Chapitre 3. Classes et objets

Objectifs

À la fin de ce chapitre, tu seras capable de :

- Comprendre les **principes de la POO** (Programmation Orientée Objet) en Python,
- Créer et manipuler des **classes** et des **objets**,
- Définir des **attributs**, des **méthodes**, et des **constructeurs/destructeurs**,
- Appliquer les notions d'**encapsulation**, de **visibilité**, et de **relations entre classes** (association, dépendance, amitié, classes imbriquées).

1. Introduction à la Programmation Orientée Objet

La POO est un paradigme de programmation basé sur la modélisation du monde réel à l'aide d'objets**.**

1.1.Concepts fondamentaux

Concept	**Description**
Classe	Modèle ou plan de construction d'un objet.
Objet	Instance concrète d'une classe.
Attributs	Données ou propriétés de l'objet.
Méthodes	Fonctions associées à une classe.
Encapsulation	Protection des données internes d'un objet.
Héritage	Possibilité de créer une nouvelle classe à partir d'une autre.
Polymorphisme	Capacité à redéfinir des méthodes selon le contexte.

Exemple

Classe : Voiture

Objets : ma_voiture, ta_voiture

Attributs : couleur, vitesse, marque

Méthodes : demarrer(), accelerer(), freiner()

2. Déclaration d'une classe et création d'objets

Syntaxe de base

```
class NomDeClasse:
    def __init__(self):
        # Constructeur
        pass
```

Exemple

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

# Création d'objets
v1 = Voiture("Toyota", "rouge")
v2 = Voiture("Renault", "bleue")
print(v1.marque)   # Toyota
print(v2.couleur)  # bleue
```

La sortie

```
Toyota
bleue
```

Exercice 1

Créer une classe Etudiant avec les attributs **nom, prenom, age**. Créer ensuite deux objets et afficher leurs info.

Solution

```
class Etudiant:
    def __init__(self,nom,prenom,age):
        self.nom=nom
        self.prenom=prenom
        self.age=age
e1=Etudiant("ahmed","dbkm",22)
e2=Etudiant("ali","univ",30)
print(e1.nom)
print(e1.prenom)
print(e1.age)
print(e2.nom)
print(e2.prenom)
print(e2.age)
```

Sortie

```
ahmed
dbkm
22
ali
```

```
univ
30
```

3. Variables et méthodes d'instance

3.1. Variables d'instance

Ce sont des variables propres à chaque objet.

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

Chaque objet possède **sa propre copie** de x et y.

3.2. Méthodes d'instance

Elles sont définies **à l'intérieur de la classe** et **opèrent sur les attributs de l'objet**.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def afficher(self):
        print(f"Point({self.x}, {self.y})")
```

Exercice 2

Créer une classe Rectangle avec les attributs longueur et largeur, et une méthode surface().

Solution

```
class Rectangle:
    def __init__(self, longueur,largeur):
        self.longueur = longueur
        self.largeur= largeur
    def surface(self):
       return  self.longueur*self.largeur


v1=Rectangle(1,2)
print("surface:",v1.surface())
```

Sortie

```
surface: 2
```

4. Droits d'accès et encapsulation

- **Public** : Attribut ou méthode accessible partout.
- **Protégé** (_nom) : Conventions indiquant que l'attribut est réservé à l'usage interne ou aux classes dérivées

- **Privé** (__nom) : inaccessible directement depuis l'extérieur. Python applique le **name-mangling** → l'attribut est renommé en _NomClasse__nom, ce qui empêche son accès direct.

Exemple

```
class Exemple:
    def __init__(self):
        self.public = "Je suis public"
        self._protege = "Je suis protégé (usage interne)"
        self.__prive = "Je suis privé (name mangling)"

    def afficher(self):
        print(self.public)
        print(self._protege)
        print(self.__prive)
e = Exemple()
# --- Accès aux attributs ---
print(e.public)       # ✓ OK
print(e._protege)     # ✓ Techniquement OK, mais déconseillé
# print(e.__prive)    # ✗ Erreur : inaccessible directement
# Accès au privé par name mangling :
print(e._Exemple__prive)  # ✓ Possible mais non recommandé
```

Exercice 3

Créer une classe Employe avec un attribut privé __salaire et une méthode augmenter().

