Programmation orientée objet

Résumé

Jusqu'ici, on séparait les données (dans des variables) et les traitements (dans des fonctions). Mais dans le monde réel, les deux sont liés : un compte bancaire a un solde *et* des opérations (déposer, retirer) ; un personnage de jeu a des points de vie *et* des actions (attaquer, se soigner). La **programmation orientée objet** (POO) réunit données et traitements dans un même concept : la **classe**. Une classe est un modèle, et chaque **objet** est un exemplaire concret de ce modèle. C'est la façon standard d'organiser les programmes complexes, utilisée dans tous les langages modernes.

Prérequis : Les fonctions, Les dictionnaires.

① Mémo

Définition 1 (Classe, objet, attribut, méthode)

Classe Modèle (plan) décrivant les données (attributs) et les comportements (méthodes) d'un type d'objet.
Objet (instance) Exemplaire concret créé à partir d'une classe.
Attribut Variable associée à un objet, accessible avec `objet.attribut`.
Méthode Fonction définie dans une classe, appelée avec `objet.methode()`.
`self` Référence à l'objet courant, premier paramètre de toute méthode.

Syntaxe d'une classe

```
class NomClasse:
    def __init__(self, param1, param2):
        self.attribut1 = param1
        self.attribut2 = param2

    def methode(self):
        return self.attribut1
```

- `__init__` est le constructeur : appelé automatiquement à la création.
- `__str__` définit l'affichage de l'objet par `print()`.
- `__repr__` définit la représentation dans la console.
- `__eq__` définit l'égalité avec `==`.

Définition 2 (Encapsulation)

On peut distinguer les attributs et méthodes :

- **publics** : accessibles partout (convention : pas de préfixe) ;
- **privés** (convention) : préfixés par `_` (avertissement) ou `__` (name mangling).

En pratique au programme de NSI, on utilise des accesseurs (`get_`) et des mutateurs (`set_`) pour contrôler l'accès aux attributs.

⚠ Pièges fréquents

- Oublier `self` comme premier paramètre des méthodes.
- Confondre la classe et l'instance : `Cercle` est une classe, `c = Cercle(5)` est une instance.
- Oublier les parenthèses à l'instanciation : `Cercle` (la classe) ≠ `Cercle()` (une instance).
- Modifier un attribut de classe partagé par toutes les instances (variable de classe vs d'instance).

⚠ Erreurs classiques

Code erroné	Code correct	Explication

`def aire(rayon):`
`  return ...`
	`def aire(self):`
	`  return ... self`
	`.rayon ...`
		Oublier `self` comme premier paramètre provoque un `TypeError` à l'appel : Python passe automatiquement l'instance en premier paramètre effectif.

`def __init__(self,`
`notes=[]):`
	`def __init__(self,`
	`notes=None):`
	`  if notes is None:`
	`  self.notes = []`
		Un paramètre mutable par défaut est partagé entre toutes les instances : modifier la liste d'un objet modifie celle de tous les autres.

`c = Cercle` (sans parenthèses)	`c = Cercle(5)`	Sans parenthèses, `c` est la classe elle-même, pas une instance. L'appel `c.aire()` échoue car il n'y a pas d'objet sur lequel appliquer la méthode.

`class Chien:`
`  pattes = 4`
puis `Chien.pattes =`
`3`
	Utiliser un attribut d'instance :
	`self.pattes = 4`
		Un attribut de classe est partagé par toutes les instances : le modifier via la classe change la valeur pour tous les objets existants.

② Exemples

Programme 1 · Classe Cercle

```python
import math

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

    def aire(self):
        return math.pi * self.rayon ** 2

    def perimetre(self):
        return 2 * math.pi * self.rayon

    def __str__(self):
        return f"Cercle de rayon {self.rayon}"

    def __eq__(self, other):
        return self.rayon == other.rayon

c1 = Cercle(5)
c2 = Cercle(3)
print(c1)              # Cercle de rayon 5
print(c1.aire())       # 78.539...
print(c1 == c2)        # False
print(c1 == Cercle(5)) # True (grâce à __eq__)
```

