Programmation Orientée Objet (POO) en Python

La programmation orientée objet (POO) est un paradigme de programmation largement utilisé dans lequel les programmes sont structurés autour d'objets qui représentent des entités réelles ou conceptuelles. En Python, la POO est mise en œuvre à l'aide de classes et d'objets.

Classes et Objets

- **Classe** : Une classe est un modèle pour créer des objets. Elle définit les propriétés et les comportements que les objets auront. Par exemple, une classe Animal pourrait définir les propriétés comme nom et âge, ainsi que des comportements comme manger et dormir.
- **Objet** : Un objet est une instance d'une classe. Il représente une occurrence spécifique de cette classe et possède ses propres valeurs pour les attributs de la classe. Par exemple, un objet Chien pourrait être une instance de la classe Animal, avec un nom spécifique comme "Rex" et un âge spécifique comme 3 ans.

Encapsulation, Héritage et Polymorphisme

- **Encapsulation** : L'encapsulation est le principe de regrouper les données (attributs) et les méthodes (fonctions) qui les manipulent dans une seule unité, c'est-à-dire la classe. Cela permet de cacher les détails d'implémentation et de protéger les données sensibles.
- **Héritage** : L'héritage permet à une classe (appelée classe enfant ou sous-classe) de hériter des attributs et des méthodes d'une autre classe (appelée classe parente ou super-classe). Cela favorise la réutilisabilité du code et permet de créer des hiérarchies de classes.
- **Polymorphisme** : Le polymorphisme permet à des objets de différentes classes de répondre de manière similaire à des messages ou à des appels de méthode. Cela permet d'écrire du code générique qui peut fonctionner avec différentes types d'objets

Exemple :

```
class Animal:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def manger(self):
        print(f"{self.nom} mange.")

class Chien(Animal):
    def aboyer(self):
        print(f"{self.nom} aboie.")

chien1 = Chien("Rex", 5)
chien1.manger()  # Output: Rex mange.
chien1.aboyer()  # Output: Rex aboie.
```

Dans cet exemple, `Animal` est la classe parente avec une méthode `manger`, et `Chien` est une classe enfant qui hérite de `Animal`. `Chien` ajoute sa propre méthode `aboyer`.

Exemple d'héritage, getter et setter

```
class Animal:
    def __init__(self, nom, age):
        self._nom = nom  # Attribut protégé
        self._age = age

    # Getter pour l'attribut nom
    def get_nom(self):
        return self._nom

    # Setter pour l'attribut nom
    def set_nom(self, nom):
        self._nom = nom

    # Méthode commune à tous les animaux
    def faire_du_bruit(self):
        pass

# Classe enfant héritant de la classe Animal
class Chien(Animal):
    def __init__(self, nom, age, race):
        super().__init__(nom, age)
        self._race = race

    # Redéfinition de la méthode faire_du_bruit
    def faire_du_bruit(self):
        return "Woof !"

    # Getter pour l'attribut race
    def get_race(self):
        return self._race

    # Setter pour l'attribut race
    def set_race(self, race):
        self._race = race

# Création d'une instance de la classe Chien
chien1 = Chien("Rex", 5, "Labrador")

# Utilisation du getter pour l'attribut nom
print(chien1.get_nom())  # Output: Rex

# Utilisation du setter pour l'attribut nom
chien1.set_nom("Buddy")
print(chien1.get_nom())  # Output: Buddy

# Utilisation du getter pour l'attribut race
```

```
print(chien1.get_race())  # Output: Labrador

# Utilisation du setter pour l'attribut race
chien1.set_race("Golden Retriever")
print(chien1.get_race())  # Output: Golden Retriever

# Appel de la méthode faire_du_bruit
print(chien1.faire_du_bruit())  # Output: Woof !
```

Dans cet exemple, la classe `Animal` définit un attribut protégé `_nom`, ainsi que des getters et des setters pour cet attribut. La classe enfant `Chien` hérite de `Animal` et définit un attribut```_race, ainsi que ses propres getters et setters. Ensuite, nous créons une instance de` Chienappeléechien1`` et utilisons les méthodes getters et setters pour accéder et modifier les attributs.

