Structures mutables : support de cours

Nous avons dit que les listes et les ensembles étaient des structures *mutables* (on dit aussi *modifiables*), alors que les tuples et les chaînes de caractères ne l'étaient pas. Dans cette section, nous allons étudier quelques propriétés des structures mutables qui peuvent sembler déroutantes au premier abord et qui sont souvent sources d'erreurs.

✔ Pour bien comprendre ce cours, je vous conseille d'aller tester les exemples proposés sur Python Tutor (http://www.pythontutor.com/visualize.html).

1 Concaténation

Nous avons vu que l'on pouvait concaténer des collections ordonnées :

Python
```
In [1]: t = (0, 1)

In [2]: t = t + (2, 3)
```

Python
```
In [3]: t += (4, 5)

In [4]: t
Out[4]: (0, 1, 2, 3, 4, 5)
```

On pourrait penser que l'on modifie le tuple `t` qui n'est pourtant pas mutable. En réalité, il ne s'agit pas d'une modification du tuple (qui est bien immuable) mais de la création de *nouveaux* tuples, liés au *même* nom de variable `t`. Vérifions-le en nous intéressant aux identifiants que l'on peut assimiler aux adresses mémoires des variables :

Python
```
In [5]: t = (0, 1)

In [6]: id(t)
Out[6]: 4383589832

In [7]: t = t + (2, 3)
```

Python
```
In [8]: id(t)
Out[8]: 4382954824

In [9]: t += (4, 5)

In [10]: id(t)
Out[10]: 4382499080
```

Il s'agit bien à chaque fois d'un nouvel identifiant. On peut faire la même chose sur les listes :

Python
```
In [11]: L = [0, 1]

In [12]: id(L)
Out[12]: 4382830984

In [13]: L = L + [2, 3]
```

Python
```
In [14]: id(L)
Out[14]: 4383533640

In [15]: L += [4, 5]

In [16]: id(L)
Out[16]: 4383533640  # ?!?!?!?
```

De manière assez surprenante, l'identifiant de la liste n'a pas changé lorsque l'on utilise l'opérateur `L += ...`, que l'on pensait pourtant être un raccourci pour `L = L + ...`. Étudions le comportement de l'affectation à un indice et des méthodes `append` et `extend`.

Python
```
In [17]: L = [0, 1, 2, 3]

In [18]: id(L)
Out[18]: 4383596168

In [19]: L[0] = 4

In [20]: id(L)
Out[20]: 4383596168

In [21]: L.append(5)

In [22]: id(L)
Out[22]: 4383596168
```

Python
```
In [23]: L.extend([6, 7, 8])

In [24]: id(L)
Out[24]: 4383596168

In [25]: L = L + [9]

In [26]: id(L)
Out[26]: 4382871880

In [27]: L += [10]

In [28]: id(L)
Out[28]: 4382871880
```

On voit que toutes les opérations *modifient* la même liste, sauf la concaténation `L + ...` qui en créé une nouvelle.

⚠ Pour les listes, l'opérateur `+=` *ajoute* les éléments à la liste (tout comme `extend`), contrairement à `L = L + ...` qui créé une nouvelle liste.

Comparaison On va comparer différentes méthodes pour créer une liste de 100000 zéros. On utilise la commande spéciale `timeit` qui calcule le temps exécution (en faisant une moyenne sur un grand nombre d'essais.)

PYTHON
```
def concatenation_naive():
    L = []
    for i in range(100000):
       L = L + [0]

In [29]: timeit concatenation_naive()
29.7 s +- 299 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

C'est très très long ! Presque 30 s pour exécuter ce programme. Le problème, c'est qu'à chaque tour de boucle i, pour créer la nouvelle liste, il faut *recopier* une liste qui contient i éléments et en ajouter un. Le nombre d'opérations est donc de l'ordre de $i+1$ à chaque tour de boucle, donc de l'ordre de $\sum_{i=0}^{n-1}(i+1) = \frac{n(n+1)}{2} \approx n^2$. Pour $n = 10^5$ il faut donc environ dix milliards d'opérations !

PYTHON
```
def concatenation_plus_egal():
    L = []
    for _ in range(100000):
        L += [0]

In [30]: timeit concatenation_plus_egal()
17.1 ms +- 2.43 ms per loop (mean +- std. dev. of 7 runs, 100 loops)
```

PYTHON
```
def concatenation_avec_extend():
    L = []
    for _ in range(100000):
        L.extend([0])

In [31]: timeit concatenation_avec_extend()
21.7 ms +- 514 us per loop (mean +- std. dev. of 7 runs, 10 loops)
```

PYTHON
```
def ajout_avec_append():
    L = []
    for _ in range(100000):
        L.append(0)

In [32]: timeit ajout_avec_append()
15.4 ms +- 280 us per loop (mean +- std. dev. of 7 runs, 100 loops)
```

Avec les trois fonctions précédentes, il n'y a qu'une seule opération par boucle et donc de l'ordre de n opérations en tout, soit pour $n = 10^5$, cent mille opérations, ce qui est bien plus raisonnable. Il est préférable ici d'utiliser la méthode `append` puisque l'on ajoute un seul élément à la fin de la liste, mais les méthodes `extend` ou `+=`, qui sont à peu près équivalentes, sont utiles lorsque l'on veut ajouter une liste plutôt qu'un seul élément.

Bien sûr, dans le cas présent, on pouvait créer directement la liste :

PYTHON
```
def creation_avec_multiplication():
    L = [0] * 100000

