Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG
DUT- IDIA 1 - S2
M124 – Programmation en Python
Année universitaire: 2024-2025

Support de cours

Programmation en Python

Chapitre IV : Programmation orientée objet en Python

Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- **INTRODUCTION À LA POO**
- **LES CLASSES ET LES OBJETS**
- **L'ENCAPSULATION**
- **L'HÉRITAGE**
- **LE POLYMORPHISME**
- **L'ABSTRACTION ET LES INTERFACES**
- **EXERCICES**

Introduction

La Programmation Orientée Objet (POO) est un paradigme qui permet de structurer le code en objets qui possèdent des attributs (données) et des méthodes (comportements).

❑ Concepts clés

- **Classe** : Modèle définissant les caractéristiques et comportements d'un objet.
- **Objet** : Instance concrète d'une classe.
- **Attribut** : Variable qui stocke des données propres à un objet.
- **Méthode** : Fonction définie dans une classe et applicable aux objets.

❑ Avantages de la POO

- **Modularité** : Diviser le code en blocs réutilisables.
- **Réutilisabilité** : Réutiliser du code avec l'héritage.
- **Encapsulation** : Protéger les données sensibles.
- **Polymorphisme** : Avoir plusieurs comportements pour une même méthode.

❑ Principaux concepts

La POO repose sur **quatre principes fondamentaux** :

- **Encapsulation** : cacher les détails internes d'un objet.
- **Héritage** : une classe peut hériter des propriétés d'une autre classe.
- **Polymorphisme** : une méthode peut avoir plusieurs comportements différents.
- **Abstraction** : cacher les détails d'implémentation et ne montrer que l'essentiel.

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

    def rouler(self):
        print(f"La {self.marque} roule !")

# Création d'un objet
ma_voiture = Voiture("Toyota", "Bleu")
ma_voiture.rouler()
```

```
La Toyota roule !
```

Les classes et les objets

- Une **classe** est un modèle qui définit les attributs et méthodes des objets.
- Un **objet** est une instance d'une classe.

❑ Déclaration d'une Classe

On définit une classe avec le mot-clé **`class`**, suivi de son nom en **MAJUSCULE** initiale par convention.

```
class NomDeLaClasse:
    # Corps de la classe (attributs et méthodes)
```

❑ Création d'un objet en Python

```
objet = NomDeLaClasse()
```

→ Le constructeur `__init__`

- Le **constructeur** est une méthode spéciale appelée **`__init__`**, exécutée automatiquement lors de la création d'un objet.
- Le mot-clé **`self`** fait référence à l'instance de l'objet en cours.

❑ Syntaxe du constructeur

```
class NomDeLaClasse:
    def __init__(self):  # Constructeur
```

- **`self`** est un paramètre spécial qui représente l'instance (objet) actuelle de la classe.
- Il permet d'accéder aux attributs et aux méthodes de l'objet.

```
class Animal:
    def __init__(self, nom):
        self.nom = nom  # `self.nom` est un attribut propre à chaque objet

animal1 = Animal("Chat")
print(animal1.nom)
```

Les classes et les objets

Exemple

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

# Création d'objets
personne1 = Personne("Ali", 19)
personne2 = Personne("Sara", 20)

print(personne1.nom, personne1.age)
print(personne2.nom, personne2.age)
```

```
Ali 19
Sara 20
```

→ Les attributs d'une classe

- Un **attribut** est une variable stockée à l'intérieur d'un objet.

❑ Types d'attributs

Il existe deux types d'attributs en POO :

- **Attributs d'instance** : Ils sont spécifiques à chaque objet et définis dans le constructeur (**`__init__`**).
- **Attributs de classe** : Partagés entre tous les objets et définis en dehors du constructeur

Exemple

```
class Voiture:
    nb_roues = 4  # Attribut de classe

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

v1 = Voiture("Toyota")
v2 = Voiture("BMW")

print(v1.marque)
print(v2.marque)
print(Voiture.nb_roues)  # 4 (même valeur pour toutes les instances)
```

```
Toyota
BMW
4
```

→ Les méthodes d'une classe

- Une **méthode** est une fonction définie dans une classe et qui opère sur ses attributs.