Type Hinting avec Python

Le type hinting est une pratique consistant à annoter les types de données dans le code Python à l'aide de commentaires spéciaux ou de déclarations syntaxiques. Ces annotations ne sont pas nécessaires pour que le code fonctionne, mais elles permettent de spécifier les types attendus pour les variables, les arguments de fonction et les valeurs de retour.

Pourquoi utiliser le type hinting ?

1. **Documentation améliorée**: Les annotations de type fournissent une documentation claire sur les types de données attendus, ce qui facilite la compréhension du code pour les autres développeurs.
2. **Détection précoce des erreurs**: Les outils statiques tels que Mypy peuvent analyser les annotations de type et détecter les incohérences de type potentielles, ce qui permet de repérer les erreurs de manière précoce dans le processus de développement.
3. **Meilleure maintenance du code**: Les annotations de type rendent le code plus explicite et réduisent les risques de bugs, ce qui facilite la maintenance du code à long terme.

Utilisation de l'import *typing*:

Le module typing fournit des outils pour annoter les types de données de manière plus expressives. Voici comment l'utiliser :

```
from typing import List, Tuple, Dict, Union, Optional, Callable, Literal
```

- List, Tuple, Dict : Annoter des conteneurs comme des listes, des tuples et des dictionnaires.
- Union : Annoter des variables avec plusieurs types possibles.
- Optional : Annoter des variables qui peuvent être de type défini ou None.
- Callable : Annoter des fonctions avec des types de fonctions attendus comme arguments ou retours.
- Literal : Limiter les valeurs d'une variable à un ensemble prédéfini de valeurs littérales.
- Any : type spécial qui indique que la variable peut être de n'importe quel type.

```
from typing import Any

a: Any = 1
a = 3.14
a = "Bonjour"
```

En utilisant ces outils, vous pouvez rendre vos annotations de type plus expressives et plus précises, ce qui améliore la qualité et la maintenabilité de votre code.

Utilisation des Type Hints:

```
vit: float = 1.5  # Annotation de type
```

- Les annotations de type permettent de spécifier le type des variables, des arguments de fonction et des valeurs de retour.
- Ils sont facultatifs et n'affectent pas l'exécution du code.
- Même si vous n'initialisez pas la variable avec une valeur float, le code peut toujours être exécuté.

Syntaxe des Annotations de Type:

```
def f(v: int, w: str = '') -> None:  # Types hints pour les paramètres et le
retour
    ...
```

- Les annotations de type sont placées après les noms de variables, de paramètres ou de valeurs de retour, suivies de :.
- L'utilisation de None comme type de retour indique qu'aucune valeur n'est retournée.

Types de Base:

```
from types import NoneType

a: int = 1
a = 3.14
a = "Bonjour"

value: NoneType = None
```

- Les types de base tels que int, float, str, NoneType, etc., peuvent être annotés directement.

Containers:

```
from typing import List, Tuple, Dict

n1: List[int] = [1, 2, 3]
n2: Tuple[int, str, float] = (1, "allo", 3.14)
n3: Dict[str, int] = {"allo": 1}
```

- Pour les conteneurs comme les listes, tuples et dictionnaires, utilisez des annotations de type spécifiques (List, Tuple, Dict, etc.).

Annotations Avancées:

```
def sum_moy(values: list[float]) -> tuple[float, float]:
    ...
```

- Vous pouvez annoter des fonctions avec des paramètres et des valeurs de retour plus complexes.

Union Types:

```
from typing import Union

nombre: Union[int, float] = 3
```

- Les types union permettent à une variable d'accepter plusieurs types de données.

Optional Types:

```
from typing import Optional

value: Optional[int] = None
```

- Les types optionnels permettent à une variable d'être de type défini ou None.

Callable Types

```
from typing import Callable

def f(i: int, r: float) -> complex:
    ...

task: Callable[[int, float], complex] = f
```

- Vous pouvez annoter les fonctions avec des types de fonctions attendues comme arguments ou retours.

