvais : Un test avec plusieurs assertions qui testent plusieurs comportements
- Solution : Un test par cas précis

3. Dépendre de l'ordre des tests

- Mauvais : Un test qui échoue si un autre test ne s'est pas exécuté avant
- Solution : Chaque test doit être indépendant

Exercice 1 : Mock d'une fonction simple

Contexte : Vous développez une application météo qui récupère la température actuelle d'un capteur. La fonction obtenir_temperature() retourne normalement une température en Celsius. Cependant, vous devez tester convertir_en_fahrenheit() qui dépend de cette fonction.

Objectif : Mockez obtenir_temperature() pour tester si convertir_en_fahrenheit() fonctionne correctement.

Exercice 2 : Vérifier les appels d'une méthode mockée

Contexte : Une application de gestion des utilisateurs envoie un email lorsqu'un nouvel utilisateur s'inscrit.

Objectif : Créez une classe ServiceUtilisateur avec une méthode envoyer_email(self, email). Testez que cette méthode est bien appelée lorsqu'un utilisateur s'inscrit.

Exercice 3 : Mock d'une API externe

Contexte : Vous devez récupérer des informations météorologiques d'une API publique via requests.get(). Cependant, l'API peut parfois être indisponible.

Objectif : Écrivez une fonction qui fait un appel à requests.get() pour récupérer des données JSON. Testez cette fonction en remplaçant requests.get() par un mock et forcez différents scénarios (réponse valide, erreur réseau, JSON mal formé).

Exercice 4 : Simulation d'une base de données

Contexte : Un service d'authentification vérifie les utilisateurs en consultant une base de données.

Objectif : Créez une classe BaseDeDonnees avec une méthode rechercher_utilisateur(nom). Testez un service ServiceAuth qui utilise BaseDeDonnees pour authentifier un utilisateur en mockant les retours possibles (utilisateur trouvé, utilisateur inexistant).

Exercice 5 : Lever une exception avec un mock

Contexte : Lorsque la base de données ne trouve pas un utilisateur, elle lève une exception. L'application doit gérer correctement cette situation.

Objectif : Modifiez BaseDeDonnees.rechercher_utilisateur() pour lever une exception si l'utilisateur n'est pas trouvé. Testez que ServiceAuth gère correctement cette exception en capturant et en traitant l'erreur.

Exercice 6 : Mock de classe entière

Contexte : Un site e-commerce a un service de paiement qui traite les transactions. Vous devez tester une classe CommandeService qui dépend du service PaiementService.

Objectif : Créez une classe PaiementService avec traiter_paiement(montant). Testez CommandeService en remplaçant traiter_paiement() par un mock et en vérifiant que la commande est bien validée après un paiement réussi.

Ces exercices permettent aux étudiants de comprendre les différentes facettes de l'utilisation des mocks dans des contextes variés et réalistes.

Exercice 7 : Mock d'un capteur à ultrasons

Contexte : Le véhicule utilise un capteur à ultrasons pour mesurer la distance aux objets environnants et ajuster sa vitesse en conséquence.

Objectif : Écrivez une fonction mesurer_distance() qui retourne une distance en centimètres. Testez cette fonction en simulant différentes distances et en vérifiant que la vitesse du véhicule s'ajuste correctement.

Exercice 8 : Simulation d'un capteur de courant

Contexte : Pour éviter de surcharger le circuit électrique du véhicule, un capteur mesure l'ampérage et déclenche une alerte si une surcharge est détectée.

Objectif : Créez une classe CapteurCourant avec une méthode mesurer_ampere(). Mockez cette méthode pour simuler différents niveaux d'ampérage et testez que le véhicule coupe l'alimentation en cas de dépassement du seuil.

Exercice 9 : Test d'un capteur infrarouge

Contexte : Vous travaillez sur un véhicule autonome qui détecte des obstacles grâce à un capteur infrarouge. La fonction detecter_obstacle() retourne True si un obstacle est détecté et False sinon.

Objectif : Mockez detecter_obstacle() pour tester si le système réagit correctement dans les deux cas (obstacle détecté ou non).

