Unit Test

Unittest est un framework de test unitaire intégré à Python. Il permet de définir, d'organiser et d'exécuter des tests pour vérifier le bon fonctionnement des parties individuelles (unités) de votre code.

Voici quelques points clés à retenir sur unittest :

1. **Organisation des tests** : Les tests sont organisés dans des classes héritant de `unittest.TestCase`. Chaque méthode de cette classe qui commence par `test_` est considérée comme un test à exécuter.
2. **Assertions** : Unittest fournit une variété d'assertions prédéfinies pour vérifier différents aspects du comportement de votre code. Par exemple, `assertEqual`, `assertTrue`, `assertFalse`, etc.
3. **Méthodes de configuration** : Unittest fournit des méthodes spéciales de configuration pour exécuter du code avant et après l'exécution des tests, telles que `setUp()` et `tearDown()`.

Voici un exemple simple illustrant l'utilisation de unittest :

```
import unittest

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

class TestAddFunction(unittest.TestCase):

    def test_add_positive_numbers(self):
        result = add(1, 2)
        self.assertEqual(result, 3)

    def test_add_negative_numbers(self):
        result = add(-1, -1)
        self.assertEqual(result, -2)

    def test_add_zero(self):
        result = add(0, 0)
        self.assertEqual(result, 0)

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

Dans cet exemple :

- Nous avons défini une fonction add qui additionne deux nombres.
- Ensuite, nous avons créé une classe TestAddFunction qui hérite de unittest.TestCase et contient plusieurs méthodes de test commençant par test_.
- Chaque méthode de test exécute l'appel à la fonction add avec des arguments spécifiques et utilise une assertion pour vérifier le résultat.
- Enfin, nous avons utilisé unittest.main() pour exécuter les tests lorsque le script est exécuté directement.

Lorsque vous exécutez ce script, unittest exécute chaque méthode de test et signale si les assertions réussissent ou échouent.

```
# Tests unitaires
#
# Le test unitaire est une technique fondamentale dans le développement
# logiciel, visant à vérifier la fiabilité et la précision des composants
# individuels d'un programme. En Python, cette pratique est facilitée par
# l'utilisation de unittest, une bibliothèque intégrée qui tire ses racines
# de JUnit, un cadre de test pour Java, reflétant une approche standardisée
# dans de nombreux langages de programmation.
#
# unittest incarne l'idée que chaque module ou fonction du logiciel devrait
# être testé isolément, assurant ainsi leur fonctionnement correct avant leur
# intégration dans des systèmes plus complexes. Cette bibliothèque fournit
# une infrastructure permettant de définir des cas de test, des ensembles de
# tests, de les exécuter et d'en vérifier les résultats. Les composants clés à
# comprendre dans cette bibliothèque incluent les "test cases" (cas de test),
# qui sont les unités de base de test encapsulant les scénarios de test
# spécifiques, et les "test suites" (suites de tests), qui permettent de
# regrouper et d'exécuter plusieurs tests comme un ensemble cohérent.
#
# L'adoption de cette méthodologie permet non seulement de détecter et de
# corriger les erreurs dès les phases initiales du développement, mais elle
# encourage également la rédaction de code plus propre, plus modulaire et plus
# facile à maintenir. En outre, les tests unitaires servent de documentation
# vivante du comportement attendu des composants du système, facilitant ainsi
# la compréhension et la collaboration au sein des équipes de développement.
#
#
#
# La bibliothèque 'unittest' de Python est conçue autour de plusieurs
# principes fondamentaux, avec des conventions de nomenclature spécifiques
# pour certains éléments afin d'optimiser le processus de test. Voici une
# présentation détaillée et structurée de ces éléments clés :
#
# - Cas de test où 'TestCase' :
#   - Présentation : Un 'TestCase' est une classe qui regroupe plusieurs tests
#                    unitaires, permettant de tester différentes
#                    fonctionnalités d'un composant logiciel de manière
#                    structurée et isolée.
#   - Héritage : Les classes de cas de test doivent hériter de
#                'unittest.TestCase'.
#   - Nomenclature : Par convention et non par obligation, les noms des classes
#                    commencent souvent par 'Test', par exemple 'TestAbc'.
#
# - Méthodes de test :
#   - Présentation : Les méthodes de test, préfixées par test_ au sein d'un
#                    'TestCase', sont des fonctions spécifiques où s'effectuent
#                    les vérifications du comportement attendu du code. Elles
#                    permettent d'appliquer des assertions pour tester chaque
```

