Chapitre 4. Héritage et polymorphisme

Objectifs

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

- Comprendre les relations d'héritage**,** d'agrégation et de composition**,**
- Créer et étendre des classes dérivées**,**
- Maîtriser le chaînage des constructeurs**,**
- Utiliser les notions de multi-héritage et de polymorphisme**,**
- Redéfinir et surcharger des méthodes,
- Manipuler des classes abstraites et des méthodes virtuelles**.**

1. Agrégation et composition d'objets

Les relations entre classes permettent de **modéliser le monde réel**.

1.1.Composition

Une classe **contient** une autre classe, et **détient sa durée de vie**.

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

class Voiture:
    def __init__(self, marque, puissance):
        self.marque = marque
        self.moteur = Moteur(puissance)  # composition

v = Voiture("Toyota", 120)
print(v.moteur.puissance)  # 120
```

Si l'objet Voiture est détruit, le Moteur l'est aussi.

1.2.Agrégation

Une classe **réutilise** une autre classe existante sans en être propriétaire.

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

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

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

Si Voiture est supprimée, le Moteur peut encore exister.

2. Héritage : principe et syntaxe

L'héritage permet de **créer une nouvelle classe** à partir d'une **classe existante** (appelée classe de base ou superclasse*)*.

Syntaxe

```
class ClasseDeBase:
    pass

class ClasseDerivee(ClasseDeBase):
    pass
```

Exemple

```
class Animal:
    def parler(self):
        print("un animal.")

class Chien(Animal):
    def aboyer(self):
        print("Wouf !")

r = Chien()
r.parler()  # Hérité de Animal
r.aboyer()
```

3. Règles d'héritage et chaînage des constructeurs

Appel au constructeur de la classe de base

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

class Chien(Animal):
    def __init__(self, nom, race):
        super().__init__(nom)  # appel du constructeur parent
        self.race = race
```

Le mot-clé **super()** permet d'appeler les méthodes de la classe mère.

Exemple

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

class Etudiant(Personne):
    def __init__(self, nom, niveau):
        super().__init__(nom)
        self.niveau = niveau

    def afficher(self):
        print(f"{self.nom} est en {self.niveau}.")
r=Etudiant("ahmed", "M1")
r.afficher()
```

4. Classes de base, attributs et méthodes protégés

Les attributs hérités peuvent être :

- **publics** : accessibles directement,
- **protégés** : précédés d'un seul _,
- **privés** : précédés de __, non accessibles par héritage direct.

Exemple:

```
class A:
    def __init__(self):
        self._protégé = "Protégé"
        self.__privé = "Privé"

class B(A):
    def afficher(self):
        print(self._protégé)
        # print(self.__privé)  # Erreur
```

5. Multi-héritage

Python permet d'hériter de plusieurs classes à la fois**.**

```
class A:
    def afficher_A(self): print("Classe A")

class B:
    def afficher_B(self): print("Classe B")

class C(A, B):
    pass

obj = C()
obj.afficher_A()
obj.afficher_B()
```

5.1. Ordre de résolution (MRO)

Python résout l'ordre d'héritage avec le **C3 Linearization (MRO)**.

C.__mro__ # Affiche l'ordre de recherche des méthodes

Exemple

```
class A:
    def dire(self):
        print("Classe A")

class B(A):
    def dire(self):
        print("Classe B")

class C(A):
    def dire(self):
        print("Classe C")

class D(B, C):
    pass  # pas de méthode dire ici

d = D()
d.dire()
print(D.__mro__)   # Affiche l'ordre de résolution
```

Sortie

```
Classe B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

Python cherche la méthode dire() dans l'ordre suivant (appelé MRO = Method Resolution Order) :

1. D (la classe elle-même)
2. B (premier parent)
3. C (deuxième parent)
4. A (parent de B et C)
5. object (racine de toutes les classes en Python)

La première dire() trouvée est dans **B**, donc il affiche : **Classe B**

Python suit un algorithme appelé **C3 Linearization**, qui garantit :

- Respect du sens d'héritage
- Pas de conflit entre classes parentes
- Résolution cohérente et sans ambiguïté

6. Surcharge et redéfinition des méthodes

6.1. Redéfinition

Une classe dérivée peut réécrire une méthode de la classe de base :

```
class Faculte:
    def parler(self):
        print("ST")

class Departement(Faculte):
    def parler(self):
        print("ELN")

c = Departement()
c.parler()
```

Sortie

```
ELN
```

6.2. Surcharge du constructeur

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

    def afficher(self):
        print(f"Classe A : x = {self.x}")

class B(A):
    def __init__(self, x, y):
        super().__init__(x)  # Appel au constructeur de A
        self.y = y

    def afficher(self):
        print(f"Classe B : x = {self.x}, y = {self.y}")

# Objet de la classe A : constructeur avec 1 paramètre
obj1 = A(10)
obj1.afficher()

# Objet de la classe B : constructeur avec 2 paramètres
obj2 = B(5, 20)
obj2.afficher()
```

Sortie

```
Classe A : x = 10
Classe B : x = 5, y = 20
```

7. Polymorphisme

Le polymorphisme permet à plusieurs classes de partager une même interface (même méthode, comportement différent).

Exemple

```python
class Science:
    def parler(self): pass

class Physique(Science):
    def parler(self): print("vibration")

class Chimie(Science):
    def parler(self): print("hydrogene!")
module = [Physique(),Chimie()]

for a in module:
    a.parler()  # Appelle la bonne méthode selon le type d'objet
```

Sortie

```
vibration
hydrogene!
```

8. Méthodes virtuelles et classes abstraites

8.1. Classe abstraite

Une classe abstraite définit une interface commune sans implémentation concrète. Une classe abstraite est un modèle, une idée générale, qui n'est pas complète. Elle sert de base pour créer d'autres classes, mais on ne peut pas créer d'objet directement à partir d'elle.

Exemple

```
from abc import ABC, abstractmethod
""" Importation de la bibliothèque abc
ABC signifie Abstract Base Class (classe de base abstraite).
abstractmethod permet de définir des méthodes obligatoires dans les classes filles.
================================================================================"""
class Forme(ABC):
    @abstractmethod
    def aire(self):
        pass
""" ✔ Forme est une classe abstraite.
✔ Elle contient une méthode aire() sans implémentation (juste un pass).
✖ On ne peut pas créer d'objet Forme, car elle est abstraite.
➡ Elle impose à ses sous-classes d'implémenter la méthode aire().
================================================================================"""

class Cercle(Forme):  #Classe Cercle qui hérite de Forme
    def __init__(self, r):
        self.r = r
    def aire(self):
        return 3.14 * self.r ** 2

    """✔ Cercle hérite de Forme.
✔ Elle doit implémenter aire(), sinon erreur.
➡ aire() calcule l'aire : 3.14*r**2
============================================================="""
#Création d'objets et utilisation
c1 = Cercle(5)
c2 = Cercle(3)

print("Aire du cercle 1 :", c1.aire())
print("Aire du cercle 2 :", c2.aire())
```

Sortie

```
Aire du cercle 1 : 78.5
Aire du cercle 2 : 28.26
```

On ne peut pas **instancier** une classe abstraite.