Programme 2 · Classe CompteBancaire (encapsulation)

```python
class CompteBancaire:
    def __init__(self, titulaire, solde=0):
        self._titulaire = titulaire
        self._solde = solde

    def get_solde(self):
        return self._solde

    def deposer(self, montant):
        if montant > 0:
            self._solde += montant

    def retirer(self, montant):
        if 0 < montant <= self._solde:
            self._solde -= montant
            return True
        return False

```

```
    def __str__(self):
        return f"Compte de {self._titulaire} : {self._solde} €"
```

Programme 3 · Classe Carte (pour un jeu de cartes)

```
class Carte:
    def __init__(self, valeur, couleur):
        self.valeur = valeur     # "As", "2", ..., "Roi"
        self.couleur = couleur   # "Pique", "œCur", ...

    def __str__(self):
        return f"{self.valeur} de {self.couleur}"

    def __repr__(self):
        return f"Carte('{self.valeur}', '{self.couleur}')"
```

③ Exercices

Exercice 1 *Classe Rectangle* Créer une classe `Rectangle` avec :

- Un constructeur prenant la largeur et la hauteur ;
- Une méthode `aire()` ;
- Une méthode `perimetre()` ;
- Une méthode `est_carre()` renvoyant `True` si le rectangle est un carré ;
- Une méthode `__str__`.

Exercice 2 *Classe Fraction* Créer une classe `Fraction` représentant une fraction irréductible avec :

- Un constructeur prenant numérateur et dénominateur, qui simplifie la fraction ;
- Une méthode `__str__` affichant par exemple `"3/4"` ;
- Une méthode `__add__(self, other)` pour additionner deux fractions ;
- Une méthode `__eq__(self, other)` pour tester l'égalité.

On pourra utiliser `math.gcd` pour le PGCD.

Exercice 3 *Classe Eleve* Créer une classe `Eleve` avec :

- Un constructeur prenant le nom et une liste de notes (vide par défaut) ;
- Une méthode `ajouter_note(note)` qui ajoute une note entre 0 et 20 ;
- Une méthode `moyenne()` ;
- Une méthode `meilleure_note()` ;
- Une méthode `__str__` affichant le nom et la moyenne.

Attention : ne pas utiliser une liste mutable comme valeur par défaut !

Solutions des exercices

Corrigé de l'exercice 1.

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

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

    def perimetre(self):
        return 2 * (self.largeur + self.hauteur)

    def est_carre(self):
        return self.largeur == self.hauteur

    def __str__(self):
        return f"Rectangle {self.largeur} × {self.hauteur}"

r = Rectangle(4, 6)
print(r.aire())       # 24
print(r.perimetre())  # 20
print(r.est_carre())  # False
c = Rectangle(5, 5)
print(c.est_carre())  # True
```

Corrigé de l'exercice 2.

```python
import math

class Fraction:
    def __init__(self, num, den):
        if den == 0:
            raise ValueError("Dénominateur nul")
        if den < 0:     # signe au numérateur
            num, den = -num, -den
        g = math.gcd(abs(num), den)
        self.num = num // g
        self.den = den // g

    def __str__(self):
        if self.den == 1:
            return str(self.num)
        return f"{self.num}/{self.den}"

    def __add__(self, other):
        return Fraction(
```

```
            self.num * other.den + other.num * self.den,
            self.den * other.den
        )

    def __eq__(self, other):
        return self.num == other.num and self.den == other.den

a = Fraction(2, 4)    # simplifié en 1/2
b = Fraction(1, 3)
print(a)              # 1/2
print(a + b)          # 5/6
print(a == Fraction(3, 6))  # True
```

Corrigé de l'exercice 3.

```
class Eleve:
    def __init__(self, nom, notes=None):
        self.nom = nom
        self.notes = notes if notes is not None else []

    def ajouter_note(self, note):
        if 0 <= note <= 20:
            self.notes.append(note)

    def moyenne(self):
        if len(self.notes) == 0:
            return 0
        return sum(self.notes) / len(self.notes)

    def meilleure_note(self):
        if len(self.notes) == 0:
            return None
        return max(self.notes)

    def __str__(self):
        return f"{self.nom} (moyenne : {self.moyenne():.1f})"
```

Piège évité : écrire `def __init__(self, nom, notes=[])` partagerait la même liste entre toutes les instances créées sans paramètre effectif.

On utilise `None` comme sentinelle et on crée une nouvelle liste dans le constructeur.