# TP Chapitre 2 : Piles et Files

**Objectifs du TP**

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

— distinguer les modes LIFO et FIFO et les associer aux piles et aux files ;
— utiliser le vocabulaire des piles et des files : empiler, dépiler, enfiler, défiler, sommet, tête ;
— écrire la classe `Pile` et la classe `File`, et les implémenter avec une liste Python ;
— distinguer l'interface d'une structure de données et son implémentation ;
— implémenter une file avec deux piles, avec exactement la même interface ;
— choisir une structure de données adaptée à une situation, en particulier des situations de la vie économique (caisse, paiements, opérations bancaires) ;
— concevoir et écrire un programme complet 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. Les classes `Pile` et `File` du cours sont à ta disposition : tu peux les recopier dans chacun de tes fichiers. 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 — La classe `Pile` : le mode LIFO

Un coffre-fort électronique ne permet de récupérer que le dernier billet déposé : pour atteindre un billet plus ancien, il faut d'abord retirer tous ceux qui sont au-dessus. C'est exactement le fonctionnement d'une **pile** : le dernier élément ajouté est le premier retiré (mode LIFO, *Last In First Out*). L'élément que l'on peut retirer s'appelle le **sommet**.

Compléter la classe `Pile` ci-dessous : chaque méthode doit réaliser l'opération indiquée en commentaire. Le programme de test à la fin doit fonctionner sans modification.

```
class Pile:
    def __init__(self):
        self.data = []

    def empiler(self, element):
        # A COMPLETER : ajouter element au sommet de la pile
        pass

    def depiler(self):
        # A COMPLETER : retirer et renvoyer l'element du sommet
        pass

    def hauteur(self):
        # A COMPLETER : renvoyer le nombre d'elements de la pile
        pass

    def est_vide(self):
```

```
        # A COMPLETER : renvoyer True si la pile est vide, False sinon
        pass

    def sommet(self):
        # A COMPLETER : renvoyer le sommet sans le retirer
        pass

p = Pile()
p.empiler(10)
p.empiler(20)
p.empiler(30)
print(p.depiler())     # le sommet est 30
print(p.hauteur())     # il reste 10 et 20
print(p.est_vide())
```

**Point de validation**

Vérifie ton programme : il doit afficher `30`, puis `2`, puis `False`.

### Exercice A2 — La classe `File` : le mode FIFO

À la caisse d'un supermarché, les clients sont servis dans l'ordre d'arrivée : le premier arrivé est le premier servi (mode FIFO, *First In First Out*). L'élément que l'on peut retirer s'appelle la **tête** de la file. Une `File` fonctionne de la même façon : on **enfile** un élément en fin de file, on **défile** l'élément en tête.

Compléter la classe `File` ci-dessous, avec les mêmes méthodes que le cours. Le programme de test à la fin doit fonctionner sans modification.

```
class File:
    def __init__(self):
        self.data = []

    def enfiler(self, element):
        # A COMPLETER : ajouter element en fin de file
        pass

    def defiler(self):
        # A COMPLETER : retirer et renvoyer l'element en tete de file
        pass

    def longueur(self):
        # A COMPLETER : renvoyer le nombre d'elements de la file
        pass

    def est_vide(self):
        # A COMPLETER : renvoyer True si la file est vide, False sinon
        pass

    def dernier(self):
        # A COMPLETER : renvoyer le dernier element sans le retirer
        pass

f = File()
f.enfiler(25)
f.enfiler(40)
f.enfiler(15)
print(f.defiler())     # le premier arrive : 25
print(f.longueur())    # il reste 40 et 15
```

```
print(f.dernier())     # le dernier arrive : 15
```

**Point de validation**

Vérifie ton programme : il doit afficher 25, puis 2, puis 15.

### Exercice A3 — Le relevé bancaire : dernier dépôt et annulation (pile)

Une application bancaire conserve l'historique des opérations d'un compte. La dernière opération est la plus récente, et une annulation doit retirer la dernière opération effectuée : une **pile** est la structure adaptée (mode LIFO).

