Structures de données - Piles et files

Rappels

Structures de données

Structure	entier	réel /flottant	booléen	n-uplet	liste	chaîne	ensemble	dictionnaire	tableau	fichier	pile	file	arbre
Type Python	int	float	bool	tuple	list	str	set	dict	array	file			

Pour mémoire, autre structure de données : base de données.

Temps d'accès

En informatique, un ***tableau*** est une structure de données de taille fixe (statique) définie par un temps d'accès à un élément en ***O(1)*** c'est-à-dire indépendant du nombre d'éléments du tableau (autrement dit, le temps d'accès est indépendant de la place de l'élément dans le tableau).

En python, les ***listes*** (type list) se comportent comme des tableaux du point de vue du temps d'accès qui est donc en ***O(1)***. Noter que les listes sont des structures de taille variable (dynamique).

Les opérations d'ajout / suppression ***en fin de liste*** (cf. pop et append ci-dessous) sont ***rapides*** tandis que ces mêmes opérations en tête de liste (insert) sont lentes (décalage des éléments de la liste).

Opérations / méthodes sur les listes

L = [1, 2, 3, 4, 5]

Attention :

M = L ***ne copie pas*** la liste L dans M : M et L sont deux noms différents (ou alias) qui pointent vers la ***même*** plage d'adresses mémoires.
Conséquence : si M est modifiée, L est modifiée !
La copie doit être effectuée par M = L[0 **:** len(L)] ou encore **M =L[:]**.

len(L) renvoie la longueur de la liste (5 ici).
L.append(8) ***ne renvoie rien*** mais ***ajoute*** un élément à la fin de la liste (L devient [1, 2, 3, 4, 5, 8]).
L.pop() ***renvoie le dernier élément*** (8 ici) et ***supprime*** cet élément de la liste (L devient [1, 2, 3, 4, 5]).
L.insert(0,e) insert l'élément *e* en tête de liste (index 0).

Attention à la syntaxe !

- ✓ len(L) est une ***fonction*** qui attend en argument un tuple, une chaîne, un ensemble, une liste.
- ✓ append() et pop() sont des ***méthodes*** de l'objet-liste L. Syntaxe = objet**.**méthode().

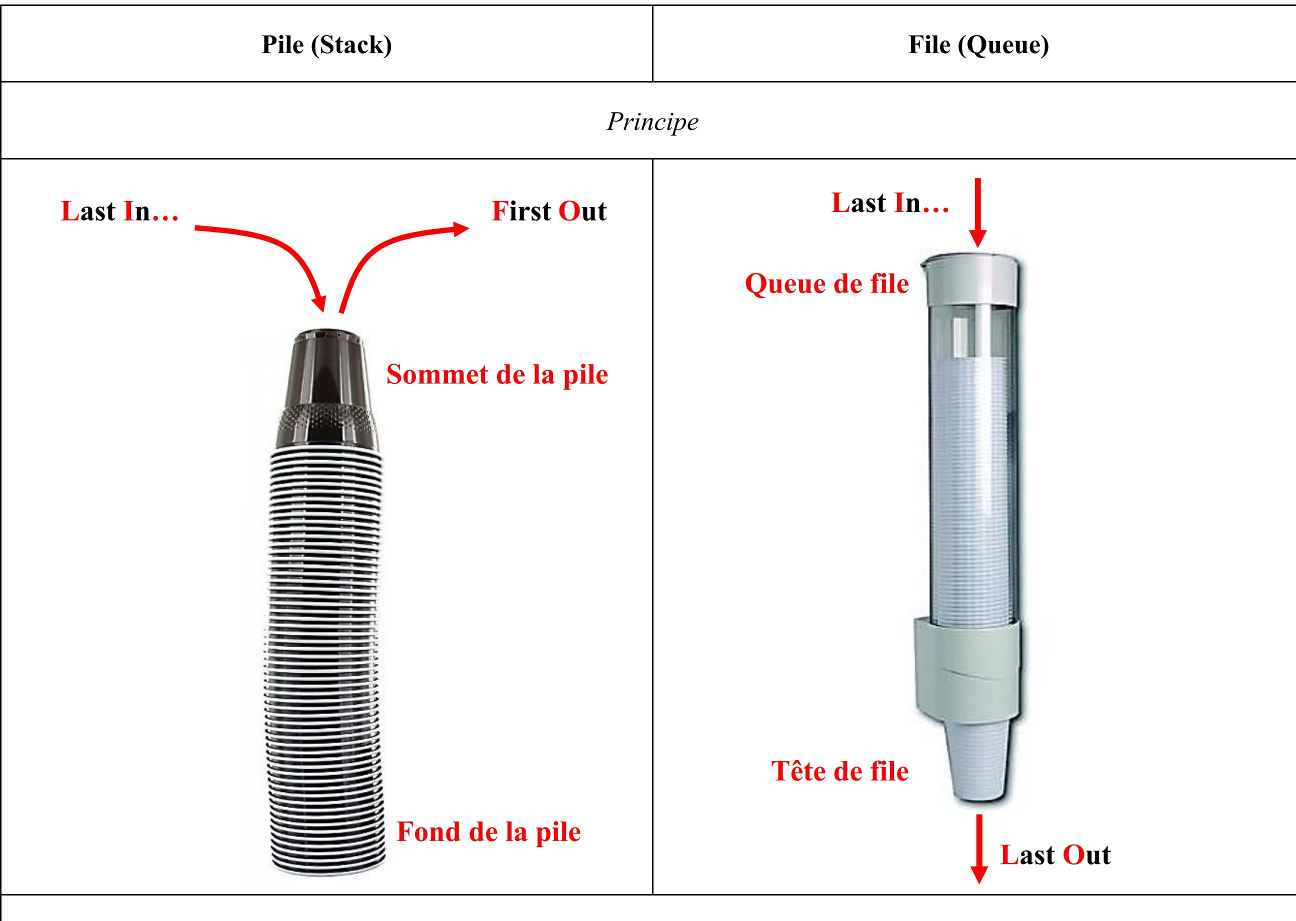
Création et remplissage d'une liste :

