Programmation Orientée Objet en Python

Pierre Jaumier

Programmation Orientée Objet en Python

- Introduction aux concepts fondamentaux
- Différences avec la programmation procédurale
- Pourquoi utiliser la POO ?

Qu'est-ce que la POO ?

> La Programmation Orientée Objet est un paradigme basé sur le concept d'**objets** qui peuvent contenir **des données (attributs)** et **du code (méthodes)**.

En résumé : - Une **classe** = modèle (plan de construction) - Un **objet** = instance concrète du modèle

Comparaison rapide :

Langage	Concept
Java	`class`, `new`, `this`
C++	`class`, `struct`, constructeurs
Python	`class`, pas de `new`, syntaxe fluide

Définir une classe simple

```
class Personne:
    pass
```

- `class` = mot-clé pour définir une nouvelle classe
- `Personne` = nom de la classe (par convention, en PascalCase)
- `pass` = placeholder (rien ne se passe)

Créer un objet :

```
alice = Personne()
print(alice)  # <__main__.Personne object at 0x...>
```

Attributs d'instance

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

# Création d'un objet
alice = Personne('Alice', 25)
print(alice.nom)  # 'Alice'
```

- `__init__()` = constructeur de l'objet
- `self` = référence vers l'objet courant
- `self.nom` = attribut d'instance

Comparaison : - Java : `this.nom = nom;`

Méthodes d'une classe

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

    def se_presenter(self):
        print(f'Bonjour, je m\'appelle {self.nom}.')

# Utilisation
alice = Personne('Alice', 25)
alice.se_presenter()  # Bonjour, je m'appelle Alice.
```

- Une méthode = fonction liée à une classe
- Le premier argument est toujours `self`

Encapsulation

En Python, l'encapsulation n'est **pas stricte**, mais on peut simuler :

```
class CompteBancaire:
    def __init__(self, solde_initial):
        self._solde = solde_initial  # Attribut privé

    def afficher_solde(self):
        print(f'Solde : {self._solde} €')
```

- `_solde` = convention pour indiquer "privé"
- `__solde` = double underscore pour mettre l'accent sur le danger d'y accéder

Attention : Python ne bloque pas l'accès. C'est une question de discipline !

Getter / Setter en Python

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    def get_celsius(self):
        return self._celsius

    def set_celsius(self, valeur):
        if valeur < -273.15:
            raise ValueError('Température invalide')
        self._celsius = valeur
```

Cette approche permet de valider les données

En Java/C++ : cette gestion est automatique via `private` + getters/setters

Propriétés avec `@property`

Getter : La méthode décorée avec @property est utilisée pour accéder à l'attribut privé _celsius comme s'il s'agissait d'un attribut public.

```python
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius
```

Setter : Le décorateur @celsius.setter permet de définir une méthode pour modifier la valeur de _celsius avec une validation.

Avantage : usage naturel

```python
t = Temperature()
print(t.celsius)  # 0
```

Le décorateur `@attribut_privé.setter`

Setter : Le décorateur @celsius.setter permet de définir une méthode pour modifier la valeur de _celsius avec une validation.

```python
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, valeur):
        if valeur < -273.15:
            raise ValueError('Température invalide')
        self._celsius = valeur
```

```python
t = Temperature(25)
t.celsius = 30  # Valide
t.celsius = -300  # Lève une erreur
```

Méthodes spéciales utiles

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

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

    def __repr__(self):
        return f'Point(x={self.x}, y={self.y})'
```

Résultat :

```
p = Point(2, 3)
print(p)          # Point(2, 3)
print(repr(p))    # Point(x=2, y=3)
```

Utile pour le debugging et l'affichage utilisateur

Comparaison entre objets

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

    def __eq__(self, autre):
        return self.nom == autre.nom and self.age == autre.age

    def __lt__(self, autre):
        return self.age < autre.age
```

Permet de faire :