❑ Types de méthodes

Il existe trois types de méthodes en POO :

- **Méthodes d'instance**
  - Utilisent `self` comme premier paramètre.
  - Peuvent accéder et modifier les attributs de l'objet (instance).
  - Peuvent appeler d'autres méthodes de l'objet.
- **Méthodes de classe (`@classmethod`)**
  - Utilisent `cls` comme premier paramètre.
  - Peuvent **accéder** et **modifier** les attributs de la classe (et non de l'objet).
  - Définies avec le décorateur `@classmethod`.
- **Méthodes statiques (`@staticmethod`)**
  - Ne prennent ni `self` ni `cls` en paramètre.
  - **Ne modifient** ni les attributs de l'objet ni ceux de la classe.
  - Définies avec le décorateur `@staticmethod`.
  - Utilisées pour des fonctions utilitaires qui n'ont pas besoin d'accéder aux données de la classe ou de l'objet.

→ Les méthodes d'une classe

Exemple

```
class Voiture:
    nb_roues = 4  # Attribut de classe

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

    def afficher_marque(self):  # Méthode d'instance
        print(f"Marque : {self.marque}")

    @classmethod
    def afficher_roues(cls):  # Méthode de classe
        print(f"Nombre de roues : {cls.nb_roues}")

    @staticmethod
    def description():  # Méthode statique
        print("Une voiture est un moyen de transport.")


ma_voiture = Voiture("Toyota")
ma_voiture.afficher_marque()

Voiture.afficher_roues()
Voiture.description()
```

```
Marque : Toyota
Nombre de roues : 4
Une voiture est un moyen de transport.
```

→ Les modificateurs d'accès

Les **modificateurs d'accès** définissent la **visibilité des attributs et méthodes**.

En Python, il existe trois types d'accès aux attributs et méthodes :

- **Public** : Accessible de partout.
- **Protégé** (`_nom`) : Accessible uniquement à l'intérieur de la classe et des sous-classes (convention, mais pas une vraie restriction).
- **Privé** (`__nom`) : Non accessible directement en dehors de la classe.

```
class Personne:
    def __init__(self, nom, age, salaire):
        self.nom = nom  # Public
        self._age = age  # Protégé
        self.__salaire = salaire  # Privé

p = Personne("Sara", 20, 5000)
print(p.nom)
print(p._age)
# print(p.__salaire)  # Erreur : attribut privé
```

```
Sara
20
```

Le modificateur privé empêche l'accès direct à l'attribut __salaire depuis l'extérieur de la classe.

L'encapsulation

L'**encapsulation** est un principe fondamental de la POO qui consiste à **cacher les données internes d'un objet** et à **limiter leur accès** depuis l'extérieur. Cela permet de protéger les informations sensibles et d'éviter les modifications involontaires.

❑ Pourquoi utiliser l'encapsulation ?

- Sécurise les attributs
- Empêche la modification involontaire
- Contrôle l'accès aux données

```python
class CompteBancaire:
    def __init__(self, solde):
        self.__solde = solde  # Attribut privé

    def afficher_solde(self):
        print(f"Solde : {self.__solde}")

compte = CompteBancaire(1000)
compte.afficher_solde()
```

```
Solde : 1000
```

Pour accéder aux attributs privés tout en préservant leur encapsulation, on utilise des ***getters***, des ***setters*** et des ***deleters*** à l'aide des décorateurs ***`@property`***, ***`@nom.setter`*** et ***`@nom.deleter`***

L'encapsulation

→ Getters, setters, deleter

- Un **getter** est une méthode qui permet d'accéder à un attribut privé sans y accéder directement.
- Un **setter** est une méthode qui permet de modifier un attribut privé tout en imposant des contraintes.
- Un **deleter** permet de supprimer un attribut en toute sécurité.

- `@property` : Transforme une méthode en getter, permettant d'accéder à un attribut privé comme une propriété.
- `@nom.setter` : Définit une méthode qui modifie la valeur d'un attribut privé.
- `@nom.deleter` : Définit une méthode pour supprimer un attribut privé.

```python
class Compte:
    def __init__(self, solde):
        self.__solde = solde

    @property
    def solde(self):  # Getter
        return self.__solde

    @solde.setter
    def solde(self, montant):  # Setter
        if montant >= 0:
            self.__solde = montant

    @solde.deleter
    def solde(self):  # Deleter
        del self.__solde
```

