

Une limite de la boucle for

Complément - niveau basique

Pour ceux qui veulent suivre le cours au niveau basique, reprenez seulement que dans une boucle `for` sur un objet mutable, **il ne faut pas modifier le sujet** de la boucle.

Ainsi par exemple il ne **faut pas faire** quelque chose comme ceci:

```
[-] # on veut enlever de l'ensemble toutes les chaines
    # qui ne commencent pas par 'a'
    ensemble = {'marc', 'albert'}

    # ceci semble une bonne idée mais ne fonctionne pas
    for valeur in ensemble:
        if valeur[0] != 'a':
            ensemble.discard(valeur)
```

Comment faire alors ?

Lorsqu'on est dans ce cas de figure, il suffit de faire la boucle sur une *shallow copy* de l'objet.

```
[-] from copy import copy
    # on veut enlever de l'ensemble toutes les chaines
    # qui ne commencent pas par 'a'
    ensemble = {'marc', 'albert'}

    # si on fait d'abord une copie tout va bien
    for valeur in copy(ensemble):
        if valeur[0] != 'a':
            ensemble.discard(valeur)

    print ensemble
```

Avertissement

Dans l'exemple ci-dessus, on voit que l'interpréteur se rend compte que l'on est en train de modifier l'objet de la boucle, et nous le signifie.

Ne vous fiez pas forcément à cet exemple, il existe des cas -- nous en verrons plus loin dans ce document -- où l'interpréteur peut accepter votre code alors qu'il n'obéit pas à cette règle, et du coup essentiellement se mettre à faire n'importe quoi.

Précisons bien la limite

Pour être tout à fait clair, lorsqu'on dit qu'il ne faut pas modifier l'objet de la boucle `for`, il ne s'agit que du premier niveau.

On ne doit pas modifier la **composition de l'objet en temps qu'itérable**, mais on peut sans souci modifier chacun des objets qui constituent l'itération.

Ainsi par exemple cette construction par contre est tout à fait valide

```
liste = [[i] for i in range(3)]
print 'avant', liste
for sous_liste in liste:
    sous_liste.append(None)
print 'après', liste
```

Dans cet exemple, les modifications ont lieu sur les éléments de `liste`, et non sur l'objet `liste` lui-même, c'est donc tout à fait légal.

Complément - niveau intermédiaire

Pour bien comprendre la nature de cette limitation, il faut bien voir que cela soulève deux types de problèmes distincts.

Difficulté d'ordre sémantique

D'un point de vue sémantique, si l'on voulait autoriser ce genre de choses, il faudrait définir très précisément le comportement attendu.

Considérons par exemple la situation d'une liste qui a 10 éléments, sur laquelle on ferait une boucle et que, par exemple au 5ème élément, on enlève le 8ème élément. Quel serait le comportement attendu dans ce cas ? Faut-il ou non que la boucle envisage alors le 8-ème élément ?

La situation serait encore pire pour les dictionnaires et ensembles pour lesquels l'ordre de parcours n'est pas spécifié; ainsi on pourrait écrire du code totalement indéterministe si le parcours d'un ensemble essayait:


- d'enlever l'élément 'b' lorsqu'on parcourt l'élément 'a'
- d'enlever l'élément 'a' lorsqu'on parcourt l'élément 'b'

On le voit, il n'est déjà pas très simple d'expliciter sans ambiguïté le comportement attendu d'une boucle `for` qui serait autorisée à modifier son propre sujet. En fait, l'apparente limitation qui consiste à interdire de modifier l'objet sur lequel on itère permet d'avoir un comportement plus facile à comprendre.

Difficulté d'implémentation

Voyons maintenant un exemple de code qui ne respecte pas la règle, et qui modifie le sujet de

la boucle en lui ajoutant des valeurs

```
 # cette boucle ne termine pas
```

```
liste = [1, 2, 3] for c in liste: if c == 3: liste.append(c)
```

Nous avons volontairement mis ce code **dans une cellule de texte** et non de code: vous **ne pouvez pas l'exécuter** dans le notebook. Si vous essayez de l'exécuter sur votre ordinateur vous constaterez qu'elle ne termine pas, en fait à chaque itération on ajoute un nouvel élément dans la liste, et du coup la boucle a un élément de plus à balayer; ce programme ne termine jamais.