ALG. Piles et Files

I/ Généralités

Vous avez découvert avec les **listes chaînées** un nouveau moyen plus souple que les tableaux pour stocker des données. Ces listes sont particulièrement flexibles car on peut insérer et supprimer des données à n'importe quel endroit, à n'importe quel moment.
Les **piles** et les **files** sont deux variantes un peu particulières des listes chaînées. Elles permettent de contrôler la manière dont sont ajoutés les nouveaux éléments. Cette fois, on ne va plus insérer de nouveaux éléments au milieu de la liste mais seulement au début ou à la fin.
Les piles et les files sont très utiles pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure.

Les structures qui seront étudiées peuvent s'envisager sous deux aspects :

- Le côté utilisateur, qui utilisera une **interface** pour manipuler les données.
- Le côté concepteur, qui aura choisi une **implémentation** pour construire la structure de données.

Par exemple, le volant et les pédales constituent (une partie de) l'interface d'une voiture. L'implémentation va désigner tous les mécanismes techniques qui sont mis en œuvre pour que le mouvement de rotation du volant aboutisse à un changement de direction des roues.

1/ La structure de type **Pile**

Une structure de **pile** (*en anglais stack*) est associée à la méthode **LIFO** (*en anglais, Last In First Out*) : les éléments sont empilés les uns au-dessus des autres, et on ne peut toujours dépiler que l'élément du haut de la pile. **Le dernier élément à être arrivé est donc le premier à être sorti**.

On peut imaginer *une pile de pièces* (figure à droite). On peut ajouter des pièces une à une en haut de la pile, mais aussi en enlever depuis le haut de la pile. Il est en revanche impossible d'enlever une pièce depuis le bas de la pile (je n'ai jamais réussi du moins 😊)

Quelques cas concrets de l'utilisation d'une pile :

- Lors d'un DS (devoir surveillé), la dernière copie est (en principe) le première corrigée.
- Dans un navigateur, la fonction « back » permet de revenir à la page précédente (on « dépile » les pages).

2/ La structure de type **File**

Les files (*en anglais, queue*) ressemblent assez aux piles, si ce n'est qu'elles fonctionnent dans le sens inverse !
Dans ce système, les éléments s'entassent les uns à la suite des autres. Le premier qu'on fait sortir de la file est le premier à être arrivé. On parle ici d'algorithme **FIFO** (*en anglais, First In First Out*), c'est-à-dire « Le premier qui arrive est le premier à sortir ».

Il est facile de faire le parallèle avec la vie courante. Quand vous allez prendre un billet de cinéma, vous faites la queue au guichet (fig. suivante). À moins d'être le frère du guichetier, vous allez devoir faire la queue comme tout le monde et attendre derrière. C'est le premier arrivé qui sera le premier servi.

Quelques cas concrets de l'utilisation d'une file :

- Le stockage de denrées périssables.
- La gestion des impressions avec une imprimante en réseau.

3/ La problématique du stockage

Dans les entrepôts de stockage, comme dans les rayons d'un supermarché, la structure naturelle est celle de la **pile** : les gens attrapent l'élément situé devant eux («en haut de la pile»). Si les employés du supermarché remettent en rayon les produits plus récents sur le dessus de la pile, les produits au bas de la pile ne seront jamais choisis et périmeront.
Ils doivent donc transformer la pile en file : lors de la mise en rayon de nouveaux produits, ceux-ci seront placés derrière («au bas de la file») afin que partent en priorité les produits à date de péremption plus courte. On passe donc du LIFO au FIFO.

Certains dispositifs permettent de le faire naturellement :
Ci-dessous, une file... de piles (électriques) et de bonbons. Le chargement par le haut du distributeur fait que celle qui sera sortie (en bas) sera celle qui aurait été rentrée en premier (par le haut). Ce FIFO est donc provoqué naturellement par la gravité.

Un distributeur de piles

Un distributeur de bonbons

Une problématique universelle : https://www.mecalux.fr/blog/methode-lifo-fifo-peps

II/ Implémentation d'une pile en Python

1/ Squelette d'une pile

L'objectif est de créer une classe *Stack*. L'appel au **constructeur** *Stack()* créera une pile vide (appel à la méthode *__init__()*). Chaque objet instancié de la classe *Stack* disposera des méthodes suivantes :

- empty() : indique si la pile est vide.
- add_back() : insère un élément en haut de la pile.
- pop_back() : renvoie la valeur de l'élément en haut de la pile ET le supprime de la pile.
- __str__() : permet de surcharger la fonction *print()*.