Exercice 10 : Simulation d'un capteur de couleur

Contexte : Un capteur de couleur est utilisé pour détecter les marquages au sol et ajuster la trajectoire du véhicule.

Objectif : Créez une classe CapteurCouleur avec une méthode detecter_couleur(). Mockez cette méthode pour simuler la détection de différentes couleurs et testez que le véhicule change correctement de direction.

Exercice 11 : Détection d'erreur avec un mock

Contexte : Si un capteur de courant retourne une valeur aberrante (ex. None ou une valeur négative), le système doit déclencher une alerte.

Objectif : Modifiez CapteurCourant.mesurer_ampere() pour lever une exception en cas de valeur incorrecte. Testez que le véhicule gère correctement cette exception et passe en mode sécurité.

Exercice 12 : Mock de l'ensemble des capteurs

Contexte : Pour s'assurer que l'ensemble des capteurs fonctionne bien ensemble, on teste leur intégration dans un module SystemeCapteurs.

Exercice 7 : Test d'un capteur à ultrasons

```
from unittest.mock import Mock
import unittest

def mesurer_distance():
    """Retourne la distance mesurée en cm"""
    pass

class TestCapteurUltrasons(unittest.TestCase):
    def test_mesurer_distance(self):
        mock_distance = Mock()
        mock_distance.return_value = 15 # Simuler une distance de 15 cm

        with unittest.mock.patch("__main__.mesurer_distance", mock_distance):
            self.assertEqual(mesurer_distance(), 15)
```

Exercice 8 : Simulation d'un capteur de courant

```
class CapteurCourant:
    def mesurer_ampere(self):
        """Retourne la mesure de l'ampérage"""
        pass

class TestCapteurCourant(unittest.TestCase):
    def test_mesurer_ampere(self):
        capteur = Mock()
        capteur.mesurer_ampere.return_value = 5 # Simuler un ampérage de 5A
        self.assertEqual(capteur.mesurer_ampere(), 5)
```

Exercice 9 : Mock d'un capteur infrarouge

```
class CapteurInfrarouge:
    def detecter_obstacle(self):
        """Retourne True si un obstacle est détecté, False sinon"""
        pass

class TestCapteurInfrarouge(unittest.TestCase):
    def test_detecter_obstacle(self):
        capteur = Mock()
        capteur.detecter_obstacle.return_value = True
        self.assertTrue(capteur.detecter_obstacle())
```

Exercice 10 : Simulation d'un capteur de couleur

```
class CapteurCouleur:  
    def detecter_couleur(self):  
        """Retourne la couleur détectée"""  
        pass  
  
class TestCapteurCouleur(unittest.TestCase):  
    def test_detecter_couleur(self):  
        capteur = Mock()  
        capteur.detecter_couleur.return_value = "Rouge"  
        self.assertEqual(capteur.detecter_couleur(), "Rouge")
```

Exercice 11 : Détection d'erreur avec un mock

```
class TestErreurCapteurCourant(unittest.TestCase):  
    def test_mesurer_ampere_exception(self):  
        capteur = Mock()  
        capteur.mesurer_ampere.side_effect = ValueError("Valeur incorrecte")  
  
        with self.assertRaises(ValueError):  
            capteur.mesurer_ampere()
```

