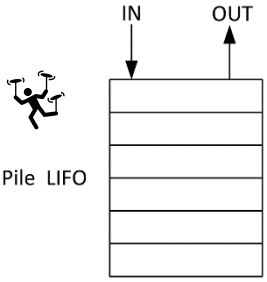
Informatique Pour Tous

Cours/ TD 21 – Gestion des piles (stacks) et des files (queues)

A / Éléments de cours : les différents types de piles

LES PILES



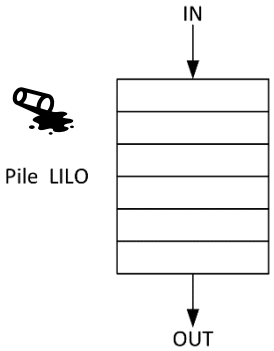
Une pile peut être de type ***LIFO*** : **L**ast **I**n **F**irst **O**ut (dernier entré, premier sorti).
C'est un mode de gestion de type pile d'assiettes. On ajoute et on enlève des éléments du même côté de la pile.

Dans ce cours/TD **nous modéliserons une pile comme une liste**, dont la tête (entrée) est le début de liste, et la queue est la fin. Il se peut que la convention inverse soit adoptée dans de futurs sujets, c'est-à-dire un empilement (*push*) par la fin de la liste.

LES FILES

Une pile peut également être de type ***LILO*** : **L**ast **I**n **L**ast **O**ut (dernier entré, dernier sorti). On parle alors de **files**, comme dans le cas des files d'attente notamment.
On les nomme également ***FIFO*** : **F**irst **I**n **F**irst **O**ut
C'est un mode de gestion de type gobelets dans un distributeur de boissons :

- On recharge en gobelets pour le dessus de la pile.
- On prend le gobelet d'en bas pour se servir une boisson.



Pour ce type de piles, on peut vouloir conserver la valeur de l'élément sortant, lors d'un dépilement (pop). *A priori*, si l'on ne précise rien, cet élément de queue de liste est supprimé et perdu. Dans ce TP, nous choisirons que le dépilement renvoie la valeur de l'élément supprimé (qui pourrait, par exemple, être stocké en tête d'une seconde pile, type mémoire secondaire) en même temps qu'il supprime le dernier élément de la pile.

Exemple

→ L'utilisation des piles est pratique pour des langages informatiques de bas niveau (proches du processeur, loin de l'application). Il existe quelques contre exemples de haut niveau, comme la gestion de l'historique d'un navigateur (commandes *page précédente* et *suivante*), ou la commande *Undo* (Ctrl + Z) ou *Redo* (Ctrl + Y) sur votre système d'exploitation, qui seraient gérées par des piles de type ***LIFO***.

→ On peut également citer la gestion de la mémoire tampon (*buffer*) par votre ordinateur lorsqu'il communique avec un périphérique d'acquisition (exemple : un oscilloscope) : un nombre donné de points, fixé par la taille de mémoire tampon, est conservé en mémoire. À chaque instant d'échantillonnage, une nouvelle acquisition est faite et le dernier point mesuré entre en début de liste, chassant le dernier point de la liste. Autrement dit, dans cette gestion de type ***LILO***, on a un *tampon* qui peut se mesurer en unité de temps (par exemple 2 secondes) et à tout instant t, chaque mesure M_i dans la liste correspond à la mesure réalisée à l'instant $t - i \times \delta t$ (avec δt période d'échantillonnage). La taille N de la liste fixe le temps de mémoire tampon ($N \times \delta t$ est la mesure la plus ancienne sauvegardée). À l'instant suivant, le « vieillissement des données » fait glisser toutes les données d'un cran dans la liste ($i \to i + 1$), une nouvelle mesure est acquise et prend la 1ère place, et la taille limite de la mémoire tampon oblige à supprimer le dernier point qui correspondrait à la mesure en $(N + 1) \times \delta t$.

Ainsi, on comprend qu'une pile a généralement une **capacité** maximale (nombre d'éléments maxi dans la pile).

- Tant que la taille de la pile n'atteint pas la capacité, on ajoute (*empiler*, ou *push*) des éléments sans en supprimer (*dépiler*, ou `pop`).
- Lorsque la taille de la pile atteint sa capacité, tout ajout implique une suppression d'élément.

