Chapitre 2
Piles et Files

Ce chapitre introduit deux structures de données fondamentales : les **piles** et les **files**. Une **pile** fonctionne comme une pile d'assiettes : on pose une assiette sur le dessus, et on reprend toujours la dernière assiette posée. Une **file** fonctionne comme la queue devant un guichet : le premier arrivé est le premier servi. Tout ceci est parfaitement conforme à l'intuition, et suggère fortement des techniques d'implémentation pour les piles et les files : nous les verrons en Python à l'aide des classes et des listes étudiées au chapitre précédent.

I. Les structures de données, interface et implémentation

1. La notion de structure de données

Définition 2.1.
Une **structure de données** est une manière d'organiser et de stocker des données, afin de pouvoir les manipuler efficacement. On connaît déjà plusieurs structures de données depuis la Première : les **listes**, les **tuples** et les **dictionnaires**.

■ **Exemple 2.1.**
La liste Python est une structure de données linéaire : les éléments sont rangés « à la suite » les uns des autres, chacun possédant un indice. On peut ajouter un élément en fin de liste avec `append`, en retirer un avec `pop`, ou accéder à un élément par son indice. ■

```
notes = [12, 15, 8]
notes.append(17)
print(notes) # affiche [12, 15, 8, 17]
print(notes.pop()) # affiche 17
```

Ce chapitre étudie deux nouvelles structures de données linéaires : les **piles** et les **files**. Leur particularité est de restreindre fortement les opérations possibles : on n'ajoute et on ne retire des éléments qu'à une seule extrémité de la structure.

2. Interface et implémentation

Définition 2.2.
L'**interface** d'une structure de données est l'ensemble des opérations que l'on peut effectuer sur elle, indépendamment de la manière dont ces opérations sont réalisées en interne. L'**implémentation** est la réalisation concrète de ces opérations : le code qui les exécute.

■ **Exemple 2.2.**
On utilise `append` sur une liste sans savoir comment Python range réellement les éléments en mémoire : la méthode `append` fait partie de l'**interface** de la liste. La façon dont Python la réalise (placement en mémoire, agrandissement du tableau) est son **implémentation**, qui nous est cachée. ■

R Une même interface peut être implémentée de plusieurs façons différentes. C'est un point central du programme de Terminale : nous verrons en fin de chapitre une même structure de données (la file) implémentée de deux manières distinctes, avec exactement la même interface.

3. Pile et file : deux structures linéaires

Une pile et une file sont toutes deux des structures linéaires : leurs éléments sont rangés à la suite les uns des autres. Elles se distinguent par le **mode d'accumulation** des données, c'est-à-dire l'ordre dans lequel les éléments sont retirés :

— dans une **pile**, le dernier élément ajouté est le premier retiré : c'est le mode **LIFO** (*Last In First Out*, « dernier entré, premier sorti »), parfois noté **FILO** (*First In Last Out*, « premier entré, dernier sorti ») ;
— dans une **file**, le premier élément ajouté est le premier retiré : c'est le mode **FIFO** (*First In First Out*, « premier entré, premier sorti »).

■ **Exemple 2.3.**
La pile d'assiettes illustre le mode LIFO : on pose les assiettes les unes sur les autres, et on reprend toujours l'assiette du dessus, c'est-à-dire la dernière posée. La queue devant un guichet illustre le mode FIFO : le premier client arrivé est le premier servi. ■

Le tableau suivant récapitule les analogies et le vocabulaire des deux structures.

	Pile	**File**
Mode d'accumulation	LIFO (dernier entré, premier sorti)	FIFO (premier entré, premier sorti)
Analogie	Pile d'assiettes	Queue devant un guichet
Ajouter un élément	Empiler	Enfiler
Retirer un élément	Dépiler	Défiler
Élément accessible	Le sommet (dernier ajouté)	La tête (premier ajouté)

II. Les piles

1. Principe : la structure LIFO

Définition 2.3.
Une **pile** est une structure de données linéaire dans laquelle le dernier élément ajouté est le premier élément retiré. On parle de mode **LIFO** (*Last In First Out*). L'élément que l'on peut retirer s'appelle le **sommet** de la pile.

▪ **Exemple 2.4.**
Une pile d'assiettes : on ne peut prendre que l'assiette du dessus, c'est-à-dire la dernière posée. Pour atteindre une assiette plus bas dans la pile, il faut d'abord retirer toutes celles qui sont au-dessus. ▪

Les deux opérations fondamentales d'une pile sont :
— **empiler** : ajouter un élément au sommet de la pile ;
— **dépiler** : retirer l'élément du sommet de la pile et le renvoyer.