```
#                   aspect du composant logiciel en question.
#   - Identification : Doivent commencer par 'test_' pour que 'unittest' les
#                      reconnaisse comme des méthodes de test à exécuter.
#   - Exemple : Une méthode vérifiant l'addition pourrait s'appeler
#               'test_addition'.
#
# - Méthodes d'assertion :
#   - Assertion = déclaration ou affirmation considérée comme vraie!
#   - Vérification des résultats : Utilisées pour comparer les résultats
#                                  obtenus aux résultats attendus dans
#                                  les tests.
#   - Variété : Incluent plusieurs méthodes utilitaires pour réaliser les
#               tests. Voir plus bas pour la liste des méthodes disponible.
#
# - Considérations de bonne pratique :
#   - Généralement, lorsqu'on réalise des test unitaires, ces derniers sont
#     produits dans un fichier externe spécifique pour les tests.
#   - Il est important que les tests unitaires soient mis à jour et reflètent
#     l'évolution du code.
#
#
# Autres considérations plus avancées pas sujet au cours :
#
# - TestRunner :
#   - Exécution des tests : Gère l'exécution des suites de tests et la
#                           présentation des résultats.
#   - Personnalisation : Peut être configuré pour modifier le comportement
#                        d'exécution et de rapport des tests.
#
# - TestSuite :
#   - Regroupement des tests : Permet de combiner plusieurs cas de test ou
#                              méthodes de test en un seul ensemble exécutable.
#   - Organisation : Facilite l'exécution groupée et l'organisation logique des
#                    tests.
#




class ChuteLibre:
    """Modélise le mouvement d'un corps en chute libre sous l'effet de la gravité
terrestre."""

    GRAVITE: float = 9.81  # Accélération due à la gravité en m/s^2

    def __init__(self, hauteur_initiale_m: float, vitesse_initiale_m_s: float =
0.0):
        """
        Initialise un nouvel objet en chute libre.

        :param hauteur_initiale_m: Hauteur initiale en mètres au-dessus du sol.
        :param vitesse_initiale_m_s: Vitesse initiale en mètres par seconde.
        :raises ValueError: Si la hauteur initiale est négative.
        :raises TypeError: Si les types des paramètres ne sont pas des flottants.
```

```
        """
        if not isinstance(hauteur_initiale_m, float) or not
isinstance(vitesse_initiale_m_s, float):
            raise TypeError("La hauteur initiale et la vitesse initiale doivent
être des nombres flottants.")
        if hauteur_initiale_m < 0:
            raise ValueError("La hauteur initiale ne peut pas être négative.")

        self.hauteur_initiale_m : float= hauteur_initiale_m
        self.vitesse_initiale_m_s : float = vitesse_initiale_m_s

    def position_apres_temps_s(self, temps_s: float) -> float:
        """
        Calcule la position du corps après un certain temps en seconde.

        :param temps_s: Temps écoulé en seconde depuis le début de la chute.
        :return: Hauteur en mètres au-dessus du sol après le temps écoulé.
        :raises ValueError: Si le temps fourni est négatif.
        :raises TypeError: Si le type du paramètre temps n'est pas un flottant.
        """
        if not isinstance(temps_s, float):
            raise TypeError("Le temps doit être un nombre flottant.")
        if temps_s < 0:
            raise ValueError("Le temps ne peut pas être négatif.")

        position_m = self.hauteur_initiale_m + self.vitesse_initiale_m_s * temps_s
- 0.5 * self.GRAVITE * temps_s ** 2
        return max(position_m, 0)  # La position ne peut pas être négative; cela
indique que l'objet est au sol.




import unittest
# Les tests unitaire sont généralement inclus dans un autre fichier :
test_chute_libre.py
# from chute_libre import ChuteLibre
#
# Les méthodes spéciales `setUp` et `tearDown` permettent la configuration
# initiale et la finalisation après chaque test.
#
# La documentation exhaustive n'est pas vraiment produite en général mais
# l'est ici à titre d'explication introductive.

class TestChuteLibre(unittest.TestCase):
    """
    Suite de tests pour la classe ChuteLibre, vérifiant le comportement de la
    simulation de chute libre.

