Chapitre 8
Classes

Ce chapitre introduit la **programmation orientée objet** (POO) : une manière de concevoir un programme en définissant des **classes**, qui servent de modèles pour créer des **objets**. Un objet regroupe des **données** (ses attributs) et des **actions** (ses méthodes). On dit que l'on **modélise** un élément du réel, comme une voiture, une personne ou un élève, pour le représenter dans un programme.

I. Le concept de programmation orientée objet

La programmation orientée objet consiste en la définition et l'interaction de briques logicielles appelées **objets** ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes.

Dès lors, l'étape de **modélisation** revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

> **R** Beaucoup des types que nous avons déjà utilisés sont considérés comme des **Classes** en Python. Par exemple, les listes, les dictionnaires et les chaînes de caractères sont des classes : chaque liste que l'on crée est un **objet** de la classe `list`.

■ Exemple 8.1.
Une liste est un objet de la classe `list` : elle possède des données (ses éléments) et des méthodes comme `append` ou `count`. ■

```
ma_liste = [1, 2, 3]
ma_liste.append(4)
print(ma_liste)
print(ma_liste.count(2))
```

■ Exemple 8.2.

La fonction `type` renvoie la classe d'un objet : elle confirme que les valeurs manipulées appartiennent bien à des classes. ■

```
print(type(ma_liste))
print(type("bonjour"))
print(type({}))
```

■ **Exemple 8.3.**
Un bel exemple d'utilisation de la programmation orientée objet est la **simulation de foule** : chaque personne de la foule est un objet, avec sa position, sa vitesse et son comportement. C'est l'interaction de tous ces objets qui produit le mouvement d'ensemble. Une vidéo de la chaîne *Fouloscopie* (https://www.youtube.com/watch?v=w-Oy4TYDnoQ) vulgarise bien ce concept. ■

À la place de manipuler uniquement des classes et des types directement donnés par Python, on peut **créer directement les objets et les types qui nous intéressent**.

II. Création d'une classe

Pour créer une nouvelle classe, on utilise le mot-clé `class` suivi du nom de la classe (par convention, en `Capitales`). On écrit ensuite son contenu, indenté comme le corps d'une fonction.

■ **Exemple 8.4.**
On définit une classe `Eleve` ; pour l'instant, son corps est vide : l'instruction `pass` ne fait rien et sert uniquement à remplir la classe. ■

```
class Eleve:
    pass
```

> **R** `pass` sera remplacé par des méthodes et des attributs dans la suite. On peut déjà créer des objets de cette classe, mais ils ne contiennent encore rien d'utile.

■ **Exemple 8.5.**
Une classe est un modèle : on peut en créer autant d'exemplaires que l'on veut, exactement comme on crée plusieurs listes différentes. ■

```
eleve_1 = Eleve()
eleve_2 = Eleve()
print(eleve_1)
print(type(eleve_1))
```

III. Attributs

Comme dit précédemment, les classes et les objets sont utiles pour réaliser des modélisations. On prend le problème de modélisation suivant : on veut modéliser des élèves de Seconde qui veulent partir en Première générale, afin de pouvoir faire un programme qui propose des répartitions d'élèves dans des classes.

Première question : que faut-il pour caractériser un tel élève ? On posera qu'il faut connaître :

— son nom ;
— son sexe ;
— ses vœux de spécialité.

Ces caractéristiques sont appelés **attributs de la classe**. Pour ajouter des attributs, on a besoin d'un **constructeur**. Ce constructeur est la fonction (méthode) : `__init__`.

Exemple 8.6.
On complète la classe `Eleve` avec son constructeur `__init__` : il donne une valeur initiale aux attributs de chaque objet créé. ■

```
class Eleve:
    def __init__(self):
        self.nom = "Mettre un nom"
        self.sexe = "Adolescent(e)"
        self.specialite = []
```

Définition 8.1.
Un **objet** est une **instance** d'une classe. Tout comme `"babar"` est une instance d'une chaîne de caractères, un objet sera une instance d'une classe.

