TP Chapitre 7 : Paradigmes de programmation

Objectifs du TP

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

- distinguer, sur des exemples, les paradigmes impératif, fonctionnel et objet ;
- reconnaître la programmation événementielle et décrire un programme par ses réactions aux événements ;
- écrire des fonctions `lambda` et utiliser `map` et les listes par compréhension sur des montants ;
- identifier un effet de bord et écrire une fonction pure (qui ne modifie aucun état) ;
- écrire une classe simple avec des attributs et des méthodes (rappels du chapitre 0) ;
- choisir le paradigme le mieux adapté au champ d'application d'un programme.

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. Les montants sont exprimés en euros. 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 total du panier (paradigme impératif)

Dans une épicerie, chaque article est scanné un par un et son prix est ajouté au total du ticket. La **programmation impérative** décrit ce traitement comme une suite d'instructions qui modifient l'état du programme : ici, la variable `total` change de valeur à chaque tour de boucle.

Écrire le programme qui calcule le total du panier :

```
prix = [12.5, 8.0, 25.5, 4.0]    # prix des articles du panier, en euros

total = 0.0
for p in prix:
    # A COMPLETER : ajouter p au total

print("Total du panier :", total, "euros")
```

Point de validation

Vérifie ton programme : il doit afficher `Total du panier : 50.0 euros` (12,5 + 8 + 25,5 + 4).

Exercice A2 — Les remises en style fonctionnel (`map` et `lambda`)

Une boutique solde tous ses articles : chaque prix subit une remise de 20 %. Le **style fonctionnel** exprime ce traitement comme l'application d'une fonction à chaque élément de la liste, sans boucle ni variable modifiée.

1. Compléter le programme avec `map` et une fonction `lambda` :

```python
prix = [10.0, 25.0, 50.0, 100.0]
remise = 0.20

# A COMPLETER : construire prix_solde avec list(map(lambda p: ..., prix))
print(prix_solde)
```

2. Compléter la liste par compréhension qui fait exactement le même calcul :

```python
# A COMPLETER : construire prix_solde2 avec une liste par comprehension
print(prix_solde2)
```

Point de validation

Les deux listes doivent afficher `[8.0, 20.0, 40.0, 80.0]` : chaque prix a été multiplié par 0,8 (remise de 20 %).

Exercice A3 — Trier des articles par prix (`lambda` comme clé de tri)

Pour afficher les articles du moins cher au plus cher, on utilise la fonction `sorted` avec un paramètre `key` : une fonction `lambda` qui extrait le prix de chaque article.

```python
articles = [("stylo", 2.5), ("cahier", 4.0), ("sac", 15.0), ("gomme", 1.0)]

# A COMPLETER : trier articles par prix croissant avec sorted et une lambda
print(tri)
```

Point de validation

Le programme doit afficher `[('gomme', 1.0), ('stylo', 2.5), ('cahier', 4.0), ('sac', 15.0)]` : les articles du moins cher au plus cher.

Exercice A4 — TVA et effet de bord : écrire une fonction pure

La fonction suivante ajoute le prix TTC d'un article à un ticket de caisse, mais elle a un **effet de bord** : elle modifie une variable globale à chaque appel. Dans le style fonctionnel, on préfère les **fonctions pures** : elles ne modifient aucun état et se contentent de renvoyer un résultat.

1. Sans exécuter le programme, prédire son affichage et expliquer pourquoi :

```python
total_ttc = 0.0

def ajouter_au_ticket(prix_ht):
    global total_ttc
    total_ttc = total_ttc + prix_ht * 1.2

ajouter_au_ticket(10)
ajouter_au_ticket(25)
print("Total TTC :", total_ttc)
```

2. Compléter la version *pure* : la fonction prend le total en paramètre et renvoie le nouveau total, sans modifier aucune variable globale.

```python
def nouveau_total(total, prix_ht):
    # A COMPLETER : renvoyer le nouveau total, sans rien modifier

```

```
print(nouveau_total(0.0, 10))
print(nouveau_total(0.0, 10))
```

Point de validation

Première question : le programme affiche `Total TTC : 42.0` ($10 \times 1{,}2 + 25 \times 1{,}2$). Deuxième question : la fonction pure doit afficher `12.0` puis `12.0` : le même appel renvoie le même résultat, car rien n'est modifié entre les deux appels.

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 — Le panier : deux versions du même calcul (impératif et fonctionnel)

Le même problème peut être traité dans deux paradigmes différents. Ici, on calcule le total d'un panier de deux façons.

Énoncé. Écrire un programme qui définit deux fonctions qui calculent le total d'un panier (une liste de prix) :

- `total_imperatif(prix)` : version impérative, avec une boucle et une variable d'accumulation ;
- `total_fonctionnel(prix)` : version fonctionnelle, en une seule expression avec `sum` et une liste par compréhension.

Le programme teste ensuite les deux fonctions avec la liste `[12.5, 8.0, 25.5, 4.0]` et affiche les deux résultats.

Spécification.