```
a = Personne('Alice', 25)
b = Personne('Alice', 25)
print(a == b)  # True
```

– Méthodes spéciales / magiques en détail (Dunder methods)

Python propose des méthodes commençant et finissant par `__` pour enrichir le comportement des objets.

Exemples courants :

- `__str__()` → affichage utilisateur
- `__repr__()` → affichage développeur
- `__eq__()` → comparaison
- `__lt__()`, `__gt__()` → tri et comparaison
- `.__len__()` → supporte la fontion len()

```
def __str__(self):
    return f'Personne: {self.nom}, {self.age} ans'

def __eq__(self, autre):
    return self.nom == autre.nom and self.age == autre.age
```

Il y en a plus de 100 (https://www.pythonmorsels.com/every-dunder-method/)

Attributs de classe vs instance

```
class Chien:
    espece = 'Canis lupus familiaris'  # Attribut de classe

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

# Utilisation
rex = Chien('Rex')
milou = Chien('Milou')

print(rex.espece)     # Canis lupus familiaris
print(milou.espece)
```

Tous les chiens partagent la même espèce

Mini-Projet : Classe Personne

Écrivez une classe `Personne` avec : - Nom, prénom, âge - Méthode `.se_presenter()` → 'Bonjour, je m'appelle Alice Martin.' - Méthode `.vieillir()` → incrémente l'âge - Gestion des accesseurs/mutateurs

Héritage simple

```
class Vehicule:
    def demarrer(self):
        print('Véhicule démarré.')

class Voiture(Vehicule):  # Hérite de Vehicule
    def rouler(self):
        print('Voiture en mouvement.')
```

La classe `Voiture` hérite de `Vehicule` et peut utiliser ses méthodes.

```
ma_voiture = Voiture()
ma_voiture.demarrer()  # Méthode héritée
```

– Méthodes statiques

```
class MathOperations:

    @staticmethod
    def add_numbers(x, y):
        return x + y

# Appel de la méthode statique sans créer d'instance de la classe
result = MathOperations.add_numbers(5, 3)
print("Le résultat est :", result)
```

Besoin d'une fonction qui a une relation logique avec la classe, mais qui n'a pas besoin d'accéder ou de modifier l'état de la classe ou de ses instances.

– Méthodes de classe

Le décorateur `@classmethod` permet de définir une **méthode qui reçoit la classe (`cls`) comme premier argument**, au lieu de l'instance (`self`).

```
class MaClasse:
    @classmethod
    def ma_methode(cls, arg1, arg2):
        ...
```

- `cls` = référence vers la **classe elle-même**
- Utile pour :
  - Créer des **constructeurs alternatifs**
  - Manipuler des **attributs de classe**
  - Des méthodes liées à la classe plutôt qu'à l'instance

Exemple : constructeur alternatif

Imaginons que tu veuilles créer une classe `Personne` où on puisse instancier facilement une personne à partir d'une chaîne de caractères.

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

    @classmethod
    def depuis_chaine(cls, donnees_str):
        nom, age = donnees_str.split(',')
        return cls(nom.strip(), int(age.strip()))


# Utilisation
a = Personne('Alice', 30)
b = Personne.depuis_chaine('Bob, 25')
print(b.nom, b.age)  # Bob 25
```

Quand utiliser `@classmethod` ?

- Pour **créer des constructeurs alternatifs**
- Pour manipuler des **attributs de classe**

- Pour des méthodes qui **doivent fonctionner même sur une sous-classe**

Un autre exemple : compteur d'instances

```
class Personne:
    nombre_de_personnes = 0

    def __init__(self, nom):
        self.nom = nom
        Personne.nombre_de_personnes += 1

    @classmethod
    def afficher_nombre(cls):
        print(f"Nombre total de personnes : {cls.nombre_de_personnes}")

# Utilisation
a = Personne('Alice')
b = Personne('Bob')

Personne.afficher_nombre()  # Nombre total de personnes : 2
```

Ici, `afficher_nombre()` est une méthode de classe qui lit un attribut de classe.