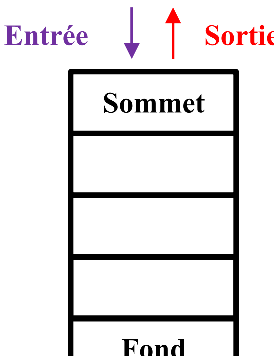
Comprendre les différences entre les 5 propositions ci-dessous pour créer la liste [0, 2, 4, 6, 8, 10, 12, 14, 16, 18].

<u>Création et remplissage simultanés</u> : `L = [2*n for n in range(10)]` (liste de 10 éléments)

<u>Création puis remplissage</u> : ***attention, l'incrémentation dépend de l'initialisation de la liste***

<table>
<tr><td></td><td colspan="2">Incrémentation utilisant des éléments précédents de la liste</td><td></td></tr>
<tr><td rowspan="2">Incrémentation par ajout d' une valeur car L[n+1] <u>ne préexiste pas</u> dans la liste</td><td>L = [0] 'liste à 1 élément
for n in range(9):
    L.append(L[n]+2) 'Ajouts</td><td>L = 10*[0] 'liste à 10 éléments
for n in range(9):
    L[n+1] = L[n]+2 'Mise à jour</td><td rowspan="2">Incrémentation par remplacement d' un élément <u>préexistant</u></td></tr>
<tr><td>L = [0]
for n in range(1,10):
    L.append(2*n)</td><td>L = 10*[0]
for n in range(1,10):
    L[n] = 2*n</td></tr>
<tr><td></td><td colspan="2">Incrémentation simple (observer les bornes de range)</td><td></td></tr>
</table>

<table>
<tr><th>Pile (Stack)</th><th>File (Queue)</th></tr>
<tr><td colspan="2">Principe</td></tr>
<tr><td colspan="2">
</td></tr>
<tr><td colspan="2">Spécifications</td></tr>
<tr><td>
Dernier arrivé, premier sorti = LIFO (Last In First Out).
Les derniers seront les premiers…(Matthieu 20:16).

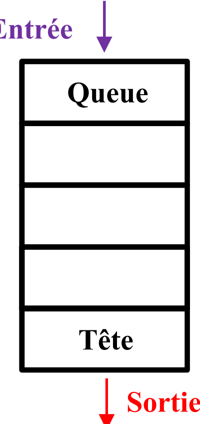


Primitives (opérations fondamentales) :

Tester si la pile est vide.
Ajouter un élément sur la pile (push).
Retirer un élément de la pile (pop).
Afficher le sommet de la pile.

Ces opérations doivent se faire en temps en O(1).

Rq : les spécifications exactes de ces opérations peuvent varier (cf. TP).
</td><td>
Dernier arrivé, dernier sorti = LILO (Last In Last Out).



