



Programmation Python

Séance 5 — Programmation orientée objet

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/programmation-python/chapitres/05-programmation-orientee-objet.html>

Version du 11 octobre 2026



Version en ligne

Programmation Python

— Séance 5

Programmation orientée objet

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

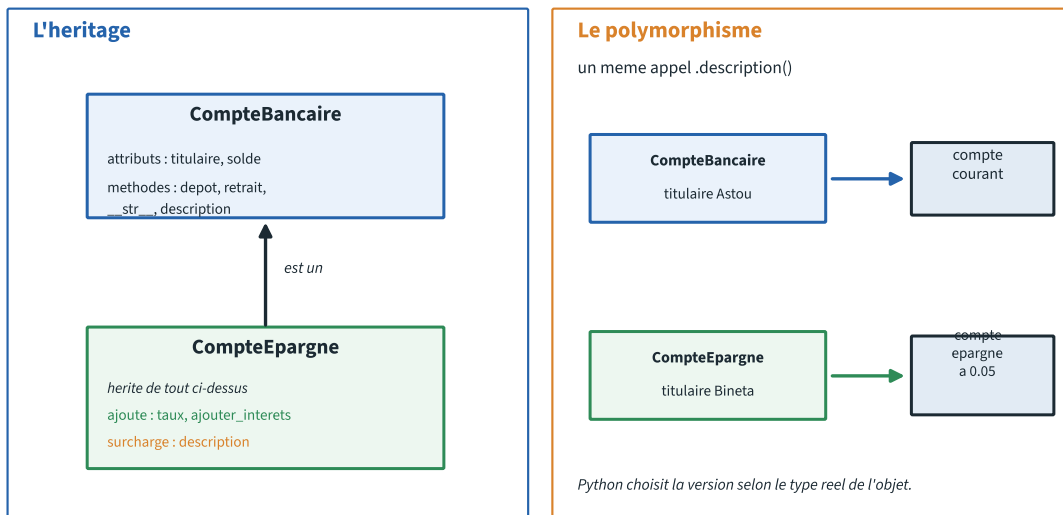
Contenu

Jusqu'ici, les *données* (variables) et les *traitements* (fonctions) étaient séparés. La programmation orientée objet les **réunit** : un objet regroupe des données et les comportements qui les manipulent. C'est le style dans lequel sont écrits `numpy`, `pandas` et la plupart des bibliothèques.

- **Classe et objet** : `class`, `__init__`, `self`, attributs.
- **Méthodes** : les comportements de l'objet ; `__str__` pour l'affichage.
- **Héritage** : spécialiser une classe sans la réécrire.
- **Polymorphisme** : un même appel, un comportement propre à chaque classe.

L'entraînement se fait dans le **notebook**, avec une cellule « À vous » après chaque notion.

Deux mécanismes clés



L'héritage (une classe fille récupère et enrichit une classe mère) et le polymorphisme (un même appel donne un résultat propre à chaque type).

Vous utilisez déjà des objets

L'idée : données + comportements

L'idée en une phrase

Un **objet** réunit des **données** et les **méthodes** (fonctions attachées) qui les manipulent. Une chaîne, une liste *sont* déjà des objets : les méthodes que vous appelez avec un point en sont la preuve.

La POO consiste à définir **nos propres** types d'objets, taillés pour notre problème — un compte bancaire, un patient, un produit — chacun avec ses données et ses comportements.

Des objets que vous connaissez déjà

```
texte = "dakar"  
print(texte.upper())  
notes = [17, 12]  
notes.append(15)  
print(notes)
```

RÉSULTAT →

```
DAKAR  
[17, 12, 15]
```

`.upper()` et `.append()` sont des **méthodes** : des fonctions qui appartiennent à l'objet et agissent sur lui.

Définir une classe

L'idée : la classe, plan de l'objet

L'idée en une phrase

Une **classe** est le **plan** d'un objet : elle décrit ses données et ses méthodes. `__init__` est le **constructeur**, exécuté à la création ; `self` y désigne l'objet en train d'être construit, sur lequel on installe les attributs avec `self.x = ...`.