La classe `Pile` est fournie. Compléter le programme : (1) afficher la dernière opération (le sommet de la pile) ; (2) annuler la dernière opération et mettre le solde à jour ; (3) afficher la nouvelle dernière opération. Tu peux écrire par exemple `print("Derniere operation :", operations.sommet(), "euros")`.

```
class Pile:
    def __init__(self):
        self.data = []

    def empiler(self, element):
        self.data.append(element)

    def depiler(self):
        return self.data.pop()

    def hauteur(self):
        return len(self.data)

    def est_vide(self):
        return self.hauteur() == 0

    def sommet(self):
        return self.data[-1]

operations = Pile()
solde = 0

operations.empiler(100)       # depot de 100 euros
solde = solde + 100

operations.empiler(-30)       # retrait de 30 euros
solde = solde + (-30)

operations.empiler(50)        # depot de 50 euros
solde = solde + 50

# A COMPLETER : afficher la derniere operation (le sommet de la pile)

# A COMPLETER : annuler la derniere operation (depiler, recuperer le
# montant annule, puis corriger le solde)

# A COMPLETER : afficher la nouvelle derniere operation

print("Solde :", solde, "euros")
```

**Point de validation**

Vérifie ton programme : il doit afficher `Derniere operation : 50 euros`, puis après l'annulation `Derniere operation : -30 euros`, puis `Solde : 70 euros`.

### Exercice A4 — La file d'impression des tickets de caisse (file)

À la sortie d'un supermarché, la caisse envoie les tickets à imprimer dans l'ordre des passages : le premier ticket envoyé est le premier imprimé. La file d'impression contient les montants des achats, et se comporte comme une **file** (mode FIFO).

La classe `File` est fournie. Compléter le programme : (1) afficher le prochain ticket à imprimer *sans le retirer* ; (2) imprimer le prochain ticket (le retirer de la file) et afficher son montant ; (3) afficher le nombre de tickets encore en attente ; (4) afficher le nouveau prochain ticket.

**Remarque.** La file du cours ne possède pas de méthode pour lire la tête sans la retirer : tu peux accéder au premier élément de la liste interne avec `tickets.data[0]`. C'est un accès direct à l'*implémentation* (l'attribut `data`), pas à l'*interface* — nous y reviendrons dans la partie B.

```
class File:
    def __init__(self):
        self.data = []

    def enfiler(self, element):
        self.data.append(element)

    def defiler(self):
        return self.data.pop(0)

    def longueur(self):
        return len(self.data)

    def est_vide(self):
        return self.longueur() == 0

    def dernier(self):
        return self.data[-1]

tickets = File()
tickets.enfiler(12.50)
tickets.enfiler(7.80)
tickets.enfiler(23.40)

# A COMPLETER : afficher le prochain ticket a imprimer (sans le retirer)

# A COMPLETER : imprimer le prochain ticket et afficher son montant

# A COMPLETER : afficher le nombre de tickets encore en attente

# A COMPLETER : afficher le nouveau prochain ticket
```

**Point de validation**

Vérifie ton programme : il doit afficher `12.5`, puis `12.5`, puis `2`, puis `7.8`.

# 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 terminal de paiement en attente (file)

Un terminal de paiement accepte les paiements des clients mais les traite dans l'ordre d'arrivée : les paiements en attente forment une **file**. Le premier paiement accepté doit être le premier traité.

**Énoncé.** Écrire un programme qui demande des montants à payer jusqu'à la saisie du mot `fin`, enfile chaque montant valide (strictement positif) dans une file, puis traite les paiements un par un dans l'ordre d'arrivée en affichant `Paiement de ... euros traite` pour chacun. À la fin, il affiche le total encaissé et le nombre de paiements restés en attente.

**Spécification.**

— *Entrées* : une suite de montants (flottants) saisie un par un, terminée par le mot `fin` ; chaque montant doit être strictement positif.

— *Traitement* : enfiler chaque montant valide dans une file ; puis défiler les paiements un à un jusqu'à ce que la file soit vide, en cumulant le total.

