Programmation Orienté Object en Python

Dr. ABOUABID Hamza

EMSI - Casablanca

5 mars 2025

Introduction à la Programmation Orientée Objet

- **Définition de la POO :** La programmation orientée objet (POO) est un paradigme de programmation basé sur le concept d'objets, qui peuvent contenir des données sous forme d'attributs et des codes sous forme de procédures ou de méthodes. Chaque objet peut être considéré comme une petite machine indépendante avec un rôle ou une responsabilité spécifique.

Introduction à la Programmation Orientée Objet

- **Principes de base de la POO :**
  1. *Encapsulation :* L'encapsulation est un mécanisme qui regroupe les données (attributs) et les méthodes qui manipulent ces données en une seule unité. Ce principe permet également de restreindre l'accès direct à certaines composantes de l'objet, ce qui est une méthode clé pour prévenir les modifications non autorisées de l'état interne de l'objet.
  2. *Héritage :* L'héritage est un moyen de former de nouvelles classes en utilisant des classes qui ont déjà été définies. Les nouvelles classes, connues sous le nom de classes dérivées, héritent des attributs et des comportements (méthodes) des classes de base, permettant la réutilisation du code et l'amélioration de la modularité.
  3. *Polymorphisme :* Le polymorphisme est le principe selon lequel les méthodes peuvent prendre plusieurs formes. Cela signifie qu'il est possible de définir plusieurs méthodes du même nom avec des comportements différents. Le polymorphisme peut être statique (aussi appelé surcharge) ou dynamique (aussi appelé liaison tardive).

Introduction aux Classes

- **Définition d'une classe :** Une classe en programmation orientée objet est un plan ou un modèle à partir duquel des objets sont créés. Elle représente un ensemble d'attributs (variables) et de méthodes (fonctions) qui définissent les caractéristiques et comportements d'un certain type d'objet.

Introduction aux Classes

- **Structure de base d'une classe en Python :** En Python, une classe est définie en utilisant le mot-clé 'class'. Les attributs sont définis dans la méthode '__init__', qui est le constructeur de la classe. Les méthodes sont des fonctions qui sont définies à l'intérieur de la classe.

Introduction aux Classes

- **Exemple : Définition d'une classe simple :**

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def afficher_details(self):
        print(f"Marque: {self.marque}, \
              Modele: {self.modele}")
```

Introduction aux Classes

- **Exercices 1 :**
  1. Créez une classe de base ‘Livre‘ avec un constructeur ‘__init__‘.
  2. Initialisez avec des attributs comme ‘titre‘ et ‘auteur‘.

Méthodes de Classe

- **Définition d'une méthode :** Une méthode est une fonction qui est définie à l'intérieur d'une classe. Elle est utilisée pour définir le comportement d'un objet ou pour interagir avec ses données internes (attributs).
- **Syntaxe des méthodes de classe :** Les méthodes de classe prennent toujours 'self' comme premier argument, qui fait référence à l'instance de la classe à laquelle la méthode appartient.

Méthodes de Classe

- **Exemple : Ajout de méthodes dans une classe :**

```
class Voiture:
    # ...[définitions précédentes]...

    def changer_modele(self, nouveau_modele):
        self.modele = nouveau_modele
```

Méthodes de Classe

- **Exercices 2 :**
  1. Ajoutez une méthode ‘get_details‘ pour retourner les détails du livre.
  2. Ajoutez une méthode ‘is_available‘ pour suivre la disponibilité du livre.

Attributs et Instances

- **Définition des attributs de classe et d'instance :** Les attributs de classe sont partagés par toutes les instances de la classe, tandis que les attributs d'instance sont spécifiques à chaque objet. Les attributs de classe sont définis directement dans la classe, hors de toute méthode, tandis que les attributs d'instance sont généralement définis dans le constructeur.

Attributs et Instances

- **Comment et quand utiliser les attributs :** Utilisez les attributs de classe pour les propriétés qui sont constantes pour toutes les instances. Les attributs d'instance sont utilisés pour des données qui varient d'un objet à l'autre.

Attributs et Instances

- **Exemple : Création d'attributs dans notre classe :**

```
class Voiture:
    categorie = 'Véhicule'  # Attribut de classe
    def __init__(self, marque, modele):
        self.marque = marque  # Attribut d'instance
        self.modele = modele  # Attribut d'instance
```

Attributs et Instances

- **Exercices 3 :**
  1. Introduisez une variable d'instance 'statut' (disponible ou emprunté).
  2. Modifiez 'is_available' pour vérifier l'attribut 'statut'.

Le Constructeur '__init__'

- **Rôle du constructeur '__init__' dans une classe :** '__init__' est utilisé pour initialiser les attributs d'instance de la classe. Il est appelé automatiquement lors de la création d'un nouvel objet.

Le Constructeur '__init__'

- **Syntaxe et utilisation de '__init__' :** Le constructeur '__init__' prend 'self' comme premier argument suivi par les paramètres nécessaires à l'initialisation de l'objet.

Le Constructeur '__init__'

- **Exemple : Initialisation d'objets avec '__init__' :**

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
ma_voiture = Voiture('Peugeot', '508')
```

Le Constructeur '__init__'

- **Exercices 4 :**
  1. Améliorez '__init__' pour inclure plus d'attributs comme 'genre', 'ISBN', et 'annee_de_publication'.
  2. Mettez à jour la création d'objets pour inclure ces nouveaux attributs.