Exemple 8.7.
On veut créer un objet élève avec les attributs : `"Kévin"`, `"Homme"`, `["Math","NSI","SES"]`. On affecte à une variable `premier_eleve` un objet de type `Eleve`. Tout d'abord on crée l'objet. ■

```
premier_eleve = Eleve()
```

Exemple 8.8.
Regardons maintenant ses attributs. Les attributs d'un objet sont des variables que l'on peut afficher avec `print`, entre autres. ■

```
print(premier_eleve.nom)
print(premier_eleve.sexe)
print(premier_eleve.specialite)
```

Exemple 8.9.
Pour modifier des attributs, on affecte une nouvelle valeur avec un point entre l'objet et le nom de l'attribut. ■

```
premier_eleve.nom = "Kévin"
premier_eleve.sexe = "Homme"
premier_eleve.specialite = ["Math","NSI","SES"]
```

R On sait que `__init__` est une fonction. Donc elle prend une entrée : ici l'entrée est `self`. Ce dernier représente l'objet en cours de création. C'est grâce à `self` que la méthode sait sur quel objet elle travaille.

Exemple 8.10.
Le constructeur peut recevoir des paramètres : on crée alors chaque objet avec ses propres valeurs, comme on appelle une fonction avec des arguments. Ici, la classe `Point` attend deux coordonnées. ■

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

p = Point(3, 4)
print(p.x)
print(p.y)
```

Exercice 8.1.
L'objectif de cet exercice est de faire une classe `Prof` et de créer deux objets : Gorce et Gibaud, avec les bons attributs.
- Trouvez les caractéristiques d'un professeur de lycée (sur papier).
- Définir la classe `Professeur` (avec son constructeur).
- Créer deux variables de type `Professeur` : une variable sera `Gibaud`, l'autre `Gorce`.
- Changer les attributs de ces deux objets pour que Gibaud et Gorce aient les bons attributs.

■

IV. Les méthodes

Toute la beauté de la programmation orientée objet est que les objets ont :
- des caractéristiques appelés **attributs** ;
- des actions et fonctions appelés **méthodes**.

Les méthodes sont des fonctions internes à une classe. Cela permet aux objets d'agir et d'interagir entre eux.

■ **Exemple 8.11.**
Pour faire une méthode (ici `dire_present` ou `__init__`), on écrit une fonction à l'intérieur de la classe. La méthode `dire_present` affiche le nom de l'élève ; la méthode `__str__` renvoie une chaîne qui décrit l'objet. ■

```
class Eleve:
    def __init__(self):
        self.nom = "Mettre un nom"
        self.sexe = "Adolescent(e)"
        self.specialite = []
    def dire_present(self):
        print(f"{self.nom} Présent !")
    def __str__(self):
        return f"Eleve {self.nom}"
```

R Toutes les méthodes prennent au moins `self` en entrée : c'est le premier paramètre, placé automatiquement par Python lors de l'appel.

■ **Exemple 8.12.**
Pour appeler une méthode, on doit déjà créer l'objet puis appeler la méthode avec un point. Ici, on change d'abord les attributs de l'objet `second_eleve`, puis on appelle `dire_present`. ■

```
second_eleve = Eleve()
second_eleve.nom = "Isma"
second_eleve.sexe = "Femme"
second_eleve.voeux = ["Math","NSI","HLP"]
second_eleve.dire_present()
```

R Dans l'exemple précédent, l'attribut `voeux` n'a pas été défini dans le constructeur : l'affectation l'ajoute à la volée, uniquement sur cet objet. Python le permet, mais c'est une pratique à éviter : on préfère déclarer tous les attributs dans `__init__`, pour que chaque objet ait la même structure.

■ **Exemple 8.13.**
Afficher plus facilement les objets ! La méthode `__str__` permet de `print`er un objet. La méthode `__str__` renvoie un string qui sera affiché quand on imprimera l'objet. ■

```
print(second_eleve)
```

R Pour qu'un objet appelle une méthode, on met un **point** entre l'objet et la méthode. Comme la méthode est une fonction, on met des parenthèses avec les arguments après la méthode. Si la méthode ne prend que `self`, on met des parenthèses vides.

■ **Exemple 8.14.**
Cependant les méthodes peuvent être plus compliquées et faire des actions plus

complexes. Par exemple `__init__` est une méthode, ou `append` qui est une méthode pour les listes et qui permet d'ajouter un élément à la fin d'une liste. On pourrait avoir la classe suivante, dont la méthode `dire_present` prend un texte en argument. ■

```
class Eleve:
    def __init__(self):
        self.nom = "Mettre un nom"
        self.sexe = "Adolescent(e)"
        self.specialite = []
    def dire_present(self,phrase):
        print(phrase)
```

■ **Exemple 8.15.**
On aura alors comme appel de `dire_present` un appel avec un argument entre parenthèses. ■

