Piles et files

Structures de données

Objectifs.
Prérequis : listes chaînées, programmation orientée objet
À l'issue de ce chapitre, vous saurez :

- définir les types abstraits *pile* (LIFO) et *file* (FIFO) et leurs opérations ;
- implémenter une pile et une file à l'aide de listes chaînées ou du type `list` de Python ;
- identifier quelle structure utiliser selon le contexte (pile d'appels, file d'attente) ;
- discuter l'efficacité des différentes implémentations en termes de complexité.

Introduction

Piles et files sont des structures de données linéaires, permettant de gérer des séquences d'éléments.

Piles et files diffèrent par le jeu d'opérations disponibles et par la politique de mémorisation des éléments dans la séquence.

Les piles sont en mode **LIFO** (Last In First Out : dernier entré premier sorti). Leur usage caractéristique en informatique est la pile des contextes d'exécution.

Les files sont en mode **FIFO** (First In First Out : premier entré, premier sorti). Leur usage caractéristique concerne les files d'attente.

Les piles

Le type abstrait pile

On définit ce type abstrait de données par ses opérations :

- création de pile vide ;
- ajout au sommet (souvent appelé `push` ou *empiler*) ;
- retrait du sommet (souvent appelé `pop` ou *depiler*) ;
- accès au nombre d'éléments.

Implémentation avec la classe `ListeChainee`

La classe `ListeChainee` est tout à fait adaptée pour les piles car les opérations les plus simples y sont l'ajout et la supression en tête de liste. Il suffit donc de poser que le sommet de la pile est la tête de la liste chaînée.

On choisit d'implémenter la création d'une pile vide et l'accès au nombre d'éléments par les méthodes spéciales `__init__` et `__len__`. Pour `empiler` et `depiler`, on réutilise les méthodes de la classe `ListeChainee`.

Remarque : on aurait pu aussi bien recopier le code des méthodes des listes chaînées pour obtenir une implémentation indépendante.

```python
from listechainee import ListeChainee

class Pile:
    def __init__(self):
        self.pile = ListeChainee()

    def __len__(self):
        return len(self.pile)

    def empiler(self, element):
        self.pile.ajout_en_tete(element)

    def depiler(self):
        if len(self) == 0:
            raise ValueError("depiler une pile vide")
        return self.pile.supprime_en_tete()
```

On peut tester l'usage de la pile en empilant puis en dépilant des éléments.

```python
p = Pile()
p.empiler(3)
p.empiler(5)
p.empiler(8)
print(p.depiler())
print(p.depiler())
print(p.depiler())
```

Activité : Redéfinir la classe `Pile` en utilisant seulement la classe `Noeud` mais sans utiliser la classe `ListeChainee`.

```python
class Noeud:
    def __init__(self, element, suivant=None):
        self.element = element
        self.suivant = suivant

class Pile:
    def __init__(self):
        """Crée une pile vide"""
        pass

    def __len__(self):
        """Donne la taille de la pile"""
        pass

    def empiler(self, element):
        """Ajoute un élément sur le sommet de la pile"""
        pass

    def depiler(self):
        """Enlève et renvoie l'élément situé sur le sommet de la pile"""
        pass
```

Implémentation avec les `list` de Python

On dispose sur les listes de la méthode `append`, qui ajoute à la fin, et de la méthode `pop` qui enlève un élément à la fin. Ce qui est important pour les piles est d'ajouter et de supprimer les éléments, à la même extrémité de la liste.