Literal Types

```
from typing import Literal

var: Literal['rouge', 'vert', 'bleu'] = 'rouge'
```

- Les types littéraux limitent les valeurs d'une variable à un ensemble prédéfini de valeurs littérales.

Forward Declaration

```
class Tower:
    def compare(other: 'Tower') -> bool:
        ...
```

- Utilisez des chaînes de caractères pour indiquer un type qui sera défini plus tard dans le code.

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)
#
```

Emballage et Déballage

L'emballage (packing) et le déballage (unpacking) sont des concepts en Python qui permettent de manipuler des collections de données telles que les listes, les tuples ou les dictionnaires de manière flexible. Voici une explication concise avec des exemples :

1. **Emballage (Packing)** : L'emballage consiste à regrouper plusieurs valeurs dans une seule collection. Cela se fait généralement avec des tuples, mais cela peut également se faire avec des listes ou des ensembles. Voici un exemple :

```
# Emballage dans un tuple
my_tuple = 1, 2, 3  # Les valeurs sont automatiquement regroupées dans un tuple
print(my_tuple)     # Output: (1, 2, 3)

# Emballage dans une liste
my_list = [1, 2, 3]  # Les valeurs sont regroupées dans une liste
print(my_list)       # Output: [1, 2, 3]
```

2. **Déballage (Unpacking)** : Le déballage consiste à extraire les valeurs d'une collection (comme un tuple ou une liste) et à les assigner à des variables individuelles. Voici un exemple :

```
# Déballage d'un tuple
my_tuple = (1, 2, 3)
a, b, c = my_tuple  # Les valeurs sont extraites du tuple et assignées à a, b et c
print(a, b, c)      # Output: 1 2 3

# Déballage d'une liste
my_list = [4, 5, 6]
x, y, z = my_list   # Les valeurs sont extraites de la liste et assignées à x, y
et z
print(x, y, z)      # Output: 4 5 6
```

L'emballage et le déballage sont des fonctionnalités très utiles en Python, car elles permettent de manipuler les données de manière concise et efficace. Elles sont couramment utilisées dans divers contextes, tels que les fonctions renvoyant plusieurs valeurs, les boucles et les opérations sur les collections de données.

Décorateurs en python

Les décorateurs sont des fonctions qui prennent une autre fonction ou méthode en argument et renvoient une nouvelle fonction ou méthode modifiée. En Python, les décorateurs sont largement utilisés pour ajouter des fonctionnalités à des fonctions ou des classes sans les modifier directement.

Voici quelques points clés sur les décorateurs en Python :

1. **Syntaxe des décorateurs** : En Python, les décorateurs sont définis en plaçant le symbole `@` suivi du nom du décorateur juste au-dessus de la fonction ou de la méthode à décorer. Par exemple :

```
@decorator
def my_function():
    pass
```

Cela équivaut à my_function = decorator(my_function).

2. **Fonctions comme objets de première classe** : En Python, les fonctions sont des objets de première classe, ce qui signifie qu'elles peuvent être assignées à des variables, passées comme arguments à d'autres fonctions et retournées comme valeurs à partir d'autres fonctions. Cela permet de manipuler facilement les fonctions dans les décorateurs.

3. **Décorateurs de fonctions et de méthodes** : Les décorateurs peuvent être utilisés pour modifier le comportement des fonctions et des méthodes. Ils peuvent être utilisés pour ajouter du logging, de la validation d'entrée, du caching, etc.

4. **Chainage de décorateurs** : Il est possible d'appliquer plusieurs décorateurs à une même fonction ou méthode en les empilant les uns au-dessus des autres avec la syntaxe `@decorator1`, `@decorator2`.

