Informatique 2 — Listes, piles, files

1 Listes en Python (rappels)

Voici la liste complète des méthodes des objets de type liste :
`L=[]` : fabrication d'une liste vide.
`L.append(x)` : ajoute un élément à la fin de la liste ; équivalent à `L[len(L):] = [x]`.
`L.extend(L)` : étend la liste en y ajoutant tous les éléments de la liste fournie ; équivalent à `L[len(L):] = L`.
`L.insert(i, x)` : insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer. Ainsi `L.insert(0,x)` insère l'élément en tête de la liste, et `L.insert(len(L),x)` est équivalent à `L.append(x)`.
`L.remove(x)` : supprime de la liste le premier élément dont la valeur est `x`. Une exception est levée s'il existe aucun élément avec cette valeur.
`L.pop([i])` : enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiqué alors `L.pop()` enlève et retourne le dernier élément de la liste. Les crochets autour du `i` dans la signature de la méthode indiquent bien que ce paramètre est facultatif, et non que vous devez placer des crochets dans votre code !
`L.index(x)` : retourne la position du premier élément de la liste ayant la valeur `x`. Une exception est levée s'il n'existe aucun élément avec cette valeur.
`L.count(x)` : retourne le nombre d'éléments ayant la valeur `x` dans la liste.
`L.sort()` : trie les éléments de la liste (des arguments optionnels permettent de choisir certains paramètres).
`L2 = sorted(L)` : trie les éléments de la liste, sans modifier la liste initiale.
`L.reverse()` : inverse l'ordre des éléments de la liste, en place.

Attention : les méthodes `insert`, `append`, `remove`, `reverse`, `sort` et `pop` modifient la liste !
Méthodes utiles : `filter` (filtre `L` selon critère), `map` (applique une fct à `L`) et `reduce` (réduit `L` selon une fonction).

ligne de commande pour `L = [75, 3, 3, 1, 1234.5]`	ce que contient `L` ou affichage
`len(L)`	affiche `5`
`print(L.count(3), L.count(75), L.count('x'))`	affiche `2 1 0` et `L` reste inchangé
`L.insert(2, -1)`	`[75, 3, -1, 3, 1, 1234.5]`
`L.append(3)`	`[75, 3, -1, 3, 1, 1234.5, 3]`
`L.index(3)`	affiche `1` et `L` reste inchangé
`L.remove(3)`	`[75, -1, 3, 1, 1234.5, 3]`
`L.reverse()`	`[3, 1234.5, 1, 3, -1, 75]`
`L.sort()`	`[-1, 1, 75, 3, 3, 1234.5]`
`L2 = sorted()`	construit `L2`, et `L` reste inchangé
`L.pop()`	affiche `1234.5` et `L` devient `[-1, 1, 75, 3, 3]`
`def f(x): return x % 2 != 0 and x % 3 != 0` `filter(f, range(2, 25))`	`[5, 7, 11, 13, 17, 19, 23]`
`def cube(x): return x*x*x` `map(cube, range(1, 5))`	`[1, 8, 27, 64]`
`def add(x,y): return x+y` `reduce(add, range(1, 11))`	`55`

2 Utiliser des listes comme des piles

Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti », ou LIFO pour « last-in, first-out »).

Deux méthodes sont déjà définies dans Python :

- Pour ajouter un élément `x` sur la pile `P`, utilisez la commande `P.append(x)`.
- Pour récupérer l'objet au sommet de la pile `P`, utilisez la commande `P.pop()`, sans indicateur de position.

Ce sont les deux opérations « licites » pour les piles, toutes les autres opérations sont à construire à partir de ces deux opérations élémentaires.

ligne de commande	ce que contient `P` ou affichage
`P = []`	`[]`
`P.append(2)`	`[2]`
`P.append(3)`	`[2, 3]`
`P.append(4)`	`[2, 3, 4]`
`P.pop()`	affiche `4` et `P` devient `[2, 3]`

Exercice 1 :

On utilisera uniquement les deux méthodes ci-dessus (`append` et `pop`) et celles qu'on programmera soi-même.