Analogie

Le plan et l'objet

La **classe** est comme le *plan d'une maison* : un seul plan, mais on peut construire autant de maisons qu'on veut. Chaque maison bâtie est une **instance** : elle suit le plan, mais possède ses propres caractéristiques (sa couleur, ses habitants). `CompteBancaire` est le plan ; `c1`, `c2` sont des comptes concrets, chacun avec son titulaire et son solde.

Définir une classe et créer un objet

```
class CompteBancaire:  
    def __init__(self, titulaire, solde):  
        self.titulaire = titulaire  
        self.solde = solde
```

```
c1 = CompteBancaire("Astou", 5000)  
print(c1.titulaire)  
print(c1.solde)
```

RÉSULTAT →

```
Astou  
5000
```

`c1` est une **instance**. On accède à ses **attributs** avec un point.

Plusieurs instances, attributs modifiables

```
c2 = CompteBancaire("Modou", 12000)
print(c1.titulaire, c1.solde)
print(c2.titulaire, c2.solde)
c2.solde = 20000
print(c2.solde)
```

RÉSULTAT →

```
Astou 5000
Modou 12000
20000
```

Chaque instance a ses propres valeurs ; un attribut se lit et se **modifie** avec un point.

Les méthodes

L'idée : les comportements de l'objet

L'idée en une phrase

Une **méthode** est une fonction définie *dans* la classe. Elle reçoit toujours `self` en premier paramètre, ce qui lui donne accès aux attributs de l'objet (`self.solde`). Une méthode peut **modifier** l'objet, ou **renvoyer** une valeur.

Des méthodes qui modifient l'objet

```
class CompteBancaire:
    def __init__(self, titulaire, solde):
        self.titulaire = titulaire
        self.solde = solde
    def depot(self, montant):
        self.solde = self.solde + montant
    def retrait(self, montant):
        if montant <= self.solde:
            self.solde = self.solde - montant
        else:
            print("Solde insuffisant")

c1 = CompteBancaire("Astou", 5000)
c1.depot(2000)
print(c1.solde)
c1.retrait(1000)
print(c1.solde)
```

Appel de méthode, et sortie

On appelle une méthode sur l'objet : `c1.depot(2000)`. Inutile de passer `self`, Python le fait : `self` sera `c1`.

RÉSULTAT →

```
7000  
6000
```

Un retrait supérieur au solde n'est pas effectué :

```
c1.retrait(50000)  
print(c1.solde)
```

```
Solde insuffisant  
6000
```

Une méthode peut renvoyer une valeur

Renvoyer plutôt que modifier

`peut_retirer` ne change rien : elle **renvoie** un booléen, utilisable dans un test.

```
def peut_retirer(self, montant):  
    return montant <= self.solde
```

```
print(c1.peut_retirer(1000))  
print(c1.peut_retirer(999999))
```

RÉSULTAT →

```
True  
False
```

Un affichage lisible

La méthode spéciale `__str__`

L'idée en une phrase

`__str__` décide ce qu'affiche `print(objet)` : elle **renvoie** une chaîne. On ajoute aussi `description`, qui renvoie un libellé (utile plus loin).

```
def __str__(self):  
    return f"Compte de {self.titulaire} : {self.solde} F"  
def description(self):  
    return "compte courant"
```

```
c1 = CompteBancaire("Astou", 5000)  
print(c1)  
print(c1.description())
```

RÉSULTAT →

```
Compte de Astou : 5000 F  
compte courant
```

L'héritage

L'idée : spécialiser sans réécrire

L'idée en une phrase

L'**héritage** crée une classe *à partir d'une autre* : elle récupère tout et peut **ajouter** ce qui lui est propre. `CompteEpargne(CompteBancaire)` se lit « un compte épargne **est un** compte bancaire, spécialisé ». `super().__init__(...)` réutilise le constructeur du parent.