```
@decorator1
@decorator2
def my_function():
    pass
```

Cela équivaut à my_function = decorator1(decorator2(my_function)).

5. **Décorateurs de classe** : En plus des fonctions, les décorateurs peuvent également être utilisés pour décorer des classes. Les décorateurs de classe peuvent être utilisés pour ajouter des méthodes supplémentaires, des propriétés calculées, ou pour modifier le comportement de l'initialisation de la classe, entre autres choses.

6. **Utilisation de décorateurs prédéfinis** : Python fournit plusieurs décorateurs prédéfinis dans les modules standard tels que `functools`, `contextlib`, `abc`, etc. Par exemple, `functools.wraps` est souvent utilisé pour conserver les métadonnées des fonctions décorées.

7. **Création de vos propres décorateurs** : Vous pouvez créer vos propres décorateurs personnalisés en définissant une fonction qui prend une autre fonction en argument, modifie son comportement selon vos besoins, et renvoie la fonction modifiée.

Les décorateurs sont un outil puissant en Python, mais ils peuvent rendre le code difficile à comprendre s'ils sont mal utilisés. Il est donc important de les utiliser avec parcimonie et de les documenter clairement pour faciliter la compréhension du code par d'autres développeurs.

Décorateurs Fondamentaux

Les décorateurs sont des fonctions spéciales en Python qui permettent de modifier le comportement d'autres fonctions ou méthodes. Voici un aperçu des principaux décorateurs :

- `@staticmethod`: Ce décorateur est utilisé pour définir une méthode statique dans une classe. Une méthode statique peut être appelée à partir de la classe ou de l'instance de la classe, mais elle ne reçoit pas implicitement l'instance de la classe (le self).

> Ce décorateur est utilisé pour définir une méthode statique dans une classe. Les méthodes statiques sont des méthodes qui peuvent être appelées sur la classe elle-même plutôt que sur une instance de la classe. Lorsqu'un objet appelle une méthode statique, aucun paramètre d'instance (comme self) n'est passé automatiquement à la méthode.

- `@classmethod`: Ce décorateur est similaire à @staticmethod, mais il reçoit implicitement la classe elle-même en tant que premier argument plutôt que l'instance de la classe.

> Ce décorateur est utilisé pour définir une méthode de classe dans une classe. Les méthodes de classe prennent la classe elle-même comme premier argument, conventionnellement appelé cls, plutôt que l'instance de la classe. Les méthodes de classe sont souvent utilisées pour créer des instances de la classe ou pour effectuer des opérations qui affectent la classe dans son ensemble.

- `@abstractmethod`: Ce décorateur est utilisé pour déclarer une méthode comme étant une méthode abstraite dans une classe abstraite. Une classe est considérée comme abstraite si elle contient au moins une méthode abstraite. Les méthodes abstraites doivent être redéfinies dans les sous-classes concrètes.

> Ce décorateur est utilisé pour déclarer une méthode comme abstraite dans une classe abstraite. Une méthode abstraite est une méthode qui doit être redéfinie dans une sous-classe, mais qui n'a pas de définition dans la classe abstraite elle-même.

- `@override`: Ce n'est pas un décorateur standard en Python, mais il est souvent utilisé comme convention pour indiquer qu'une méthode redéfinit une méthode de sa classe mère. Ce décorateur n'est pas requis, mais il est souvent utile pour documenter le code et éviter les erreurs de frappe lors de la redéfinition de méthodes.

> Utilisé dans le contexte de la programmation orientée objet pour indiquer qu'une méthode dans une classe enfant redéfinit une méthode héritée de la classe parente.

Les décorateurs sont utilisés en plaçant simplement le nom du décorateur au-dessus de la fonction ou de la méthode à laquelle il s'applique. Voici un exemple d'utilisation des décorateurs :

```
from abc import ABC, abstractmethod

class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method")

    @classmethod
    def class_method(cls):
        print(f"This is a class method of {cls}")

class MyAbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

class ConcreteClass(MyAbstractClass):
    def abstract_method(self):
        print("Implementation of abstract_method")

    @staticmethod
    def static_method():
        print("This is a static method")

    @classmethod
    def class_method(cls):
        print(f"This is a class method of {cls}")