```
second_eleve = Eleve()
second_eleve.nom = "Rayan"
second_eleve.sexe = "Homme"
second_eleve.voeux = ["Math","NSI","HGGSP"]
second_eleve.dire_present("Je suis ici Monsieur, vous m'avez vu")
```

R On remarque que cette fois `dire_present` prend un argument en entrée : cet argument est `phrase`. Comme pour une fonction classique, on peut passer des valeurs à une méthode.

■ **Exemple 8.16.**
Une méthode peut aussi **modifier** les attributs de l'objet. La classe `Compteur` ci-dessous augmente son attribut `valeur` de 1 à chaque appel. ■

```
class Compteur:
    def __init__(self):
        self.valeur = 0
    def incrementer(self):
        self.valeur = self.valeur + 1
```

■ **Exemple 8.17.**
On crée un compteur, on l'incrémente deux fois, puis on lit son attribut. ■

```
c = Compteur()
c.incrementer()
c.incrementer()
print(c.valeur)
```

■ **Exemple 8.18.**
Une méthode peut **renvoyer** une valeur avec `return`, comme une fonction classique. La classe `Rectangle` calcule l'aire d'un rectangle. ■

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

■ **Exemple 8.19.**
On instancie un rectangle et on affiche son aire. ■

```
r = Rectangle(3, 5)
print(r.aire())
```

■ **Exemple 8.20.**
Une méthode peut utiliser les attributs d'un autre objet passé en argument. Ici, chaque chien connaît son nom et sait aboyer. ■

```
class Chien:
    def __init__(self, nom):
        self.nom = nom
    def aboyer(self):
        print(self.nom + " aboie !")
```

■ **Exemple 8.21.**
On crée deux chiens et on fait aboyer le premier. ■

```
rex = Chien("Rex")
medor = Chien("Medor")
rex.aboyer()
```

■ **Exemple 8.22.**
La classe `Cercle` combine tout cela : ses méthodes `perimetre` et `aire` lisent l'attribut `rayon` et renvoient le résultat du calcul. ■

```
class Cercle:
    def __init__(self, rayon):
        self.rayon = rayon
    def perimetre(self):
        return 2 * 3.14 * self.rayon
    def aire(self):
        return 3.14 * self.rayon ** 2
```

■ **Exemple 8.23.**
On crée un cercle de rayon 5 et on affiche son périmètre et son aire. ■

```
cercle = Cercle(5)
print(cercle.perimetre())
print(cercle.aire())
```

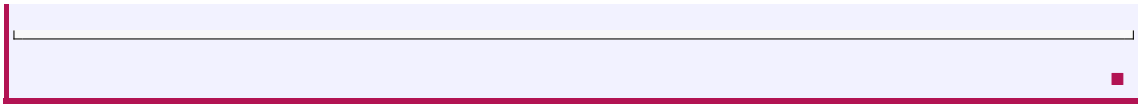
■ **Exercice 8.2.**
Ajouter à la classe `Professeur` une méthode prenant en argument un texte (qui sera un texte d'exercice) et qui affiche cet exercice. ■

■ **Exercice 8.3.**

— Faire une classe `Joueur` avec comme attributs un identifiant (qui sera un entier positif ou nul) et un score (qui sera aussi un entier).
— Ajouter à cette classe `Joueur` une méthode (vous choisirez les arguments de la méthode) qui affiche l'identifiant et le score (dans un message lisible).
— Ajouter une méthode `vol_de_point` qui prend en entrée un autre `Joueur` et qui lui vole un nombre de points aléatoire pris entre 0 et 10. (Attention : les points ne peuvent pas être négatifs.)
— Faire une classe `Groupe_de_Joueurs` dont l'attribut est une liste de Joueurs et qui aura comme méthode `trier_la_liste` (qui trie les joueurs par nombre de points croissant) et une méthode `voler_dans_tous_les_sens` (qui prendra en argument n et qui fera tourner n vols entre joueurs et qui triera ensuite la liste des joueurs).

On rappelle que l'on peut obtenir un entier aléatoire entre 0 et 10 inclus avec le code suivant.

```
import random
points_voles = random.randint(0, 10)
```



V. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : calcul mental d'expressions Python, lecture de code court, prédiction d'affichage, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 8.4.

Types et calculs : répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
2 / 3
```

2. Que vaut l'expression suivante ?

```
17 // 5
```

3. Que vaut l'expression suivante ?

```
"ab" + "cd"
```

4. Que vaut l'expression suivante ?

```
5 <= 3
```

■

■ Exercice 8.5.

Classes : compléter et lire du code.

