Cours_mutabilite_et_effet_de_bord

April 1, 2021

1 Illustration des effets de bords

[1]:
```
liste1 = [1,2,3]
liste2 = liste1

print(liste1)
print(liste2)
```

```
[1, 2, 3]
[1, 2, 3]
```

[2]:
```
liste1[0] = 10
print(liste1)
```

```
[10, 2, 3]
```

[3]:
```
print(liste2)
```

```
[10, 2, 3]
```

`liste2` a été modifiée alors qu'on ne l'a pas demandé !!! On dit que `liste2` a été modifiée par **effet de bord**.

1.0.1 Que s'est-il passé ?

Pour le comprendre, il faut savoir 3 choses :

1. Contrairement à l'analogie "simpliste" donnée en début d'année, les **noms de variables ne sont pas des "étiquettes" collées sur des "boîtes" cases mémoire**. En fait, les **noms de variables sont des références qui POINTENT vers l'endroit de la mémoire où est stocké l'objet.**
2. La mémoire (RAM) de l'ordinateur est une ressource limitée : Quand plusieurs variables "contiennent" le même objet, python ne duplique pas les objets en mémoire. Au lieu de cela ,python crée juste plusieurs variables qui pointent vers le même objet stocké en mémoire. Ceci est rendu visible dans pythonTutor à l'aide de flèches entre les noms de variables et les objets qu'elles référencent
3. Les listes ne contiennent pas les objets mais uniquement les références vers les objets qu'elles "contiennent". Ceci encore pour économiser la mémoire de l'ordinateur.

Plus d'informations (hors programme) :

- mécanisme de référence_partagée
- notion de pointeurs

[4]:
```
#!pip install nbtutor
#!jupyter nbextension install --overwrite --py nbtutor
#!jupyter nbextension enable nbtutor --py
%load_ext nbtutor
```

[5]:
```
%%nbtutor -r -f

x = 3
y = 4
z = 4
```

On constate bien que l'objet `4` stocké en mémoire est référencé à la fois par `y` et `z`. L'objet `4` n'existe donc qu'une seule fois en mémoire.

Ce mécanisme est particulièrement intéressant sur les listes qui en pratique occupe beaucoup plus de place en mémoire. (il n'est pas rare d'avoir des listes de plusieurs milliers d'éléments)

Reprenons l'exemple du début à l'aide de pythonTutor pour bien comprendre pourquoi liste2 a été modifié alors qu'on ne l'avait pas explicitement demandé

[6]:
```
%%nbtutor -r -f

liste1 = [1,2,3]
liste2 = liste1
liste1[0] = 10
```

2 Le danger des types mutables

Tous les objets mutables sont susceptibles d'être affectés par les effets de bord, Les objets immuables non. (Pour l'instant les listes sont les seuls objets mutables que vous connaissez)

- Pas de danger particulier quand on utilise des types immuables : Le programme peut "planter" mais au moins on sait qu'il ne marche pas.
- Il faut être très vigilant dès qu'on manipule des types mutables, car si on ne fait pas attention, des effets de bords peuvent se produire sans qu'on ne s'en rende compte. C'est un peu comme si Python faisait des choses "dans notre dos". Le programme ne "plante" pas mais aura un comportement très étrange... cette situation est beaucoup plus délicate à gérér qu'un programme qui "plante franchement"

La méthode `copy` permettent de faire de "vraies" copies de liste et de se protéger contre l'effet de bord :

[7]:
```
%%nbtutor -r -f

liste1 = [1,2,3]
liste2 = liste1.copy()
liste1[0] = 10
```

3 Fonctions utilisant les effets de bord

Certaines méthodes utilisent les effets de bord comme la méthode `sort` (La méthode `sort` permet de trier les éléments d'une liste). Normalement tout effet de bord d'une fonction doit être écrit explicitement dans la documentation

[8]:
```
liste = [7,3,8,2,4]

liste_triee = liste.sort()

print(liste_triee)

# Erreur courante : sort modifie la liste par effet de bord mais ne renvoie pas␣
↪une nouvelle liste !!
```

```
None
```

[9]:
```
liste = [7,3,8,2,4]

liste.sort()

print(liste)

# On voit bien que liste a été modifiée !!
```

```
[2, 3, 4, 7, 8]
```

Remarquez comme dans la documentation de `sort`, on insiste sur le fait que le trie se fait **IN PLACE** (en place) : une façon de dire que la liste est modifée par effet de bord...

[10]:
```
help(list.sort)
```

```
Help on method_descriptor:

sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

La fonction `sorted` **renvoie** une nouvelle liste.

[11]:
```
liste = [7,3,8,2,4]

liste_triee = sorted(liste)

print(liste)
print(liste_triee)
```

```
[7, 3, 8, 2, 4]
[2, 3, 4, 7, 8]
```

En résumé :

- Avec la fonction `sorted` :
  - Il n'y a donc pas d'effet de bord donc pas de danger
  - mais on a 2 listes en mémoires plutôt qu'une seule (plus de ressource mémoire utilisée)
- Avec la méthode `sort` :
  - Il y a un effet de bord donc danger (si on ne fait pas attention)
  - On a 1 seule liste en mémoire (optimisation de la ressource mémoire)
  - La liste initiale (désordonnée) est définitivement perdue !

(Conseil : En tant que débutant en programmation, éloignez-vous des effets de bord. On préférera donc `sorted` plutôt que `sort`)

3.0.1 Remarque : Attention à l'opérateur * sur les listes : il cache un effet de bord

[12]:
```
#Création d'une matrice 5*5 de 0
liste = [[0]*5]*5

print(liste)
```

```
[[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]]
```

[13]:
```
%%nbtutor -r -f

#Création d'une matrice 5*5 de 0
liste = [[0]*5]*5

# Malheureusement toutes les listes "internes" sont des références partagées

liste[0][0] = 1

print(liste)
```

```
[[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]]
```