Cours : Attributs et Méthodes en Python

1. Introduction

En Python, une **classe** est un modèle pour créer des objets. Les objets créés à partir de cette classe peuvent avoir des caractéristiques (appelées **attributs**) et des actions (appelées **méthodes**).

Les **méthodes** et **attributs** peuvent être définis à trois niveaux différents :

- **Attributs et méthodes d'instance** : Ceux-ci sont propres à chaque objet créé à partir de la classe.
- **Attributs et méthodes de classe** : Ceux-ci sont partagés par tous les objets de la classe.
- **Méthodes statiques** : Celles-ci ne dépendent ni de l'instance, ni de la classe. Elles sont simplement liées à l'espace de nom de la classe.

2. Attributs et Méthodes d'Instance

Les **attributs d'instance** sont propres à chaque objet créé à partir de la classe. Chaque objet peut avoir une valeur différente pour ces attributs.

Les **méthodes d'instance** sont des fonctions qui opèrent sur une instance particulière de la classe. Elles doivent obligatoirement prendre self comme premier paramètre, qui représente l'instance de l'objet.

Exemple d'attributs et méthodes d'instance

```python
class Personne:
    def __init__(self, nom, age):  # Méthode constructeur
        self.nom = nom  # Attribut d'instance
        self.age = age  # Attribut d'instance

    def se_presenter(self):  # Méthode d'instance
        return f"Je m'appelle {self.nom} et j'ai {self.age} ans."

# Création d'une instance
personne1 = Personne("Alice", 30)
personne2 = Personne("Bob", 25)

# Appel de la méthode d'instance
print(personne1.se_presenter())  # "Je m'appelle Alice et j'ai 30 ans."
print(personne2.se_presenter())  # "Je m'appelle Bob et j'ai 25 ans."
```

Ici, chaque objet de type Personne a ses propres attributs nom et age.

3. Attributs et Méthodes de Classe

Les **attributs de classe** sont partagés par toutes les instances de la classe. Les **méthodes de classe** agissent sur la classe elle-même, et non sur une instance particulière. Elles prennent cls comme premier paramètre.

Exemple d'attributs et méthodes de classe

```
class CompteBancaire:
    taux_interet = 0.05  # Attribut de classe (partagé entre toutes les instances)

    def __init__(self, solde):
        self.solde = solde  # Attribut d'instance

    @classmethod
    def modifier_taux(cls, nouveau_taux):  # Méthode de classe
        cls.taux_interet = nouveau_taux

    def afficher_solde(self):
        return f"Le solde du compte est {self.solde} €."

# Création d'instances
compte1 = CompteBancaire(1000)
compte2 = CompteBancaire(2000)

# Affichage du taux d'intérêt de classe
print(compte1.taux_interet)  # 0.05
print(compte2.taux_interet)  # 0.05

# Modification du taux d'intérêt via la méthode de classe
CompteBancaire.modifier_taux(0.07)

# Vérification de la modification
print(compte1.taux_interet)  # 0.07
print(compte2.taux_interet)  # 0.07
```

Ici, taux_interet est un attribut de classe partagé entre toutes les instances. La méthode modifier_taux est une méthode de classe qui permet de modifier cet attribut.

4. Méthodes Statiques

Les **méthodes statiques** ne dépendent ni de l'instance ni de la classe. Elles ne prennent ni self ni cls comme paramètres. Elles sont utilisées pour des fonctions qui ne nécessitent pas d'interagir avec l'état de l'objet ou de la classe, mais qui sont liées à la classe de manière logique.

Exemple de méthode statique

```python
class MathOperations:
    @staticmethod
    def addition(a, b):  # Méthode statique
        return a + b

    @staticmethod
    def soustraction(a, b):  # Méthode statique
        return a - b

# Appel des méthodes statiques sans créer d'instance
print(MathOperations.addition(3, 4))  # 7
print(MathOperations.soustraction(10, 5))  # 5
```

Les méthodes statiques ne peuvent pas accéder ni modifier des attributs d'instance ou de classe. Elles sont généralement utilisées pour encapsuler des fonctions qui sont logiquement liées à la classe, mais qui ne nécessitent pas d'accéder à son état interne.