Dans le cadre de notre programme d'informatique, nous ferons beaucoup appel à des algorithmes récursifs, la gestion des variables pour les appels successifs est alors gérée par un système de pile. Vous pourrez parfois, lorsqu'il y a trop d'appels récursifs, voir apparaître le code d'erreur « attention, capacité de la pile excédée », qui se traduit en anglais par : `Warning, stack overflow` *(référence pour ceux qui connaissent cet excellent forum).*

B / Pré-requis : notion d'arguments optionnels

Lorsque nous utilisons `range(a,b,p)`, nous pouvons l'utiliser avec un, deux ou trois arguments (cf. cours 19). Par défaut, *Python* comprend donc que :

- Si on appelle `range(b)` ceci sous-entend que `a = 0` et `p = 1`
  Exemple : `range(5)` renvoie `[0,1,2,3,4]` # liste de 5-0 = 5 éléments
- Si on appelle `range(a, b)` ceci sous-entend que `p = 1`
  Exemple : `range(2,6)` renvoie `[2,3,4,5]` # liste de 6-2 = 4 éléments
- L'appel peut être fait en spécifiant les 3 valeurs
  Exemple : `range(-2,10,3)` renvoie `[-2,1,4,7]` # liste de $\frac{10-(-2)}{3}$ = 4 éléments

Quand on code des fonctions à plusieurs arguments, il arrive que certains de ces arguments prennent le plus souvent une valeur donnée. Auquel cas, pour simplifier l'appel de la fonction, on peut donner à ces arguments une valeur **par défaut**.

Exemple 1 :
```
def fonc(x,b) :
    return x**b
```

➔ Imaginons qu'en pratique, on veuille se servir de cette fonction quasiment exclusivement pour calculer x^2, mais en pratique on a codé la fonction de sorte à pouvoir calculer d'autres puissances de x au cas où on ait besoin de s'en resservir plus tard, dans un autre cadre, ou de la mettre à disposition d'autres utilisateurs (bibliothèque).

➔ Alors, on peut écrire, plutôt :

```
def fonc(x,b = 2) :  # b a pour valeur par défaut 2
    return x**b
```

➔ Cette affectation lors de la déclaration de la fonction est interprétée comme suit :

- Si l'appel est fait avec un seul argument, alors on interprète que `b = 2`
- Si l'appel est fait avec 2 arguments, alors on interprète le second argument comme la valeur de *b*

➔ L'appel de la fonction pourra alors se faire de plusieurs façons :

- `fonc(3)` → on comprend que `b = 2`, donc ceci renverra $3^2 = 9$
- `fonc(3, 2)` → idem
- `fonc(3, 3)` → on fixe alors `b = 3`, donc ceci renverra $3^3 = 27$
- `fonc(3, b=3)` → idem, pour comprendre, voir ci-dessous

Un problème se pose s'il y a plusieurs arguments facultatifs… Imaginons qu'on ait défini une fonction avec l'en-tête

```
def fonc2(x,a = 2, b = 3) :
    ...
```

Si on appelle `fonc2(7,4)`, il est clair que `x = 7`, mais à quelle variable `a` ou `b` affecte-t-on la valeur 4 ?
En d'autres termes, doit-on comprendre que `a = 4` et que `b = 3` (valeur par défaut) ou bien `a = 2` (valeur par défaut) et que `b = 4` ?

➔ Dans le cas des fonctions à plusieurs arguments optionnels, on fera l'appel en spécifiant la variable (*kwarg*) à laquelle est affectée la valeur qui n'est pas celle par défaut.

- `fonc2(3)` → on comprend que `x = 3` et `a = 2, b = 3` (valeurs par défaut)
- `fonc2(-2, a = 5)` → on comprend que `x = -2` et `a = 5` et `b = 3` (valeur par défaut)
- `fonc2(1, b = 0)` → on comprend que `x = 1`, et `a = 2` (valeur par défaut) et `b = 0`
- `fonc2(1, a = 5, b = 0)` est alors équivalent à `fonc2(1,5,0)`

L'intérêt est que si `fonc2` a énormément d'arguments optionnels (voir par exemple `plt.plot(X,Y,**kwargs)` il y a des dizaines d'arguments optionnels !!), on n'est pas obligés de faire l'appel en spécifiant toutes les valeurs de ces arguments.

