Les piles et les files

I Introduction

Nous utilisons en Python des objets élémentaires de type ***int, float, str, bool***, ou plus complexes comme les types ***list*** ou ***tuple***. La construction et l'utilisation de ces objets, avec des propriétés, des opérations, des méthodes, des fonctions, sont déjà prévues par le langage.
Mais nous pouvons aussi construire et utiliser nos propres objets ou structures de données. Cela signifie dans ce cas que nous devons implémenter de manière explicite un ensemble d'objets avec toutes les fonctions utiles ou nécessaires, en particulier des opérations de construction, d'accès et de modification.

Dans ce chapitre, nous allons étudier les **piles**, ("stack" en anglais), qui seront construites en utilisant les objets de type list. Une liste a un début et une fin et il est possible d'accéder à chacun de ses éléments par l'intermédiaire d'un index. Les **piles** peuvent être considérées comme des listes avec des conditions d'utilisation restreintes : il est possible d'insérer ou de supprimer un élément uniquement à la fin, (en anglais "LIFO", acronyme de "Last In First Out" : dernier entré, premier sorti).
En modifiant la condition d'insertion et de suppression, nous pouvons obtenir une file ("queue" en anglais). C'est encore un objet qui peut être considéré comme la restriction d'une liste.

Une **file** autorise l'insertion d'un côté et la suppression de l'autre (en anglais "FIFO", acronyme de "First In First Out" : premier entré, premier sorti). Ceci correspond par exemple à la file d'attente à une caisse.
Il est important d'avoir toujours en tête l'image d'une pile concrète pour bien comprendre ce qu'il est possible de faire. Si nous disposons d'assiettes que nous pouvons manier seulement une par une, nous commençons par choisir un endroit où les poser : c'est la création d'une pile vide. Ensuite, nous posons une première assiette à l'endroit choisi, puis une deuxième sur la première et ainsi de suite. Nous empilons les assiettes une par une en les posant sur la pile : ce sera le rôle de la fonction empiler. Nous pouvons les reprendre dans l'ordre inverse en commençant par la dernière ajoutée, toujours une par une : ce sera le rôle de la fonction dépiler. Si les assiettes sont empilées dans un placard, la capacité est limitée par la hauteur du placard.
Si nous sommes en extérieur, il n'y a pas de limites, à part la taille de l'échelle dont nous aurons besoin pour empiler ou dépiler et un problème de stabilité ! Enfin, nous pouvons constater si une pile est vide ou pas et nous pouvons compter le nombre d'assiettes empilées pour obtenir la taille de la pile.
Cette structure de pile est utilisée par la plupart des microprocesseurs : la pile correspond à une zone de la mémoire, et le processeur retient l'adresse du dernier élément. Nous la rencontrerons également dans le chapitre sur la récursivité.

La structure de **file** est elle aussi utilisée dans un ordinateur pour mémoriser temporairement une suite d'actions en attente qui seront traitées suivant l'ordre d'arrivée des demandes. C'est le cas, par exemple, avec les mémoires tampons ("buffers" en anglais), les serveurs d'impression et les moteurs multitâches d'un système d'exploitation.

A retenir :

Les **piles** et les **files** sont des **listes** particulières : on accède aux éléments par les **extrémités**, c'est-à-dire au début ou à la fin.

Elles sont utilisées par exemple pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure.

On distingue :
- les piles (« *stacks* ») : le premier empilé est le dernier à être dépilé, « LIFO » (Last In first Out),
- les files (« *queue* ») : le premier entré est le premier à sortir, « FIFO » (First In First Out).

L'implémentation des piles et des files peut se faire avec des **listes chaînées** ou des **tableaux**.

II Rappel sur les listes

Les objets de type list, que nous appelons des listes, sont étudiés en première année. Ils sont très souvent utilisés et il est bon de faire quelques rappels.

1 Définition

Une liste est un ensemble ordonné d'éléments éventuellement hétérogènes dont les valeurs peuvent être modifiées.
Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

2 Création d'une liste

```
liste1=[] # une liste vide
# ou liste1=list()
liste2=['a'] # une liste contenant un unique élément
liste3=['a','bonjour',17]
liste4=['d','e']
liste3[1]='b' # modification d'un élément
print(liste3) # affiche ['a','b',17]
liste5=liste3+liste4
print(liste5) # affiche ['a','b',17,'d','e']
liste6=3*liste4 # soit ['d','e','d','e','d','e']
```

La fonction ***list()*** permet de convertir certains objets en listes. Nous pouvons aussi l'utiliser avec la fonction ***range*** pour initialiser une liste d'entiers :

```
liste=list('bonjour') # ['b','o','n','j','o','u','r']
liste1=list(range(4)) # [0,1,2,3]
liste2=list(range(1,4)) # [1,2,3]
liste3=list(range(2,14,3)) # [2,5,8,11]
```

Construction par compréhension :

```
liste=[2*x-1 for x in range(1,5)] # construit [1,3,5,7]
```

Nous pouvons construire une autre liste en itérant sur les éléments d'une liste :

```
liste2=[2*x for x in liste] # construit [2,6,10,14]
```