La différence principale entre **les méthodes statiques** et **les méthodes de classe** réside dans la manière dont elles interagissent avec la classe elle-même.

1. Méthodes statiques :

- **Indépendance** : Elles ne dépendent ni de l'instance de la classe (l'objet) ni de la classe elle-même.
- **Pas d'accès aux attributs ou méthodes de la classe** : Elles ne peuvent pas accéder ou modifier les attributs ou les méthodes de la classe ou de ses instances.
- **Utilisation** : Elles sont principalement utilisées pour des fonctions utilitaires ou des opérations qui ne nécessitent pas d'accès aux données de la classe ou des objets.

```python
class MathOperations:
    @staticmethod
    def addition(a, b):
        return a + b

# Utilisation sans avoir besoin de créer une instance de la classe
resultat = MathOperations.addition(3, 4)  # 7
print(resultat)
```

2. Méthodes de classe :

- **Dépend de la classe** : Elles dépendent de la classe, mais pas des instances (objets). La méthode de classe prend comme premier argument cls, qui fait référence à la classe elle-même.
- **Accès aux attributs de classe** : Elles peuvent accéder et modifier les attributs de la classe, mais pas les attributs spécifiques aux objets.
- **Utilisation** : Elles sont utilisées lorsque vous voulez que la méthode interagisse avec la classe (et ses attributs) plutôt qu'avec les instances.

```python
class CompteBancaire:
    taux_interet = 0.05  # Attribut de classe

    @classmethod
    def modifier_taux(cls, nouveau_taux):
        cls.taux_interet = nouveau_taux  # Modifie l'attribut de la classe

# Utilisation de la méthode de classe
CompteBancaire.modifier_taux(0.07)  # Modifie l'attribut de la classe
print(CompteBancaire.taux_interet)  # 0.07
```

Résumé des différences :

Caractéristique	Méthodes statiques	Méthodes de classe
Accès à la classe	Non, elles ne peuvent pas accéder à cls	Oui, elles peuvent accéder et modifier cls
Accès aux instances	Non, elles ne peuvent pas accéder à self	Non, elles n'ont pas accès aux instances
Utilisation typique	Fonctions utilitaires ou indépendantes	Manipulation d'attributs ou comportements partagés par toutes les instances
Premier argument	Aucun, pas de self ni cls	cls, représentant la classe

Les méthodes de classe sont souvent utilisées quand on veut que la méthode affecte des choses au niveau de la classe (comme un attribut de classe), tandis que les méthodes statiques sont plus comme des fonctions normales, mais organisées dans la classe pour des raisons de lisibilité et de structure.

Les fonctions magiques

Les "dunder functions" (ou "double underscore functions") en Python sont des méthodes spéciales qui commencent et se terminent par deux underscores (__). Elles permettent de personnaliser le comportement des objets d'une classe. Ces méthodes sont également appelées "magic methods" ou "special methods" et sont appelées implicitement par Python lorsqu'une opération particulière est effectuée sur un objet.

Voici une explication des méthodes __str__, __repr__, __eq__ et d'autres dunder functions courantes, avec des exemples :

1. __str__ : Représentation de l'objet sous forme de chaîne

La méthode __str__ est utilisée pour définir une représentation sous forme de chaîne de caractères d'un objet, qui sera renvoyée lors de l'utilisation de la fonction str() ou lors de l'affichage de l'objet avec print().