— *Sorties* : pour chaque paiement traité, `Paiement de ... euros traite` ; en fin de programme, le total encaissé et le nombre de paiements restés en attente.

— *Contraintes* : écrire la classe `File` complète ; boucle `while` pour la saisie et boucle `while` pour le traitement ; un montant non valide est refusé (message d'erreur) mais ne termine pas la saisie ; les montants ne sont pas stockés ailleurs que dans la file.

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

— Pourquoi une file est-elle adaptée pour traiter des paiements « dans l'ordre d'arrivée » ?

— Que renvoie `input()` ? Pourquoi faut-il comparer la saisie au mot `"fin"` *avant* de la convertir en `float` ?

— Que se passerait-il si l'on utilisait une pile pour stocker les paiements en attente ? L'ordre de traitement serait-il le même ?

— Quelle condition permet de savoir qu'il ne reste plus de paiement à traiter ?

**Point de validation**

Vérifie ton programme : avec les saisies `10`, `20`, `fin`, il doit afficher `Paiement de 10.0 euros traite`, `Paiement de 20.0 euros traite`, `Total encaisse : 30.0 euros`, `Paiements en attente : 0`.

## Exercice B2 — Le compte bancaire et l'annulation d'opérations (pile)

Une application bancaire permet d'annuler la dernière opération effectuée sur un compte : dépôt, retrait, ou erreur de saisie. Les opérations sont conservées dans une **pile** : la dernière opération est au sommet.

**Énoncé.** Écrire un programme de gestion de compte qui affiche un menu répété : `1` dépôt, `2` retrait, `3` annuler la dernière opération, `4` solde, `0` quitter. Chaque opération est empilée (un retrait est stocké comme un montant négatif) et le solde est mis à jour immédiatement. Annuler une opération, c'est la

dépiler puis annuler son effet sur le solde. Si l'on demande une annulation alors qu'aucune opération n'est enregistrée, le programme affiche `Aucune operation a annuler.`

**Spécification.**

— *Entrées* : un choix (entier entre 0 et 4) à chaque tour du menu ; un montant (flottant) pour les dépôts et les retraits.

— *Traitement* : une pile d'opérations (montants signés) et un solde ; dépôt : empiler le montant, solde + ; retrait : empiler l'opposé, solde − ; annulation : dépiler et corriger le solde.

— *Sorties* : la confirmation de chaque opération, le message d'erreur d'annulation sur pile vide, et le solde avec le nombre d'opérations (choix `4`).

— *Contraintes* : écrire la classe `Pile` complète ; boucle `while` pour le menu ; ne jamais dépiler une pile vide ; la boucle se termine par le choix `0`.

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

— Pourquoi une pile est-elle adaptée pour annuler ? Quel est l'ordre de retrait des opérations ?

— Que signifie « annuler l'effet d'une opération » ? Si l'opération annulée vaut $m$, comment corriger le solde ?

— Pourquoi faut-il tester `est_vide()` avant de dépiler ?

— Dans quel cas une file serait-elle plus adaptée que la pile pour ce programme ? (pense à l'exercice B1)

**Point de validation**

Vérifie ton programme avec la séquence : dépôt 100, dépôt 50, retrait 20, puis deux annulations. Après la première annulation le solde affiché doit être `150.0` ; après la seconde, `100.0` ; une troisième annulation doit afficher `Aucune operation a annuler.`

## Exercice B3 — La caisse du supermarché (simulation FIFO)

Un supermarché ouvre une caisse. Une file de clients attend ; chaque client a un panier d'un montant aléatoire entre 5 et 100 € et un temps de passage en caisse aléatoire entre 3 et 10 minutes. Les clients sont servis dans l'ordre d'arrivée.

**Énoncé.** Écrire un programme qui demande le nombre de clients $N$ ($1 \leq N \leq 50$), crée $N$ clients aléatoires et les enfile, puis traite la file dans l'ordre : pour chaque client, il affiche le montant de son panier et sa durée de passage. À la fin, il affiche le temps total (somme des durées), le total encaissé, le montant moyen d'un panier et la durée moyenne par client.

