Programmation orientée objet : pour aller plus loin

Programmation orientée objet

Note : les notions d'héritage, de polymorphisme et de généricité présentées dans cette fiche sont **hors programme** de terminale NSI. Elles sont proposées à titre de culture informatique et d'approfondissement.

1 Les principes de la programmation orientée objet

La notion d'objet en programmation prolonge celle de type abstrait algébrique. Un objet regroupe données et traitements. Voici les grands principes de la POO :

— Une **classe** est un modèle de création d'instances.

— Les objets « discutent » entre eux par envoi de messages.

— **Persistance** : les données d'un objet continuent d'exister même lorsque l'objet n'est pas « actif ».

— **Encapsulation** : les données d'un objet lui appartiennent et peuvent être protégées des interventions extérieures.

— **Héritage** : une classe peut disposer automatiquement des méthodes d'autres classes, ce qui permet la réutilisation du code.

— **Polymorphisme** : une classe peut manipuler des données dont le type n'est pas précisé dans la définition de la classe.

— **Généricité** : grâce à l'héritage et au polymorphisme, l'étude d'un problème conduit à une solution abstraite facilitant la résolution de problèmes similaires.

2 En Python, les types sont des classes

La classe détermine les attributs et les méthodes applicables à un objet.

```
objet = [1, 2, 3]
print(type(objet))    # <class 'list'>
objet.append(4)       # OK : list a la méthode append

objet = (1, 2, 3)
print(type(objet))    # <class 'tuple'>
objet.append(4)       # Erreur : tuple n'a pas append
```

3 En Python, tout est dynamique

3.1 Des objets « déformés »

On peut ajouter dynamiquement des attributs et des méthodes à une classe existante ou à des objets déjà créés.

```
class ObjetVide:
```

```
    pass

un_objet = ObjetVide()
un_objet.nom = "Objet 1"
un_objet.description = "Mon premier objet"

autre_objet = ObjetVide()
autre_objet.nom = "Objet 2"
autre_objet.prix = 10000
```

Chaque objet définit un espace de nommage distinct.

3.2 Encapsulation non garantie

En Python, un attribut dont le nom commence par un tiret bas (`_solde`) est considéré par convention comme interne, mais rien n'empêche d'y accéder. Avec deux tirets bas (`__solde`), Python renomme l'attribut en `_CompteBancaire__solde` : l'écriture `compte.__solde` échoue alors hors de la classe (erreur `AttributeError`), mais `compte._CompteBancaire__solde` reste accessible. L'accès est gêné, pas interdit.

3.3 Attributs de classe

```
ObjetVide.numero = 12
print(un_objet.numero)       # 12

ObjetVide.numero += 1
print(un_objet.numero)       # 13
print(autre_objet.numero)    # 13
```

Attention : le mélange d'attributs de classe et d'attributs d'instance rend rapidement le code incompréhensible. Il est conseillé de ne définir les attributs d'instance qu'à l'intérieur de `__init__`.

④ Héritage

Complément hors programme : l'héritage n'est pas exigible au baccalauréat ; il est présenté ici parce qu'il est omniprésent dans les bibliothèques Python.

Quand une classe B hérite d'une classe A, toutes les méthodes de A sont disponibles dans B, qui peut éventuellement en redéfinir certaines.

4.1 Exemple : polynômes et fonctions affines

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

    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]
```

On peut utiliser la classe `Affine` avec toutes les méthodes héritées de `Polynome` :

```
f = Affine(2, 4)
f.eval(12)    # 28
```