▪ **Exemple 2.5.**
On empile successivement les entiers 1, 2 puis 3 dans une pile vide. Le sommet est alors 3. On dépile : on obtient 3, et la pile contient 1 et 2. On dépile encore : on obtient 2. Les éléments sortent donc dans l'ordre inverse de leur arrivée : 3, 2, 1. ▪

```
p = Pile()
p.empiler(1)
p.empiler(2)
p.empiler(3)
print(p.hauteur()) # affiche 3
print(p.depiler()) # affiche 3
print(p.depiler()) # affiche 2
```

2. Implémentation en Python

On implémente une pile en Python à l'aide d'une **liste** : les éléments sont stockés à la suite dans un attribut `data`. La fin de la liste représente le sommet de la pile. On définit une classe `Pile` dont les méthodes réalisent les opérations de la pile. Les méthodes s'ajoutent les unes aux autres dans la classe `Pile`.

Le constructeur

La méthode `__init__` est le constructeur : on initialise la pile à une liste vide.

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

Empiler

La méthode `empiler` ajoute un élément au sommet de la pile. Comme le sommet est la fin de la liste, on utilise la méthode `append` des listes.

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

R `append` ajoute l'élément en **fin** de liste : c'est bien le sommet de la pile qui reçoit le nouvel élément.

Hauteur

La méthode `hauteur` renvoie le nombre d'éléments de la pile.

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

Test de vacuité

La méthode `est_vide` renvoie un booléen : `True` si la pile est vide, `False` sinon.

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

Dépiler

La méthode `depiler` retire l'élément du sommet de la pile et le renvoie. La méthode `pop` des listes, appelée sans argument, retire et renvoie le dernier élément : c'est exactement le comportement attendu.

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

R Si la pile est vide, `pop()` provoque une erreur `IndexError` : on ne peut pas retirer un élément qui n'existe pas. Le traitement de ce cas est proposé en exercice.

Sommet

La méthode `sommet` renvoie le dernier élément de la pile, sans le retirer. L'indice `-1` désigne le dernier élément d'une liste.

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

Pour résumer

Le tableau suivant récapitule les méthodes de la classe `Pile` et leur rôle.

Méthode	Rôle
`__init__(self)`	Construit une pile vide
`empiler(self, element)`	Ajoute `element` au sommet de la pile
`hauteur(self)`	Renvoie le nombre d'éléments de la pile
`est_vide(self)`	Renvoie `True` si la pile est vide
`depiler(self)`	Retire et renvoie l'élément du sommet
`sommet(self)`	Renvoie l'élément du sommet sans le retirer

3. Applications des piles

Les piles sont très présentes en informatique, partout où l'on a besoin de « revenir en arrière » dans l'ordre inverse des actions.

■ **Exemple 2.6.**
La pile d'appels de fonctions. Lorsqu'un programme exécute une fonction, Python mémorise l'endroit où reprendre l'exécution en empilant une « frame » d'appel. Quand la fonction se termine, la frame est dépilée et le programme reprend son cours. C'est une pile : la dernière fonction appelée est la première terminée. ■

■ **Exemple 2.7.**
L'historique du navigateur. Le bouton « précédent » d'un navigateur affiche la page visitée juste avant la page courante : les pages visitées sont empilées, et chaque retour en arrière dépile la page courante. ■

■ **Exemple 2.8.**
Les annulations d'un éditeur de texte. Le raccourci `Ctrl+Z` annule la dernière action effectuée, puis l'avant-dernière, et ainsi de suite : les actions sont empilées et l'annulation les dépile dans l'ordre inverse. ■

■ **Exemple 2.9.**
La vérification des parenthèses. Pour vérifier qu'une expression comme `(3 + (2 * 5))` a ses parenthèses correctement équilibrées, on peut empiler chaque parenthèse ouvrante et dépiler à chaque parenthèse fermante : si la pile est vide à la fin, les parenthèses sont équilibrées. Cette idée est reprise en exercice. ■

III. Les files

1. Principe : la structure FIFO

Définition 2.4.
Une **file** est une structure de données linéaire dans laquelle le premier élément ajouté est le premier élément retiré. On parle de mode **FIFO** (*First In First Out*). L'élément que l'on peut retirer s'appelle la **tête** de la file.

■ **Exemple 2.10.**
La queue devant un guichet : les clients arrivent les uns derrière les autres, et le premier arrivé est le premier servi. Un nouveau client se place toujours en fin de file.
■

Les deux opérations fondamentales d'une file sont :
— **enfiler** : ajouter un élément en fin de file ;
— **défiler** : retirer l'élément en tête de file et le renvoyer.