L'encapsulation

→ Getters, setters, deleter

Exemple

```
class Compte:
    def __init__(self, solde):
        self.__solde = solde  # Attribut privé

    @property
    def solde(self):  # Getter
        return self.__solde

    @solde.setter
    def solde(self, montant):  # Setter
        if montant >= 0:
            self.__solde = montant
        else:
            print("Impossible d'avoir un solde négatif !")

    @solde.deleter
    def solde(self):  # Deleter
        del self.__solde

c = Compte(1000)
print(c.solde)  # Accès comme un attribut normal
c.solde = 500   # Modification autorisée
print(c.solde)
del c.solde     # Suppression autorisée
```

```
1000
500
```

Cette approche protège les données tout en permettant un accès contrôlé, sans exposer directement **`__solde`**

Sans @property, on devrait appeler **`c.solde()`** (comme une fonction), mais grâce à **`@property`**, on peut simplement faire **`c.solde`**

`@solde.setter` permet de modifier solde directement (**`c.solde = 500`**).

L’héritage

L'héritage, concept fondamental de la POO, permet à une classe enfant d'acquérir et de réutiliser les attributs et méthodes d'une classe parent.

- La classe parent contient des attributs et méthodes de base.
- La classe enfant hérite de la classe parent et peut ajouter ou modifier des fonctionnalités.

❑ Pourquoi utiliser l’héritage ?

- Évite la duplication du code.
- Facilite l’extension des fonctionnalités.
- Simplifie la maintenance du code.
- Rend le programme plus modulaire.

→ Pour créer une relation d’héritage, il suffit de spécifier le nom de la classe parent **entre parenthèses** lors de la déclaration de la classe enfant.

❑ Syntaxe d'une classe parent et une classe enfant

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

    def afficher_nom(self):
        print(f"Nom : {self.nom}")

class Enfant(Parent):  # La classe Enfant hérite de Parent
    def __init__(self, nom, age):
        super().__init__(nom)  # Appelle le constructeur de la classe Parent
        self.age = age

    def afficher_infos(self):
        print(f"Nom : {self.nom}, Age : {self.age}")
```

- ***super().__init__(nom)*** appelle le constructeur du parent pour éviter de réécrire du code.
- Sans ***super()***, il aurait fallu redéfinir nom manuellement.

→ Utilisation de *super()*

- La fonction **super()** permet d'accéder aux méthodes de la classe parent sans mentionner explicitement son nom. Cela facilite la maintenance du code, surtout en cas de modification du nom de la classe parent.
- **super()** peut aussi être utilisé pour appeler des méthodes de la classe parent. et pas seulement le constructeur.

```
class Vehicule:
    def demarrer(self):
        print("Le véhicule démarre.")

class Camion(Vehicule):
    def demarrer(self):
        super().demarrer()  # Appel à la méthode parent
        print("Le camion est prêt à rouler.")

camion1 = Camion()
camion1.demarrer()
```

```
Le véhicule démarre.
Le camion est prêt à rouler.
```

→ Héritage multiple

- Python autorise l'héritage multiple.
- Une classe peut hériter de plusieurs classes à la fois.

Exemple

```
class A:
    def afficher_A(self):
        print("Méthode de A")

class B:
    def afficher_B(self):
        print("Méthode de B")

class C(A, B):  # Héritage multiple
    pass

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

```
Méthode de A
Méthode de B
```

Quand on utilise **l'héritage multiple**, un problème peut apparaître : **dans quel ordre les classes parentes sont-elles parcourues ?**

→ Python calcule l'ordre d'exécution des méthodes grâce au **MRO** (Method Resolution Order) en utilisant ***C3 Linearization***.

→ Héritage multiple

❑ MRO (Method Resolution Order)

- Le **MRO** détermine l'ordre dans lequel les classes sont recherchées quand on appelle une méthode.
- Dans le cas de l'héritage multiple, Python doit choisir quelle classe parent exécuter en premier.

Exemple

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

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

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

class D(B, C):  # Héritage multiple
    pass

obj = D()
obj.montrer()
```

→ Héritage multiple

❑ MRO (Method Resolution Order)

- On utilise `print(Classe.__mro__)` ou `print(Classe.mro())` pour voir l'ordre exact suivi par Python lors de la résolution des méthodes en héritage multiple.
- Cela affiche la hiérarchie des classes parentes et permet de mieux comprendre l'ordre d'exécution des méthodes. C'est un outil utile pour détecter d'éventuels conflits ou redondances dans la hiérarchie des classes.

Exemple

```
print(D.__mro__)
print(D.mro())
```

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

→ Python commence par examiner **D**, puis **B**, ensuite **C**, et enfin **A**, car **B** et **C** héritent de **A**. La classe **object** est la classe mère par défaut.

→ Python applique ***C3 Linearization*** pour garantir un ordre logique et éviter les répétitions.

→ La redéfinition des méthodes (Overriding)

La **redéfinition** d'une méthode consiste à remplacer une méthode héritée de la classe parent par une nouvelle version dans la classe enfant. Cela permet d'adapter le comportement d'une méthode sans modifier la classe parent.