Créer une classe fille

```
class CompteEpargne(CompteBancaire):
    def __init__(self, titulaire, solde, taux):
        super().__init__(titulaire, solde)
        self.taux = taux
    def ajouter_interets(self):
        self.solde = self.solde + self.solde * self.taux

e1 = CompteEpargne("Bineta", 10000, 0.05)
print(e1)
e1.depot(5000)
e1.ajouter_interets()
print(e1.solde)
```

RÉSULTAT →

```
Compte de Bineta : 10000 F
15750.0
```

Tout le parent est hérité

`print(e1)` utilise le `__str__` **hérité** ; `e1.depot(...)` et `e1.retrait(...)` sont des méthodes **héritées**. Seuls `taux` et `ajouter_interets` sont propres à la classe fille.

```
e1.retrait(2000)
print(e1.solde)
```

RÉSULTAT →

13750.0

Le retrait, écrit une seule fois dans `CompteBancaire`, fonctionne sur le compte épargne sans une ligne de plus.

Le polymorphisme

L'idée : un même appel, des résultats différents

L'idée en une phrase

Surcharger une méthode, c'est la **redéfinir** dans la classe fille. Le **polymorphisme** en découle : un *même* appel de méthode produit un résultat **propre au type réel** de l'objet. Python choisit automatiquement la bonne version.

Surcharger une méthode

```
class CompteEpargne(CompteBancaire):
    def __init__(self, titulaire, solde, taux):
        super().__init__(titulaire, solde)
        self.taux = taux
    def description(self):
        return f"compte epargne a {self.taux}"

comptes = [CompteBancaire("Astou", 5000), CompteEpargne("Bineta", 10000, 0.05)]
for c in comptes:
    print(c.titulaire, ":", c.description())
```

RÉSULTAT →

```
Astou : compte courant
Bineta : compte epargne a 0.05
```

Le polymorphisme à travers une fonction

```
def presenter(compte):  
    print("Il s'agit d'un", compte.description())  
  
presenter(comptes[0])  
presenter(comptes[1])
```

RÉSULTAT →

```
Il s'agit d'un compte courant  
Il s'agit d'un compte epargne a 0.05
```

La fonction `presenter` accepte n'importe quel compte : elle appelle `description()` sans se soucier du type exact. C'est toute la souplesse du polymorphisme.

Synthèse

Pièges fréquents

- **Oublier `self`** : toute méthode le prend en premier paramètre, et l'on écrit `self.attribut` pour lire les données de l'objet.
- **Confondre la classe et l'instance** : `CompteBancaire` est le plan ; `c1 = CompteBancaire(...)` est un objet concret. On appelle les méthodes sur l'instance.
- **Méthode sans parenthèses** : `c1.description` seul désigne la méthode ; il faut `c1.description()` pour l'exécuter.
- **Oublier `super().__init__`** dans une classe fille : les attributs du parent ne sont alors pas installés.

À retenir — Séance 5

L'essentiel

Classe : le plan d'un objet (`class`). **Instance** : un objet concret.

`__init__(self, ...)` : le constructeur ; installe les **attributs** (`self.x = ...`).

self : l'objet courant ; toute méthode le reçoit en premier.

Méthode : fonction dans la classe, appelée par `objet.methode(...)` ; modifie l'objet ou **renvoie** une valeur. `__str__` : la chaîne affichée par `print`.

Héritage : `class Fille(Parent)` récupère tout et ajoute ; `super().__init__(...)` réutilise le constructeur parent.

Polymorphisme : une méthode **surchargée** donne, pour un même appel, un comportement propre à chaque classe.

Prochaine séance

Séance 6 — NumPy

On sait organiser données et comportements en objets. La Séance 6 ouvre la **science des données** avec `numpy` : le **tableau** (`array`), le calcul **vectorisé** (une opération sur tout le tableau, sans boucle) et les agrégations. C'est l'outil sur lequel repose `pandas`.

À faire d'ici là : refaire les cellules « À vous » du notebook de la séance.

Références

- Severance, C. — *Python for Everybody*, 2016.
- Downey, A. — *Think Python*, 2e éd., O'Reilly, 2015.
- Matthes, E. — *Python Crash Course*, 3e éd., No Starch Press, 2023.
- Python Software Foundation — *The Python Tutorial* (Classes), 2024.

Ressources de la séance

- **Notebook** · [TP 5 — Programmation orientée objet](#) (18 Ko)