Exemple :

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

# Utilisation
p = Person("Alice", 30)
print(p)  # Alice, 30 years old
```

2. __repr__ : Représentation "officielle" de l'objet

La méthode __repr__ est utilisée pour définir une représentation officielle de l'objet qui devrait être détaillée et, idéalement, permettre de recréer l'objet. Elle est utilisée dans des contextes comme la fonction repr() ou lors de l'inspection interactive dans un shell Python.

Exemple :

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person('{self.name}', {self.age})"

# Utilisation
p = Person("Alice", 30)
print(repr(p))  # Person('Alice', 30)
```

3. __eq__ : Comparaison d'égalité

La méthode __eq__ est utilisée pour définir la comparaison d'égalité entre deux objets. Elle est appelée par l'opérateur ==.

Exemple :

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name and self.age == other.age
        return False

# Utilisation
p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2)  # True
print(p1 == p3)  # False
```

4. __lt__, __le__, __gt__, __ge__ : Comparaisons de grandeur

Ces méthodes permettent de définir le comportement des opérateurs de comparaison <, <=, >, et >=.

Exemple :

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __lt__(self, other):
        if isinstance(other, Person):
            return self.age < other.age
        return False

# Utilisation
p1 = Person("Alice", 30)
p2 = Person("Bob", 25)

print(p1 < p2)  # False
```

5. __add__ : Surcharge de l'opérateur d'addition

La méthode __add__ permet de définir l'opération d'addition (+) entre deux objets.

Exemple :

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

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        return NotImplemented

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

# Utilisation
p1 = Point(2, 3)
p2 = Point(4, 1)
p3 = p1 + p2
print(p3)  # Point(6, 4)
```

6. __len__ : Longueur d'un objet

La méthode __len__ permet de définir ce que signifie la "longueur" d'un objet, utilisée par la fonction len().

```python
class MyList:
    def __init__(self, items):
        self.items = items

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

# Utilisation
lst = MyList([1, 2, 3, 4])
print(len(lst))  # 4
```

7. __getitem__ et __setitem__ : Accès aux éléments

Ces méthodes permettent de définir le comportement des opérations d'accès (obj[index]) et de modification (obj[index] = value) pour un objet.

```python
class MyList:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        self.items[index] = value

# Utilisation
lst = MyList([1, 2, 3])
print(lst[1])  # 2
lst[1] = 5
print(lst[1])  # 5
```

8. __call__ : Rendre un objet callable

La méthode __call__ permet de rendre un objet callable, comme une fonction.

Exemple :

```python
class Adder:
    def __init__(self, value):
        self.value = value

    def __call__(self, x):
        return self.value + x

# Utilisation
add_five = Adder(5)
print(add_five(10))  # 15
```

1. Encapsulation

L'encapsulation est un principe fondamental de la POO qui consiste à regrouper les données (attributs) et les méthodes (fonctions) qui agissent sur ces données au sein d'une même unité appelée **classe**. Cela permet de protéger les données internes de l'objet en restreignant l'accès direct à ses attributs et en fournissant des méthodes pour interagir avec eux. Les attributs sont généralement rendus privés (en les préfixant par un double underscore __), et l'accès aux données se fait via des méthodes publiques (getters et setters).

Exemple :

```python
class Person:
    def __init__(self, name, age):
        self.__name = name  # Attribut privé
        self.__age = age    # Attribut privé

    # Getter pour le nom
    def get_name(self):
        return self.__name

    # Setter pour l'âge
    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age invalid")

    # Méthode publique
    def display_info(self):
        print(f"Name: {self.__name}, Age: {self.__age}")

# Création d'un objet
person = Person("Alice", 30)
person.display_info()  # Affiche les informations
person.set_age(35)     # Modifie l'âge via le setter
person.display_info()  # Affiche les nouvelles informations
```

Explication : Ici, __name et __age sont des attributs privés. L'accès et la modification de l'âge se font via la méthode set_age, assurant ainsi que seules des valeurs valides sont attribuées.

2. Héritage

L'héritage permet à une classe d'hériter des attributs et des méthodes d'une autre classe. Cela favorise la réutilisation du code et permet de créer des hiérarchies de classes. La classe qui hérite est appelée classe dérivée ou enfant, et la classe qui est héritée est appelée classe de base ou parent.

Exemple :

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Appel du constructeur de la classe parent
        self.breed = breed

    def speak(self):  # Redéfinition de la méthode speak
        print(f"{self.name} says Woof!")

# Création d'un objet Dog
dog = Dog("Buddy", "Golden Retriever")
dog.speak()  # Affiche: Buddy says Woof!
```