```
class Media:
    def afficher_type(self):
        print("Ceci est un média.")

class Livre(Media):
    def afficher_type(self):
        print("Ceci est un livre.")  # Redéfinition de la méthode

livre1 = Livre()
livre1.afficher_type()
```

```
Ceci est un livre.
```

Le polymorphisme

Le **polymorphisme** permet à une méthode d'être utilisée de manière différente selon la classe à laquelle elle appartient.

❑ Pourquoi utiliser le Polymorphisme ?

- Il permet d'écrire un code plus générique et flexible.
- Il favorise le principe d'extensibilité sans modifier le code existant.

❑ Types de Polymorphisme en Python

- Polymorphisme des Méthodes
- Polymorphisme des Classes

Le polymorphisme

→ Polymorphisme des méthodes

Plusieurs classes peuvent avoir des méthodes avec le même nom, mais avec des comportements différents.

Exemple

```
class PDF:
    def ouvrir(self):
        print("Ouverture du fichier PDF.")

class Word:
    def ouvrir(self):
        print("Ouverture du document Word.")

# Fonction polymorphique
def ouvrir_fichier(fichier):
    fichier.ouvrir()

pdf = PDF()
word = Word()

ouvrir_fichier(pdf)
ouvrir_fichier(word)
```

```
Ouverture du fichier PDF.
Ouverture du document Word.
```

Le polymorphisme

→ Polymorphisme des classes

On peut utiliser une même méthode sur différents types d'objets.

Exemple

```
class Rectangle:
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur

    def aire(self):
        return self.largeur * self.hauteur

class Cercle:
    def __init__(self, rayon):
        self.rayon = rayon

    def aire(self):
        return 3.14 * self.rayon * self.rayon

# Fonction polymorphique
def afficher_aire(forme):
    print("Aire :", forme.aire())

r = Rectangle(5, 10)
c = Cercle(7)

afficher_aire(r)
afficher_aire(c)
```

```
Aire : 50
Aire : 153.86
```

L’abstraction et les interfaces

L'abstraction consiste à définir une classe qui ne peut pas être instanciée directement et qui impose à ses classes dérivées d'implémenter certaines méthodes.

- Python ne supporte pas directement les classes abstraites, mais il permet de les définir avec le module ***abc*** (***Abstract Base Class***).
- Une classe abstraite est une classe qui contient au moins une méthode abstraite (méthode définie mais non implémentée).
- Une classe abstraite ne peut pas être instanciée directement.

❑ Pourquoi utiliser l’abstraction ?

- Forcer les classes dérivées à implémenter certaines méthodes.
- Standardiser l’implémentation dans différentes classes.
- Cacher les détails d’implémentation et ne montrer que ce qui est essentiel.

Exemple

```
from abc import ABC, abstractmethod  # Importation du module abc

class Forme(ABC):  # Définition d'une classe abstraite
    @abstractmethod
    def aire(self):  # Méthode abstraite
        pass
```

- **`@abstractmethod`** est un décorateur qui marque une méthode comme abstraite.
- Toute classe qui hérite d'une classe abstraite doit obligatoirement redéfinir les méthodes marquées **`@abstractmethod`**, sinon elle ne pourra pas être instanciée.

→ Sans **`@abstractmethod`**, la méthode n'est pas vraiment abstraite et Python n'oblige pas à la redéfinir.

→ Avec @abstractmethod, on force l'implémentation dans les classes dérivées, évitant ainsi les oublis et garantissant la cohérence du code.

Attention : Si la classe enfant n'implémente pas toutes les méthodes abstraites, Python lèvera une erreur.

❑ Implémentation des classes abstraites

Exemple d'Utilisation d'une Classe Abstraite