Primitives :

Tester si la file est vide
Ajouter un élément dans la file.
Retirer un élément de la file.
Afficher l'élément en tête de file

Ces opérations doivent se faire en temps en O(1).
</td></tr>
</table>

<table>
<tr><th colspan="2">Applications</th></tr>
<tr><td>Structure de données utilisée par :
- les processeurs ;
- les langages (paramètres des fonctions, variables locales, fonctions récursives…) ;
- les navigateurs web (page précédente) ;
- les logiciels (fonction « Undo ») ;
- certains éditeurs de texte pour la vérification des parenthèses (Spyder, Notepad++).</td><td>Structure de données utilisée par :
- Queue à la caisse !
- les mémoires tampons (buffers) ;
- les systèmes d'exploitation multitâches qui répartissent le temps-machine entre différentes tâches ;
- les serveurs d'impression qui traitent les requêtes dans l'ordre d'arrivée.</td></tr>
<tr><td colspan="2">Dans les deux cas, ces structures sont destinées au stockage temporaire de données en cours de traitement (le stockage durable utilise des fichiers).</td></tr>
<tr><th colspan="2">Implémentation en python</th></tr>
<tr><td>Grâce au type list : pile sans limite de taille (taille dynamique).
Grâce au type array : pile de taille fixe, il faut alors gérer le cas de la pile pleine.

Rq : le message « stack oveflow » qui signale un débordement de pile n'est jamais bon signe…</td><td>Le type list n'est pas vraiment adapté (il s'agit alors d'une simulation et non d'une implémentation) car les opérations d'ajout / suppression en début de liste sont lentes.</td></tr>
</table>

Pour implémenter ou simuler les piles et les files à l'aide de listes, il faut garder à l'esprit les schémas suivants :

Piles	**Files**
Fond → [8, 5, 9, 2] ← Sommet Fond → [1er, …, dernier] ← Sommet	Queue → [8, 5, 9, 2] ← Tête Queue→ [dernier, …, 1er] ← Tête
Dernier élément de la liste = sommet	Dernier élément de la liste = tête de file
Ajouts et retraits par la droite	Ajout par la gauche – Retrait par la droite

Piles sans limite de taille

Bibliothèque de fonctions pour les piles

Écrire en python les fonctions suivantes en utilisant les listes et les opérations associées :

1. creer_pile() (renvoie une pile vide), la création de pile dans un programme se fait par : P = creer_pile () ;
2. taille(p) (renvoie la hauteur de la pile) ;
3. pile_vide(p) (teste si la pile p est vide, renvoie un booléen) ;
4. empiler(p, e) (ajoute l'élément e à la pile p, la pile p est donc modifiée par la fonction) ;
5. depiler(p) (renvoie l'élément dépilé, la pile p est modifiée par la fonction) ;
6. sommet(p) (renvoie l'élément au sommet de la pile p sans le dépiler) ;
7. copier(p) (renvoie une copie de la pile p), la copie dans un programme se fait par : Q = copier(P).

Rq : ces fonctions peuvent paraître triviales (certains points de syntaxe sont néanmoins à méditer) mais il s'agit ici de construire une ***bibliothèque de fonctions dédiées aux piles à l'exclusion de toute autre***.

Tests des fonctions

1. Comment fait-on pour afficher l'élément dépilé ?
2. Interpréter les messages d'erreurs «IndexError : pop from empty list » et « IndexError : list index out of range » et modifier les fonctions concernées si nécessaire.
3. Comparer les solutions proposées et celles que vous avez écrites.

Exercices

Seules les fonctions de la bibliothèque ci-dessus sont autorisées (il n'est plus question de listes).

1. Écrire une fonction inverser_pile(p) qui retourne une pile inversée *sans modifier p* (procéder à cette opération, à la main, avec quelques feuilles de papier : l'algorithme en découle !).
2. Écrire une fonction depilerK(p, k) qui désempile k éléments si la pile p en contient au moins k et désempile toute la pile sinon sans provoquer d'erreur.
3. Écrire une fonction depilerE(p, e) qui désempile la pile p tant que l'élément e (entier) n'est pas rencontré et que p n'est pas vide.
4. ***Facultatif -*** Écrire une bibliothèque analogue pour la structure de file.

Facultatif - Piles de taille finie

But :

> Implémenter des fonctions analogues aux précédentes en utilisant la structure *array* de numpy pour des piles de **taille finie**.
>
> Comme le tableau est de dimension fixe, la taille de la pile doit être stockée dans le tableau (et actualisée à chaque empilage/dépilage), on utilise la ***première case*** du tableau pour mémoriser cette taille.