    Les tests incluent la vérification de l'initialisation correcte des
    instances, la précision des calculs de position après un temps donné et la
    gestion des entrées invalides ou des types incorrects.

    Méthodes:
```

```
    - setUp: Prépare l'environnement de test avant chaque méthode de test,
      créant une instance de ChuteLibre.
    - tearDown: Nettoie l'environnement de test après chaque méthode de test,
      garantissant l'indépendance des tests.
    - test_initialisation: Vérifie que l'initialisation avec des paramètres
      invalides lève les exceptions appropriées.
    - test_position_apres_temps_s: Confirme que la méthode de calcul de la
      position retourne des résultats attendus pour des entrées valides et gère
      correctement les types de données incorrects.
    - test_atterrissage: Assure que l'objet est considéré au sol après un temps
      suffisant, simulant l'atterrissage.

    Chaque méthode de test est conçue pour être indépendante, permettant
    l'exécution sélective et le débogage facilité.
    """

    def setUp(self):
        """Configuration avant chaque test.

        Crée une instance de ChuteLibre avec une hauteur initiale de 100.0
        mètres pour être utilisée dans les tests suivants.

        Considération technique:
        La méthode setUp est appelée avant l'exécution de chaque méthode de
        test, permettant de réinitialiser l'état avant chaque test pour éviter
        les dépendances entre eux.
        """
        self.chute = ChuteLibre(100.0)

    def tearDown(self):
        """Nettoyage après chaque test.

        Libère les ressources ou effectue toute autre opération de nettoyage
        nécessaire après chaque test.

        Ici, l'exemple est purement académique et le code produit n'est pas
        nécessaire puisque dans le subséquent appel de setUp, la variable
        'self.chute' sera réinitialisée à une nouvelle instance.

        Considération technique:
        La méthode tearDown est exécutée après chaque méthode de test,
        permettant de garantir qu'aucun état persistant ne fausse les
        résultats des tests suivants.
        """
        del self.chute

    def test_initialisation(self):
        """Teste l'initialisation de la classe ChuteLibre avec des valeurs
        valides et invalides.

        Objectif du test:
        - Vérifier que l'initialisation avec une hauteur négative lève une
ValueError.
        - Confirmer que l'initialisation avec un type incorrect pour la hauteur
```

```
lève une TypeError.

        Considération technique:
        Ce test utilise assertRaises pour vérifier que les exceptions attendues
        sont bien levées lors de conditions d'erreur spécifiques.
        """

        self.assertEqual(self.chute.hauteur_initiale_m, 100.0)
        self.assertEqual(self.chute.vitesse_initiale_m_s, 0.0)

        with self.assertRaises(ValueError):
            ChuteLibre(-10.0)
        with self.assertRaises(TypeError):
            ChuteLibre("100")

    def test_position_apres_temps_s(self):
        """Vérifie le calcul de la position après un certain temps.

        Objectif du test:
        - S'assurer que le calcul de la position est correct après 2 secondes.
        - Vérifier que fournir un type incorrect pour le temps lève une TypeError.

        Considération technique:
        Utilise assertAlmostEqual pour comparer les valeurs flottantes et
        assertRaises pour tester la gestion des types incorrects.
        """
        position = self.chute.position_apres_temps_s(2.0)
        self.assertAlmostEqual(position, 80.38, places=2)
        with self.assertRaises(TypeError):
            self.chute.position_apres_temps_s("2")

    def test_atterrissage(self):
        """Teste le cas où l'objet atteint le sol après un certain temps.