- *Entrées* : une liste de prix fixée dans le programme (aucune saisie clavier).
- *Traitement* : deux fonctions `total_imperatif(prix)` et `total_fonctionnel(prix)`, chacune renvoie la somme des prix ; puis deux appels avec la même liste.
- *Sorties* : deux lignes : `Total imperatif : ... euros` et `Total fonctionnel : ... euros`.
- *Contraintes* : la version impérative utilise une boucle `for` et un accumulateur ; la version fonctionnelle tient en une seule expression ; aucune variable globale ; les deux fonctions sont pures.

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

- Dans la version impérative, quelle variable joue le rôle d'accumulateur ? Où doit-elle être initialisée ?
- Pourquoi la version fonctionnelle n'a-t-elle besoin d'aucune variable d'accumulation ?
- Quelle version modifie un état ? Laquelle est la plus courte ? Laquelle est la plus facile à tester ?
- Comment vérifier que les deux fonctions donnent le même résultat ?

Point de validation

Le programme doit afficher `Total imperatif : 50.0 euros` puis `Total fonctionnel : 50.0 euros`.

Exercice B2 — Le compte bancaire en programmation objet

La programmation objet regroupe les données et les opérations qui les concernent dans des **classes** : chaque objet possède des *attributs* (ses données) et des *méthodes* (ses opérations).

Énoncé. Écrire un programme qui définit une classe `CompteBancaire` :

— attributs : `titulaire` (chaîne) et `solde` (flottant, initialisé à 0.0) ;

— méthode `deposer(montant)` : ajoute le montant au solde si le montant est strictement positif ;

— méthode `retirer(montant)` : retire le montant si le solde est suffisant ; sinon affiche `Retrait refuse : solde insuffisant.` sans modifier le solde ;

— méthode `afficher()` : affiche `Compte de ... : ... euros`.

Le programme crée ensuite un compte au nom d'Aya, dépose 500 €, retire 120 €, tente de retirer 600 €, et affiche le compte après chaque opération.

Spécification.

— *Entrées* : aucune saisie clavier ; les opérations sont écrites en dur dans le programme.

— *Traitement* : définition de la classe, création d'un objet, appels successifs aux méthodes.

— *Sorties* : les lignes de `afficher()` et le message de refus éventuel.

— *Contraintes* : programmation objet obligatoire (classe, attributs, méthodes) ; le solde ne doit jamais devenir négatif.

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

— Quels sont les attributs d'un compte ? Quelles sont ses méthodes ? Pourquoi regrouper le solde et ses opérations dans une même classe ?

— Que doit faire `retirer()` si `montant > solde` ? Pourquoi ne pas autoriser le découvert ?

— Quelle différence avec une version impérative où le solde serait une variable globale ? (relire l'exercice A4)

— Combien d'objets `CompteBancaire` peux-tu créer ? Les soldes sont-ils indépendants ?

Point de validation

Le programme doit afficher : `Compte de Aya : 500.0 euros` (après le dépôt) ; `Compte de Aya : 380.0 euros` (après le retrait de 120) ; `Retrait refuse : solde insuffisant.` ; puis `Compte de Aya : 380.0 euros`.

Exercice B3 — La caisse événementielle

La **programmation événementielle** décrit un programme par ses réactions aux événements qui peuvent se produire. Une caisse de supermarché est un bon exemple : elle ne fait rien tant qu'aucun événement ne se produit (achat, paiement, fermeture).

Énoncé. Écrire un programme qui simule une caisse de supermarché. Le programme lit une suite d'événements saisis au clavier, un par ligne, et réagit à chacun :

— `achat`, suivi d'un montant sur la ligne suivante : le montant est ajouté au total du ticket en cours, et le programme affiche `Total : ... euros` ;

— `payer` : le programme affiche `A payer : ... euros`, ajoute ce montant au total encaissé de la journée, remet le ticket à zéro et affiche `Paiement recu.` ;

— `quitter` : le programme s'arrête et affiche le total encaissé de la journée.

Spécification.

— *Entrées* : une suite d'événements, un par ligne : `achat` (suivi d'une ligne contenant le montant), `payer` ou `quitter`.

— *Traitement* : boucle `while` ; lecture d'un événement ; réaction par `if` / `elif` selon l'événement ; deux accumulateurs : `total` (ticket en cours) et `encaisse` (total de la journée).

— *Sorties* : après chaque achat, `Total : ... euros` ; après `payer`, `A payer : ... euros` et `Paiement recu.` ; à la fin, `Caisse fermee. Total encaisse : ... euros`.

— *Contraintes* : boucle `while` (on ne connaît pas à l'avance le nombre d'événements) ; un seul événement traité à la fois (file d'événements) ; aucun autre événement n'est prévu.

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

— En quoi ce programme est-il événementiel ? Quels sont les événements et les réactions associées ?

— Pourquoi une boucle `while` et pas une boucle `for` ?

— Pourquoi deux variables `total` et `encaisse` ? Que devient `total` après l'événement `payer` ?

— Que se passe-t-il si l'utilisateur tape un mot inconnu ? Que décider pour que le programme reste simple ?

— Comment garantir que le programme se termine ?

