# TP Chapitre 0 : POO

**Objectifs du TP**

**Objectifs du TP.** À l'issue de cette séance, tu dois être capable de :

- distinguer les notions de classe, d'objet (instance), d'attribut et de méthode ;
- définir une classe avec le mot-clé `class` et son constructeur `__init__` ;
- créer des objets (instancier une classe) et accéder à leurs attributs ;
- modifier les attributs d'un objet ;
- définir et appeler des méthodes, avec ou sans arguments, en utilisant `self` ;
- écrire une méthode `__str__` pour afficher proprement un objet ;
- concevoir et écrire un petit programme orienté objet à partir d'une spécification (entrées, traitement, sorties).

**Consignes générales.** Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. En programmation orientée objet, vérifie en particulier : le mot-clé `class`, le paramètre `self` de chaque méthode, le point entre l'objet et la méthode, et les parenthèses après le nom de la méthode. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

## Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

### Exercice A1 — Ma première classe : le porte-monnaie (`class`, constructeur, attributs)

Tu veux modéliser ton porte-monnaie avec la programmation orientée objet. Un porte-monnaie possède deux caractéristiques : le nom de son propriétaire (le *titulaire*) et la somme d'argent qu'il contient (le *montant*).

Écrire la classe `PorteMonnaie` : son constructeur reçoit le titulaire et le montant, et les range dans les attributs de l'objet (avec `self`). Créer ensuite un objet, puis afficher ses attributs.

```
class PorteMonnaie:
    def __init__(self, titulaire, montant):
        # A COMPLETER : creer les attributs titulaire et montant

monnaie = PorteMonnaie("Aya", 25)
print(monnaie.titulaire)
# A COMPLETER : afficher le montant du porte-monnaie
```

**Point de validation**

Vérifie ton programme : il doit afficher `Aya` puis `25`.

### Exercice A2 — Le compte bancaire : la méthode `deposer` (méthodes et `self`)

On modélise un compte bancaire : chaque compte possède un titulaire et un solde. La classe `CompteBancaire` ci-dessous contient déjà son constructeur ; il reste à écrire la méthode `deposer`, qui ajoute une somme

au solde.

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

    def deposer(self, somme):
        # A COMPLETER : ajouter somme au solde

compte = CompteBancaire("Leo", 100)
compte.deposer(50)
print(compte.solde)
```

**Point de validation**

Vérifie ton programme : il doit afficher `150`.

### Exercice A3 — Le retrait contrôlé (méthode avec condition)

La méthode `retirer` retire une somme du solde, mais uniquement si le solde reste « positif ou nul ». Sinon, elle affiche un message d'erreur et ne modifie pas le solde.

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

    def retirer(self, somme):
        # A COMPLETER : si le solde reste positif ou nul, retirer ;
        # sinon afficher "Retrait refuse : solde insuffisant."

compte = CompteBancaire("Leo", 100)
compte.retirer(130)
compte.retirer(40)
print(compte.solde)
```

**Point de validation**

Vérifie ton programme : il doit afficher `Retrait refuse : solde insuffisant.` puis `60`. (Le retrait de 130 est refusé car le solde deviendrait négatif ; celui de 40 est accepté : $100 - 40 = 60$.)

### Exercice A4 — Afficher un compte proprement (méthode `__str__`)

La méthode `__str__` renvoie une chaîne de caractères décrivant l'objet : c'est cette chaîne qui est affichée quand on écrit `print(compte)`. Compléter la méthode pour qu'elle renvoie la description du compte.

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

    def __str__(self):
        # A COMPLETER : renvoyer "Compte de Leo : 100 euros"
        # avec une f-string utilisant self.titulaire et self.solde

```