ConcreteClass.static_method()
ConcreteClass.class_method()
```

Dans cet exemple, ConcreteClass implémente abstract_method, et elle peut également appeler les méthodes statiques et de classe définies dans MyClass.

Notes sur les Docstrings en Python

Les docstrings en Python sont des chaînes de caractères placées immédiatement après la déclaration d'un module, d'une classe, d'une fonction ou d'une méthode. Elles servent à documenter le code et sont accessibles via l'attribut **doc** ou la fonction help().

Exemple :

```
def my_function():
    '''Demonstrates triple double quotes
    docstrings and does nothing really.'''

    return None
```

Pour consulter la documentation, on peut utiliser :

```
print(my_function.__doc__)
```

ou :

```
help(my_function)
```

Par convention, les docstrings sont encadrées de triple guillemets et devraient être concises, avec un style one-liner suivi d'une explication plus détaillée.

Exemple PEP257 :

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
```

La documentation joue un rôle crucial dans la compréhension du code. Par exemple, numpy est largement populaire en partie grâce à sa documentation claire et complète.

Principes à retenir :

- WORO (Write Once, Read Often) : Écrire la documentation une fois pour qu'elle puisse être lue plusieurs fois.
- Les docstrings permettent de générer des simili-tests unitaires avec la librairie doctest.

Exemple d'utilisation de doctest :

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Modèle de docstring :

```
'''
phrase qui décrit le plus simplement possible

détails.
exceptions possibles
exemple (doc test -> simili test unitaire)
'''
```

L'attribut **doc** est une variable interne d'une fonction, classe ou module contenant la docstring.

Plus d'info, provenant de **ChatGPT** :

Types d'annotations :

```
Avec l'introduction des annotations de type dans Python 3, il est possible
d'enrichir davantage les docstrings en spécifiant les types attendus pour les
paramètres et les valeurs de retour des fonctions.
```

Exemples d'utilisation :

```
Il est recommandé de garder les docstrings à jour et de les réviser chaque fois
que des modifications sont apportées au code correspondant. Cela garantit que la
documentation reste précise et utile.
```

Clarté et Concision :

```
Les docstrings doivent être claires, concises et faciles à comprendre. Elles
doivent fournir suffisamment d'informations pour permettre à l'utilisateur de
```

```
comprendre rapidement le but et le fonctionnement de la fonction, de la classe ou
du module documenté.
```

Utilisation de Triple Quotes :

```
Utilisez des triple quotes (''' ou """) pour encadrer les docstrings. Cela permet
d'inclure des sauts de ligne et de formatter le texte de manière plus lisible.
```

Description du But :

```
Décrivez brièvement le but de la fonction, de la classe ou du module documenté
dans la première ligne de la docstring. Utilisez un style one-liner pour cette
description.
```

Doctest

Doctest est un module intégré à la bibliothèque standard de Python qui permet de tester du code Python en utilisant des exemples de session interactifs que l'on trouve souvent dans la documentation. Voici une explication succincte de ce qu'est Doctest :

- **Objectif** : Doctest permet de vérifier que les exemples de code inclus dans la documentation d'un programme Python fonctionnent comme prévu. Il s'agit de tester le code de manière automatisée en utilisant les exemples fournis dans les docstrings.
- **Fonctionnement** : Pour utiliser Doctest, vous incluez des exemples de code Python dans les docstrings de vos fonctions, classes ou modules. Ces exemples de code ressemblent à des sessions interactives que vous auriez dans l'interpréteur Python. Doctest exécute ces exemples et compare les résultats obtenus avec les résultats attendus, tels qu'ils sont affichés dans les docstrings.
- **Avantages** : Doctest permet de s'assurer que les exemples de code fournis dans la documentation sont toujours valides et fonctionnent correctement. Cela aide à maintenir la documentation à jour et garantit que les utilisateurs peuvent faire confiance aux exemples fournis pour comprendre et utiliser le code.
- **Utilisation** : Pour exécuter les tests Doctest, vous pouvez utiliser la ligne de commande en exécutant le module Python avec l'option -m doctest ou vous pouvez intégrer les tests Doctest dans votre suite de tests existante en utilisant le module doctest dans le cadre de vos tests unitaires. En résumé, Doctest est un outil simple mais puissant pour tester du code Python en utilisant les exemples de session interactifs inclus dans la documentation, ce qui contribue à garantir la qualité et la précision de votre code et de votre documentation.