Explication : Dog hérite de la classe Animal. Le constructeur de Dog appelle celui de Animal via super(), et la méthode speak est redéfinie dans Dog pour afficher un message spécifique.

3. Polymorphisme

Le polymorphisme permet à des objets de différentes classes d'être traités comme des objets de la même classe de base. Il se manifeste principalement par le fait qu'une même méthode peut avoir des comportements différents en fonction de l'objet qui l'appelle. Il existe deux types de polymorphisme :

- **Polymorphisme de méthode** : une méthode ayant le même nom dans différentes classes, mais un comportement différent.
- **Polymorphisme d'objet** : lorsqu'un objet de sous-classe peut être traité comme un objet de la classe parente.

Exemple :

```python
class Cat(Animal):
    def speak(self):
        print(f"{self.name} says Meow!")

# Création de différents animaux
animals = [Dog("Buddy", "Golden Retriever"), Cat("Whiskers")]

# Polymorphisme : appel de la méthode speak pour différents objets
for animal in animals:
    animal.speak()  # Affiche : Buddy says Woof! et Whiskers says Meow!
```

Explication : Bien que Dog et Cat soient des objets de classes différentes, ils partagent une méthode speak qui est appelée de manière polymorphique. Le comportement de cette méthode dépend du type réel de l'objet (Dog ou Cat).

Exercices pratiques :

TP 1 : Gestion des informations d'une personne

Objectif : Appliquer les attributs et méthodes d'instance.

Exercice : Créez une classe Personne avec les attributs nom et age. Implémentez une méthode afficher_informations() qui affiche les informations de la personne sous forme de chaîne de caractères. Ensuite, créez deux objets Personne avec des valeurs différentes et affichez leurs informations.

TP 2 : Comptes bancaires et taux d'intérêt

Objectif : Appliquer les attributs et méthodes de classe.

Exercice : Créez une classe CompteBancaire qui a un attribut de classe taux_interet (initialement 0.05) et un attribut d'instance solde. Implémentez une méthode de classe modifier_taux qui permet de modifier le taux d'intérêt pour tous les comptes. Créez plusieurs instances de CompteBancaire et testez l'effet de la modification du taux d'intérêt sur toutes les instances.

TP 3 : Méthodes statiques pour calculs financiers

Objectif : Appliquer les méthodes statiques.

Exercice : Dans la classe CompteBancaire, ajoutez une méthode statique calculer_interet(solde) qui calcule l'intérêt d'un solde donné en fonction du taux d'intérêt actuel. Testez cette méthode sans créer d'instance de la classe.

TP 4 : Représentation d'objets avec __str__ et __repr__

Objectif : Appliquer les méthodes magiques pour personnaliser l'affichage.

Exercice : Dans la classe Personne, implémentez les méthodes __str__ et __repr__. La méthode __str__ doit afficher un message simple, comme "Nom: John, Âge: 30". La méthode __repr__ doit afficher quelque chose de plus détaillé, comme "Personne(nom='John', age=30)". Testez les deux méthodes avec différents objets.

TP 5 : Comparaison d'objets avec **__eq__**

Objectif : Appliquer la méthode magique __eq__.

Exercice : Créez une classe Rectangle avec des attributs longueur et largeur. Implémentez la méthode __eq__ pour comparer deux objets Rectangle en fonction de leur surface (longueur * largeur). Créez deux objets Rectangle et vérifiez si leur surface est égale.

TP 6 : Héritage et polymorphisme

Objectif : Appliquer l'héritage et le polymorphisme.

