

Programmation orientée objet : pour aller plus loin

Fiche de synthèse

❗ Important. L'héritage, le polymorphisme et la généricité sont hors programme : ils sont présentés pour la culture informatique.

① LES PRINCIPES DE LA PROGRAMMATION ORIENTÉE OBJET

- **Encapsulation** : les données d'un objet lui appartiennent et ne sont modifiées que par ses méthodes, qui forment son interface.
- **Persistence** : les données d'un objet existent tant que l'objet existe, même quand aucune de ses méthodes ne s'exécute.
- **Héritage** : une classe peut reprendre automatiquement les méthodes d'une autre classe, sans les réécrire.
- **Polymorphisme** : une classe peut manipuler des données de types variés ; une pile, par exemple, peut contenir des entiers comme des chaînes de caractères.
- **Généricité** : une solution écrite pour un problème se réutilise pour des problèmes semblables.

② EN PYTHON, LES TYPES SONT DES CLASSES

La classe d'un objet détermine les méthodes qu'on peut lui appliquer : `type([1, 2])` donne `<class 'list'>`, et la classe `list` possède une méthode `append`, que la classe `tuple` ne possède pas. `(1, 2).append(3)` provoque donc une erreur `AttributeError`.

③ ENCAPSULATION NON GARANTIE

- Un nom qui commence par un tiret bas, comme `_solde`, est interne **par convention** : rien n'empêche d'écrire `compte._solde = -1000`.
- Avec deux tirets bas, `__solde`, Python renomme l'attribut en `_CompteBancaire__solde` : `compte.__solde` échoue hors de la classe, mais le nouveau nom reste accessible. L'accès est généré, pas interdit.

④ UN MODÈLE OBJET DYNAMIQUE

Python permet d'ajouter des attributs à un objet, ou des méthodes à une classe, après leur création.

```
class ObjetVide:
    pass

a = ObjetVide()
a.nom = "Objet 1"  # attribut ajouté
```

```
def affiche(self):
    print(self.nom)

ObjetVide.affiche = affiche  # méthode ajoutée
a.affiche()                 # affiche Objet 1
```

Chaque objet a son propre espace de noms : les attributs créés pour `a` n'existent pas pour un autre objet de la même classe.

⚠ Attention. Tout ce qui est possible n'est pas conseillé. On définit les attributs d'instance dans `__init__`, pour que tous les objets d'une classe aient les mêmes attributs.

⑤ ATTRIBUTS DE CLASSE ET D'INSTANCE

Définition 1 (Attributs de classe et d'instance) Un **attribut d'instance** appartient à un seul objet, comme `a.nom`. Un **attribut de classe** est défini sur la classe et partagé par toutes ses instances, comme `ObjetVide.numero`.

```
b = ObjetVide()
ObjetVide.numero = 12
ObjetVide.numero += 1
print(a.numero)  # 13 : valeur partagée
b.numero += 1    # crée un attribut d'instance
print(a.numero)  # 13
print(b.numero)  # 14
```

L'instruction `b.numero += 1` lit l'attribut de classe, puis crée pour `b` seul un attribut d'instance de même nom. Ce mélange devient vite incompréhensible : on limite les attributs de classe aux cas où ils sont nécessaires.

⑥ HÉRITAGE

Définition 2 (Héritage) Quand une classe `B` **hérite** d'une classe `A`, toutes les méthodes de `A` sont disponibles dans `B`, qui peut en redéfinir certaines. On écrit la classe mère entre parenthèses : `class B(A):`.

```
class Polynome:
    def __init__(self, coefs):
        self.coefs = coefs  # degré décroissant

    def eval(self, x):
        px = 0
        for c in self.coefs:
            px = px * x + c
```

```

        return px

class Affine(Polynome):
    def __init__(self, a, b):
        self.coefs = [a, b]

f = Affine(2, 4)
print(f.eval(12))    # 28, méthode héritée

```

La classe `Affine` ne redéfinit que le constructeur : la méthode `eval`, et toutes les autres méthodes de `Polynome`, s'appliquent à ses instances.

À retenir.

- En Python, les types de base sont des classes.
- L'encapsulation n'est qu'une convention : les attributs internes restent accessibles.
- Un attribut de classe est partagé par toutes les instances ; un attribut d'instance est propre à un objet et se définit dans `__init__`.
- Une classe fille hérite des méthodes de sa classe mère et peut les redéfinir.