TP Chapitre 8 : Classes

Objectifs du TP

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

- expliquer ce qu'est une **classe**, un **objet** (une instance) et une modélisation du réel ;
- créer une classe avec le mot-clé `class` et son constructeur `__init__` ;
- utiliser `self` et les attributs : lire et modifier un attribut avec le point (ex. `objet.attribut`) ;
- écrire et appeler des **méthodes**, avec ou sans argument, qu'elles modifient l'objet ou qu'elles renvoient une valeur (`return`) ;
- utiliser la méthode spéciale `__str__` pour décrire proprement un objet ;
- identifier les erreurs classiques : oubli de `self`, attribut manquant, appel de méthode sans parenthèses ;
- concevoir 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. Une classe se définit avec le mot-clé `class` ; un objet (une instance) se crée en appelant la classe comme une fonction, par exemple `mon_objet = MaClasse()`. 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 — Le compte bancaire (attributs et méthodes)

Une banque modélise les comptes courants de ses clients. Chaque compte possède un solde : c'est son attribut. On veut pouvoir déposer de l'argent, en retirer, et afficher le solde. Complète la classe `Compte` : le constructeur fixe le solde initial ; chaque méthode commence par `self` pour agir sur l'objet courant.

```
class Compte:
    def __init__(self, solde_initial):
        self.solde = solde_initial

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

    def retirer(self, montant):
        # A COMPLETER : soustraire montant du solde

    def afficher(self):
        # A COMPLETER : afficher "Solde : ... euros"

c = Compte(100)
c.deposer(30)
c.retirer(50)
c.afficher()
```

Point de validation

Vérifie ton programme : il doit afficher `Solde : 80 euros`. Teste aussi avec un dépôt de 0 : le solde ne doit pas changer.

Exercice A2 — Le prix TTC (méthode qui renvoie une valeur)

Un magasin affiche ses prix hors taxes et applique la TVA à 20 % en caisse : le prix TTC vaut prix HT × 1,2. Complète la classe `Article` : le constructeur enregistre le nom et le prix hors taxes ; la méthode `prix_ttc` doit *renvoyer* le prix TTC avec `return`, sans modifier le prix.

```
class Article:
    def __init__(self, nom, prix_ht):
        self.nom = nom
        self.prix_ht = prix_ht

    def prix_ttc(self):
        # A COMPLETER : renvoyer le prix_ht multiplie par 1.2
        # (pense au mot-cle return)

article = Article("Cahier", 25)
print(article.prix_ttc())
```

Point de validation

Vérifie ton programme : il doit afficher `30.0` pour un cahier à 25 € HT. Teste aussi avec un stylo à 2 € : il doit afficher `2.4`.

Exercice A3 — Le livret d'épargne (méthodes qui modifient et `__str__`)

Léa place 200 € sur un livret rémunéré à 3 % par an. Complète la classe `Livret` : `interets` renvoie le montant des intérêts de l'année (solde × taux) ; `appliquer_interets` ajoute ces intérêts au solde ; la méthode spéciale `__str__` renvoie une chaîne de caractères qui décrit l'objet, pour qu'un simple `print(livret)` affiche une phrase lisible.

```
class Livret:
    def __init__(self, solde, taux):
        self.solde = solde
        self.taux = taux

    def interets(self):
        # A COMPLETER : renvoyer solde * taux

    def appliquer_interets(self):
        # A COMPLETER : ajouter les interets au solde

    def __str__(self):
        # A COMPLETER : renvoyer la chaine "Livret : ... euros"

livret = Livret(200, 0.03)
print(livret.interets())
livret.appliquer_interets()
print(livret)
```

Point de validation

Vérifie ton programme : il doit afficher `6.0` (les intérêts de la première année), puis `Livret : 206.0 euros` (le solde après application des intérêts).

Exercice A4 — Le panier d'achat (attribut liste et boucle)

Un site marchand modélise le panier d'un client par une liste de prix. Complète la classe `Panier` : le constructeur crée une liste vide ; `ajouter` range un nouveau prix dans la liste ; `total` calcule la somme des prix avec une boucle `for` ; `nb_articles` renvoie le nombre d'articles.

```
class Panier:
    def __init__(self):
        self.prix = []

    def ajouter(self, montant):
        # A COMPLETER : ajouter montant a la liste self.prix

    def total(self):
        # A COMPLETER : parcourir self.prix avec une boucle for
        # et additionner tous les prix (accumulateur initialise a 0)
        # Pense a arrondir avec round(total, 2)

    def nb_articles(self):
        # A COMPLETER : renvoyer le nombre de prix de la liste

panier = Panier()
panier.ajouter(12.50)
panier.ajouter(7)
panier.ajouter(9.99)
print(panier.total())
print(panier.nb_articles())
```

Point de validation

Vérifie ton programme : il doit afficher un total d'environ `29.49` (un léger écart du type `29.490000000000002` est normal sans arrondi, c'est la représentation des flottants) puis `3`.