Copie d'une liste : pour créer une nouvelle liste, copie d'une liste existante, le code est le suivant :

```
liste1=[0,2,4,6,8]
liste2=list(liste1)
# ou liste2=liste1[:]
```

Attention : ce code peut poser problème si les éléments de la liste sont eux-mêmes des listes.

```
liste1=[[0,1],[2,3],[4,5]]
liste2=list(liste1)
liste2[1][0]=8 # modifie aussi liste1
print(liste1) # affiche [[0,1],[8,3],[4,5]]
```

Pour éviter cela, nous devons plutôt écrire :

```
liste1=[[0,1],[2,3],[4,5]]
liste2=[list(x) for x in liste1]
```

Le module copy propose la fonction ***deepcopy*** pour effectuer une vraie copie en profondeur :

```
from copy import deepcopy
liste1=[[0,1],[2,3],[4,5]]
liste2=deepcopy(liste1)
```

Insertion et extraction

La méthode append permet d'ajouter un élément à la fin d'une liste :

```
liste=['a','b']
liste.append('c') # (liste=['a','b','c'])
```

La méthode insert permet d'insérer un objet dans une liste.

```
liste=['a','b','d','e']
liste.insert(2,'c') # l'élément 'c' est inséré à l'index 2
print(liste) # affiche ['a','b','c','d','e'],
# 'd' et 'e' sont décalés vers la droite
```

La syntaxe pour extraire une sous-liste est la suivante :

```
liste=['a','b','c','d','e','f']
liste2=liste[1:4] # liste2 est la liste ['b','c','d']
# liste[-1] est le dernier élément soit 'f'
```

Suppression d'un élément : pour supprimer et récupérer un élément d'une liste, nous utilisons la méthode ***pop*** qui supprime l'élément dont l'indice est passé en paramètre et le renvoie.

```
liste=['a','b','d']
x=liste.pop(2) # l'élément d'indice 2 est affecté dans x
# et il est supprimé de la liste
print(liste) # affiche ['a','b','d']
print(x) # affiche 'c'
```

La valeur par défaut du paramètre de la fonction pop, (l'indice), est -1. Donc à l'exécution de l'instruction ***liste.pop(),*** l'élément en fin de liste est supprimé et renvoyé.

La méthode ***remove*** permet de supprimer un élément de valeur donnée :

```
liste=['a','b','c','d','c','e']
liste.remove('c') # l'élément 'c' est supprimé
# seul le premier rencontré !
print(liste) # affiche ['a','b','d','c','e'],
# 'd', 'c' et 'e' sont décalés vers la gauche
```

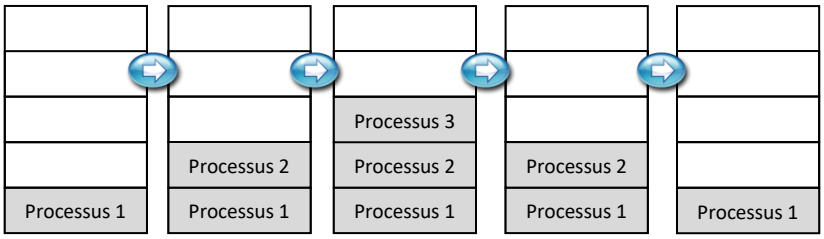
III Les piles

Dernier arrivé Premier traité

Exemple 1 : Empilement de dossiers

Dans une pile de dossiers, le dernier arrivé est le premier traité.

Exemple 2 : Gestion de processus par un système d'exploitation.
Dans un ordinateur, lorsqu'un « processus 1 » fait appel à un « processus 2 » qui fait lui-même appel à un « processus 3 », l'ensemble est stocké dans une table des processus, propre au noyau du système d'exploitation. Lorsque le « processus 3 » se termine, le système sait qu'il doit revenir au « processus 2 », puis au « processus 1 ».



NOTA : On retrouve cet empilement dans les appels récursifs de fonction. Lors d'un débordement de pile, le message « stack overflow » apparaît.

Exemple 3 :
Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur désempile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente »

Les principales fonctions associées aux piles sont :
- **Ajouter** au sommet : empiler (« **push** »),
- **Supprimer** du sommet : dépiler (« **pop** »),

Algorithme 1 Ajouter au sommet de la pile
Données : T : un tableau [1..n]
x : un élément à empiler
Résultat : le tableau T avec l'élément supplémentaire « x » empilé [1..n+1]

Empiler(T,x)

Algorithme 2 Supprimer du sommet de la pile
Données : T : un tableau [1..n]
Résultat : le tableau T avec le dernier élément supprimé [1..n-1]
r : le dernier élément

Depiler(T)

NOTA : Avant d'enlever un élément, il faut s'assurer que le nombre de données empilées n'est pas nul.

Exemple 4 : Expressions postfixées.
Une expression est dite postfixée si l'opérateur suit les opérandes.
En notation inverse polonaise (Reverse Polish Notation), l'opérateur suit toujours le deuxième opérande.

