Piles et files

Résumé

Une pile d'assiettes dans un placard, une file d'attente à la boulangerie : ces situations de la vie quotidienne correspondent à deux structures de données fondamentales en informatique. La **pile** (on ne peut accéder qu'au dernier élément ajouté) et la **file** (on ne peut accéder qu'au premier élément ajouté) sont partout : la touche « annuler » de votre éditeur de texte utilise une pile, et l'imprimante gère les documents en attente avec une file. Comprendre ces structures, c'est comprendre comment les programmes gèrent l'ordre dans lequel les tâches sont traitées.

Prérequis : Les listes, Programmation orientée objet.

① Mémo

Définition 1 (La pile (stack) – LIFO)

Une pile fonctionne selon le principe **Last In, First Out** : le dernier élément empilé est le premier dépilé (comme une pile d'assiettes).

Opérations fondamentales :

`empiler(x)` Ajoute `x` au sommet (push).
`depiler()` Retire et renvoie l'élément au sommet (pop).
`est_vide()` Teste si la pile est vide.
`sommet()` Consulte l'élément au sommet sans le retirer (peek).

Toutes ces opérations sont en O(1).

Définition 2 (La file (queue) – FIFO)

Une file fonctionne selon le principe **First In, First Out** : le premier élément enfilé est le premier défilé (comme une file d'attente).

Opérations fondamentales :

`enfiler(x)` Ajoute `x` à la fin (enqueue).
`defiler()` Retire et renvoie l'élément en tête (dequeue).
`est_vide()` Teste si la file est vide.
`tete()` Consulte l'élément en tête sans le retirer.

Implémentation avec une liste Python

```
# Pile avec une liste
pile = []
pile.append(x)       # empiler
x = pile.pop()       # dépiler
len(pile) == 0       # est_vide
pile[-1]             # sommet

# File avec une liste (simple mais inefficace)
file = []
file.append(x)       # enfiler
x = file.pop(0)      # défiler (O(n) !)
```

Attention : `pop(0)` est en O(n) car Python décale tous les éléments. Pour une file efficace, utiliser `collections.deque`.

Applications

Pile Historique de navigation (bouton « retour »), évaluation d'expressions postfixées, appels récursifs, vérification de parenthèses.

File File d'impression, gestion de requêtes serveur, parcours en largeur d'un graphe (BFS).

Erreurs classiques

Code erroné	Code correct	Explication
`pile.pop()` sans vérifier `est_vide()`	`if not pile.est_vide(): pile.depiler()`	Dépiler (ou défiler) une structure vide lève une erreur `IndexError`. Toujours vérifier que la structure n'est pas vide avant.
Utiliser `file.pop(0)` pour défiler	Utiliser `collections.deque` et `popleft()`	`pop(0)` est en O(n) car Python décale tous les éléments. Avec `deque`, `popleft()` est en O(1).
Confondre pile et file : utiliser `pop()` pour une file	`pop()` pour une pile (LIFO), `popleft()` pour une file (FIFO)	`pop()` retire le *dernier* élément (pile, LIFO). Pour une file (FIFO), il faut retirer le *premier* élément avec `popleft()`.
Inverser l'ordre des opérandes : `a - b` au lieu de `b - a` (éval. postfixée)	Dépiler `b` puis `a`; calculer `a - b`	En notation postfixée, le premier opérande est empilé en premier : il est donc sous le second. L'ordre de dépilement est inversé.

② Exemples

Programme 1 · Implémentation d'une pile avec interface propre

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

    def empiler(self, x):
        self._data.append(x)

    def depiler(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self._data.pop()

    def sommet(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self._data[-1]

    def est_vide(self):
        return len(self._data) == 0

    def __len__(self):
        return len(self._data)
```

```
23
24      def __str__(self):
25          return str(self._data)
```

Programme 2 · Implémentation d'une file avec deque

```
1   from collections import deque
2
3   class File:
4       def __init__(self):
5           self._data = deque()
6
7       def enfiler(self, x):
8           self._data.append(x)
9
10      def defiler(self):
11          if self.est_vide():
12              raise IndexError("File vide")
13          return self._data.popleft()
14
15      def tete(self):
16          if self.est_vide():
17              raise IndexError("File vide")
18          return self._data[0]
19
20      def est_vide(self):
21          return len(self._data) == 0
```

Programme 3 · Vérification de parenthèses avec une pile

```
1   class Pile:
2       def __init__(self):
3           self._data = []
4
5       def empiler(self, x):
6           self._data.append(x)
7
8       def depiler(self):
9           if self.est_vide():
10              raise IndexError("Pile vide")
11          return self._data.pop()
12
13      def est_vide(self):
14          return len(self._data) == 0
15
16  def parentheses_valides(expr):
17      pile = Pile()
18      corresp = {')': '(', ']': '[', '}': '{'}
19      for c in expr:
```