1. Créer une fonction `EstVide` qui vérifie si une pile est vide, sans modifier la pile.
2. Créer une fonction `Peek` qui renvoie l'élément sur le dessus de la pile, sans modifier la pile (peek = coup d'oeil).
3. Créer une fonction `Hauteur` qui détermine la hauteur d'une pile (sans utiliser `len(P)` !), sans modifier la pile.
4. Créer une fonction `CopiePile` qui renvoie la copie d'une pile, sans modifier la pile.
   Évaluer le coût en mémoire et le nombre d'opérations de cette fonction.
5. Créer une fonction `RenversePile` qui renvoie la copie renversée d'une pile, sans modifier la pile.
   Évaluer le coût en mémoire et le nombre d'opérations de cette fonction.
6. Créer une fonction `PermutPile` qui renvoie la pile après une permutation circulaire (`["a","b","c"]` devient `["c","a","b"]` : on prend une assiette sur le dessus de la pile et on la met sous la pile).
   Évaluer le coût en mémoire et le nombre d'opérations de cette fonction.
7. Créer une fonction `NPermutPile` qui prend en argument un entier n et une pile `P` et qui renvoie la pile après n permutation circulaire (`["a","b","c","d"]` devient `["b","c","d","a"]` pour $n = 3$: on prend n assiettes sur le dessus de la pile et on les met sous la pile).
   Évaluer le coût en mémoire et le nombre d'opérations de cette fonction.

Exercice 2 :

Dans un logiciel de calcul formel ou, plus généralement dans un éditeur de texte, il y a une gestion dynamique (i.e. « à la volée ») du parenthésage : si une parenthèse fermante de trop est ajoutée ou si elle n'est pas de même nature (une accolade au lieu d'un crochet par exemple) que la parenthèse ouvrante associée, alors un message d'erreur est envoyé. Par exemple, parmi les trois expressions suivantes :

$$3 * (x + 4), \quad 4 * (x - y], \quad (2 * x - \{4/3) + z\}$$

seule la première est correctement parenthésée.

Écrire une fonction `ParenthesageOK` qui reçoit comme argument une chaîne de caractères `C`, en analyse la correction du parenthésage et renvoie à l'utilisateur un booléen adapté (`True` si bon parenthésage, `False` sinon).

L'analyse de la chaîne `C` de caractères se fera caractère après caractère. On commencera par convertir `C` en une liste : `L=list(C)`. Puis on gèrera une pile contenant les parenthèses ouvrantes, et on comparera chaque parenthèse fermante au sommet (éventuel) de la pile.

3 Utiliser des listes comme des files

Il est également possible d'utiliser une liste comme une file (d'attente), où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti », ou FIFO pour « first-in, first-out »). Les deux instructions sont :
`F.append(x)` : enfile `x` sur la file `F`.
`F.pop(0)` : défile `F`, c'est-à-dire fait sortir le premier élément de la file `F`.

Ce sont les deux opérations « licites » pour les files, toutes les autres opérations sont à construire à partir de ces deux opérations élémentaires.

Exercice 3 :

Écrire des méthodes `CopieFile` et `PermutationFile` analogues à `CopiePile` et `PermuteFile`.

Toutefois, les listes ne sont pas très efficaces pour ce type de traitement. Alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertion ou de retrait en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).

On utilisera plutôt les objets `deque` (*double ended queue*, queue à deux bouts) :

```
from collections import deque   ♯ on importe le total
F = deque(L)   ♯ fabrique une file à partir de la liste L
F.append(x)   ♯ enfile x sur F
F.popleft()   ♯ défile l'élément en tête de file
```

ligne de commande	ce que contient `F` ou affichage
`F = deque(["Agnès", "Edith"])`	`["Agnès", "Edith"]`
`F.append("Margot")`	`["Agnès", "Edith", "Margot"]`
`F.popleft()`	affiche `"Agnès"` et `F` devient `["Edith", "Margot"]`