```
compte = CompteBancaire("Leo", 100)
print(compte)
```

**Point de validation**

Vérifie ton programme : il doit afficher `Compte de Leo : 100 euros`.

## Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

### Exercice B1 — La caisse du petit commerce : classe `Article` (méthodes et `return`)

Un magasin vend des articles au prix hors taxes et applique la TVA à 20 % en caisse. Le gérant veut un petit programme orienté objet pour calculer les prix TTC et appliquer des remises fidélité.

**Énoncé.** Écrire un programme qui définit la classe `Article` avec :

— les attributs `nom` et `prix_ht`, initialisés par le constructeur ;

— une méthode `prix_ttc()` qui *renvoie* le prix TTC (le prix HT multiplié par 1,2) ;

— une méthode `remise(pourcent)` qui réduit le prix hors taxes de `pourcent` % (elle modifie l'attribut `prix_ht`).

Le programme crée ensuite deux articles — une carte graphique à 250 € HT et un clavier à 40 € HT —, applique une remise de 10 % sur la carte graphique, puis affiche pour chaque article son nom et son prix TTC.

**Spécification.**

— *Entrées* : aucune saisie au clavier ; les deux articles sont créés directement dans le programme.

— *Traitement* : instancier deux objets `Article` ; appliquer la remise sur la carte graphique ; calculer le prix TTC de chaque article avec la méthode `prix_ttc`.

— *Sorties* : deux lignes, une par article : le nom puis le prix TTC en euros.

— *Contraintes* : classe `Article` ; méthode `prix_ttc()` sans argument ; méthode `remise(pourcent)` qui modifie l'attribut `prix_ht` ; valeurs de test de l'énoncé.

**Pour t'aider — questions de conception (sans donner le code)**

— Quels attributs pour un article ? Pourquoi le prix TTC est-il calculé par une méthode plutôt que rangé dans un attribut ?

— Pourquoi la méthode `remise` doit-elle *modifier* l'attribut `prix_ht` (et pas seulement afficher un nouveau prix) ?

— Que fait le mot-clé `return` dans la méthode `prix_ttc` ? Que se passerait-il si on affichait le résultat avec `print` à la place ?

— Peut-on appeler `prix_ttc()` avant d'avoir créé un objet `Article` ? Pourquoi ?

**Point de validation**

Vérifie ton programme : il doit afficher `Carte graphique : 270.0 euros` puis `Clavier : 48.0 euros`.

### Exercice B2 — La conversion de devises : classe `Convertisseur`

Tu pars en voyage et tu veux convertir des euros dans la monnaie locale. Chaque convertisseur est caractérisé par le nom de la devise et son taux de change (le nombre d'unités étrangères obtenues pour 1 euro).

**Énoncé.** Écrire un programme qui définit la classe `Convertisseur` avec :

— les attributs `devise` et `taux`, initialisés par le constructeur ;

— une méthode `convertir(montant)` qui *renvoie* le montant converti (`montant` × `taux`).

Le programme crée ensuite deux convertisseurs (dollar : taux 1,08 ; yen : taux 162,0), puis affiche la conversion de 50 € en dollars et de 100 € en yens.

**Spécification.**

— *Entrées* : aucune saisie au clavier ; les deux convertisseurs sont créés directement dans le programme.

— *Traitement* : instancier deux objets `Convertisseur` ; appeler la méthode `convertir` sur chacun.

— *Sorties* : la conversion de 50 € en dollars puis celle de 100 € en yens (le texte des messages est libre, les valeurs doivent être exactes).

— *Contraintes* : classe `Convertisseur` ; méthode `convertir(montant)` ; les valeurs de test de l'énoncé.

**Pour t'aider — questions de conception (sans donner le code)**

— Quelle est la différence entre un *attribut* (`taux`) et un *paramètre* de méthode (`montant`) ? Où chacun est-il connu ?

— Pourquoi `convertir` doit-elle *renvoyer* la valeur (avec `return`) plutôt que l'afficher directement ?

— Comment créer deux convertisseurs indépendants à partir de la même classe ?

— Que vaut `dollar.taux` après la création de l'objet ? Comment le vérifier en une ligne ?

**Point de validation**

Vérifie ton programme : 50 € doivent donner `54.0` dollars et 100 € doivent donner `16200.0` yens.

### Exercice B3 — L'abonnement de streaming : classe `Abonnement` (`__str__`)

Tu souscris des abonnements (streaming, salle de sport, forfait téléphone…). Chaque abonnement a un nom, un prix mensuel et une durée en mois.

**Énoncé.** Écrire un programme qui définit la classe `Abonnement` avec :

— les attributs `nom`, `prix_mensuel` et `nb_mois`, initialisés par le constructeur ;

— une méthode `cout_total()` qui *renvoie* le coût total (`prix_mensuel` × `nb_mois`) ;

— une méthode `__str__` qui renvoie une chaîne décrivant l'abonnement : nom, prix mensuel et durée.

Le programme crée ensuite deux abonnements — Netflix à 12 €/mois pendant 12 mois et Salle de sport à 30 €/mois pendant 10 mois —, affiche la description de chacun avec `print(abonnement)`, puis son coût total.

**Spécification.**

— *Entrées* : aucune saisie au clavier ; les deux abonnements sont créés directement dans le programme.

— *Traitement* : instancier deux objets `Abonnement` ; calculer le coût total de chacun avec la méthode `cout_total`.