Par Exemples :

Cette notation est très pratique car elle est non ambigüe.
On lit l'expression de gauche à droite :
-
-

Algorithme 3 Evaluation d'une expression postfixée
Données : T : un tableau de nombres et d'opérateurs, de taille n
Résultat : r : le résultat du calcul de l'expression postfixée

Evaluer(T)

Comment implémenter les opérations sur les piles en Python ?

Rappelons tout d'abord quelques spécificités des listes sous Python :

- Suppression de termes : l'instruction $del\ L[i:j]$ supprime les éléments d'indices i à j-1.
- Copie d'une liste : la syntaxe est $M = list(L)$. Attention à l'instruction $M = L$:

```
L=[0,1,2,3,4,5,6]
M=list(L)
del L[0:2]
print(L)
print(M)
```

Mais :

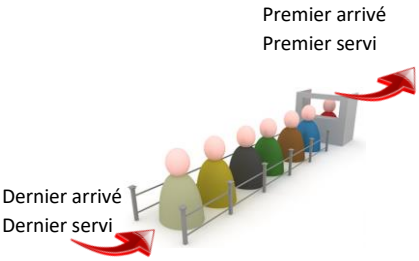
```
L=[0,1,2,3,4,5,6]
M=L
del L[0:2]
print(L)
print(M)
```

Spécification informelle.	Implémentation sous Python.
Créer une pile vide.	
Empiler un élément (push) : pas d'autre façon d'insérer un élément que de la placer au sommet de la pile	
Saisir le sommet de la pile (peek): pas d'autre élément accessible dans une pile.	
Désempiler(pop) : on enlève le sommet de la pile, qui est aussi le dernier placé dans la pile (LIFO)	
Tester si la pile est vide : nécessaire pour ne pas dépiler une pile déjà vide.	
Affichage de la pile :	

IV Les files

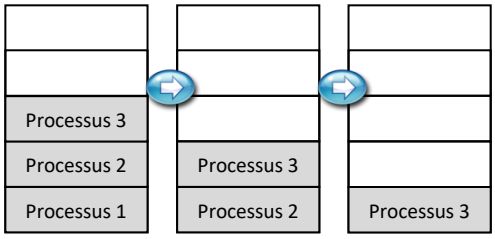
Exemple 5 : File d'attente

Dans une file d'attente, le premier arrivé est le premier servi.



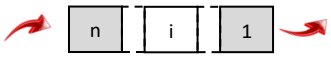
Exemple 6 : Gestion de processus par un système d'exploitation
Dans un ordinateur, lorsque des appels aux processus « 1 », « 2 », puis « 3 » se succèdent, l'ensemble est stocké dans une table des processus, propre au noyau du système d'exploitation. Lorsque le « processus 1 » se termine, le système sait qu'il doit passer au « processus 2 », puis au « processus 3 ».

Processus 1 en cours
Processus 2 et 3 en attente



NOTA : On retrouve cette notion de file dans les appels successifs de fonction, la gestion des travaux d'impression, les « buffers », etc.

Les principales fonctions associées aux files sont :
- **Ajouter** en queue : enfiler (« **enqueue** »),
- **Supprimer** tête : défiler (« **dequeue** »),
- **Est_vide**, **Est_pleine**.



Algorithme 4 Ajouter en queue de file
Données : T : un tableau [1..n]
x : un élément à enfiler
Résultat : le tableau T avec l'élément supplémentaire « x » enfilé [1..n+1]

Enfiler(T,x)

NOTA : Avant d'ajouter un nouvel élément, il faut s'assurer que le nombre de données enfilées n'est pas trop grand.

Algorithme 5 : Supprimer la tête de la pile
Données : T : un tableau [1..n]
Résultat : le tableau T avec le premier élément supprimé [1..n-1]
r : le premier élément

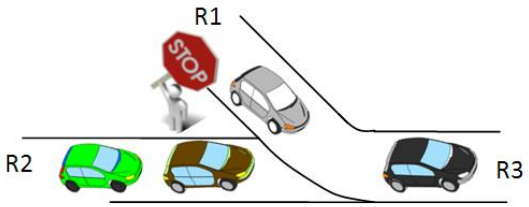
Defiler(T)

NOTA :

Exemple 7 : Croisement routier

Problème : Pour simuler un croisement routier, à sens unique, on utilise 3 files f1, f2 et f3 représentant respectivement les voitures arrivant sur des routes R1 et R2, et les voitures partant sur la route R3. La route R2 a un STOP. Les voitures de la file f2 ne peuvent avancer que s'il n'y a aucune voiture sur la route R1, donc dans la file f1. On souhaite écrire un algorithme qui simule le départ des voitures sur la route R3, modélisée par la file f3.

On utilise trois tableaux f1, f2 et f3 de valeurs booléennes :
- « 1 » symbolise la présence d'une voiture,
- « 0 » l'absence de véhicule.



Algorithme 6 : Croisement routier
Données : f1, f2 : deux tableaux de booléens [1..n1] et [1..n2]
Résultat : le tableau f3 simulant le départ des voitures sur la route R3 [1..n1+n2]

Croisement_routier(f1,f2)
    i1 ← 1
    i2 ← 1
    Tant que i1<=n1 et i2<=n2 **faire**