```python
from abc import ABC, abstractmethod

class Forme(ABC):  # Classe abstraite
    @abstractmethod
    def aire(self):
        pass

class Cercle(Forme):  # Classe concrète qui hérite de Forme
    def __init__(self, rayon):
        self.rayon = rayon

    def aire(self):  # Implémentation de la méthode abstraite
        return 3.14 * self.rayon * self.rayon

# cercle = Forme() - Impossible : on ne peut pas instancier une classe abstraite
c1 = Cercle(5)
print(c1.aire())
```

```
78.5
```

→ Les interfaces

- Une interface est une classe qui définit uniquement des méthodes sans implémentation.
- Contrairement aux classes abstraites, les interfaces ne contiennent généralement aucun attribut et servent uniquement à imposer un comportement aux classes qui les implémentent.
- Une interface est une classe qui définit un ensemble de méthodes que les classes dérivées doivent implémenter.
- En Python, il n’existe pas de mot-clé spécifique pour les interfaces, mais on peut les implémenter avec les classes abstraites contenant uniquement des méthodes abstraites.

❑ Pourquoi utiliser une interface ?

- Garantir que certaines méthodes sont présentes dans une classe.
- Fournir une structure commune à plusieurs classes.
- Faciliter la modularité du code.

→ Les interfaces

❑ Définition d'une interface en Python

```
from abc import ABC, abstractmethod

class Vehicule(ABC):  # Interface
    @abstractmethod
    def demarrer(self):
        pass

    @abstractmethod
    def arreter(self):
        pass
```

→ Les interfaces

❑ Implémentation de l'interface

```
class Voiture(Vehicule):
    def demarrer(self):
        print("La voiture démarre.")
    def arreter(self):
        print("La voiture s'arrête.")

class Moto(Vehicule):
    def demarrer(self):
        print("La moto démarre.")
    def arreter(self):
        print("La moto s'arrête.")

# Création des objets
v = Voiture()
m = Moto()
v.demarrer()
m.demarrer()
```

```
La voiture démarre.
La moto démarre.
```

la classe **`Vehicule`** agit comme une interface, car elle impose aux classes **`Voiture`** et **`Moto`** d'implémenter les méthodes **`demarrer`** et **`arreter`**.

→ Les interfaces multiples

Puisque Python permet l'héritage multiple, une classe peut implémenter plusieurs interfaces.

Exemple

```
from abc import ABC, abstractmethod
class Chargeable(ABC):
    @abstractmethod
    def charger(self):
        pass
class Connectable(ABC):
    @abstractmethod
    def connecter(self):
        pass
class Smartphone(Chargeable, Connectable):  # Implémente deux interfaces
    def charger(self):
        print("Le smartphone est en charge.")

    def connecter(self):
        print("Le smartphone est connecté au Wi-Fi.")

s = Smartphone()
s.charger()
s.connecter()
```

Python n'a pas besoin d'un mot-clé spécial pour les interfaces, il suffit d'hériter de plusieurs classes abstraites

Exercice 1

Un site de e-commerce souhaite gérer plusieurs types de paiements (Carte bancaire, PayPal). Créer un programme en Python pour gérer ces paiements.

1. Définir une interface **`MoyenPaiement`** avec une méthode **`payer(montant)`**, qui devra être implémentée par chaque type de paiement.
2. Créer deux classes :
   - **`CarteBancaire`** : Demande un numéro de carte et simule un paiement.
   - **`PayPal`** : Demande une adresse e-mail et simule un paiement.
3. Tester le programme en créant un panier et en permettant à l'utilisateur de choisir un mode de paiement.

Exercice 2

Une entreprise souhaite un programme pour gérer ses employés et distinguer les managers des employés classiques. Créer un programme en Python permettant de structurer et gérer les employés et managers.

1. Créer une classe **`Employe`** avec les attributs : **`nom`** et **`salaire`**.
2. Ajouter une méthode **`afficher_infos`**() pour afficher le nom et le salaire de l'employé.
3. Créer une classe **`Manager`** qui hérite de **`Employe`** et qui possède en plus :
   - Un attribut **`equipe`** (liste des employés sous sa responsabilité).
   - Une méthode **`ajouter_employe`**(**`emp`**) pour ajouter un employé à l'équipe du manager.
4. Une méthode **`afficher_equipe`**() pour afficher la liste des employés sous sa gestion.
5. Tester le programme en créant des employés et des managers, et en affichant leurs informations.