Exercice 12 : Mock de l'ensemble des capteurs

```
class SystemeCapteurs:
    def __init__(self, capteur_infrarouge, capteur_courant, capteur_ultrasons,
    capteur_couleur):
        self.capteur_infrarouge = capteur_infrarouge
        self.capteur_courant = capteur_courant
        self.capteur_ultrasons = capteur_ultrasons
        self.capteur_couleur = capteur_couleur

    def verifier_systeme(self):
        return {
            "Obstacle": self.capteur_infrarouge.detecter_obstacle(),
            "Ampérage": self.capteur_courant.mesurer_ampere(),
            "Distance": self.capteur_ultrasons.mesurer_distance(),
            "Couleur": self.capteur_couleur.detecter_couleur()
        }

class TestSystemeCapteurs(unittest.TestCase):
    def test_verifier_systeme(self):
        capteur_infrarouge = Mock()
        capteur_courant = Mock()
        capteur_ultrasons = Mock()
        capteur_couleur = Mock()

        capteur_infrarouge.detecter_obstacle.return_value = True
        capteur_courant.mesurer_ampere.return_value = 3
        capteur_ultrasons.mesurer_distance.return_value = 20
        capteur_couleur.detecter_couleur.return_value = "Bleu"

        systeme = SystemeCapteurs(capteur_infrarouge, capteur_courant,
        capteur_ultrasons, capteur_couleur)
        resultats = systeme.verifier_systeme()

        self.assertEqual(resultats, {
            "Obstacle": True,
            "Ampérage": 3,
            "Distance": 20,
            "Couleur": "Bleu"
        })
```

BA2

Introduction à l'utilisation des mocks pour une voiture sur Raspberry Pi

Lorsqu'on programme une voiture autonome sur **Raspberry Pi**, on doit interagir avec de nombreux composants matériels comme :

- **Capteurs :**
 - **Capteurs infrarouges** (détection de ligne, obstacles proches).
 - **Capteurs ultrason** (mesure de distance).
 - **Capteurs de couleur** (suivi de ligne colorée, reconnaissance d'objets).
 - **Capteurs INA** (mesure de courant consommé par les moteurs).
- **Moteurs :**
 - **Moteurs DC** (propulsion).
 - **Servo-moteurs** (direction, levage d'objets, etc.).

Le problème est que ces composants **nécessitent du matériel branché** pour tester le programme.

C'est ici que les **Mocks (unittest.mock)** entrent en jeu !

Pourquoi utiliser des mocks ?

- Simuler les capteurs sans avoir le matériel branché.
- Tester le comportement des moteurs sans les faire tourner.
- Créer des scénarios contrôlés (ex: obstacle toujours détecté à 10 cm).
- Automatiser des tests unitaires sans risquer d'endommager le matériel.

Exemple 1 : Mock d'un capteur ultrason

Dans un programme réel, un capteur ultrason **renvoie une distance en centimètres**.

Problème :

Si on veut tester un programme qui freine à moins de **10 cm**, on doit **placer un objet devant le capteur** à chaque test... Pas pratique !

1. Code standard sans mock :

```
import RPi.GPIO as GPIO
import time

TRIG = 23
ECHO = 24

def mesurer_distance():
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(TRIG, GPIO.OUT)
    GPIO.setup(ECHO, GPIO.IN)
```

```

GPIO.output(TRIG, True)
time.sleep(0.00001)
GPIO.output(TRIG, False)

while GPIO.input(ECHO) == 0:
    start_time = time.time()

while GPIO.input(ECHO) == 1:
    stop_time = time.time()

distance = (stop_time - start_time) * 17150
return round(distance, 2)

```

Problème : Cette fonction ne peut pas être testée sans un vrai capteur !

2. Code avec unittest.mock

On va remplacer `mesurer_distance()` par un **mock** pour tester la réaction du programme.

```

from unittest.mock import patch
import unittest

def freiner_si_obstacle():
    distance = mesurer_distance()
    if distance < 10:
        return "STOP"
    return "AVANCE"

class TestVoiture(unittest.TestCase):
    @patch("__main__.mesurer_distance", return_value=5)
    def test_freinage_obstacle(self, mock_mesurer):
        self.assertEqual(freiner_si_obstacle(), "STOP")

    @patch("__main__.mesurer_distance", return_value=15)
    def test_pas_de_freinage(self, mock_mesurer):
        self.assertEqual(freiner_si_obstacle(), "AVANCE")

if __name__ == "__main__":
    unittest.main()

```

Explication :

- `@patch("__main__.mesurer_distance", return_value=5)` : simule un obstacle à **5 cm**.
- `freiner_si_obstacle()` utilise la version **mockée** de `mesurer_distance()`.
- Le test vérifie que la voiture **s'arrête bien ("STOP")** à moins de **10 cm**.