Solution

```
class Employe:
    def __init__(self, nom, salaire):
        self.nom = nom
        self.__salaire = salaire  # attribut privé

    def augmenter(self, pourcentage):
        """Augmente le salaire d'un certain pourcentage."""
        if pourcentage > 0:
            self.__salaire *= (1 + pourcentage / 100)

    def afficher_salaire(self):
        """Méthode d'accès contrôlée au salaire."""
        return self.__salaire
# Exemple d'utilisation
e = Employe("Alice", 3000)
print("Salaire initial :", e.afficher_salaire())
e.augmenter(10)
print("Après augmentation :", e.afficher_salaire())
```

Sortie
Salaire initial : 3000
Après augmentation : 3300.0000000000005

5. Constructeur et destructeur

5.1. Constructeur (__init__)

Méthode appelée automatiquement lors de la création d'un objet.

Rôle :

- Initialiser les attributs de l'objet.
- Préparer l'objet pour être utilisé.

Exemple

```
class Exemple:
    def __init__(self, nom):
        self.nom = nom
        print("Objet créé :", self.nom)
e = Exemple("Test")  # Appelle automatiquement __init__
```

5.2. Destructeur (__del__)

Méthode appelée lors de la suppression d'un objet (ou à la fin du programme).

Rôle :

- Libérer des ressources (fichiers, connexions…),
- Exécuter des actions finales avant la destruction.

```
class Exemple:
    def __init__(self, nom):
        self.nom = nom
        print("Objet créé :", self.nom)

    def __del__(self):
        print("Objet supprimé :", self.nom)

e = Exemple("Test")
del e  # Force la destruction
```

6. Objets en interaction

Une classe peut **contenir un objet d'une autre classe** :

```
class Moteur:
    def __init__(self, puissance):
        self.puissance = puissance

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

m = Moteur(120)
v = Voiture("Peugeot", m)
```

```
print(v.moteur.puissance)  # 120
```

l'attribut moteur **n'est pas un nombre ou une chaîne**, mais **un objet complet** de type Moteur.

7. Égalité et clonage d'objets

7.1. Comparaison d'objets

Par défaut, == compare les **adresses mémoire**.

Quand vous comparez deux objets avec ==, Python compare **leurs références**, c'est-à-dire **leur emplacement en mémoire**, et non leur contenu.

Exemple

```
a = Point(1, 2)
b = Point(1, 2)
print(a == b)   # False par défaut
```

Même si a et b ont les mêmes valeurs, ce sont **deux objets différents**, donc == renvoie False.

Redéfinir __eq__() pour comparer le contenu

Pour comparer les objets **en fonction de leurs attributs**, on redéfinit la méthode spéciale **__eq__().**

Exemple :

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

    def __eq__(self, autre):
        return self.x == autre.x and self.y == autre.y
```

Maintenant :

```
p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2)   # True
```

- Python compare désormais **le contenu** (x et y).
- On peut donc choisir ce que signifie « égalité » pour nos objets.

Pourquoi redéfinir __eq__() ?

- Pour rendre les objets comparables selon leur **état**, pas leur identité.
- Pour utiliser les objets dans des tests, collections, structures de données.
- Pour définir des types personnalisés cohérents.

7.2. Méthodes de comparaison en Python

Tu peux redéfinir ces méthodes pour personnaliser la comparaison entre objets.

__eq__(self, other)

Définit l'opérateur **==**

__ne__(self, other)

Définit **!=**

Si __ne__ n'est pas défini, Python utilise automatiquement **not eq()**.

__lt__(self, other)

Définit <

__le__(self, other)

Définit <=

__gt__(self, other)

Définit >

__ge__(self, other)

Définit >=

Exemple

```
import math

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

    def distance(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, autre):
        return self.x == autre.x and self.y == autre.y

    def __lt__(self, autre):
        return self.distance() < autre.distance()

    def __le__(self, autre):
        return self.distance() <= autre.distance()

    def __gt__(self, autre):
        return self.distance() > autre.distance()

    def __ge__(self, autre):
        return self.distance() >= autre.distance()
p1 = Point(1, 2)
p2 = Point(2, 2)
p3 = Point(1, 2)

print(p1 == p3)   # True
print(p1 == p2)   # False
print(p1 < p2)    # True  (distance 2.2 < 2.8)
print(p2 > p1)    # True
print(p1 is p3)   # False (objets différents)
```

Sortie

