Informatique Pour Tous

TD 8 – Rappels sur la portée des variables, type « classiques » et « conteneurs »

Dans cette partie nous nous focaliserons sur le cas particulier des listes, mais la plupart des objets permettant de stocker plusieurs valeurs rencontreront le même problème : la plupart des fonctions *natives* de *Python,* et que nous coderons, seront des fonction *procédurales,* qui affecteront la liste en argument sans avoir nécessairement besoin d'un return !

Pour les différents tests de cette partie, nous utiliserons la liste de test : `L_test = list(range(10))` ce qui affecte à *L_test* la valeur [0, 1, 2, 3, …, 9].

A / Renverser une liste

Tout comme l'affectation, la permutation de deux éléments a par défaut une portée globale, qui affecte la liste en argument à l'extérieur de la fonction.
Pour réaliser une permutation de deux éléments (ce qui est classique quand on fait du tri en place, ce qui est justement l'objectif puisqu'on souhaite affecter la liste sans en faire de copie), on tape `L[i],L[j] = L[j],L[i]`

Q1 – Compléter les **lignes 10, 11** pour obtenir une fonction `reverse(L)` qui prend en argument une liste *L* et qui en inverse le sens, par permutation d'éléments 2 à 2. Compléter ensuite la **ligne 16** pour tester votre fonction.

Q2 – Compléter les **lignes 21 à 24** pour obtenir une fonction `reverse2(L)` similaire, mais qui ne modifie pas la liste en argument.

Aide : la ligne 22 vous interdit l'utilisation d'index… et donc la permutation des éléments 2 à 2. Vous pourrez en revanche utiliser les méthodes `.append()` et/ou `.insert()`, en partant d'une liste L2 vide.

Rappel : on a souvent envie d'écrire `L.insert(élément, index)` mais non, c'est l'ordre inverse pour les arguments : `L.insert(index_apres_insertion, élément_a_inserer)`

B / Méthodes *.append()* et *.insert()*

Ces deux méthodes affectent directement la liste en argument, sans return. On souhaite en coder des équivalents, cette fois-ci on n'utilisera pas des copies (ce serait trop facile).

Q3 – Compléter la **ligne 34** pour que la fonction `append2(L, a)` réalise la même opération que `L.append(a)` mais en renvoyant un résultat, et sans affecter la liste *L*. Compléter ensuite la **ligne 36** pour tester votre fonction, en adjoignant la valeur `10` à la liste `[2,4,6,8]`.

Q4 – Compléter la **ligne 38** pour que la fonction `insert2(L, i, a)` réalise la même opération que `L.insert(i, a)` mais en renvoyant un résultat, et sans affecter la liste *L*. Compléter ensuite les **lignes 43 à 45** pour tester votre fonction (les indices d'insertion et la liste sont indiqués dans le commentaire).

C / Méthode *.pop()* et fonction *del()*

La méthode `L.pop(i)` est extrêmement utile pour les piles et files, ce dont nous nous servirons notamment dans le tri de listes ou les parcours de graphes au semestre 2. Cette fonction fait **simultanément** deux opérations :

1. Elle supprime le *ième* élément de *L* de la liste (à portée globale)
2. Elle renvoie la valeur de cet élément (ce qui permet, éventuellement, de le stocker ailleurs).

Exemple :

```
>>> Liste = [1,5,7,9]
>>> valeur_gauche = Liste.pop(0)      # valeur_gauche ⟵ 1
>>> print(Liste)                      # affiche [5,7,9]
>>> valeur_droite = Liste.pop(-1)     # valeur_droite ⟵ 9
>>> print(Liste)                      # affiche [5,7]
```

Q5 – ⌨ Compléter **les lignes 52 à 54** pour que la fonction `pop(L, i)` réalise exactement la même chose que `L.pop(i)` (c'est-à-dire ici une fonction qui affecte la liste en argument). Il est bien entendu hors sujet d'utiliser `L.pop(i)` pour ce faire.