— *Sorties* : pour chaque abonnement, sa description (via `print(abonnement)`) puis son coût total.

— *Contraintes* : méthode `cout_total()` sans argument ; méthode `__str__` ; les valeurs de test de l'énoncé.

**Pour t'aider — questions de conception (sans donner le code)**

— Que fait Python quand on écrit `print(abonnement)` ? À quoi sert la méthode `__str__` ?

— Pourquoi `__str__` doit-elle *renvoyer* une chaîne (avec `return`) et pas l'afficher avec `print` ?

— Quelle formule pour le coût total ? Pourquoi le coût total n'est-il pas rangé dans un attribut ?

— Quel est le coût total de chaque abonnement de l'énoncé ? (calcule d'abord à la main)

**Point de validation**

Vérifie ton programme : le coût total de Netflix doit être `144` euros et celui de Salle de sport `300` euros. La description de chaque abonnement doit contenir le nom, le prix mensuel et la durée.

### Exercice B4 — Le portefeuille d'actions : classes `Action` et `Portefeuille`

Un investisseur suit la valeur de son portefeuille d'actions. Chaque action est modélisée par un objet : le nom de l'entreprise, la quantité possédée et le prix d'achat unitaire. Le portefeuille est modélisé par un objet qui contient une *liste* d'actions.

**Énoncé.** Écrire un programme qui :

1. définit la classe `Action` (attributs `nom`, `quantite`, `prix_achat`) avec une méthode `valeur()` qui renvoie `quantite` × `prix_achat` et une méthode `__str__` qui décrit l'action ;
2. définit la classe `Portefeuille` (attribut `actions`, une liste) avec une méthode `ajouter(action)` qui ajoute une action à la liste et une méthode `valeur_totale()` qui renvoie la somme des valeurs de toutes les actions ;
3. crée deux actions — 10 actions TotalEnergies à 58 € et 5 actions Airbus à 140 € —, les ajoute au portefeuille, puis affiche le contenu du portefeuille et sa valeur totale.

**Spécification.**

— *Entrées* : aucune saisie au clavier ; les actions et le portefeuille sont créés directement dans le programme.

— *Traitement* : instancier des objets `Action` et `Portefeuille` ; ajouter les actions ; sommer leurs valeurs.

— *Sorties* : la description de chaque action (sa méthode `__str__`), puis la valeur totale du portefeuille.

— *Contraintes* : deux classes ; la méthode `ajouter` utilise `append` ; la méthode `valeur_totale` utilise une boucle `for` et appelle la méthode `valeur` de chaque action ; les valeurs de test de l'énoncé.

**Pour t'aider — questions de conception (sans donner le code)**

— Pourquoi faut-il deux classes ? Que modélise chacune ?

— Quel est le type de l'attribut `actions` ? Quelle méthode de la classe `list` permet d'ajouter un élément ?

— Pourquoi `valeur_totale` appelle-t-elle la méthode `valeur` de chaque objet `Action` plutôt

que de recalculer la formule ?

— Que doit afficher le programme si le portefeuille est vide ? (réponds en une phrase)

**Point de validation**

Vérifie ton programme : il doit afficher la description des deux actions, puis `Valeur totale du portefeuille : 1280 euros` ($10 \times 58 + 5 \times 140 = 1280$).

## Partie C — Exercice de réflexion (à finir à la maison)

**À finir à la maison**

**Mon gestionnaire de budget mensuel.**

**Contexte.** Tu veux suivre tes dépenses du mois pour ne pas dépasser ton budget. Chaque dépense a un libellé (ex. « Courses »), une catégorie (ex. « alimentation », « transport », « loisirs ») et un montant. Ton budget mensuel est un ensemble de dépenses, avec un plafond à ne pas dépasser.

**Objectif.** Écrire un programme orienté objet qui modélise ce budget et réponde à des questions comme : combien ai-je déjà dépensé ? Me reste-t-il de la marge avant le plafond ? Ai-je dépassé mon budget ? (bonus : combien par catégorie ?)

**Questions à se poser avant de coder.**

— Quelles classes créer ? Quels attributs pour une dépense ? Pour le budget ?

— Quelles méthodes sont utiles : ajouter une dépense, calculer le total, calculer le reste, afficher le budget ?

— Comment afficher proprement une dépense et le budget (méthode `__str__` ou méthode d'affichage) ?

— Comment vérifier que le programme est juste ? (penser à un cas simple calculable à la main)

**Étapes suggérées.**

1. Version 0 : classe `Depense` seule (attributs + `__str__`) ; créer deux ou trois dépenses et les afficher.
2. Version 1 : classe `Budget` avec une liste de dépenses et une méthode `ajouter` ; afficher la liste des dépenses.
3. Version 2 : méthode `total()` ; vérifier avec des dépenses connues (ex. 85,5 + 12 + 40 = 137,5).
4. Version 3 : attribut `plafond` et méthode `reste()` ; afficher un message si le budget est dépassé.
5. Bonus : méthode `total_par_categorie()` qui renvoie le total des dépenses d'une catégorie donnée.

**Contraintes.** Montants strictement positifs ; programme testé avec des valeurs calculables à la main et commenté ; les messages peuvent être écrits sans accents (ex. `Depense`, `total`) si ton éditeur pose problème.

*Fin du TP — la partie C est à finir à la maison.*