Exercice : Créez une classe Animal avec une méthode parler(). Ensuite, créez deux sous-classes : Chien et Chat, chacune redéfinissant la méthode parler() avec des messages différents ("Woof!" pour le chien et "Meow!" pour le chat). Créez une fonction faire_parler() qui accepte un objet Animal et appelle sa méthode parler(). Testez avec des instances de Chien et Chat.

TP 7 : Encapsulation avec getters et setters

Objectif : Appliquer l'encapsulation.

Exercice : Créez une classe Voiture avec un attribut privé __vitesse_max (vitesse maximale). Implémentez des méthodes get_vitesse_max() et set_vitesse_max() pour accéder et modifier la vitesse maximale. Testez ces méthodes en créant un objet Voiture.

Ces TP permettent de couvrir les concepts clés du cours et d'aider à maîtriser les fonctionnalités de la POO en Python.

TP Global : Système de Gestion d'une Bibliothèque

Objectif : Créer un système de gestion d'une bibliothèque en utilisant les concepts de la programmation orientée objet, y compris les attributs et méthodes d'instance, de classe et statiques, ainsi que l'encapsulation, l'héritage, le polymorphisme et les méthodes magiques.

Description du Système

La bibliothèque dispose de plusieurs types de livres, et chaque livre a un titre, un auteur, un nombre de pages et un statut (disponible ou emprunté). La bibliothèque permet aux utilisateurs de rechercher un livre, emprunter un livre, et de gérer les livres en fonction de leur statut.

Les classes à créer sont les suivantes :

1. Livre : Représente un livre de la bibliothèque.
2. Utilisateur : Représente un utilisateur qui emprunte un livre.
3. Bibliotheque : Représente la bibliothèque qui gère les livres.
4. LivreEmprunte : Une sous-classe de Livre pour gérer les livres empruntés.

Étapes du TP

1. Classe Livre

Cette classe doit avoir les attributs suivants :

- titre (chaîne de caractères)
- auteur (chaîne de caractères)
- nombre_pages (entier)
- statut (chaîne de caractères : "disponible" ou "emprunté")

Méthodes :

- __init__() : Constructeur pour initialiser les attributs du livre.
- emprunter() : Change le statut du livre à "emprunté".
- retourner() : Change le statut du livre à "disponible".
- __str__() : Retourne une chaîne qui décrit un livre.
- __repr__() : Retourne une représentation détaillée du livre.

2. Classe Utilisateur

Cette classe doit avoir les attributs suivants :

- nom (chaîne de caractères)
- livres_empruntes (liste des livres empruntés)

Méthodes :

- __init__() : Constructeur pour initialiser le nom et la liste des livres empruntés.
- emprunter_livre(livre) : Permet à un utilisateur d'emprunter un livre.
- retourner_livre(livre) : Permet à un utilisateur de retourner un livre.
- __str__() : Retourne une chaîne avec le nom de l'utilisateur et les titres des livres empruntés.
- __repr__() : Retourne une représentation détaillée de l'utilisateur.

3. Classe Bibliotheque

Cette classe doit avoir les attributs suivants :

- livres (liste de tous les livres disponibles dans la bibliothèque)

Méthodes :

- __init__() : Constructeur pour initialiser la liste des livres.
- ajouter_livre(livre) : Ajoute un livre à la bibliothèque.
- rechercher_livre(titre) : Recherche un livre par son titre.
- afficher_livres_disponibles() : Affiche tous les livres qui sont disponibles à l'emprunt.
- afficher_livres_empruntes() : Affiche tous les livres qui ont été empruntés.

4. Classe LivreEmprunte

Cette classe doit hériter de Livre et ajouter l'attribut :

- date_emprunt (la date à laquelle le livre a été emprunté)

Méthodes :

- __init__() : Constructeur qui initialise le livre emprunté avec la date d'emprunt.
- __str__() : Redéfinir pour afficher les informations d'un livre emprunté, y compris la date d'emprunt.