        Objectif du test:
        Confirmer que l'objet est considéré comme étant au sol (position 0) après
un temps suffisamment long.

        Considération technique:
        Le test vérifie que la position retournée est égale à 0 pour simuler
        l'atterrissage, en utilisant assertEqual.
        """
        position = self.chute.position_apres_temps_s(10.0)
        self.assertEqual(position, 0)



def main():
    unittest.main() # exécute un TestRunner préconfiguré de base


if __name__ == '__main__':
    main()
```

```
# Voici une liste plus ou moins complète des méthodes d'assertion fournies par
# 'unittest' :
#
# - assertEqual(a, b) : Confirme que 'a' est égal à 'b'.
# - assertNotEqual(a, b) : Vérifie que 'a' n'est pas égal à 'b'.
# - assertTrue(x) : Assure que 'x' est vrai.
# - assertFalse(x) : Assure que 'x' est faux.
# - assertIs(a, b) : Confirme que 'a' est 'b'.
# - assertIsNot(a, b) : Vérifie que 'a' n'est pas 'b'.
# - assertIsNone(x) : Assure que 'x' est 'None'.
# - assertIsNotNone(x) : Vérifie que 'x' n'est pas 'None'.
# - assertIn(a, b) : Confirme que 'a' se trouve dans 'b'.
# - assertNotIn(a, b) : Vérifie que 'a' n'est pas dans 'b'.
# - assertIsInstance(a, b) : Confirme que 'a' est une instance de 'b'.
# - assertNotIsInstance(a, b) : Vérifie que 'a' n'est pas une instance de 'b'.
# - assertAlmostEqual(a, b) : Vérifie que 'a' est presque égal à 'b'.
# - assertNotAlmostEqual(a, b) : Vérifie que 'a' n'est pas presque égal à 'b'.
# - assertGreater(a, b) : Assure que 'a' est plus grand que 'b'.
# - assertGreaterEqual(a, b) : Vérifie que 'a' est plus grand ou égal à 'b'.
# - assertLess(a, b) : Assure que 'a' est moins que 'b'.
# - assertLessEqual(a, b) : Vérifie que 'a' est moins ou égal à 'b'.
# - assertRegex(text, regex) : Vérifie que 'text' correspond à 'regex'.
# - assertNotRegex(text, regex) : Assure que 'text' ne correspond pas à 'regex'.
# - assertCountEqual(a, b) : Vérifie que 'a' et 'b' ont les mêmes éléments, sans
considération d'ordre.
# - assertRaises(exception) : Vérifie qu'une exception est levée.
# - assertWarns(warning) : Vérifie qu'un avertissement est émis.
# - assertLogs(logger, level) : Assure que des messages de log sont émis au niveau
spécifié.




#
#
#
# La documentation qui suit est complémentaire pour votre culture personnelle
# et n'est pas spécifiquement sujet au cours.
#
#
#
# Un `TestRunner` est un composant qui orchestre l'exécution des tests et est
# responsable de la collecte des résultats. Voici une description plus
# détaillée de ses fonctionnalités et de son utilisation :
#
# - Rôle principal : Le `TestRunner` prend en charge l'exécution des suites de
#                    tests définies dans les classes `TestCase`. Il parcourt
#                    chaque test, l'exécute et recueille les résultats, tels
#                    que les réussites, les échecs et les erreurs.
#
```