Méthodes abstraites avec `abc`

- `ABC` = classe de base abstraite
- `@abstractmethod` = décorateur pour définir une méthode qui doit être **implémentée dans les sous-classes**

Cela permet de créer des interfaces ou des classes partiellement implémentées, typiquement utilisées en Programmation Orientée Objet (POO).

Imaginons que tu veuilles modéliser différents animaux. Tous doivent avoir une méthode `.parler()` mais chacun la définit différemment.

Étape 1 : Définir la classe abstraite

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def parler(self):
        pass
```

Cette classe ne peut **pas être instanciée directement**.

Oblige les sous-classes à implémenter certaines méthodes

Méthodes abstraites

Étape 2 : Créer des sous-classes concrètes

```python
class Chat(Animal):
    def parler(self):
        print('Miaou')

class Chien(Animal):
    def parler(self):
        print('Wouaf')
```

Les méthodes abstraites sont maintenant **implémentées** dans chaque sous-classe.

Méthodes abstraites

Étape 3 : Utiliser le polymorphisme

```python
animaux = [Chat(), Chien()]

for animal in animaux:
    animal.parler()
```

Résultat :

```
Miaou
Wouaf
```

Méthodes abstraites

Ce que tu ne peux pas faire

```
a = Animal()  #  Erreur : on ne peut pas instancier une classe abstraite
```

Python empêche cela car la méthode `.parler()` n'est pas implémentée dans `Animal`.

Pourquoi utiliser `ABC` et `@abstractmethod` ?

Cela permet de : - Imposer une **structure commune** aux sous-classes - Garantir qu'une méthode existe dans toutes les sous-classes - Créer des **interfaces** en Python (comme en Java/C++)

Héritage multiple

```
class A:
    def methode(self):
        print('A')

class B:
    def methode(self):
        print('B')

class C(A, B):
    pass
```

Quel résultat donne `C().methode()` ?
Python suit **la MRO (Method Resolution Order)**.

```
print(C.__mro__)  # Affiche l'ordre de résolution
```

Astuce : évitez l'héritage multiple complexe sauf cas bien identifiés.

Polymorphisme

Le polymorphisme permet d'utiliser des objets selon leur interface commune :

```
class Chat:
    def parler(self):
        print('Miaou')

class Robot:
    def parler(self):
        print('Bip Bip')

for truc in [Chat(), Robot()]:
    truc.parler()
```

Résultat :

```
Miaou
Bip Bip
```

Très utile pour écrire du code générique et extensible.

Mini-Projet : Générateur de formulaires

Créez un système de formulaires basé sur des classes :

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

    def valider(self):
        raise NotImplementedError('Doit être implémenté')

class ChampTexte(Champ):
    def valider(self):
        return len(self.nom.strip()) > 0
```

Objectif : - Créer différentes sous-classes (`ChampEmail`, `ChampNombre`) - Implémenter une méthode `.valider()` cohérente - Créer une classe `Formulaire` qui gère plusieurs champs

---

Bonnes pratiques

Utilisez : - Des **classes** pour modéliser des entités logiques - Des **propriétés** plutôt que des getters/setters manuels - Des **méthodes abstraites** pour imposer une structure

Évitez : - L'héritage multiple trop complexe - Les classes trop longues (respectez SRP: Single Responsibility Principle)

Lisibilité avant performance !

Conclusion

Ce que vous avez appris aujourd'hui : - Définir une classe avec Python - Créer des objets et leur donner des attributs - Ajouter des méthodes - Simuler l'encapsulation - Utiliser les propriétés - Utiliser les méthodes spéciales (`__str__`, `__eq__`, etc.) - Organiser des hiérarchies de classes - Appliquer l'héritage et le polymorphisme - Gérer les attributs privés et les validations

Questions?