■ **Exemple 2.11.**
On enfile successivement les lettres A, B puis C dans une file vide. La tête est alors A. On défiler : on obtient A, et la file contient B et C. On défiler encore : on obtient B. Les éléments sortent donc dans l'ordre de leur arrivée : A, B, C. ■

```
f = File()
f.enfiler("A")
f.enfiler("B")
f.enfiler("C")
print(f.longueur()) # affiche 3
print(f.defiler()) # affiche A
print(f.defiler()) # affiche B
```

2. Implémentation en Python

On implémente une file en Python à l'aide d'une **liste** : les éléments sont stockés à la suite dans un attribut `data`. La fin de la liste reçoit les nouveaux éléments, et le début de la liste (indice 0) est la tête de la file. On définit une classe `File` dont les méthodes réalisent les opérations de la file.

Le constructeur

La méthode `__init__` est le constructeur : on initialise la file à une liste vide.

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

Enfiler

La méthode `enfiler` ajoute un élément en fin de file, avec `append`.

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

Longueur

La méthode `longueur` renvoie le nombre d'éléments de la file.

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

Test de vacuité

La méthode `est_vide` renvoie `True` si la file est vide, `False` sinon.

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

Défiler

La méthode `defiler` retire l'élément en tête de file et le renvoie. La méthode `pop` appelée avec l'argument `0` retire et renvoie le premier élément de la liste : c'est exactement le comportement attendu.

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

R Si la file est vide, `pop(0)` provoque une erreur `IndexError`, comme pour la pile. Le traitement de ce cas est proposé en exercice.

Dernier élément

La méthode `dernier` renvoie le dernier élément de la file, sans le retirer.

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

Pour résumer

Le tableau suivant récapitule les méthodes de la classe `File` et leur rôle.

Méthode	Rôle
`__init__(self)`	Construit une file vide
`enfiler(self, element)`	Ajoute `element` en fin de file
`longueur(self)`	Renvoie le nombre d'éléments de la file
`est_vide(self)`	Renvoie `True` si la file est vide
`defiler(self)`	Retire et renvoie l'élément en tête de file
`dernier(self)`	Renvoie le dernier élément sans le retirer

R **Coût de l'opération `defiler`.** Retirer le premier élément d'une liste avec `pop(0)` oblige Python à décaler tous les éléments restants d'un cran vers la gauche : le coût de l'opération est proportionnel à la longueur de la file. L'implémentation d'une file avec une liste Python n'est donc pas la plus efficace ; d'autres implémentations, comme celle présentée à la section suivante, évitent ce décalage.

3. Applications des files

Les files apparaissent partout où des éléments doivent être traités dans l'ordre de leur arrivée.

■ **Exemple 2.12.**
La file d'impression. Les documents envoyés à une imprimante sont traités dans l'ordre d'arrivée : le premier document envoyé est le premier imprimé. C'est une file. ■

■ **Exemple 2.13.**
La file d'attente des clients. Dans un supermarché, les clients font la queue et sont dirigés vers une caisse dans l'ordre d'arrivée. C'est une file, et la simulation de ce système fait l'objet d'un exercice. ■

■ **Exemple 2.14.**
Les tâches d'un système. Les travaux envoyés à un serveur d'impression ou les paquets en attente d'envoi sur un réseau sont souvent gérés en file, dans l'ordre où ils arrivent. ■

IV. Une même interface, plusieurs implémentations

La section I a introduit la distinction entre **interface** et **implémentation**. Nous allons l'illustrer : la file définie à la section précédente a été implémentée avec une liste Python. On peut implémenter une file avec exactement la même interface en utilisant **deux piles**.

■ **Exemple 2.15.**
Une file peut être réalisée 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. ■

```
def enfiler(element):
    pile_entree.empiler(element)

def defiler():
    si pile_sortie est vide :
        transferer pile_entree vers pile_sortie
    renvoyer pile_sortie.depiler()
```

■ **Exemple 2.16.**
Les deux implémentations de la file (avec une liste ou avec deux piles) offrent la même interface : les méthodes `enfiler`, `defiler`, `longueur`, `est_vide` et `dernier` existent dans les deux cas. Un programme qui utilise une file n'a pas besoin de savoir laquelle des deux implémentations est utilisée : il n'utilise que l'interface. ■

R C'est le principe de **modularité** : on peut remplacer une implémentation par une autre sans modifier le reste du programme, du moment que l'interface est respectée. C'est aussi ce qui permet d'utiliser une bibliothèque ou une API sans connaître son fonctionnement interne.

V. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à écrire des programmes propres et à tester vos classes en créant des objets. Pour les exercices « avec les cartes », on peut matérialiser la pile ou la file avec des cartes à jouer, comme en classe.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire des piles et des files, lecture de code court, prédiction d'affichage, complétion de code. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 2.1.

Dire si les affirmations suivantes sont vraies ou fausses.

1. Dans une pile, le dernier élément ajouté est le premier retiré (mode LIFO).
2. Dans une file, le dernier élément ajouté est le premier retiré (mode FIFO).
3. La méthode `pop()` d'une liste retire et renvoie le premier élément.
4. La méthode `pop(0)` d'une liste retire et renvoie le dernier élément.
5. Une pile et une file se comportent exactement de la même manière.

■

■ Exercice 2.2.

On exécute le programme suivant, la classe `Pile` étant celle du cours :

```
p = Pile()
p.empiler(5)
p.empiler(7)
p.empiler(9)
print(p.depiler())
p.empiler(11)
print(p.hauteur())
```

Que va afficher ce programme ? Justifier en détaillant le contenu de la pile à chaque étape. ■

■ Exercice 2.3.

On exécute le programme suivant, la classe `File` étant celle du cours :

```
f = File()
f.enfiler("a")
f.enfiler("b")
print(f.defiler())
```

```
f.enfiler("c")
print(f.defiler())
print(f.dernier())
```

Que va afficher ce programme ? Justifier en détaillant le contenu de la file à chaque étape. ■

■ **Exercice 2.4.**
Compléter les phrases suivantes avec le bon verbe : `empiler`, `dépiler`, `enfiler` ou `défiler`.

1. Dans une pile, ajouter un élément au sommet, c'est ...
2. Dans une file, retirer l'élément en tête, c'est ...
3. Dans une file, ajouter un élément en fin de file, c'est ...
4. Dans une pile, retirer l'élément du sommet, c'est ...

■

■ **Exercice 2.5.**
Relier chaque expression Python à son rôle. Rappel de Première : ces expressions agissent sur une liste `lst`.

```
append(x) retire et renvoie le premier element de lst
pop() retire et renvoie le dernier element de lst
pop(0) ajoute x en fin de lst
len(lst) renvoie le dernier element de lst sans le retirer
lst[-1] renvoie le nombre d'elements de lst
```

■

Exercices d'application

Les exercices d'application demandent d'écrire des classes et des fonctions complètes, de simuler le fonctionnement des piles et des files, et d'utiliser les méthodes du cours.

■ **Exercice 2.6.**
Regrouper toutes les méthodes de la classe `Pile` vues dans le cours, puis ajouter une méthode `afficher` qui affiche la pile (afficher ce qui vous paraît le plus intéressant : par exemple les éléments du sommet vers le fond, avec leur nombre).

1. Avec les cartes : expliquer comment on lit une pile de cartes.
2. En pseudo-code : décrire les étapes de la méthode `afficher`.
3. Sur les ordinateurs, en Python : écrire la classe `Pile` complète et la tester.

■

■ **Exercice 2.7.**
Regrouper toutes les méthodes de la classe `File` vues dans le cours, puis ajouter une méthode `afficher` qui affiche la file (afficher ce qui vous paraît le plus intéressant : par exemple les éléments de la tête vers la fin, avec leur nombre).

1. Avec les cartes : expliquer comment on lit une file de cartes.
2. En pseudo-code : décrire les étapes de la méthode `afficher`.
3. Sur les ordinateurs, en Python : écrire la classe `File` complète et la tester.

■

■ **Exercice 2.8.**
Faire une méthode qui retourne l'ordre de la pile, c'est-à-dire le nombre de cartes qu'elle contient. On utilisera uniquement les méthodes de la classe `Pile` définies dans le cours.

1. Avec les cartes : compter les cartes d'une pile de cartes.
2. En pseudo-code.
3. Sur les ordinateurs, en Python.

■

■ **Exercice 2.9.**
Faire une fonction qui divise une pile en deux piles de même taille (si le nombre de cartes est impair, une pile aura une carte de plus que l'autre). On utilisera uniquement les méthodes de la classe `Pile`.

1. Avec les cartes : expliquer la méthode utilisée.
2. En pseudo-code.
3. Sur les ordinateurs, en Python.

■

■ **Exercice 2.10.**
Faire une fonction qui fusionne deux piles en alternant une carte de chaque pile. Par exemple, si la première pile contient A, B, C et la seconde D, E, la pile obtenue contient A, D, B, E, C. On utilisera uniquement les méthodes de la classe `Pile`.