5. Méthodes supplémentaires

Méthodes statiques :

- afficher_total_livres_empruntes() (dans Bibliotheque) : Retourne le nombre total de livres empruntés.

6. Exemple d'implémentation

```
import datetime


class Livre:
    def __init__(self, titre, auteur, nombre_pages):
        self.titre = titre
        self.auteur = auteur
        self.nombre_pages = nombre_pages
        self.statut = "disponible"

    def emprunter(self):
        if self.statut == "disponible":
            self.statut = "emprunté"
        else:
            print(f"Le livre '{self.titre}' est déjà emprunté.")

    def retourner(self):
        if self.statut == "emprunté":
```

```
            self.statut = "disponible"
        else:
            print(f"Le livre '{self.titre}' n'est pas emprunté.")

    def __str__(self):
        return f"{self.titre} de {self.auteur} ({self.nombre_pages} pages)"

    def __repr__(self):
        return f"Livre(titre='{self.titre}', auteur='{self.auteur}', pages={self.nombre_pages},
statut='{self.statut}')"

class Utilisateur:
    def __init__(self, nom):
        self.nom = nom
        self.livres_empruntes = []

    def emprunter_livre(self, livre):
        if livre.statut == "disponible":
            livre.emprunter()
            self.livres_empruntes.append(livre)
        else:
            print(f"Le livre '{livre.titre}' est déjà emprunté.")

    def retourner_livre(self, livre):
        if livre in self.livres_empruntes:
            livre.retourner()
            self.livres_empruntes.remove(livre)
        else:
            print(f"Vous n'avez pas emprunté le livre '{livre.titre}'.")
```

```
    def __str__(self):
        livres = [livre.titre for livre in self.livres_empruntes]
        return f"{self.nom} a emprunté: {', '.join(livres)}"

    def __repr__(self):
        return f"Utilisateur(nom='{self.nom}', livres={self.livres_empruntes})"

class Bibliotheque:
    def __init__(self):
        self.livres = []

    def ajouter_livre(self, livre):
        self.livres.append(livre)

    def rechercher_livre(self, titre):
        for livre in self.livres:
            if livre.titre == titre:
                return livre
        return None

    def afficher_livres_disponibles(self):
        return [livre for livre in self.livres if livre.statut == "disponible"]

    def afficher_livres_empruntes(self):
        return [livre for livre in self.livres if livre.statut == "emprunté"]

    @staticmethod
```

```
    def afficher_total_livres_empruntes():
        return sum(1 for livre in bibliotheque.afficher_livres_empruntes() if livre.statut == "emprunté")


# Test du système
bibliotheque = Bibliotheque()
livre1 = Livre("Python pour débutants", "John Doe", 200)
livre2 = Livre("Apprendre Django", "Jane Smith", 150)


bibliotheque.ajouter_livre(livre1)
bibliotheque.ajouter_livre(livre2)


utilisateur = Utilisateur("Alice")
utilisateur.emprunter_livre(livre1)
utilisateur.retourner_livre(livre1)
```

Explications :

1. **Attributs et Méthodes d'Instance** : Utilisez des méthodes comme emprunter() et retourner() qui modifient l'état de chaque objet individuellement.
2. **Attributs et Méthodes de Classe** : Implémentez la méthode afficher_total_livres_empruntes() qui est une méthode statique pour compter les livres empruntés dans la bibliothèque.
3. **Encapsulation** : Les livres ont un attribut statut privé que vous modifiez via les méthodes d'instance.
4. **Héritage** : La classe LivreEmprunte hérite de Livre et ajoute une fonctionnalité supplémentaire liée à la gestion des emprunts.
5. **Polymorphisme** : Les objets de type Livre et LivreEmprunte peuvent être traités de manière polymorphique grâce à l'héritage.
6. **Méthodes Magiques** : Redéfinissez les méthodes __str__() et __repr__() pour les classes Livre et Utilisateur afin de personnaliser l'affichage des objets.