Exemple 2 :

```
import math as m # car on va avoir besoin de m.ceil() : arrondi supérieur

def range2(b, a=0,p=1) :   # je veux re-coder range() moi-même, version « home made »
                          # petit problème, les arguments optionnels doivent toujours être déclarés après les
                          # arguments non optionnels, on n'aura donc pas exactement le même ordre d'arguments
                          # que range(a,b,p)
    N_pts = m.ceil((b-a)/p)
    L_sortie = [a] # on commence forcément à a
    while len(L_sortie) < N_pts : # on quitte dès que L_sortie bonne taille
        Nouvelle_val = L_sortie[-1] + p # dernière valeur + le pas p
        L_sortie.append(Nouvelle_val)
    return L_sortie
```

Pour retrouver les appels que nous avons vus en début de partie, nous aurions :

- `range2(5)` serait l'équivalent de `range(5)` (par défaut, `a=0` et `p=1`)
- `range2(6,a=2)` serait l'équivalent de `range(2,6)` (par défaut, `p=1`)
- `range2(10,a=-2, p=3)` ou `range2(10,-2,3)` serait l'équivalent de `range(-2,10,3)`

C / Gestion des deux types de piles : LIFO et FIFO

Q1 – Décrire précisément le fonctionnement de la fonction ci-dessous (lignes 4 et 5 variantes experts seulement).

```
1  def LIFO(L, val = "OSEF", depil = False) : # dernier entré, premier sorti
2      # On remplit par la gauche, on vide par la gauche
3      global capacite
4      if len(L) > capacite :                                                    variante
5          raise NameError("La taille de la liste excede la capacite" )
6      if depil :
7          val_sortie = L[0]
8          del L[0]
9          return val_sortie
10     elif len(L) < capacite :
11         L.insert(0, val)
12     else :
13         print("La capacité étant de ", capacite, "la valeur ", val," n'a pas pu
                   etre ajoutee a la liste" )
```

Q2 – En vous appuyant sur une structure identique (hormis lignes 4 et 5), écrire une fonction de gestion de piles **FIFO**, `FIFO(L, val = "OSEF", depil = False)`. On adoptera la convention d'entrée par la gauche et de sortie par la droite de liste.

Remarque : dans le cas précédent (pile **LIFO**) un élément ajouté qui aurait fait « sortir » de la capacité de la liste était dans le même temps ajouté et supprimé de la liste, d'où l'affichage du message en ligne 13. Ici, l'ajout de l'élément se fait par la gauche, mais dans ce cas on doit également dépiler par la droite. L'élément supprimé sera renvoyé par la fonction.

D / Quelques utilisations de piles

Dans la suite de ce sujet, nous allons nous forcer à utiliser uniquement les fonctions **LIFO** et **FIFO** codées précédemment. Ces fonctions peuvent être incluses dans des boucles. Les fonctions qui vont suivre seront ***procédurales*** (pas de **return**) et modifieront la liste d'entrée (en argument).

Q3 – Écrire et tester une fonction **`permut_circ( L, n )`** prenant en argument une liste *L* et un nombre entier naturel *n*. Cette fonction fera *n* permutations circulaires de la liste d'entrée. Comme souvent, pensez à faire des `print` intermédiaire pour « voir » ce qu'il se passe !

Exemple : pour `L = [1,2,3,4,5]`

Après avoir tapé `permut_circ( L, 1 )` la liste *L* vaudra `[5,1,2,3,4]`

Ou, après avoir tapé `permut_circ( L, 3 )` la liste *L* vaudra `[3,4,5,1,2]`

Variante experts : faire tenir cette fonction sur une seule ligne de code (excepté la « def … » évidemment).

Q4 – Écrire et tester une fonction **`echange_listes( L1, L2 )`** prenant en argument deux listes *L1* et *L2* de même taille, qui permute terme à terme les deux listes.

Exemple : pour `A = [1,2,3]` et `B = [4,5,6]`

Alors après avoir tapé `echange_listes(A, B)` on aura `A` valant `[4,5,6]` et `B` valant `[1,2,3]`

Q5 – Écrire et tester une fonction **`depiler_jusqua( L, v_cherchee )`** prenant en argument une liste *L* et une valeur *v_cherchée* (pouvant être contenue ou non dans la liste *L*).

- Cette fonction va dépiler (par la gauche) la pile jusqu'à trouver la valeur cherchée. Cette valeur ne sera pas supprimée de la liste, mais toutes les valeurs à sa gauche le seront.
- Si la valeur n'est pas trouvée, on videra totalement la liste d'entrée.

Exemple : pour `L = [1,10,5,3,2]`

Après avoir tapé `depiler_jusqua( L, 3 )` la liste *L* vaudra `[3 , 2]`

Ou, après avoir tapé `depiler_jusqua ( L, 17 )` la liste *L* vaudra `[ ]`

Variante experts : appuyez-vous sur un « `while 1 :` » et arrangez-vous pour que le code ne soit jamais en exécution infinie.