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 — L'abonnement (classe simple)

Un service de streaming vidéo facture 13,49 € par mois. On veut modéliser un abonnement : son nom, son prix mensuel, et pouvoir calculer son coût annuel.

Énoncé. Écrire un programme qui demande le nom de l'abonnement et son prix mensuel, crée un objet de la classe `Abonnement`, puis affiche une description (nom et prix mensuel) et le coût annuel (prix mensuel × 12).

Spécification.

— *Entrées* : le nom de l'abonnement (chaîne de caractères), son prix mensuel (flottant strictement positif).

— *Traitement* : créer un objet `Abonnement` à partir des valeurs saisies ; coût annuel = prix mensuel × 12.

— *Sorties* : une description de l'abonnement, puis son coût annuel.

— *Contraintes* : classe `Abonnement` avec `__init__` ; méthode `cout_annuel()` qui *renvoie* le résultat ; méthode `afficher()` qui affiche la description ; la saisie se fait avec `input()`.

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

— Quels attributs donner à la classe `Abonnement` ?

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

— Quelle instruction crée l'objet à partir des valeurs saisies ?

Point de validation

Vérifie ton programme : pour « Streaming video » à 13,49 € par mois, il doit afficher un coût annuel de `161.88` ; pour « Salle de sport » à 25 €, un coût annuel de `300`.

Exercice B2 — La conversion de devises (objets et méthode avec argument)

Tu pars en voyage : aux États-Unis, 1 euro vaut 1,09 dollar ; au Japon, 1 euro vaut 163,5 yens. On modélise une devise par un objet de la classe `Devise` : son nom et son taux de change (combien d'unités de cette devise pour 1 euro).

Énoncé. Écrire un programme qui demande un montant en euros, crée deux objets `Devise` (le dollar et le yen), puis affiche la conversion du montant dans chacune des deux devises.

Spécification.

— *Entrées* : un montant en euros (flottant positif).

— *Traitement* : créer deux objets `Devise` avec des taux réalistes ; conversion = montant × taux.

— *Sorties* : deux lignes du type `100.0 euros = 109.0 dollars`.

— *Contraintes* : classe `Devise` avec `__init__(nom, taux)` ; méthode `convertir(montant_euros)` qui *renvoie* le résultat ; l'arrondi `round(..., 2)` est conseillé pour un affichage propre.

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

— Que représente l'attribut `taux` pour un objet `Devise` ?

— Pourquoi la méthode `convertir` prend-elle un argument en plus de `self` ?

— Où placer la saisie du montant : avant ou après la création des objets ? Pourquoi ?

Point de validation

Vérifie : pour 100 euros, le programme doit afficher une conversion à `109.0` dollars et une à `16350.0` yens (un léger écart du type `109.00000000000001` est normal sans arrondi : c'est la représentation des flottants).

Exercice B3 — Le compte épargne et les intérêts composés (boucle dans une méthode)

Une banque propose un livret rémunéré à 3 % par an : chaque année, le solde est multiplié par 1,03. On veut une classe `CompteEpargne` capable d'afficher l'évolution du solde sur plusieurs années.

Énoncé. Écrire un programme qui demande le solde initial, le taux annuel (en pourcentage : 3 pour 3 %) et le nombre d'années, crée un objet `CompteEpargne`, puis affiche le solde après chacune des années demandées.

Spécification.

— *Entrées* : solde initial (flottant), taux annuel en pourcentage (flottant), nombre d'années (entier ≥ 1).

— *Traitement* : à chaque année, solde $\leftarrow$ solde $\times(1 + \text{taux}/100)$; la boucle `for` se trouve dans une méthode `simuler(annees)`.

— *Sorties* : une ligne par année : `Annee 1 : ... euros`, `Annee 2 : ... euros`, etc.

— *Contraintes* : classe `CompteEpargne` avec `__init__(solde, taux)` ; méthode `appliquer_interets()` qui *modifie* le solde ; méthode `simuler(annees)` qui contient la boucle ; affichage arrondi au centime avec `round(solde, 2)`.

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

— Comment convertir un pourcentage (3) en taux de multiplication (1,03) ?