TP Complet et Détaillé : Gestion d'un Système de Réservations d'Hôtels

Objectif : Créer un système de gestion pour un hôtel en utilisant les concepts de la programmation orientée objet en Python. Ce TP inclut la création, la gestion, et l'utilisation de classes, ainsi que les notions de méthode d'instance, méthode statique, encapsulation, héritage, polymorphisme, et méthodes magiques.

Contexte du Problème

Un hôtel dispose de plusieurs chambres. Chaque chambre peut être réservée par un client pour une ou plusieurs nuits. Les clients peuvent :

- Consulter les chambres disponibles.
- Réserver une chambre.
- Annuler une réservation.

L'hôtel doit également pouvoir :

- Gérer ses chambres (ajouter, afficher les chambres disponibles, etc.).
- Consulter les réservations faites par des clients.

Étapes de Réalisation

1. Classe Chambre

Cette classe représente une chambre de l'hôtel.

Attributs :

- numero (int) : numéro de la chambre.
- type_chambre (str) : type de la chambre (simple, double, ou suite).
- prix (float) : prix par nuit de la chambre.
- disponible (bool) : indique si la chambre est disponible.

Méthodes :

- __init__() : Initialise les attributs de la chambre.
- changer_statut() : Change la disponibilité de la chambre (True -> False ou inversement).
- __str__() : Retourne une description lisible de la chambre.
- __repr__() : Retourne une représentation détaillée de l'objet.

---

2. Classe Client

Cette classe représente un client de l'hôtel.

Attributs :

- nom (str) : nom du client.
- email (str) : email du client.
- reservations (list) : liste des chambres réservées par le client.

Méthodes :

- __init__() : Initialise les attributs du client.
- ajouter_reservation(chambre) : Ajoute une réservation pour une chambre.
- annuler_reservation(chambre) : Annule une réservation pour une chambre.
- __str__() : Affiche les informations du client et ses réservations.

---

3. Classe Hotel

Cette classe représente l'hôtel lui-même.

Attributs :

- nom (str) : nom de l'hôtel.
- adresse (str) : adresse de l'hôtel.
- chambres (list) : liste des chambres de l'hôtel.

Méthodes :

- __init__() : Initialise les attributs de l'hôtel.
- ajouter_chambre(chambre) : Ajoute une nouvelle chambre à la liste.
- afficher_chambres_disponibles() : Retourne les chambres disponibles.
- rechercher_chambre_par_type(type_chambre) : Retourne toutes les chambres d'un type spécifique.
- afficher_reservations() : Affiche toutes les chambres réservées.

---

4. Classe Reservation

Une classe dédiée pour gérer les réservations de manière précise.

Attributs :

- client (Client) : Le client qui fait la réservation.
- chambre (Chambre) : La chambre réservée.
- date_debut (str) : Date de début de la réservation.
- duree (int) : Durée de la réservation en nuits.

Méthodes :

- __init__() : Initialise les informations de la réservation.
- calculer_prix_total() : Calcule le prix total de la réservation en fonction du prix de la chambre et de la durée.
- __str__() : Affiche les détails de la réservation.
- __repr__() : Fournit une représentation détaillée de la réservation.

Fonctionnalités Demandées

1. **Ajout et gestion des chambres** :
    - Ajouter des chambres à l'hôtel.
    - Afficher toutes les chambres disponibles ou les chambres d'un type spécifique.
2. **Réservations** :
    - Permettre à un client de réserver une chambre.
    - Calculer le coût total d'une réservation.
    - Annuler une réservation.
3. **Affichage des données** :
    - Afficher les informations d'un client, y compris ses réservations.
    - Afficher toutes les réservations dans l'hôtel.
4. **Gestion avancée** :
    - Ajouter des méthodes statiques pour calculer le revenu total généré par les réservations.