Point de validation

Avec la suite `achat 10`, `achat 25`, `payer`, `achat 5`, `quitter`, le programme doit afficher : `Total : 10.0 euros` ; `Total : 35.0 euros` ; `A payer : 35.0 euros` ; `Paiement recu.` ; `Total : 5.0 euros` ; puis `Caisse fermee. Total encaisse : 35.0 euros`.

Exercice B4 — La boutique en ligne : objets et fonctions ensemble (multiparadigme)

Dans un même programme Python, on peut utiliser plusieurs paradigmes. Ici, on combine la programmation objet (une classe `Article`) et le style fonctionnel (tri, filtrage et total sans boucle).

Énoncé. Écrire un programme qui :

1. définit une classe `Article` avec les attributs `nom` (chaîne) et `prix_ht` (flottant), et une méthode `prix_ttc()` qui renvoie le prix TTC (prix HT × 1,2) ;
2. crée la liste d'articles : stylo (2,5 €), cahier (4 €), sac (15 €), gomme (1 €) ;
3. affiche la liste des articles triée du moins cher au plus cher au prix TTC (`sorted` avec une `lambda`) ;
4. affiche la liste des articles qui coûtent moins de 5 € TTC (liste par compréhension) ;
5. affiche le total TTC du panier complet (`sum` et une liste par compréhension).

Spécification.

— *Entrées* : les articles sont fixés dans le programme (aucune saisie clavier).

— *Traitement* : classe `Article` ; tri par prix TTC ; filtre à moins de 5 € TTC ; total TTC.

— *Sorties* : trois lignes : la liste triée, la liste filtrée, le total TTC.

— *Contraintes* : trois paradigmes utilisés — objet (la classe `Article`), fonctionnel (`lambda`, compréhensions, `sum`), impératif (le programme principal) ; aucune variable globale.

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

— Pourquoi une classe `Article` est-elle adaptée ici ? Quels attributs et quelles méthodes ?

— Pourquoi trier sur le prix TTC et pas sur le prix HT ?

— Où trouve-t-on du fonctionnel dans ce programme ? Où trouve-t-on de l'objet ?

— Comment vérifier le total TTC à la main ? (calculer $1{,}2 \times (2{,}5 + 4 + 15 + 1)$)

Point de validation

Le programme doit afficher : la liste triée `[('gomme', 1.2), ('stylo', 3.0), ('cahier', 4.8), ('sac', 18.0)]` ; la liste des articles à moins de 5 € TTC : `stylo`, `cahier`, `gomme` (dans l'ordre de la liste initiale) ; et `Total TTC : 27.0 euros`.

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

À finir à la maison

Ma banque en ligne : comptes, virements et intérêts.

Contexte. Tu crées un mini programme bancaire : des comptes (titulaire, numéro, solde), des opérations (dépôt, retrait, virement entre comptes) et le calcul des intérêts annuels. L'objectif est de *choisir* et de *combiner* les paradigmes du chapitre : la programmation objet pour modéliser les comptes, le fonctionnel pour les calculs purs (intérêts), l'impératif pour le programme principal.

Objectif. Écrire un programme qui crée deux comptes, effectue des dépôts, des retraits et un virement, affiche les relevés, puis crédite chaque compte des intérêts annuels à 3 % (avec une fonction pure). Le programme doit refuser tout retrait ou virement qui rendrait un solde négatif.

Questions à se poser avant de coder.

— Quels attributs pour un compte ? (titulaire, numéro, solde) Quelles méthodes ?

— Un virement : qui appelle qui ? Comment écrire `virer(autre, montant)` à partir de `retirer` et `deposer` ?

— Comment garantir qu'un retrait ou un virement ne rend jamais un solde négatif ?

— Pourquoi le calcul des intérêts est-il un bon candidat pour une fonction pure ? Que doit-elle prendre en paramètre ? Que renvoie-t-elle ? (rappel : une fonction pure ne modifie aucun état)

— Quelle serait la version impérative avec variables globales ? Pourquoi est-elle risquée (effets de bord) ?

— Comment vérifier le programme ? (cas simples : dépôt, retrait, virement ; taux 0 %, comme dans le TP du chapitre 1)

Étapes suggérées.

1. Version 1 : classe `Compte` avec `deposer`, `retirer`, `afficher` ; tester avec un seul compte.
2. Version 2 : méthode `virer(autre, montant)` ; gérer le refus si le solde est insuffisant.
3. Version 3 : fonction pure `interets_annuels(solde, taux)` qui renvoie le solde après un an ; l'utiliser pour créditer les deux comptes.
4. Version 4 : afficher un relevé complet : titulaire, numéro, solde.
5. Bonus : refuser les montants négatifs ; décider si le découvert est autorisé ou non ; comparer avec une version impérative à variables globales et justifier le choix de l'objet.

Contraintes. Montants positifs ; un virement ne doit pas créer de découvert ; les intérêts sont arrondis au centime (`round(solde * (1 + taux), 2)`) ; programme testé avec des cas simples et commenté ; les messages peuvent être écrits sans accents (ex. `virement`, `interets`) si ton éditeur pose problème.

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