— Pourquoi la méthode `simuler` doit-elle appeler `appliquer_interets()` à chaque tour de boucle ?

— Pourquoi arrondir l'affichage avec `round(solde, 2)` quand on manipule de l'argent ?

Point de validation

Vérifie : solde initial 200 €, taux 3 %, 3 années → `Annee 1 : 206.0 euros`, `Annee 2 : 212.18 euros`, `Annee 3 : 218.55 euros` (arrondis au centime).

Exercice B4 — Le portefeuille d'actions (deux classes qui interagissent)

Un investisseur possède des actions : chacune a un nom (l'entreprise), une quantité et un prix d'achat unitaire. Son portefeuille est la liste de toutes ses actions. On modélise chaque action par un objet de la classe `Action` et le portefeuille par un objet de la classe `Portefeuille`.

Énoncé. Écrire un programme qui crée un portefeuille contenant deux actions — 10 actions TechCorp à 25,50 € et 5 actions GreenEnergy à 12,00 € — puis affiche le détail de chaque action (nom et valeur) et la valeur totale du portefeuille.

Spécification.

— *Entrées* : aucune saisie clavier : les actions sont créées directement dans le programme avec les valeurs de l'énoncé.

— *Traitement* : classe `Action` avec attributs `nom`, `quantite`, `prix_achat` et méthode `valeur()` (quantité $\times$ prix) ; classe `Portefeuille` avec attribut liste d'actions, méthode `ajouter(action)` et méthode `valeur_totale()` qui parcourt la liste.

— *Sorties* : la valeur de chaque action, puis la valeur totale du portefeuille.

— *Contraintes* : deux classes ; la liste d'actions est un attribut initialisé à `[]` dans le constructeur ; la valeur totale se calcule avec une boucle `for` sur la liste.

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

— Pourquoi une action et un portefeuille sont-ils deux classes différentes ? Quels attributs pour chacune ?

— Que doit renvoyer la méthode `valeur()` de la classe `Action` ?

— Dans `valeur_totale()`, quelle variable joue le rôle d'accumulateur ? Où doit-elle être initialisée ?

— Comment ajouter une action au portefeuille : quelle méthode de liste utiliser ?

Point de validation

Vérifie : le programme doit afficher `TechCorp : 255.0 euros`, `GreenEnergy : 60.0 euros`, puis `Valeur totale : 315.0 euros`.

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

À finir à la maison

Mon gestionnaire de budget personnel.

Contexte. Chaque mois, tu reçois une somme fixe (argent de poche, petit boulot, stage…) et tu veux suivre tes dépenses pour ne pas dépasser ton budget : alimentation, transport, loisirs, etc. On modélise chaque dépense (un libellé, un montant, une catégorie) par un objet de la classe `Depense`, et le budget du mois (la liste des dépenses) par un objet de la classe `Budget`.

Objectif. Écrire un programme qui permet de saisir plusieurs dépenses, de les ranger dans une liste, et de répondre à des questions comme : quel est le total du mois ? Combien ai-je dépensé dans chaque catégorie ? Quelle est ma plus grosse dépense ? Ai-je dépassé mon budget ?

Questions à se poser avant de coder.

— Quels attributs pour une dépense ? Pour le budget ?

— Faut-il une classe ou deux ? Justifie ton choix.

— Où calculer le total : dans une méthode de `Budget` ou ailleurs ?

— Comment parcourir la liste des dépenses pour additionner les montants d'une catégorie ?

— Que doit afficher le programme si aucune dépense n'a encore été saisie ?

— Comment vérifier que le programme est juste ? (penser à un petit exemple calculé à la main)

Étapes suggérées.

1. Version 0, sans classes : une liste de tuples `(libelle, montant, categorie)` et une boucle pour le total ; vérifier le résultat à la main (ex. 3 dépenses).
2. Version 1 : classe `Depense` avec son constructeur et une méthode `description()`.
3. Version 2 : classe `Budget` avec `ajouter(depense)` et `total()`.
4. Version 3 : `total_categorie(categorie)` et un bilan affichant le total par catégorie ; `plus_grosse_depense()`.
5. Bonus : saisie interactive des dépenses avec `input()`, comparaison avec un budget maximal, et message d'alerte si le total dépasse le budget.

Contraintes. Données réalistes (montants en euros, catégories comme `alimentation`, `transport`, `loisirs`) ; programme testé et commenté ; les messages peuvent être écrits sans accents (ex. `depense`, `categorie`) si ton éditeur pose problème.

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