Exemples:

Supposons que vous avez une fonction add qui prend deux nombres en entrée et renvoie leur somme. Voici à quoi pourrait ressembler la docstring de cette fonction avec des exemples Doctest :

```
def add(a, b):
    """
    Cette fonction prend deux nombres en entrée et renvoie leur somme.

    Exemples :
    >>> add(1, 2)
    3
    >>> add(-1, 1)
    0
    >>> add(0.5, 0.5)
    1.0
    """
    return a + b
```

Dans cet exemple :

- La docstring de la fonction add contient des exemples d'appels à la fonction add avec différents arguments.
- Les résultats attendus sont également inclus dans les exemples.
- Lorsque vous exécutez Doctest sur cette fonction, il exécutera automatiquement les exemples de code inclus dans la docstring et vérifiera si les résultats correspondent aux résultats attendus.

Voici comment vous pouvez exécuter Doctest pour cette fonction :

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

L'exécution de cette partie de code vérifiera si les exemples inclus dans la docstring de la fonction add produisent les résultats attendus. Si tout se passe bien, aucun message ne sera affiché, ce qui signifie que tous les tests sont passés avec succès. Sinon, Doctest affichera des messages d'erreur indiquant les cas où les résultats obtenus ne correspondent pas aux résultats attendus.

Dunder Functions

Les "dunder functions" (méthodes spéciales en Python) sont des méthodes dont le nom est encadré par deux double underscores (d'où le terme "dunder", qui vient de "double underscore"). Ces méthodes spéciales sont utilisées pour définir des comportements spécifiques pour les objets de classe et sont invoquées implicitement dans certaines situations par l'interpréteur Python.

Voici quelques points importants à retenir sur les dunder functions :

1. **Syntaxe** : Les dunder functions ont un nom spécifique précédé et suivi de deux underscores, par exemple :

```
__init__
__repr__
__add__
```

2. **Fonctionnalité** : Chaque dunder function a une fonctionnalité spécifique qui lui est associée. Par exemple, `__init__` est utilisé pour initialiser un nouvel objet, `__repr__` pour obtenir une représentation sous forme de chaîne de caractères de l'objet, `__add__` pour définir l'addition entre deux objets, etc.

3. **Invocation** : Les dunder functions sont invoquées automatiquement dans des situations particulières. Par exemple, lorsque vous créez une nouvelle instance d'une classe en utilisant `obj = MaClasse()`, Python appelle automatiquement la méthode `__init__` pour initialiser l'objet.

4. **Surcharge** : Vous pouvez surcharger (redéfinir) les dunder functions dans vos propres classes pour modifier le comportement par défaut. Cela vous permet d'adapter le comportement de vos objets aux besoins spécifiques de votre application.

5. **Fonctionnalités avancées** : Les dunder functions permettent d'implémenter des fonctionnalités avancées telles que la surcharge d'opérateurs, la gestion du cycle de vie des objets, la conversion de types, l'itération sur des objets, etc.

Exemples

1. init et new :

Utilisez `__new__` pour la création d'un nouvel objet et `__init__` pour initialiser cet objet avec des valeurs par défaut. Veillez à bien comprendre la différence entre ces deux méthodes.

Exemple :

```
class MaClasse:
    def __new__(cls):
        print("__new__ est appelé")
        instance = super().__new__(cls)
        return instance

    def __init__(self):
        print("__init__ est appelé")

# Création d'une instance de MaClasse
objet = MaClasse()
```

Résultat:

```
__new__ est appelé
__init__ est appelé
```

Dans cet exemple, **new** est appelé en premier pour créer une nouvelle instance de la classe. Ensuite, **init** est appelé pour initialiser cette instance.

2. repr et str :

Définissez `__repr__` pour une représentation détaillée de l'objet, souvent utilisée pour le débogage. Utilisez `__str__` pour une représentation plus lisible destinée aux utilisateurs finaux.