Remarque : On aurait aussi bien pu ajouter et supprimer au début avec les méthodes `insert(0, element)` et `pop(0)`.

```
class Pile:
    def __init__(self):
        self.pile = list()

    def __len__(self):
        return len(self.pile)

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

    def depiler(self):
        if len(self) == 0:
            raise ValueError("depiler une pile vide")
        return self.pile.pop()
```

On peut tester l'usage de la pile en empilant puis en dépilant des éléments.

```
p = Pile()
p.empiler(3)
p.empiler(5)
p.empiler(8)
print(p.depiler())
print(p.depiler())
print(p.depiler())
```

Exemple d'usage de la pile

Pour transformer la fonction récursive suivante

```
def fact (n) :
    return 1 if n == 0 else n * fact(n-1)
```

en fonction itérative, on décrémente `n` jusqu'à `0` en empilant ses valeurs successives, puis on dépile les valeurs pour multiplier le résultat intermédiaire, jusqu'à ce que la pile soit vide.

```
def fact (n):
    contexte = Pile()
    while not n == 0 :
        contexte.empiler(n)
        n = n - 1
    r = 1
    while len(contexte) > 0:
        n = contexte.depiler()
        r = n * r
```

```
    return r

fact(10)
```

Les piles sont extrêmement utiles en informatique et vous les utilisez quotidiennement, parfois même sans vous en rendre compte :

— La fonction annuler (Ctrl-Z) de votre traitement de textes par exemple est une pile : quand vous tapez Ctrl-Z, vous annulez la dernière opération effectuée. Quand vous faites une nouvelle opération, celle-ci est mémorisée au sommet de la pile. Vous ne pouvez pas annuler l'avant dernière opération sauf si vous annuler la dernière.
— Le bouton retour de votre navigateur internet fonctionne également à l'aide d'une pile. Les pages web consultées lors de votre navigation sur une page sont empilées et le bouton retour permet d'accéder à la dernière page présente sur la pile.
— Certaines calculatrices fonctionnent à l'aide d'une pile pour stocker les arguments des opérations : c'est le cas de beaucoup de calculatrices de la marque HP, dont la première calculatrice scientifique de poche : la HP 35 de 1972.

Les files

Le type abstrait file

On définit le type abstrait de données par ses opérations :

— création de file vide ;
— ajout en queue de file : enfiler (ou `enqueue`) ;
— retrait du premier élément de la file : défiler (ou `dequeue`) ;
— accès au nombre d'éléments.

Implémentation avec la classe `ListeChainee`

La classe `ListeChainee` peut être adaptée pour les files car on a programmé toutes les opérations d'ajout ou de suppression en tête ou en queue de liste chaînée.

Il faut cependant faire un choix. Si on choisit de défiler en tête (opération en temps constant) alors il faut enfiler en queue de liste chaînée (opération en temps linéaire).

On choisit d'implémenter la création d'une file vide et l'accès au nombre d'éléments par les méthodes spéciales `__init__` et `__len__`. Pour `enfiler` et `defiler`, on réutilise les méthodes de la classe `ListeChainee`.

Remarque : on aurait pu aussi bien faire le choix opposé en enfilant en tête de liste (opération en temps constant) pour défiler en queue (opération en temps linéaire).

```
from listechainee import ListeChainee

class File:
    def __init__(self):
        self.file = ListeChainee()

    def __len__(self):
        return len(self.file)

    def enfiler(self, element):
```

```
            self.file.ajout_en_queue(element)

        def defiler(self):
            if len(self) == 0:
                raise ValueError("defiler sur une file vide")
            return self.file.supprime_en_tete()
```

On peut tester la file, pour observer si les éléments sont bien *défilés*, dans l'ordre où ils ont été *enfilés*.

```
f = File()
f.enfiler(3)
f.enfiler(5)
f.enfiler(8)
print(f.defiler())
print(f.defiler())
print(f.defiler())
```

L'implémentation des files proposée avec les listes chaînées n'est pas satisfaisante car l'ajout en queue n'est pas efficace. Ce problème peut être simplement résolu en gardant pour une file deux références, une vers le premier maillon et une vers le dernier.