Rappel :

L'instruction np.zeros((n), dtype=np.int) crée un tableau de n entiers.

Deux versions de cet exercice sont proposées

*Exercice version 1 (5/2) – Recherche de code (difficulté **) :*

Ecrire la bibliothèque de fonctions pour gérer les piles de taille finie et ajouter une fonction pile_pleine(p) renvoyant True si la pile est pleine et False sinon.

*Exercice version 2 (3/2) – Analyse de code (difficulté *) :*

1. Répondre aux questions placées en commentaires dans ce script (les noms des fonctions sont les mêmes que précédemment à la lettre finale « F » près signifiant « pile de taille **F**inie »).
2. Écrire une fonction sommetF(p) qui renvoie l'élément au sommet de la pile sans le dépiler *en utilisant uniquement les fonctions ci-dessous*.

```
1  import numpy as np
2
3  def creer_pileF(dim) :
4      return np.zeros((dim+1),dtype=np.int)    # Pourquoi dim + 1 ?
5
6  def tailleF(p) :
7      return p[0]
8
9  def pile_videF(p) :
10     return tailleF(p)==0
11
12 def pile_pleineF(p) :
13     return tailleF(p)==len(p)-1              # Pourquoi len(p) - 1 ?
14
15 def afficheF(p, txt='') :
16     print(txt, p[1:tailleF(p)+1])           # Justifier les bornes
17
18 def empilerF(p,e) :
19     if not pile_pleineF(p) :
20         p[0] += 1                   # Attention à l'ordre de ces 2 instructions
21         p[tailleF(p)] = e           # Modification si interverties
22     else :
23         return None
24
25 def depilerF(p) :
26     if not pile_videF(p) :
27         i = p[0]
28         p[0] -= 1
29         return p[i]
30     else :
31         return None
```

Application 1 : vérification des expressions parenthésées

Dans un texte constitué uniquement de parenthèses, de nombres et de symboles des opérations usuelles (+, -, /, *) on souhaite vérifier que chaque parenthèse ouvrante est bien associée à une parenthèse fermante.
Dans un second temps, le script précédent pourra être généralisé aux textes contenant des crochets et des accolades.

Exemples :
L'analyse par le script du « texte » 74*(1-9/(7+1.2)) ne doit générer aucune erreur.
L'analyse de 74*(1-9/7+1.2)) ou de 74*(1-9/(7+1.2) doit provoquer un message d'erreur.

Analyse du problème
1. Quelle(s) structure(s) de données peut-on envisager pour stocker la grandeur d'entrée (le « texte ») ?
2. Proposer au moins une méthode (utilisant une pile…) pour comptabiliser les parenthèses ouvrantes et/ou fermantes.
3. Ecrire une fonction renvoyant le booléen True si le parenthésage est correct et False sinon.
4. Tester cette fonction avec les 3 exemples donnés ci-dessus.

Application 2 : notation post-fixée ou polonaise inversée

La notation polonaise inversée ou notation post-fixée permet de représenter sans parenthèses, de façon non ambigüe, les expressions algébriques (habituelles ou infixées). Ces expressions deviennent des suites d'opérandes et d'opérateurs, les opérateurs étant placés après leurs opérandes (cf. exemples ci-dessous).

Exemples :

Notation infixée	Notation post-fixée
a + b	a, b, +
a + b*c	a, b, c, *, +
(a+b)*c	a, b, +, c, *
a*(b+c)	a, b, c, +, *

Principe :
L'évaluation de l'expression utilise une pile. L'expression est lue de gauche à droite :
- si le terme lu est un opérande (nombre), il est empilé ;
- si le terme est un opérateur, les deux termes du haut de la pile sont dépilés, le calcul est effectué et empilé.

Expression à évaluer : 7, 5, 1, +, *					
Terme lu	7	5	1	+	*
Pile					
			1		
		5	5	6	
	7	7	7	7	42

On souhaite écrire un programme qui donne la valeur d'une expression en notation post-fixée. Dans un premier temps, on restreindra le problème à des expressions composées d'entiers pour les seules opérations d'addition et de multiplication (ℕ , +, *).

Analyse du problème
1. Structures de données - L'expression sera fournie sous la forme d'un tuple de la forme (1, 2, 3, '+', '*') ou 1, 2, 3, '+', '*' (noter les guillemets qui transforment les symboles des opérateurs en chaîne.
2. Le script est directement issu du fonctionnement illustré ci-dessus.