In [33]: timeit creation_avec_multiplication()
377 us +- 17.2 us per loop (mean +- std. dev. of 7 runs, 1000 loops)
```

2 Modification par effet de bord

Considérons le programme suivant, qu'il faut essayer sur PYTHON TUTOR pour bien comprendre son fonctionnement interne :

PYTHON
```
L1 = [0, 1, 2]
L2 = L1
L2[0] = 1
print(L1)
print(L2)
```

Lors de l'affectation `L2 = L1`, il n'y a pas de copie de la liste `L1`. Le nom de variable `L2` est lié au contenu de la variable `L1` qui est une liste. Donc les variables `L1` et `L2` *pointent* vers la *même* liste. Modifier l'une modifie donc l'autre.

Considérons maintenant le programme suivant, qu'il faut aussi essayer sur Python Tutor.

Python
```
def efface(liste):
    """Réinitialise une liste à '0'."""
    for i in range(len(liste)):
        liste[i] = 0
    return liste

notes = list(range(8, 15))
notes_effacees = efface(notes)
print(notes, notes_effacees)
```

On a dit que le passage des paramètres se faisait par valeur. C'est donc le contenu de la variable `notes` qui est passé à la fonction `efface`, ce contenu étant une liste. Dans la fonction `efface` on *modifie* cette *valeur* (une liste est une valeur modifiable).

✔ La variable `notes` n'a pas été modifiée par la fonction, par contre sa valeur (une liste) a été modifiée !

⚠ Lorsque l'on passe des valeurs mutables à une fonction (en particulier des listes), leur valeur peut être modifiée à l'intérieur de la fonction.

Exercice 1

Prévoir et expliquer :

Python
```
def efface(liste):
    """Réinitialise une liste à '0'."""
    liste = [0] * len(liste)
    return liste

notes = list(range(8, 15))
notes_effacees = efface(notes)
print(notes, notes_effacees)
```

3 Copie d'une liste

Nous avons déjà vu que l'on pouvait copier une liste avec la fonction `list` (on peut aussi lui ajouter une liste vide, mais c'est manquer de classe) :

Python
```
In [34]: L1 = [0, 1, 2]

In [35]: L2 = L1

In [36]: L3 = list(L1)

In [37]: L4 = [] + L
```

Python
```
In [38]: L1[0] = 42

In [39]: print(L1, L2, L3, L4,
 ↪  sep='\n')
[42, 1, 2]
[42, 1, 2]
[0, 1, 2]
[0, 1, 2]
```

✔ Si une fonction modifie une liste en argument, *toujours* se poser la question s'il ne vaut mieux pas travailler sur une copie de la liste et bien se souvenir qu'une affectation ne copie pas la liste.

4 Listes de listes

On peut mettre ce que l'on souhaite dans une liste. En particulier... des listes !

Python
```
In [40]: L = [[0, 1], [2], []]

In [41]: L[0]  # Composante d'indice 0 de L : une liste !
Out[41]: [0, 1]

In [42]: L[0][1]  # Composante d'indice 1 de la composante d'indice 0
Out[42]: 1
```

♥ On représente une matrice (un tableau à double entrée), une grille, une image, etc., par une liste de listes (de mêmes tailles). On accède à l'élément à la ligne `i` et à la colonne `j` en écrivant `L[i][j]`. La i^e ligne est `L[i]`.

Exercice 2

Écrire une fonction qui renvoie une liste formée de la colonne `j` d'une matrice.

⚠

Comment créer une matrice de taille $n \times n$ initialisée à 0 ?

Première idée :

Python

```
In [43]: n = 5

In [44]: matrice = [[0] * n] * n

In [45]: matrice
Out[45]:
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]

In [46]: matrice[0][0] = 1

In [47]: matrice
Out[47]:
[[1, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [1, 0, 0, 0, 0]]  # ?!?!?!?
```

⚠

L'évaluation de l'expression `[[0] * n] * n` commence par créer une liste, appelons-la `ligne`, à `n` éléments initialisés à 0. Puis on créé la liste matrice dont les éléments sont initialisés à `ligne`... La *même* liste !

Exercice 3

Essayer sur Python Tutor, comprendre, réessayer, comprendre, réessayer, et ce, jusqu'à saturation.

Voici une version correcte, qui construit la matrice élément par élément.

Python

```
def initialise_matrice(n, valeur):
    """Initialise une matrice ‘n x n‘ avec la valeur par défaut."""
    matrice = []
    for _ in range(n):
        ligne = []
        for _ in range(n):
            ligne.append(valeur)
        matrice.append(ligne)
    return matrice

matrice = initialise_matrice(5, 0)
```

On peut aussi construire chaque ligne directement. Ce qui est important est de ne pas lier toutes les lignes.

Python

```
def initialise_matrice(n, valeur):
    """Initialise une matrice ‘n x n‘ avec la valeur par défaut."""
    matrice = []
    for _ in range(n):
        matrice.append([valeur] * n)
    return matrice

matrice = initialise_matrice(5, 0)
```

Exercice 4

Écrire une fonction qui copie une matrice. Cette fonction renvoie donc une nouvelle matrice, identique à celle passée en arguments, sans modifier cette dernière.