2/ Création à partir du type *list*

Ce type est tout à fait puisqu'il possède déjà les méthodes nécessaires :

- La méthode *len()* permettra de savoir si la pile est vide (elle renvoie 0 en cas de liste vide).
- La méthode *append()* permet d'ajouter un élément en fin de liste.
- La méthode *pop()* permet de supprimer l'élément en fin de liste et de renvoyer sa valeur.

En savoir plus sur les méthodes du type construit *list* ici :
https://docs.python.org/fr/3/tutorial/datastructures.html#more-on-lists

Voici une implémentation possible de la classe *Stack* :

```
# Définition de la classe de type `Pile`
class Stack :
    def __init__(self):
        self.data = []

    # Si la pile n'a aucun élément alors elle est vide
    def empty(self):
        return len(self.data) == 0

    # Ajout d'un élément en fin de pile
    def add_back(self,x):
        self.data.append(x)

    # Suppression du dernier élément de la pile
    # et renvoi de sa valeur
    def pop_back(self):
        # Gestion du cas d'une pile vide
        if self.empty() == True :
            raise IndexError("La pile est vide")
        else :
            return self.data.pop()

    # Surcharge de la fonction `print()`
    def __str__(self):
        s = "|"
        for k in self.data :
            s = s + str(k) + "|"
        return s
```

```
31 # Jeu de tests
32 myPile = Stack()
33 print(myPile.empty())  # Attendu : True
34 myPile.add_back("Yop")
35 print(myPile.empty())  # Attendu : False
36 myPile.add_back(5)
37 print(myPile) # Attendu : | Yop | 5 |
38 print(myPile.pop_back())  # Attendu : 5
39 print(myPile)   # Attendu : | Yop |
40 myPile.pop_back()
41 print(myPile.empty())  # Attendu : True
42
```

```
True
False
|Yop|5|
5
|Yop|
True
```

3/ Création à partir de la classe *Cell* (Cellule)

On peut utiliser la définition d'une liste chaînée pour créer une classe *Stack*.

Rappel : une liste chaînée permet d'accéder simplement à la valeur courante et à l'élément suivant.

Voici une implémentation possible à partir de la classe *Cell* :

```
# Classe gérant une liste chaînée
class Cell :
    def __init__(self, val, nex):
        self.value = val
        self.next = nex
```

```
# Classe gérant une pile
class Stack:
    def __init__(self):
        self.data = None

    def empty(self):
        return self.data == None

    def add_back(self, x):
        self.data = Cell(x,self.data)

    def pop_back(self):
        # Cas d'une pile vide
        if self.empty() :
            raise IndexError("Pile vide")

        v = self.data.value # Récupèration de la valeur à renvoyer
        self.data = self.data.next  # Suppression la 1ère cellule
        return v                    # et renvoi de sa valeur

    def __str__(self):
        s = "|"
        c = self.data
        while c != None :
            s += str(c.value) + "|"
            c = c.next
        return s
```

```
33 # Jeu de tests
34 myPile = Stack()
35 myPile.add_back(5)
36 myPile.add_back("yop")
37 print(myPile) # Attendu : |yop|5|
38 print(myPile.empty()) # Attendu : False
39 myPile.pop_back()
40 print(myPile) # Attendu : |5|
41 print(myPile.pop_back())  # Attendu : 5
42 print(myPile.empty())   # Attendu : True
43
```

```
|yop|5|
False
|5|
5
True
```

III/ Implémentation d'une file en Python

1/ Squelette d'une file

L'objectif est de créer une classe *File*. L'appel au **constructeur** *File()* créera une file vide (appel à la méthode *__init__()*). Chaque objet instancié de la classe *File* disposera des méthodes suivantes :

- empty() : indique si la file est vide.
- add_back() : insère un élément à la fin de la file.
- pop_front() : renvoie la valeur de l'élément au début de la file ET le supprime de la file.
- __str__() : permet de surcharger la fonction *print()*.

2/ Création d'une file à partir d'une liste chaînée

La structure linéaire de liste chaînée donne une manière de réaliser efficacement une file à condition de pouvoir accéder **efficacement à l'élément à sa fin** sans devoir la parcourir en entier. Le plus simple est de conserver un pointeur vers ce dernier élément dans la file.

Pour l'ajout d'un élément en début de la file, la liste chaînée correspond parfaitement.

Remarque : contrairement à la pile, le fait de devoir ajouter un élément en début de la file rend la classe *list* du langage Python peu adaptée.

