Cours de Programmation Orientée Objet (POO) en Python

Plan du Cours :

1. **Introduction à la Programmation Orientée Objet**
    - Définition de la POO
    - Concepts clés : Classes, Objets, Attributs, Méthodes
    - Principes fondamentaux de la POO : Encapsulation, Héritage, Polymorphisme
2. **Exemple pratique 1 : Approche inspirée des jeux vidéo**
    - Création d'une classe `Personnage`
    - Attributs, méthodes, et objets liés aux personnages
    - Interaction entre plusieurs personnages
3. **Exemple pratique 2 : Approche inspirée de la vie réelle**
    - Création d'une classe `Voiture`
    - Gestion de plusieurs voitures avec différents comportements
    - Simulation d'interactions entre objets (voitures, conducteurs)
4. **Conclusion et récapitulation**

1. Introduction à la Programmation Orientée Objet (POO) en Python

Qu'est-ce que la POO ?

La Programmation Orientée Objet (POO) est un paradigme qui permet d'organiser et structurer le code de manière modulaire en utilisant des objets. Dans la POO, le code est découpé en **classes**, et chaque classe est utilisée pour créer des **objets**. Ces objets représentent des entités avec des **attributs** (des données) et des **méthodes** (des fonctions associées aux objets) qui définissent leur comportement.

Concepts Clés de la POO en Python :

1. Classe

Une **classe** est un modèle ou un plan de construction qui définit la structure et le comportement des objets. En d'autres termes, une classe sert de "gabarit" pour créer des objets. Une fois qu'une classe est définie, vous pouvez créer plusieurs objets basés sur cette classe. Chaque objet créé à partir de cette classe est appelé une **instance**.

- **Définir une classe** : En Python, une classe est définie à l'aide du mot-clé `class`.

```
class Personne:
    pass  # Une classe vide pour l'instant
```

Dans l'exemple ci-dessus, nous définissons une classe `Personne`, mais elle ne contient encore ni attributs ni méthodes. C'est une coquille vide qui servira de base pour créer des objets.

2. Objet

Un **objet** est une instance d'une classe. Une fois qu'une classe est définie, vous pouvez créer des objets en appelant la classe comme une fonction.

- **Créer un objet** : Une instance est créée en appelant la classe.

```
individu = Personne()  # Création d'un objet de la classe Personne
```

Dans cet exemple, `individu` est un objet (ou une instance) de la classe `Personne`. Chaque objet peut avoir ses propres caractéristiques et comportements définis dans la classe.

3. Attributs

Les **attributs** sont des variables définies au sein d'une classe et représentent les caractéristiques des objets. Ils sont définis dans une méthode spéciale appelée `__init__`, qui est appelée automatiquement lors de la création d'un objet. Cette méthode est aussi appelée le **constructeur**.

- **Ajouter des attributs à une classe** :

```
class Personne:
    def __init__(self, nom, age):
```

```
        self.nom = nom  # Attribut 'nom' de l'objet
        self.age = age  # Attribut 'age' de l'objet
```

Dans cet exemple, la méthode `__init__` initialise les objets de la classe `Personne` avec deux attributs : `nom` et `age`. Le mot-clé `self` fait référence à l'instance actuelle de la classe, ce qui permet de différencier les attributs de chaque objet.

- **Créer des objets avec des attributs** :

```
individu1 = Personne("Alice", 30)
individu2 = Personne("Bob", 25)

print(individu1.nom)  # Affiche 'Alice'
print(individu2.age)  # Affiche '25'
```

Dans cet exemple, `individu1` et `individu2` sont deux objets distincts, chacun ayant son propre nom et âge.

4. Méthodes

Les **méthodes** sont des fonctions définies à l'intérieur d'une classe, qui définissent les actions ou comportements que les objets peuvent effectuer. Les méthodes peuvent utiliser et manipuler les attributs de l'objet.

- **Définir des méthodes** :

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

    def se_presenter(self):
        return f"Bonjour, je m'appelle {self.nom} et j'ai {self.age} ans."
```

Dans cet exemple, la méthode `se_presenter` retourne une phrase présentant le nom et l'âge de l'objet. L'appel de cette méthode sur un objet permet de voir ces informations.

- **Utiliser une méthode sur un objet** :

```
individu1 = Personne("Alice", 30)
print(individu1.se_presenter())  # Affiche : "Bonjour, je m'appelle Alice
et j'ai 30 ans."
```

Chaque objet peut utiliser cette méthode pour effectuer une action spécifique.

Principes fondamentaux de la POO

1. **Encapsulation**
    - **Définition :** L'encapsulation consiste à regrouper les données (attributs) et les méthodes (fonctions) au sein d'une même classe. Cela permet de protéger les données d'un objet en limitant l'accès à ces données depuis l'extérieur. En Python, on peut

indiquer qu'un attribut est privé (et donc accessible uniquement à l'intérieur de la classe) en commençant son nom par deux underscores (__).

- **Exemple d'encapsulation** :

```
class Personne:
    def __init__(self, nom, age):
        self.__nom = nom  # Attribut privé
        self.__age = age  # Attribut privé

    def afficher_nom(self):
        return self.__nom
```

Ici, `__nom` et `__age` sont des attributs privés et ne sont pas accessibles directement depuis l'extérieur de la classe. Pour accéder au nom, il faut utiliser la méthode `afficher_nom`.

2. **Héritage**

- **Définition :** L'héritage permet à une classe (classe dérivée ou enfant) d'hériter des attributs et des méthodes d'une autre classe (classe de base ou parent). Cela permet de réutiliser du code et d'ajouter ou de modifier les comportements dans les sous-classes.
- **Exemple d'héritage** :

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

    def faire_bruit(self):
        pass

class Chien(Animal):
    def faire_bruit(self):
        return "Woof!"
```

Ici, la classe `Chien` hérite de la classe `Animal`. Elle réutilise l'attribut `nom` et redéfinit la méthode `faire_bruit` pour spécifier un comportement propre au chien.

3. **Polymorphisme**

- **Définition :** Le polymorphisme permet à des objets de différentes classes d'être manipulés de manière identique. En POO, cela signifie qu'une méthode ou une fonction peut être définie dans plusieurs classes et se comporter différemment selon la classe.
- **Exemple de polymorphisme** :

```
class Chat(Animal):
    def faire_bruit(self):
        return "Miaou!"

animaux = [Chien("Rex"), Chat("Whiskers")]
```

```
for animal in animaux:
    print(animal.faire_bruit())
```

Ici, `faire_bruit` est appelé de la même manière sur des objets de types différents (`Chien` et `Chat`), mais les résultats sont spécifiques à chaque classe (`Woof!` pour `Chien` et `Miaou!` pour `Chat`).

Résumé de l'introduction à la POO en Python

- **Classe** : Gabarit pour créer des objets.
- **Objet** : Instance d'une classe.
- **Attributs** : Données spécifiques à chaque objet.
- **Méthodes** : Fonctions associées aux objets, définissant leurs comportements.
- **Encapsulation** : Regrouper les données et méthodes dans une classe pour protéger les données.
- **Héritage** : Une classe peut hériter des attributs et méthodes d'une autre.
- **Polymorphisme** : Permet de manipuler différents types d'objets de manière uniforme.

Objectif atteint : Vous avez maintenant une vue d'ensemble des concepts essentiels de la POO en Python et êtes prêt à les appliquer dans des exemples pratiques.

Cette introduction pose une base solide pour passer à des exemples plus concrets et comprendre comment la POO structure le code et favorise la réutilisation.