1. Quelle ligne complète correctement le constructeur ?

```
class Eleve:
    def __init__(self):
        self.nom = "Inconnu"
```

2. Quelle instruction crée un objet de type `Point` de coordonnées `(3, 4)` ?

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

3. Qu'affiche le programme suivant ?

```
p = Point(3, 4)
```

```
print(p.x)
```

4. Combien d'attributs possède la classe `Point` définie ci-dessus ?

■

■ **Exercice 8.6.**

Rappels : boucles et listes.

1. Combien d'itérations effectue la boucle `for i in range(2, 6)` ?
2. Que vaut `t[-1]` pour `t = [5, 8, 2]` ?
3. Que renvoie `len([1, 2, 3, 4])` ?
4. Quelle instruction permet d'ajouter la valeur 9 à la fin d'une liste `t` ?

■

■ **Exercice 8.7.**

Attribut ou méthode : répondre aux questions suivantes.

1. Dans l'expression `v.vitesse`, `vitesse` est-il un attribut ou une méthode ?
2. Parmi les expressions suivantes, laquelle est un appel de méthode ?

```
v = Voiture("Renault", 50)
```

   (a) `v.accelerer()`
   (b) `v.vitesse`
   (c) `v.marque`
   (d) `Voiture`

3. Que représente `self` dans la définition d'une méthode ?
4. Qu'affiche le programme suivant ?

```
class Compteur:
    def __init__(self):
        self.valeur = 0
    def incrementer(self):
        self.valeur = self.valeur + 1

c = Compteur()
c.incrementer()
print(c.valeur)
```

■

■ **Exercice 8.8.**

Rappels : dictionnaires, booléens et compréhension.

1. Que renvoie `d["b"]` pour `d = {"a": 1, "b": 2}` ?
2. Que vaut l'expression `True and False` ?
3. Qu'affiche le programme suivant ?

```
print([x for x in range(6) if x % 3 == 0])
```

4. Que vaut l'expression `not (3 > 1)` ?

■

Exercices d'application

Les exercices d'application créent des classes et manipulent des objets, du plus simple au plus difficile. Les trois premiers reprennent les exercices du chapitre.

■ **Exercice 8.9.**
Définir la classe `Professeur` avec son constructeur, puis créer deux objets `Gorce` et `Gibaud` et leur donner les bons attributs (trouvez d'abord, sur papier, les caractéristiques d'un professeur de lycée).

```
class Professeur:
    def __init__(self):
        # Écrire ici les attributs d'un professeur

Gorce = Professeur()
Gibaud = Professeur()
# Changer les attributs de Gorce et Gibaud
```

■

■ **Exercice 8.10.**
Ajouter à la classe `Professeur` une méthode prenant en argument un texte (qui sera un texte d'exercice) et qui affiche cet exercice. Tester avec un exemple.

```
class Professeur:
    def __init__(self):
        # Attributs déjà définis àl'exercice précédent
    def donner_exercice(self, texte):
        # Écrire ici la méthode
```

■

■ **Exercice 8.11.**
Faire la classe `Joueur` (identifiant, score, méthode d'affichage, méthode `vol_de_point`)

puis la classe `Groupe_de_Joueurs` (liste de joueurs, `trier_la_liste`, `voler_dans_tous_les_sens`) décrites dans le cours. Tester avec un petit groupe de joueurs.

```
import random

class Joueur:
    # Écrire ici la classe Joueur

class Groupe_de_Joueurs:
    # Écrire ici la classe Groupe_de_Joueurs
```

■

■ **Exercice 8.12.**
Définir une classe `Chien` dont le constructeur reçoit un nom, avec une méthode `aboyer` qui affiche `"Ouaf !"`. Créer deux chiens `Rex` et `Medor`, puis faire aboyer `Rex`.

```
class Chien:
    # Écrire ici la classe Chien

rex = Chien("Rex")
medor = Chien("Medor")
rex.aboyer()
```

■

■ **Exercice 8.13.**
Définir une classe `Point` dont le constructeur reçoit deux coordonnées `x` et `y`, avec une méthode `afficher` qui affiche les coordonnées. Créer deux points `a = Point(1, 2)` et `b = Point(4, 6)`, puis afficher le point `a`. On rappelle que la distance entre deux points se calcule avec le théorème de Pythagore.

```
class Point:
    # Écrire ici la classe Point

a = Point(1, 2)
b = Point(4, 6)
a.afficher()
# Calculer la distance entre a et b
```