La Méthode '__str__' et autres Méthodes Spéciales

- **Importance de '__str__' et autres méthodes spéciales :** La méthode '__str__' est utilisée pour définir la représentation sous forme de chaîne de l'objet lorsqu'il est converti en chaîne ou imprimé. D'autres méthodes spéciales comme '__len__' et '__repr__' fournissent des moyens supplémentaires pour définir des comportements spécifiques pour les objets de classe.

La Méthode '__str__' et autres Méthodes Spéciales

- **Comment personnaliser la représentation d'un objet :** En définissant '__str__', vous pouvez contrôler la manière dont les objets sont représentés sous forme de chaîne, ce qui est utile pour le débogage et le journalisation. '__repr__' est destiné à être une représentation officielle de l'objet et devrait idéalement être une expression Python valide.

La Méthode '__str__' et autres Méthodes Spéciales

- **Exemples de méthodes spéciales comme '__len__', '__repr__' :**

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
    def __str__(self):
        return f'Voiture: {self.marque}\
         {self.modele}'
    def __repr__(self):
        return f'Voiture("{self.marque}",\
         "{self.modele}")'
    def __len__(self):
        return len(self.marque) + len(self.modele)
```

La Méthode '__str__' et autres Méthodes Spéciales

- **Exercices 5 :**
  1. Implémentez la méthode '__str__' pour un affichage convivial des détails du livre.
  2. Ajoutez une méthode '__eq__' pour comparer deux livres basés sur l'ISBN.

Construction du Projet Final

- Créez une classe ‘Bibliothèque‘ qui peut contenir plusieurs objets ‘Livre‘.
- Implémentez des méthodes pour ajouter des livres, les emprunter, et les retourner.
- Utilisez toutes les méthodes et attributs précédemment créés dans ‘Livre‘ et intégrez-les dans ‘Bibliothèque‘.

Introduction à l'Encapsulation

- **Définition de l'encapsulation :** L'encapsulation est un concept fondamental en programmation orientée objet. Elle consiste à regrouper les données (attributs) et les méthodes qui manipulent ces données dans une seule unité, ou classe. Ce principe permet de masquer les détails d'implémentation internes d'une classe et d'exposer uniquement les éléments nécessaires pour le reste du programme.

Introduction à l'Encapsulation

- **Pourquoi l'encapsulation est fondamentale en POO :**
  L'encapsulation aide à protéger l'intégrité des données en empêchant leur accès direct de l'extérieur de la classe. Elle permet également de réduire la complexité et d'accroître la réutilisabilité du code. En encapsulant les détails de mise en œuvre, les modifications peuvent être faites de manière indépendante sans affecter les autres parties du programme.

Introduction à l'Encapsulation

- **Exemple conceptuel d'encapsulation :** Considérez une classe 'Voiture'. Les détails internes, tels que la mise en œuvre du système de transmission ou le type de moteur, sont cachés. Seules les fonctionnalités nécessaires, comme démarrer la voiture ou changer de vitesse, sont exposées.

Introduction à l'Encapsulation

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque  # Attribut privé
        self.modele = modele  # Attribut privé
    def demarrer(self):
        # Méthode publique
        print(f"La voiture {self.marque} \
              {self.modele} a démarré.")
```

Dans cet exemple, les attributs ‘marque‘ et ‘modele‘ sont encapsulés dans la classe ‘Voiture‘, et la méthode ‘demarrer‘ est une interface publique pour interagir avec l'objet ‘Voiture‘.

Implémentation de l'Encapsulation en Python

- **Utilisation des variables privées et protégées :** En Python, les attributs privés sont précédés par deux underscores (ex : '__privateVar'). Les attributs protégés, utilisés dans le cadre de l'héritage, sont précédés par un seul underscore (ex : '_protectedVar').

Implémentation de l'Encapsulation en Python

- **Syntaxe des getters et setters en Python :** Les getters et setters sont des méthodes utilisées pour accéder et modifier les attributs privés. Python utilise '@property' pour définir des getters et des setters.

Implémentation de l'Encapsulation en Python

- **Exemple : Ajout d'encapsulation dans la classe 'Book' :**

```
class Book:
    def __init__(self, title, author):
        self.__title = title
        self.__author = author

    @property
    def title(self):
        return self.__title

    @title.setter
    def title(self, value):
        self.__title = value
```

Dans cet exemple, '__title' est un attribut privé, et 'title' est une propriété avec un getter et un setter.

Exercice 6 :

- **Objectif :** Améliorer la classe 'Book' avec des principes d'encapsulation.
- **Tâches :**
  1. Ajouter des attributs privés (par exemple, '__status').
  2. Créer des getters et setters pour ces attributs.
  3. Assurer que le statut du livre est modifié de manière contrôlée.
- **Extension de l'exercice :**
  - Ajouter une logique pour empêcher des changements inappropriés, comme changer le statut d'un livre déjà emprunté.

Avantages de l'Encapsulation

- **Sécurité des données :** Prévention des accès non autorisés. L'encapsulation aide à protéger les données internes de la classe contre les manipulations extérieures non intentionnelles ou malveillantes.
- **Flexibilité et évolutivité :** Facilite la modification et l'extension des classes sans affecter les utilisateurs de la classe. Les modifications internes n'affectent pas les parties du code qui utilisent la classe.
- **Exemple :** Comment l'encapsulation dans 'Book' facilite les modifications futures. Par exemple, ajouter une logique supplémentaire dans les setters pour valider les modifications avant de les appliquer.