```
True
False
```

```
True
True
False
```

7.3. Clonage (copie d'objet)

En Python, on peut dupliquer un objet en utilisant le module copy.

Il existe **deux types de copie** :

- **copie superficielle (shallow copy)**
- **copie profonde (deep copy)**

1. Copie superficielle : copy.copy()

p2 = copy.copy(p1)

Ce qui est copié :

- L'objet lui-même est dupliqué.
- **Mais tous les sous-objets internes sont partagés** avec l'original.

Exemple :

Si l'objet contient une liste, un dictionnaire ou un objet interne, la copie et l'original **pointeront vers les mêmes sous-objets**.

C'est une duplication « au premier niveau ».

2. Copie profonde : copy.deepcopy()

p3 = copy.deepcopy(p1)

Ce qui est copié :

- L'objet principal est dupliqué.
- **Tous les sous-objets internes sont également copiés récursivement.**

→ La copie est totalement indépendante de l'original.

Illustration simple avec la classe Point

Si la classe Point ne contient que des entiers (x et y), alors copy et deepcopy donneront le même résultat, car les entiers sont immuables. Mais si Point contient un objet interne mutable, la différence apparait.

Exemple

```
import copy

class Point:
    def __init__(self, x, y, tags):
        self.x = x
        self.y = y
        self.tags = tags  # liste mutable

p1 = Point(2, 3, ["A", "B"])

p2 = copy.copy(p1)        # Copie superficielle
p3 = copy.deepcopy(p1)    # Copie profonde
```

```
p1.tags.append("C")

print(p1.tags)   # ['A', 'B', 'C']
print(p2.tags)   # ['A', 'B', 'C']   (partage les mêmes tags !)
print(p3.tags)   # ['A', 'B']        (copie indépendante)
```

Sortie

```
['A', 'B', 'C']
['A', 'B', 'C']
['A', 'B']
```

8. Envoi de messages (appels de méthodes)

Les objets communiquent entre eux en s'envoyant des **messages**. Un message, c'est simplement **un appel de méthode**.

Exemple

```
class Robot:
    def parler(self, message):
        print("Robot dit :", message)


r1 = Robot()
r1.parler("Bonjour")
```

9. Relations entre classes

9.1. Association

Une classe **utilise** une autre classe sans en dépendre directement.

Exemple

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


class Livre:
    def __init__(self, titre, auteur):
        self.titre = titre
        self.auteur = auteur


a = Auteur("Victor Hugo")
l = Livre("Les Misérables", a)
print(l.titre)
print(l.auteur.nom)
```

9.2. Dépendance

Une classe **a besoin** d'une autre classe pour fonctionner ponctuellement. La **dépendance** est une relation **ponctuelle** entre deux classes :

- Une classe **a besoin d'une autre classe pour réaliser une tâche**, mais elle **ne la conserve pas comme attribut**.
- La relation est temporaire et limitée à l'exécution d'une méthode.
- On parle parfois de **« utilise »** ou **« a besoin de »**.

```
class Imprimante:
    def imprimer(self, doc):
        print(f"Impression du document {doc.titre}")
```

- La classe Imprimante **dépend de l'objet doc** pour imprimer son titre.
- La dépendance est **locale** à la méthode imprimer() : Imprimante n'a pas besoin de stocker le document, elle l'utilise seulement au moment de l'appel.

Supposons que l'on ait une classe Document :

```
class Document:
    def __init__(self, titre):
        self.titre = titre

doc1 = Document("Rapport annuel")
imp = Imprimante()

imp.imprimer(doc1)  # Dépendance ponctuelle à l'objet
```

- Imprimante fonctionne uniquement lorsque la méthode imprimer est appelée avec un document.
- Si on n'appelle pas la méthode, il n'y a pas de lien permanent entre Imprimante et Document.

10. Classes amies (concept simulé)

Python ne possède pas le concept de "classe amie" (comme en C++). Mais on peut simuler ce comportement via **méthodes publiques dédiées** ou **attributs protégés**.

```
class A:
    def __init__(self):
        self._secret = 42


class B:
    def afficher_secret(self, a):
        print("Secret :", a._secret)
```

- A a un attribut _secret **protégé**, signalant qu'il est **interne à la classe**.
- B peut quand même y accéder car Python **n'impose pas de restriction stricte**.

- On simule donc un **accès "amical"**, sans violer la convention d'encapsulation, mais en restant **transparent et volontaire**.

11. Classes imbriquées (ou internes)

Une classe peut être **définie à l'intérieur d'une autre**.

```
class Externe:
    class Interne:
        def message(self):
            print("Classe interne")

e = Externe()
i = e.Interne()
i.message()
```

Utilisation : encapsulation logique ou structuration d'objets complexes.