Exemple :

```
class MaClasse:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"MaClasse({self.x}, {self.y})"

    def __str__(self):
        return f"({self.x}, {self.y})"

# Création d'une instance de MaClasse
objet = MaClasse(10, 20)

# Affichage de l'objet
print(repr(objet))  # Utilisation de __repr__
print(str(objet))   # Utilisation de __str__
```

Résultat :

```
MaClasse(10, 20)
(10, 20)
```

Dans cet exemple, **repr** est utilisé pour obtenir une représentation détaillée de l'objet, tandis que **str** est utilisé pour obtenir une représentation plus lisible de l'objet.

3. Conversion de types :

Utilisez `__int__`, `__float__`, `__complex__` pour définir la conversion de votre objet en types numériques. Cela rend votre objet plus flexible et compatible avec les opérations mathématiques standard.

Exemple `__int__` :

```
class MaClasse:
    def __init__(self, x):
        self.x = x

    def __int__(self):
        return int(self.x)

# Création d'une instance de MaClasse
objet = MaClasse(3.14)

# Conversion en entier en utilisant int()
entier = int(objet)

# Affichage du résultat
print(entier)  # Affiche : 3
```

Dans cet exemple, **int** est utilisé pour définir la conversion de l'objet en un entier en utilisant la fonction int().

Exemple `__float__` :

```
class MaClasse:
    def __init__(self, x):
        self.x = x

    def __float__(self):
        return float(self.x)

# Création d'une instance de MaClasse
objet = MaClasse(3)
```

```
# Conversion en float en utilisant float()
flottant = float(objet)

# Affichage du résultat
print(flottant)  # Affiche : 3.0
```

4. Comparaison :

Implémentez les méthodes de comparaison telles que `__eq__`, `__lt__`, `__gt__`, etc., pour permettre une comparaison logique entre vos objets. Cela est utile lors du tri ou de la recherche dans des collections d'objets.

Exemple `__eq__` (Égalité) :

```
class MaClasse:
    def __init__(self, x):
        self.x = x

    def __eq__(self, other):
        return self.x == other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(5)
objet2 = MaClasse(5)

# Comparaison d'égalité entre les deux objets
print(objet1 == objet2)  # Affiche : True
```

Dans cet exemple, **eq** est utilisé pour définir le comportement de l'opérateur d'égalité (==). Il retourne True si les attributs x des deux objets sont égaux, et False sinon.

Exemple `__lt__` (Infériorité) :

```
class MaClasse:
    def __init__(self, x):
        self.x = x

    def __lt__(self, other):
        return self.x < other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(3)
objet2 = MaClasse(5)

# Comparaison d'infériorité entre les deux objets
print(objet1 < objet2)  # Affiche : True
```

Exemple `__gt__` (Supériorité) :

```
class MaClasse:
    def __init__(self, x):
        self.x = x

    def __gt__(self, other):
        return self.x > other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(3)
objet2 = MaClasse(5)

# Comparaison de supériorité entre les deux objets
print(objet1 > objet2)  # Affiche : False
```

Dans cet exemple, `__gt__` est utilisé pour définir le comportement de l'opérateur de comparaison de supériorité (>). Il retourne True si l'attribut x de l'objet 1 est supérieur à celui de l'objet 2, et False sinon.

5. Itération :

Si votre objet représente une séquence ou une collection, implémentez `__iter__` et `__next__` pour le rendre itérable. Cela permet d'utiliser des boucles for et d'autres fonctionnalités d'itération.

Exemples :

```
class MaClasse:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.limit:
            self.current += 1
            return self.current
        else:
            raise StopIteration

# Création d'une instance de MaClasse
obj = MaClasse(5)

# Utilisation de l'objet dans une boucle for
for valeur in obj:
    print(valeur)
```

```
#Résultat au terminal :
1
2
3
4
5
```

Dans cet exemple, nous définissons une classe MaClasse avec les dunder functions `__iter__` et `__next__`. Lorsque la méthode `__iter__` est appelée, elle retourne l'objet lui-même (ce qui permet à l'objet d'être son propre itérateur). La méthode `__next__` définit la logique pour générer les éléments de l'itérateur. À chaque appel de `__next__`, l'objet génère la valeur suivante jusqu'à ce qu'elle atteigne la limite spécifiée. Lorsque la limite est atteinte, une exception StopIteration est levée, signalant à Python que l'itération est terminée.