1. Avec les cartes.
2. En pseudo-code.
3. Sur les ordinateurs, en Python.

■

■ **Exercice 2.11.**
Faire une fonction `melange_americain` qui fusionne deux piles en alternant une ou deux cartes (le 1 ou le 2 choisi aléatoirement) de chaque pile. On utilisera `random.randint(1, 2)` pour le choix aléatoire, et uniquement les méthodes de la classe `Pile`.

1. Avec les cartes.
2. En pseudo-code.
3. Sur les ordinateurs, en Python.

■

■ **Exercice 2.12.**
Considérons la suite d'instructions suivantes, où `F` est une file :

```
F <- Creer_file
enfiler 21 a F
enfiler 22 a F
N <- defiler F
enfiler 24 a F
enfiler 25 a F
N <- defiler F
```

À la fin de cette séquence :

1. Que contient `N` ?
2. Que contient la file `F` ?

■

■ **Exercice 2.13.**
Les méthodes `depiler` et `defiler` provoquent une erreur `IndexError` si la pile ou la file est vide.

1. Expliquer pourquoi Python renvoie cette erreur lorsque l'on appelle `pop()` ou `pop(0)` sur une liste vide.
2. Modifier les méthodes `depiler` et `defiler` pour qu'elles affichent un message d'erreur et renvoient `None` lorsque la structure est vide, au lieu de provoquer une erreur.
3. Tester sur une pile et une file vides.

■

Exercices de réflexion

Les exercices de réflexion demandent de concevoir des programmes plus complets et d'analyser des situations complexes, notamment des simulations.

■ **Exercice 2.14.**
Dans un supermarché, il y a 5 caisses et une file d'attente commune. Dès qu'une caisse est libre, le client en tête de file y est envoyé. Le temps de passage en caisse est aléatoirement compris entre 3 et 10 minutes. Il y a dix clients dans la file d'attente.

1. Réaliser une simulation de leurs passages en caisses et afficher en combien de minutes tous les clients sont passés.
2. Refaire quelques essais avec plus de clients.
3. Comparer le temps total pour 10, 20 et 50 clients : que constate-t-on ?

■

■ **Exercice 2.15.**
Un poste de contrôle est alimenté en pièces toutes les 7 minutes. Ces pièces sont de quatre types qui nécessitent respectivement 4, 6, 8 et 9 minutes pour être contrôlées. Les fréquences des quatre types sont identiques.

1. À l'aide d'une file, simuler le fonctionnement du poste de contrôle pendant 1 heure, puis 80 heures.
2. Calculer le nombre de pièces en attente de contrôle.
3. Déterminer la durée de séjour des pièces dans le poste de contrôle. Afficher celle de la dernière pièce.
4. Le contrôleur fait une pause de 15 minutes toutes les 4 heures. Déterminer le nombre d'heures effectives de travail.

■

■ **Exercice 2.16.**
Écrire une fonction `parentheses_correctes(expression)` qui utilise une pile pour vérifier que les parenthèses d'une expression sont correctement équilibrées. On empilera chaque parenthèse ouvrante `(`, et on dépilera à chaque parenthèse fermante `)`. L'expression est correcte si, à la fin, la pile est vide et si l'on n'a jamais tenté de dépiler une pile vide.

1. Expliquer avec les mains pourquoi la pile est la structure adaptée.
2. En pseudo-code.
3. Sur les ordinateurs, en Python, avec la classe `Pile` du cours.

■

■ **Exercice 2.17.**
On veut implémenter une file avec deux piles, comme expliqué dans le cours : une

pile d'entrée `pile_entree` et une pile de sortie `pile_sortie`.

1. Écrire une classe `File2Piles` dont le constructeur crée les deux piles, avec les méthodes `enfiler`, `defiler`, `longueur`, `est_vide` et `dernier`. On utilisera uniquement les méthodes de la classe `Pile`.
2. Vérifier que la classe obtenue a la même interface que la classe `File` du cours : mêmes noms de méthodes, mêmes paramètres, mêmes comportements.
3. Tester sur l'exemple de la section III : enfiler A, B, C, puis défiler deux fois.

■

■ **Exercice 2.18.**

Pour chacune des situations suivantes, indiquer si une pile ou une file est la structure de données adaptée, et justifier la réponse en une phrase.

1. Le bouton « annuler » (`Ctrl+Z`) d'un éditeur de texte.
2. La file d'attente d'un guichet de banque.
3. L'historique de navigation d'un navigateur web.
4. Les documents envoyés à une imprimante partagée.
5. Les appels de fonctions successifs d'un programme Python.

■