```
# - Résultats et rapports : Après l'exécution des tests, le `TestRunner`
#                           compile les résultats dans un format compréhensible,
#                           souvent sous forme de résumé indiquant le nombre de
#                           tests réussis et échoués. Certains `TestRunner`
#                           peuvent également générer des rapports plus
#                           détaillés, incluant des informations spécifiques sur
#                           les erreurs ou les échecs rencontrés.
#
# - Personnalisation : Python fournit un `TestRunner` par défaut, mais il est
#                      possible de créer des `TestRunner` personnalisés pour
#                      répondre à des besoins spécifiques. Par exemple, un projet
#                      pourrait nécessiter un format de rapport particulier ou
#                      intégrer l'exécution des tests dans des systèmes
#                      spécifiques (comme CI/CD - Intégration Continue/Livraison
#                      Continue).
#
# - Exemple d'utilisation : Bien que dans de nombreux cas, l'utilisation de la
#                           fonction `unittest.main()` suffise pour lancer les
#                           tests avec le `TestRunner` par défaut, il est
#                           également possible d'instancier et d'utiliser
#                           explicitement un `TestRunner`, comme dans l'exemple
#                           ci-dessous :
#
# import unittest
# from my_test_case import MyTestCase
#
# # Création de la suite de tests
# suite = unittest.TestLoader().loadTestsFromTestCase(MyTestCase)
#
# # Instanciation et utilisation d'un TestRunner
# runner = unittest.TextTestRunner(verbosity=2)
# runner.run(suite)
#
# - Extensions et outils tiers : Il existe des extensions et des outils tiers qui
#                                étendent les capacités des `TestRunner` de
`unittest`,
#                                offrant des fonctionnalités avancées telles que
#                                l'intégration avec des frameworks spécifiques, la
#                                génération de rapports HTML, ou la prise en
charge de
#                                tests parallèles.
#
#
#
#
#
#
#
# Une suite de test, `TestSuite`, est une collection de cas de test, ou de
# suites de tests, qui peuvent être exécutées ensemble. Cette abstraction
# permet de regrouper et d'organiser les tests de manière logique, facilitant
# leur exécution sélective et la génération de rapports consolidés.
# Voici une description sommaire des `TestSuite` :
#
```

```
# Composition et utilisation
# - Regroupement : Une `TestSuite` peut contenir des instances individuelles
#                  de `TestCase` ou d'autres `TestSuite`, permettant une
#                  hiérarchie flexible et une organisation modulaire des tests.
# - Flexibilité : Les développeurs peuvent créer des suites de tests
#                 personnalisées pour regrouper des tests spécifiques selon des
#                 critères tels que les fonctionnalités testées, les conditions
#                 préalables nécessaires, ou la séparation des tests rapides et
#                 lents.
#
# Création d'une TestSuite
# - Manuellement : Les développeurs peuvent créer des instances de `TestSuite`
#                  et y ajouter des cas de test ou d'autres suites de tests en
#                  utilisant la méthode `addTest()` ou `addTests()` pour une
#                  liste de tests.
# - Automatiquement : La méthode `unittest.TestLoader().loadTestsFromTestCase()`
#                     permet de charger automatiquement tous les tests d'une
#                     classe `TestCase` donnée dans une `TestSuite`.
#
# Avantages
# - Organisation : Les `TestSuite` permettent une structuration claire et
#                  logique des tests, améliorant la maintenabilité et la
#                  lisibilité.
# - Sélectivité : Elles offrent la possibilité d'exécuter un sous-ensemble
#                 spécifique de tests, ce qui est particulièrement utile dans
#                 de grands projets avec de nombreux tests.
# - Réutilisabilité : Les développeurs peuvent réutiliser des suites de tests
#                     dans différents contextes, comme différentes
#                     configurations d'environnement ou différents points
#                     d'entrée de test.
#
# Exécution d'une TestSuite
# - Pour exécuter une `TestSuite`, les développeurs utilisent un `TestRunner`.
#   Le `TestRunner` parcourt chaque test ou sous-suite inclus dans la
#   `TestSuite` et exécute les tests individuellement, en collectant et en
#   rapportant les résultats.
#
# Exemple de création et d'exécution d'une TestSuite
#
# import unittest
#
# class MonPremierTestCase(unittest.TestCase):
#     def test_un(self):
#         ...
#
# class MonSecondTestCase(unittest.TestCase):
#     def test_deux(self):
#         ...
#
# # Création d'une TestSuite
# suite = unittest.TestSuite()
#
# # Ajout des tests
# suite.addTest(MonPremierTestCase('test_un'))
```

```
# suite.addTest(MonSecondTestCase('test_deux'))
#
# # Exécution de la TestSuite
# runner = unittest.TextTestRunner()
# runner.run(suite)
#
```