**Spécification.**

— *Entrées* : un entier $N$ (nombre de clients, $1 \leq N \leq 50$).

— *Traitement* : pour chaque client, générer un couple aléatoire (montant, durée) et l'enfiler ; puis défiler les clients un à un jusqu'à ce que la file soit vide, en cumulant le total encaissé et le temps total.

— *Sorties* : pour chaque client, `Client servi : panier ... euros, ... minutes` ; en fin de programme, le temps total, le total encaissé, le montant moyen d'un panier et la durée moyenne par client.

— *Contraintes* : écrire la classe `File` complète ; utiliser `random.randint` ; un client est représenté par un tuple `(montant, duree)` ; aucun client ne doit être servi avant ceux qui sont devant lui.

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

— Pourquoi représenter un client par un tuple ? Pourquoi ne pas utiliser deux files (une pour les montants, une pour les durées) ?

— Pourquoi une file est-elle adaptée ici ? Que se passerait-il avec une pile ?

— Pourquoi la boucle de traitement est-elle « tant que la file n'est pas vide » ? Quelle méthode permet de tester cela ?

— Comment vérifier que le programme est juste sans dépendre du hasard ? (pense à remplacer temporairement la génération aléatoire par des valeurs fixes)

**Point de validation**

Pour vérifier, remplace temporairement la génération aléatoire par trois clients fixes : `(10, 5)`, `(20, 4)`, `(30, 6)`. Le programme doit afficher un temps total de `15` minutes, un total encaissé de `60` euros, un montant moyen de `20.0` euros et une durée moyenne de `5.0` minutes.

## Exercice B4 — Une file construite avec deux piles (interface et implémentation)

Le cours montre qu'une file peut être implémentée avec **deux piles** : une pile d'« entrée » et une pile de « sortie ». Enfiler revient à empiler dans la pile d'entrée. Pour défiler, on transfère d'abord *tous* les éléments de la pile d'entrée vers la pile de sortie (ce qui les retourne), puis on dépile la pile de sortie : on obtient bien le premier élément entré.

**Énoncé.** Écrire la classe `File2Piles` complète, avec les méthodes `enfiler`, `defiler`, `longueur`, `est_vide` et `dernier`, en n'utilisant *uniquement* les méthodes de la classe `Pile` (fournie ci-dessous). La classe obtenue doit avoir exactement la même *interface* que la classe `File` du cours : mêmes noms de méthodes, mêmes paramètres, mêmes comportements. Le programme de test fourni doit fonctionner sans modification.

**Spécification.**

— *Entrées* : aucune (on écrit une classe).

— *Traitement* : le constructeur crée les deux piles (et éventuellement un attribut pour mémoriser le dernier élément entré) ; `enfiler` empile dans la pile d'entrée ; `defiler` transfère si nécessaire puis dépile la pile de sortie ; `longueur` et `est_vide` combinent les deux piles ; `dernier` renvoie le dernier élément entré sans le retirer.

— *Sorties* : comportements identiques à la classe `File`.

— *Contraintes* : uniquement les méthodes de la classe `Pile` pour manipuler les piles (pas d'accès direct à `data`) ; le programme de test doit afficher les résultats de l'encadré vert.

**Programme de test** (à recopier à la fin de ton fichier ; la classe `Pile` est rappelée, l'exercice porte sur la classe `File2Piles` que tu dois écrire entièrement) :

```
class Pile:
    def __init__(self):
        self.data = []
    def empiler(self, element):
        self.data.append(element)
    def depiler(self):
        return self.data.pop()
    def hauteur(self):
        return len(self.data)
    def est_vide(self):
        return self.hauteur() == 0
    def sommet(self):
        return self.data[-1]
```

```
14
15 # A COMPLETER : ecrire ici la classe File2Piles complete
16
17 f = File2Piles()
18 f.enfiler("A")
19 f.enfiler("B")
20 f.enfiler("C")
21 print(f.defiler())     # doit afficher A
22 print(f.defiler())     # doit afficher B
23 print(f.dernier())     # doit afficher C
24 print(f.longueur())    # doit afficher 1
25 f.enfiler("D")
26 print(f.defiler())     # doit afficher C
27 print(f.dernier())     # doit afficher D
28 print(f.longueur())    # doit afficher 1
```