Activité : Redéfinir la classe `File`, en utilisant seulement la classe `Noeud`, mais sans utiliser la classe `ListeChainee` et en créant deux attributs pour chaque file, pour garder la référence vers les premier et dernier maillon de la chaîne.

```
class Noeud:
    def __init__(self, element, suivant=None):
        self.element = element
        self.suivant = suivant

class File:
    def __init__(self):
        """Crée une file vide"""
        self.premier = None
        self.dernier = None

    def __len__(self):
        """Donne la taille de la file"""
        pass

    def enfiler(self, element):
        """Ajoute un élément à la fin de la file"""
        pass

    def defiler(self):
        """Enlève et renvoie l'élément situé au début de la file"""
        pass
```

Tester le bon fonctionnement de cette implémentation de la file, en vérifiant que les deux opérations `enfiler` et `defiler` s'exécutent bien en temps constant.

```
f = File()
f.enfiler(3)
f.enfiler(5)
```

```
f.enfiler(8)
print(f)
print(f.defiler())
print(f.defiler())
print(f.defiler())
```

Implémentation avec les `list` de Python

On dispose sur les listes de la méthode `append`, qui ajoute à la fin, et de la méthode `pop(0)` qui enlève un élément au début de la liste.

Attention : En Python, l'ajout en fin de liste avec `append` est en temps constant, mais le `pop(0)` est en temps linéaire. Pour une évaluation de la complexité des opérations élémentaires de Python voir le site : Python.org - Time complexity

Remarque : On aurait aussi bien pu ajouter au début et supprimer à la fin, avec les méthodes `insert(0, element)` et `pop()`.

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

    def __len__(self):
        return len(self.file)

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

    def defiler(self):
        if len(self) == 0:
            raise ValueError("defiler sur une file vide")
        return self.file.pop(0)
```

On peut tester la file, pour observer si les éléments sont bien *défilés*, dans l'ordre où ils ont été *enfilés*.

```
f = File()
f.enfiler(3)
f.enfiler(5)
f.enfiler(8)
print(f.defiler())
print(f.defiler())
print(f.defiler())
```

Exemples d'usages des files

Les file d'attentes simples sont utilisées en informatique, dès qu'une ressource en accès exclusif est partagée entre plusieurs utilisateurs.

Exemple : File d'attente d'impression. Chaque utilisateur soumet une tâche en l'enfilant dans la file d'impression. Le serveur d'imprimante, défile une tâche dès que l'imprimante est disponible.

Bilan

Pour les **piles** et les **files**, on a vu successivement des implémentations du type abstrait, utilisant une classe de listes chainées, les listes de Python ou programmées directement sous forme de listes chaînées.

L'efficacité de ces différentes implémentations est à discuter en fonction de la complexité des opérations élémentaires utilisées.

Pour les piles, on a programmé les opérations efficacement, mais pour les files, on a dû faire un compromis entre les opérations, une seule des deux pouvant être efficace selon le choix effectué.

Pour une implémentation efficace des files, il suffit de garder les deux attributs `premier` et `dernier`, comme proposé dans l'activité : `enfiler` et `defiler` sont alors en temps constant. Python fournit aussi la classe `deque` du module `collections`, une *file à deux extrémités* dont les méthodes `append` et `popleft` sont en temps constant.

À retenir.

- Une **pile** (LIFO) n'autorise l'ajout et le retrait qu'au sommet ; une **file** (FIFO) enfile en queue et défile en tête.
- Avec des listes chaînées, les opérations de pile (en tête) sont en O(1). Pour les files, une implémentation naïve impose un compromis : l'une des deux opérations est en O(n).
- Pour obtenir `enfiler` et `defiler` toutes deux en O(1), on conserve deux références (`premier` et `dernier`) ou on utilise la classe `deque` du module `collections`.
- Usages typiques : la pile modélise la pile d'appels, l'annulation (Ctrl-Z), le bouton retour du navigateur ; la file modélise les files d'attente d'impression, les tampons réseau, l'ordonnancement des processus.