Aide : vous aurez besoin de la fonction **`del`**, qui permet d'effacer la valeur d'une variable. Entre autres, si vous disposez d'une liste `L`, alors `del(L[i])` permet de supprimer le *ième* élément de la liste.

Q6 – ⌨ Compléter (à l'aide d'un appel à votre fonction `pop()` – pour la tester)

- la **ligne 57** pour supprimer la dernière valeur de la liste `L_test` et la stocker dans une variable `DERN_VAL`
  - → la **ligne 58** va donc afficher la valeur `DERN_VAL` ainsi supprimée (9), et la nouvelle valeur de la liste `L_test` qui a été modifiée en place par votre fonction `pop()` `([0,1,2,…,7,8])`
- la **ligne 59** pour supprimer la 1ère valeur de la liste `L_test` (sans rien en faire, on veut juste vider la pile).
  - → la **ligne 60** va donc afficher la valeur de la liste `L_test` qui a été modifiée en place par votre fonction `pop()` `([1,2,…,7,8])` (on ne fait rien de la valeur 0 qui a été dépilée ici).

Parenthèse importante : nous aurons souvent besoin de fonctions qui renvoient plusieurs valeurs. Exemple :

```
def f(x) :
    return 2*x+1 , 3*x**2 -1 # les parenthèses (2*x+1 , 3*x**2 -1) sont
                             # sous-entendues
```

L'appel d'une telle fonction renvoie donc 2 éléments. Imaginons qu'on veuille faire le produit des deux termes en sortie de `f(x)` en `x = 3`. On peut, bien sûr, écrire :

```
L_sortie = f(3)  # L_sortie contient 2 éléments, un peu comme une liste
produit = L_sortie[0] * L_sortie[1]
```

… mais en pratique, une double affectation est beaucoup plus lisible, je la recommande vraiment !

```
a, b = f(3) # cette ligne crée et affecte 2 variables a←(2*3+1) ; b←(2*3**2-1)
produit = a*b
```

Q7 – ⌨ Compléter **les lignes 63 à 65** pour que la fonction `pop2(L, i)` réalise la même chose que `pop(L,i)`, mais renvoie deux valeurs : la valeur de l'élément supprimé (élément en $i^{ème}$ index dans la liste *L*), et aussi la liste modifiée, sans que cela n'affecte la liste *L* en argument.

Q8 – ⌨ Compléter alors la **ligne 68**, avec une double affectation, de sorte à être cohérent avec le `print` en **ligne 69**.

D / Dégagement des bords de listes

À cause des effets de bord, on a souvent besoin d'élaguer les bords de liste (les premiers ou les derniers éléments).

On souhaite écrire une fonction qui prenne en argument `(i, L, j)` : une liste *L* et 2 entiers *i* et *j*. On admet que lors de l'appel, $i + j < \text{len}(L)$, il n'est pas nécessaire de faire un test en ce sens.

Ce que l'on souhaite obtenir c'est la liste *L* amputée de ses *i* premiers éléments et de ses *j* derniers éléments. On admet que l'appel sera fait pour *i* et *j* strictement positifs.

Q9 – ⌨ Compléter **les lignes 76, 77** pour que la fonction `elag_liste(i,L,j)` réalise l'objectif, en affectant *L* à portée globale, de façon procédurale (sans **`return`**).

Contrainte : vous utiliserez uniquement des **`del`**.

Q10 – ⌨ Tester votre fonction `elag_liste(i,L,j)` en **lignes 80, 81**, en supprimant les 2 premières et les 3 dernières valeurs de `L_test`.

Q11 – ⌨ Compléter **la ligne 87** pour que la fonction `elag_liste2(i,L,j)` réalise la même chose en une seule ligne, sans copie et sans que cela n'affecte la liste *L*. Puis, tester votre fonction en **ligne 90** (voir commentaire).