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

— Pourquoi le transfert de la pile d'entrée vers la pile de sortie retourne-t-il l'ordre des éléments ? (pense à ce qui se passe quand on dépile puis empile)

— Pourquoi ne transférer que lorsque la pile de sortie est vide ? Que se passerait-il si l'on transférait alors qu'elle contient encore des éléments ?

— Comment connaître le dernier élément entré sans vider les piles ? (une piste : un attribut mis à jour à chaque `enfiler`)

— Pourquoi peut-on dire que `File2Piles` et `File` ont la même interface ? Quel programme peut utiliser l'une ou l'autre sans être modifié ?

**Point de validation**

Le programme de test doit afficher : `A`, `B`, `C`, `1`, puis `C`, `D`, `1`. Si tu obtiens cette séquence, ta file à deux piles se comporte comme une vraie file.

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

**À finir à la maison**

**Le supermarché et ses caisses.**

**Contexte.** Un supermarché dispose de plusieurs caisses. Les clients arrivent avec leur panier et se placent dans une file d'attente ; dès qu'une caisse est libre, le client en tête de file s'y rend. Chaque client a un panier d'un montant aléatoire et un temps de passage en caisse aléatoire. La question économique est centrale : combien de temps faut-il pour servir tous les clients ? Combien la journée rapporte-t-elle ? Que se passe-t-il si l'on ouvre une caisse de plus ?

**Objectif.** Écrire un programme qui simule le fonctionnement des caisses d'un supermarché, avec une ou plusieurs files de clients, et qui affiche un bilan : temps total pour servir tout le monde, total encaissé, clients servis par caisse. Le nombre de caisses et le nombre de clients doivent être faciles à modifier (constantes ou saisie).

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

— Quelles structures de données utiliser ? (une file de clients ; comment gérer plusieurs caisses ?)

— Comment représenter un client ? (un tuple `(montant, duree)` ?)

— Quand une caisse est-elle libre ? Comment simuler le passage du temps ?

— Comment garantir que la simulation se termine ? (tu as déjà vu l'importance d'un garde-fou dans les exercices sur les boucles `while`)

— Comment vérifier que la simulation est juste ? (penser à un cas simple avec des valeurs fixes, comme dans l'exercice B3)

**Étapes suggérées.**

1. Version 1, une seule caisse : $N$ clients déjà en file ; servir tout le monde et afficher le temps total et le total encaissé. Vérifier à la main avec des valeurs fixes (ex. 3 clients `(10, 5)`, `(20, 4)`, `(30, 6)` : 15 minutes et 60 euros).
2. Version 2, plusieurs caisses : $K$ caisses ($1 \leq K \leq 8$), chacun avec sa propre file ; répartir les clients en les envoyant à la file la plus courte (ou à la première caisse libre).
3. Version 3, bilan détaillé : afficher pour chaque caisse le nombre de clients servis et le total encaissé, ainsi que la longueur des files à la fin.
4. Version 4 (défi) : les clients arrivent au fil du temps (un client toutes les 2 ou 3 minutes) au lieu d'être déjà en file ; observer ce qui se passe quand le rythme d'arrivée est trop rapide par rapport au nombre de caisses.
5. Bonus : comparer le temps total pour servir 50 clients avec 1, 3 et 5 caisses ; que constate-t-on ? Quel est l'effet d'un montant moyen de panier plus élevé ?

**Contraintes.** Données crédibles ($N$ entre 1 et 200, $K$ entre 1 et 8, montants entre 5 et 150 €, durées entre 2 et 10 minutes) ; garde-fou pour garantir la terminaison ; programme commenté et testé sur des cas simples ; les messages peuvent être écrits sans accents (ex. `Client servi`, `Total encaisse`) si ton éditeur pose problème.

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