Exemple 2 : Mock d'un moteur DC

Problème :

Un moteur DC tourne grâce à une **commande PWM** envoyée sur un GPIO. Mais on ne peut **pas réellement tester** le moteur dans un test unitaire.

1. Code standard sans mock

```
import RPi.GPIO as GPIO

MOTEUR_PIN = 18
GPIO.setmode(GPIO.BCM)
GPIO.setup(MOTEUR_PIN, GPIO.OUT)
pwm = GPIO.PWM(MOTEUR_PIN, 100)

def demarrer_moteur(vitesse):
    pwm.start(vitesse)
```

Problème : Cette fonction **nécessite un Raspberry Pi avec un moteur branché**.

2. Code avec mock

On va **mock le moteur** et tester **les valeurs envoyées** :

```
from unittest.mock import patch
import unittest

def demarrer_moteur(vitesse):
    print(f"Moteur activé à {vitesse}%")

class TestMoteur(unittest.TestCase):
    @patch("builtins.print")
    def test_moteur_vitesse_max(self, mock_print):
        demarrer_moteur(100)
        mock_print.assert_called_with("Moteur activé à 100%")

    @patch("builtins.print")
    def test_moteur_vitesse_basse(self, mock_print):
        demarrer_moteur(20)
        mock_print.assert_called_with("Moteur activé à 20%")

if __name__ == "__main__":
    unittest.main()
```

Explication :

- `@patch("builtins.print")` **intercepte** les affichages `print()`.
- On vérifie que **la bonne commande** est envoyée au moteur **sans réellement l'exécuter**.

Exemple 3 : Mock d'un capteur de couleur

Problème :

Un capteur de couleur (comme le **TCS3200**) retourne une **fréquence spécifique** correspondant à une couleur.

Il est **impossible** de tester correctement sans une vraie source lumineuse.

1. Code standard sans mock

```
def detecter_couleur():
    return "Rouge" # En réalité, il renverrait une valeur RGB
```

Problème : On ne peut **pas tester différents scénarios sans modifier le programme**.

2. Code avec mock

On va **forcer des couleurs différentes** et tester les réactions.

```
from unittest.mock import patch
import unittest

def verifier_feu_tricolore():
    couleur = detecter_couleur()
    if couleur == "Rouge":
        return "STOP"
    elif couleur == "Vert":
        return "AVANCE"
    return "ATTENTE"

class TestCapteurCouleur(unittest.TestCase):
    @patch("__main__.detecter_couleur", return_value="Rouge")
    def test_stop_rouge(self, mock_couleur):
        self.assertEqual(verifier_feu_tricolore(), "STOP")

    @patch("__main__.detecter_couleur", return_value="Vert")
    def test_avancer_vert(self, mock_couleur):
        self.assertEqual(verifier_feu_tricolore(), "AVANCE")

    @patch("__main__.detecter_couleur", return_value="Orange")
    def test_attente_orange(self, mock_couleur):
        self.assertEqual(verifier_feu_tricolore(), "ATTENTE")

if __name__ == "__main__":
    unittest.main()
```

Explication :

- On **mocke detecter_couleur()** pour tester les **3 couleurs** (Rouge, Vert, Orange).
- **verifier_feu_tricolore()** renvoie "STOP", "AVANCE", ou "ATTENTE", **sans capteur branché**.

Conclusion

Capteur/Moteur	Testable avec unittest ?	Solution
Ultrason (distance)	Impossible sans objet physique	Mock de la distance
Infrarouge (détection de ligne)	Impossible sans une piste noire/blanche	Mock de la valeur renvoyée
Capteur de couleur	Impossible sans source lumineuse	Mock des couleurs détectées
Moteur DC	Impossible sans alimentation	Mock des commandes PWM
Servo-moteur	Impossible sans bras articulé	Mock de l'angle donné

En utilisant `unittest.mock`, on peut tester toutes les logiques du programme sans avoir besoin du matériel réel ! **Donc on peut arrêter de râler sur les piles !!!!!**