L'utilisation de ces dunder functions permet à l'objet MaClasse d'être utilisé dans une boucle for ou avec d'autres structures d'itération en Python.

6. Gestion de la taille :

Implémentez `__len__` pour permettre à votre objet d'être utilisé avec la fonction len() et de fournir des informations sur sa taille ou sa longueur.

Exemples :

```
class MaListe:
    def __init__(self, elements):
        self.elements = elements

    def __len__(self):
        return len(self.elements)

# Création d'une instance de MaListe
liste = MaListe([1, 2, 3, 4, 5])

# Utilisation de len() sur l'instance
print(len(liste))  # Utilisation de __len__
```

Résultat:

```
5
```

Dans cet exemple, **len** est utilisé pour permettre l'utilisation de la fonction len() sur des instances de la classe MaListe, ce qui renvoie le nombre d'éléments dans la liste.

7. Cycle de vie de l'objet :

Utilisez `__del__` avec précaution, car il est appelé lorsque l'objet est supprimé de la mémoire. Évitez de vous appuyer sur cette méthode pour la gestion des ressources, car le moment de sa déclenchement n'est pas garanti.

Exemple :

```
class MaClasse:
    def __init__(self, nom):
        self.nom = nom

    def __del__(self):
        print(f"L'objet de la classe MaClasse avec le nom '{self.nom}' a été
détruit.")

# Création d'une instance de MaClasse
objet = MaClasse("MonObjet")

# Suppression explicite de l'objet
del objet
```

8. Autres méthodes spéciales :

Explorez d'autres méthodes spéciales en fonction des besoins spécifiques de votre objet. Par exemple, `__call__` pour rendre l'objet appelable comme une fonction, `__getitem__` pour permettre l'indexation de l'objet comme une liste, etc.

More :

1. `__name__` :

- La dunder function `__name__` est utilisée pour accéder au nom d'une fonction, d'une classe, d'une méthode ou d'un module en Python.
- Elle retourne le nom de l'objet sous forme de chaîne de caractères.
- Utile pour obtenir dynamiquement le nom de l'objet à des fins de documentation, de débogage ou d'introspection.

```
def ma_fonction():
    pass

print(ma_fonction.__name__)  # Affiche : ma_fonction
```

2. `__doc__` :

- La dunder function `__doc__` est utilisée pour accéder à la documentation d'une fonction, d'une classe, d'une méthode ou d'un module en Python.
- Elle retourne la docstring associée à l'objet sous forme de chaîne de caractères.
- Utile pour fournir des informations sur l'objet aux utilisateurs et aux développeurs.

```
def ma_fonction():
    '''Cette fonction ne fait rien de spécial.'''
    pass

print(ma_fonction.__doc__)  # Affiche : Cette fonction ne fait rien de spécial.
```

3. `__class__` :

- La dunder function `__class__` est utilisée pour accéder à la classe d'une instance en Python.
- Elle retourne la classe à laquelle l'instance appartient.
- Utile pour accéder aux attributs et méthodes de la classe à partir d'une instance.

```
class MaClasse:
    pass

objet = MaClasse()
print(objet.__class__)  # Affiche : <class '__main__.MaClasse'>
```

4. `__dict__` :

- La dunder function **dict** est utilisée pour accéder au dictionnaire d'attributs d'un objet en Python.
- Elle retourne un dictionnaire contenant les attributs de l'objet et leurs valeurs.
- Utile pour inspecter dynamiquement les attributs d'un objet et pour effectuer des opérations de manipulation d'attributs.

```
class MaClasse:
    def __init__(self, x, y):
        self.x = x
        self.y = y

objet = MaClasse(5, 10)
print(objet.__dict__)  # Affiche : {'x': 5, 'y': 10}
```