```
        if c in "([{":
            pile.empiler(c)
        elif c in ")]}":
            if pile.est_vide() or pile.depiler() != corresp[c]:
                return False
    return pile.est_vide()

# parentheses_valides("((a+b)*[c-d])") donne True
# parentheses_valides("((a+b])")        donne False
```

Programme 4 · Inverser une chaîne avec une pile

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

    def empiler(self, x):
        self._data.append(x)

    def depiler(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self._data.pop()

    def est_vide(self):
        return len(self._data) == 0

def inverser_pile(chaine):
    p = Pile()
    for c in chaine:
        p.empiler(c)
    resultat = ""
    while not p.est_vide():
        resultat += p.depiler()
    return resultat
```

③ Exercices

Exercice 1 *Simulation manuelle* On exécute les opérations suivantes sur une pile initialement vide. Donner l'état de la pile après chaque opération et la valeur renvoyée par chaque `depiler` :

`empiler(3), empiler(7), empiler(1), depiler(),`

`empiler(4), depiler(), depiler().`

Exercice 2 *Évaluation postfixée* En notation postfixée (polonaise inverse), les opérateurs sont écrits après leurs opérandes : `3 4 + 2 *` signifie $(3 + 4) \times 2 = 14$.

1. Évaluer manuellement `5 1 2 + 4 * + 3 -` en montrant l'état de la pile à chaque étape.
2. Écrire une fonction `evaluer_postfixe(expression)` qui prend une chaîne en notation postfixée et renvoie le résultat.

Exercice 3 *File : simulation* Même exercice avec une file. On exécute :

`enfiler(3), enfiler(7), enfiler(1), defiler(),`

`enfiler(4), defiler(), defiler().`

Comparer les résultats avec ceux de la pile.

Solutions des exercices

Corrigé de l'exercice 1.

`empiler(3)` Pile : `[3]`.
`empiler(7)` Pile : `[3, 7]`.
`empiler(1)` Pile : `[3, 7, 1]`.
`depiler()` Renvoie 1. Pile : `[3, 7]`.
`empiler(4)` Pile : `[3, 7, 4]`.
`depiler()` Renvoie 4. Pile : `[3, 7]`.
`depiler()` Renvoie 7. Pile : `[3]`.

Le sommet est toujours à droite. On dépile toujours le dernier élément ajouté (LIFO).

Corrigé de l'exercice 2. **1. Trace de `5 1 2 + 4 * + 3 -` :**

Jeton lu	Action	État de la pile
5	empiler 5	`[5]`
1	empiler 1	`[5, 1]`
2	empiler 2	`[5, 1, 2]`
+	dépiler 2 et 1, empiler $1 + 2 = 3$	`[5, 3]`
4	empiler 4	`[5, 3, 4]`
*	dépiler 4 et 3, empiler $3 \times 4 = 12$	`[5, 12]`
+	dépiler 12 et 5, empiler $5 + 12 = 17$	`[17]`
3	empiler 3	`[17, 3]`
-	dépiler 3 et 17, empiler $17 - 3 = 14$	`[14]`

Résultat : 14.

2. Implémentation :

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

    def empiler(self, x):
        self._data.append(x)

    def depiler(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self._data.pop()

    def est_vide(self):
        return len(self._data) == 0

def evaluer_postfixe(expression):
```

```
    pile = Pile()
    for token in expression.split():
        if token in "+-*/":
            b = pile.depiler()
            a = pile.depiler()
            if token == '+': pile.empiler(a + b)
            elif token == '-': pile.empiler(a - b)
            elif token == '*': pile.empiler(a * b)
            elif token == '/': pile.empiler(a // b)
        else:
            pile.empiler(int(token))
    return pile.depiler()
```

Attention : l'ordre de dépilement compte pour la soustraction et la division : on dépile d'abord `b`, puis `a`, et on calcule `a - b` (et non `b - a`).

Corrigé de l'exercice 3.

`enfiler(3)` File : `[3]`.
`enfiler(7)` File : `[3, 7]`.
`enfiler(1)` File : `[3, 7, 1]`.
`defiler()` Renvoie 3. File : `[7, 1]`.
`enfiler(4)` File : `[7, 1, 4]`.
`defiler()` Renvoie 7. File : `[1, 4]`.
`defiler()` Renvoie 1. File : `[4]`.

Comparaison : la pile a renvoyé 1, 4, 7 (dernier entré, premier sorti). La file renvoie 3, 7, 1 (premier entré, premier sorti).

Les mêmes opérations donnent des résultats différents selon la structure utilisée.