Voici une implémentation possible à partir de la classe *Cell* :

```
# Classe gérant une liste chaînée
class Cell :
    def __init__(self, val, nex):
        self.value = val
        self.next = nex


# Classe gérant une file
class File:
    def __init__(self):
        self.first = None
        self.last = None # Accès permanent au dernier élément

    def empty(self):
        return self.first == None

    # Ajout en fin de la "queue"
    def add_back(self, x):
        new_cell = Cell(x,None)

        # Si la file est vide
        if self.empty() :
            self.first = new_cell
        else :
            self.last.next = new_cell

        self.last = new_cell
```

Remarques :

- Si la file est vide à l'appel de la méthode *add_back(self, x)*, le premier élément est la cellule nouvellement créée *new_cell*, sinon, on indique que la cellule nouvellement créée est maintenant l'élément suivant de l'ancien dernier élément de la file.
- Dans tous les cas, *new_cell* devient le premier élément de la file (dernier arrivé à la queue pour schématiser).

```python
    def pop_front(self):
        # Cas d'une file vide
        if self.empty() :
            raise IndexError("File vide")

        v = self.first.value # Récupèration de la valeur à renvoyer
        self.first = self.first.next  # Suppression la 1ère cellule

        # Si la file est maintenant vide,
        # le dernier élément est aussi `None`
        if self.first == None :
            self.last = None
        return v                    # et renvoi de sa valeur
```

```python
    def __str__(self):
        s = "|"
        c = self.first
        while c != None :
            s += str(c.value) + "|"
            c = c.next
        return s
```

```python
# Jeu de tests
myFile = File()
myFile.add_back(5)
myFile.add_back("yop")
print(myFile) # Attendu : |yop|5|
print(myFile.empty()) # Attendu : False
myFile.pop_front()
print(myFile) # Attendu : |yop|
print(myFile.pop_front())  # Attendu : yop
print(myFile.empty())   # Attendu : True
```

```
|5|yop|
False
|yop|
yop
True
```

3/ Réalisation d'une file à partir de deux piles

On peut réaliser **une file avec deux piles** (elles-mêmes à partir d'une liste chaînée ou d'une *list* en Python). Le principe est celui-ci :

- Une **première pile** (pile d'entrée) est celle qui permet **d'ajouter des éléments**.
- La **seconde pile** (pile de sortie) est celle qui **renvoie les éléments** lors d'un dépilement.

Si on souhaite ajouter un élément : il suffit de le mettre au-dessus de la pile d'entrée.
Si on souhaite dépiler un élément, il y a deux possibilités :

- Si la pile de sortie n'est pas vide, on renvoie le dernier élément.
- Si elle vide, on **dépile toute la pile d'entrée** dans la pile de sortie et on renvoie le dernier élément.

Activité : simuler cette réalisation à l'aide d'un jeu de cartes.

Exemple de programme de réalisation d'une classe *File* à l'aide deux piles :

```
# Définition de la classe de type `Pile`
class Stack :
    def __init__(self):
        self.data = []

    # Si la pile n'a aucun élément alors elle est vide
    def empty(self):
        return len(self.data) == 0

    # Ajout d'un élément en fin de pile
    def add_back(self,x):
        self.data.append(x)

    # Suppression du dernier élément de la pile
    # et renvoi de sa valeur
    def pop_back(self):
        # Gestion du cas d'une pile vide
        if self.empty() == True :
            raise IndexError("La pile est vide")
        else :
            return self.data.pop()
```

```
class File :
    def __init__(self) :
        self.entry = Stack()
        self.exit = Stack()

    # Les deux piles doivent être vides pour que
    # la file le soit
    def empty(self) :
        return self.entry.empty() and self.exit.empty()

    # Ajout d'un élément dans la pile d'entrée
    def add_back(self,val) :
        self.entry.add_back(val)
```

```
    # Suppression du dernier élément
    def pop_front(self) :
        # S'il n'y a aucun élément
        if self.empty() :
            raise IndexError("La file est vide !")

        # Si la pile de sortie n'est pas vide
        if not self.exit.empty() :
            return self.exit.pop_back()

        # On dépile la pile d'entrée dans la pile
        # de sortie : le dernier élément sera bien
        # le premier de la pile d'entrée
        while not self.entry.empty() :
            self.exit.add_back(self.entry.pop_back())
        return self.exit.pop_back()
```

Remarque : l'affichage de cette file et le jeu de tests feront l'objet d'un exercice.

A savoir : La pile est une structure de données linéaire de type « LIFO » (dernier entré, premier sorti). Elle peut être réalisée à partir d'une liste chaînée ou d'une *list* (tableau) en Python.
La file est aussi une structure de données linéaire mais de type « FIFO » (premier entré, premier sorti). Elle peut être réalisée à partir d'une liste chaînée mutable ou encore avec deux piles.