■

■ **Exercice 8.14.**
Définir une classe `Compte` dont le constructeur reçoit un solde initial, avec les méthodes `deposer(montant)` et `retirer(montant)` qui modifient le solde, et une méthode `afficher` qui affiche le solde. Tester avec un compte à 100, un dépôt de 30 puis un retrait de 50.

```
class Compte:
    # Écrire ici la classe Compte

c = Compte(100)
c.deposer(30)
c.retirer(50)
c.afficher()
```

■

■ **Exercice 8.15.**
Définir une classe `Cercle` dont le constructeur reçoit un rayon, avec les méthodes `perimetre` et `aire` qui **renvoient** les résultats (on prendra $\pi \approx 3.14$). Afficher le périmètre et l'aire d'un cercle de rayon 5.

```
class Cercle:
    # Écrire ici la classe Cercle

cercle = Cercle(5)
print(cercle.perimetre())
print(cercle.aire())
```

■

■ **Exercice 8.16.**
Définir une classe `Livre` dont le constructeur reçoit un titre, un auteur et une année de parution, avec une méthode `description` qui **renvoie** une chaîne du type `"Titre : ... , auteur : ... , annee : ..."`. Créer un livre et afficher sa description.

```
class Livre:
    # Écrire ici la classe Livre

livre = Livre("Le Petit Prince", "Saint-Exupery", 1943)
print(livre.description())
```

■

■ **Exercice 8.17.**
Définir une classe `Compteur` dont le constructeur initialise la valeur à 0, avec une méthode `incrementer` qui augmente la valeur de 1. Créer un compteur, l'incrémenter 5 fois, puis afficher sa valeur.

```
class Compteur:
    # Écrire ici la classe Compteur

c = Compteur()
for i in range(5):
    c.incrementer()
print(c.valeur)
```

■

■ **Exercice 8.18.**
Définir une classe `Eleve` dont le constructeur initialise une liste de notes vide, avec les méthodes `ajouter_note(n)` et `moyenne` qui renvoie la moyenne des notes. Créer un élève, lui ajouter les notes 12, 15 et 9, puis afficher sa moyenne.

```
class Eleve:
    # Écrire ici la classe Eleve

e = Eleve()
e.ajouter_note(12)
e.ajouter_note(15)
e.ajouter_note(9)
print(e.moyenne())
```

■

Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : déroulement de programmes, repérage d'erreurs, questions ouvertes sur les classes et les objets. Ils sont classés du plus abordable au plus exigeant.

■ **Exercice 8.19.**
Le programme suivant provoque une erreur.

```
class Chien:
    def aboyer():
        print("Ouaf")

c = Chien()
```

```
c.aboyer()
```

1. Expliquer pourquoi l'appel `c.aboyer()` lève une erreur `TypeError`.
2. Corriger la définition de la méthode.

■

■ **Exercice 8.20.**

Le programme suivant provoque une erreur.

```
class Compte:
    def __init__(self, solde):
        self.solde = solde
    def afficher(self):
        print(solde)

c = Compte(100)
c.afficher()
```

1. Expliquer pourquoi l'erreur est une `NameError`.
2. Corriger la méthode `afficher`.

■

■ **Exercice 8.21.**

Le programme suivant provoque une erreur.

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

e = Eleve()
```

1. Expliquer pourquoi l'appel `Eleve()` lève une erreur.
2. Écrire un appel correct.

■

■ **Exercice 8.22.**

On exécute le programme suivant.

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

p1 = Personne("Lea", 16)
p2 = Personne("Lea", 16)
p1.age = 17
print(p2.age)
```

1. Qu'affiche ce programme ?
2. Pourquoi modifier `p1` ne modifie-t-il pas `p2` ?

■

■ **Exercice 8.23.**
On veut modéliser une playlist avec une classe `Playlist` dont l'attribut est une liste de titres.

```
class Playlist:
    def __init__(self):
        self.titres = []

    def ajouter(self, titre):
        self.titres.append(titre)

    def nb_titres(self):
        return len(self.titres)
```

1. Créer une playlist, ajouter trois titres, puis afficher `nb_titres()`.
2. Écrire une méthode `dernier_titre` qui renvoie le dernier titre ajouté, sans le retirer de la liste.

■

■ **Exercice 8.24.**
On exécute le programme suivant, sans méthode `__str__`.

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

p = Point(3, 4)
print(p)
```

1. Pourquoi l'affichage ressemble-t-il à `<__main__.Point object at 0x...>` ?
2. Ajouter une méthode `__str__` qui renvoie `"Point(3, 4)"` et vérifier le nouvel affichage.

■