

Informatique Pour Tous

Cours 8 – Rappels sur la portée des variables, type « classiques » et « conteneurs » (cours à compléter en classe)

A / Portée des variables de type *bool*, *str*, *int*, *float*

On a vu ensemble la différence entre la portée locale (uniquement dans une fonction) et globale d'une variable. Ci-dessous, quelques exemples de codes « mal écrits » (dans le sens où ils sont ambigus) :

Code 1.1

```
def f(x) :  
    K = 2  
    print(K)  
    return K*x  
  
K = 3  
f(2)  
print(K)
```

Code 1.2

```
K = 4  
def f(x) :  
    return K*x  
  
K = 3  
print(f(2))  
  
K = 1  
print(f(2))
```

Code 1.3

```
def f(x) :  
    x = x - 1  
    return 2*x  
  
x = 4  
print(f(x))  
print(x)
```

Code 1.4

```
def f(x) :  
    global K  
    K = 2  
    return K*x  
  
K = 3  
print(f(2))  
print(K)
```

Code 1.5

```
def f(x) :  
    global sortie  
    sortie = 3*x  
  
f(2)  
print(sortie)
```

Code 1.6

```
def f(x) :  
    global sortie  
    sortie = 3*x  
  
sortie = f(2)  
print(sortie)
```

Remarque : la commande `global` est hors programme en CPGE, et interdite aux écrits.

2 / Rappel sur les méthodes spécifiques au type *list*

Pour une liste *L*, les méthodes classiques : *L.append(valeur)*, *L.insert(index, valeur)*, *L.pop(index)* (ainsi que la commande `del L[index]` # ou `del L[slicing]`) ne renvoient rien ! On dit qu'elles sont **procédurales**, elle modifient la liste **en place** sans effectuer de **return**.

Code 2.1

```
Lst = [1,3,5]  
Lst = Lst.append(7)  
print(Lst)
```

Code 2.2

```
Lst = [1,3,5]  
Lst = Lst + [7]  
print(Lst)
```

Code 2.3

```
Lst = [1,3,5]  
Lst.append(7)  
print(Lst)
```

3 / Cas particulier pour les conteneurs (listes, *np.array*, dictionnaires) qui sont en argument d'une fonction

Pour les conteneurs qui sont placés en **argument** d'une fonction, la portée de la plupart des opérations est automatiquement **globale** ! Mais ce n'est pas le cas de toutes les opérations...

Exemple, pour les listes :

- **Sont forcément à portée globale** : les méthodes, la commande `del`, la modification de valeur d'un élément (`L[index] = ...`), donc *a fortiori* la permutation de deux éléments (`L[i], L[j] = L[j], L[i]`).
- **Ne sont pas à portée globale** : les autres opérations, comme la concaténation `L = L + L2`

Code 3.1

```
def f(L) :  
    L[0] = 6  
  
Lst = [1,2,3]  
Lst = f(Lst)  
print(Lst)
```

Code 3.2

```
def f(L) :  
    L[0], L[-1] = L[-1], L[0]  
  
Lst = list(range(5))  
f(Lst)  
print(Lst)
```

Code 3.1

```
def f(L) :  
    L = L + [4]  
  
Lst = [1,2,3]  
f(Lst)  
print(Lst)
```

Code 3.4

```
def f(L) :  
    L[0] = 6  
    return L  
  
Lst = [1,2,3]  
Lst2 = f(Lst)  
print(Lst2)  
print(Lst)
```

Code 3.5

```
def f(L) :  
    L = L + [4] # ce serait + clair : Ls = L + [4]  
    return L    # ... auquel cas ici : return Ls  
  
Lst = [1,2,3]  
Lst2 = f(Lst)  
print(Lst2)  
print(Lst)
```

Remarque importante : pour éviter que vous vous posiez trop de questions, on vous recommandera de toujours faire une **copie profonde** des conteneurs (listes, *np.array*, dictionnaires) sur lesquels vous travaillez. Voir §5.

4 / Autre cas particulier des conteneurs (listes, *np.array*, dictionnaires) : la copie VS l'alias !

Jusqu'ici, avec les objets de type *bool*, *str*, *int*, *float*, nous avons eu l'habitude que la copie se réalise ainsi :

Code 4.1

```
a = 4 # prenons un int par exemple  
b = a # on fait une copie de a  
  
a = 6 # Dès lors, si on modifie a  
print(b)
```

Code 4.2

```
a = "hello" # autre ex avec str  
b = a # on fait une copie de a  
  
a = a.replace('l', 'y') -  
print(b)
```

Si *L* est un conteneur (par exemple une liste), alors la commande *L2 = L* ne crée pas une copie mais un **alias** ! C'est-à-dire que la variable *L2* devient comme un second nom pour pointer vers la même case mémoire que *L*.

➔ Si l'on veut créer une copie (profonde) de *L* on utilisera la commande : *L2 = L.copy()* (ou *L2 = L[:]*)

Code 4.3

```
LA = [1,3,5]  
LB = LA # on fait un alias  
  
LA.append(7) # si on LA  
print(LB)
```

Code 4.4

```
LA = [1,3,5]  
LB = LA.copy() # copie profonde  
  
LA.append(7) # si on modifie LA  
print(LB)
```

Remarque : pour finir de vous convaincre qu'il vaut mieux utiliser une copie qu'un alias, ajoutons un petit complément.

En pratique, la liste aliassée n'est pas toujours modifiée comme la liste originale. Comme pour le §3, les méthodes et fonctions que touchent la liste originale de façon globale affectent aussi l'alias, mais ce n'est par exemple pas le cas pour une concaténation...

Code 4.5

```
LA = [1,3,5]  
LB = LA # on fait un alias  
  
LA = LA + [7] # si on modifie LA  
print(LA)  
print(LB)
```

➔ **Bref : faites des copies !**

Remarque : Supposons que la méthode `.copy()` n'existe pas et qu'on doive la coder nous-même (à l'aide d'une copie terme par terme, balayé par une boucle *for*). Compléter les deux fonctions suivantes :

Code 4.6

```
def copie(L) : # prend en argument une liste L et renvoie sa copie terme à terme
```

Code 4.7

```
def copie(V) : # prend en argument un vecteur (np.array) V et fait la copie profonde
```

5 – Comment procéder lorsqu'on travaille avec des copies

Dans le TD qui va suivre, vous allez être invités, pour plusieurs opérations simples sensés modifier une liste, à faire ces opérations de deux manières :

- de manière procédurale : c'est-à-dire en écrivant votre fonction `def f(L) :` et en vous arrangeant pour modifier la liste `L` (en argument) **en place**, sans que votre fonction n'effectue de **return**.

Remarque : dans ce cas, l'appel de votre fonction sur une liste donnée `Ltst` se fera en tapant `f(Ltst)` (et surtout pas `Ltst = f(Ltst)`), et vous devrez faire attention aux portées de vos opérations.

- En utilisant une copie : c'est-à-dire en écrivant votre fonction sous la forme :

Code 5.1

```
def f(L) :  
    Lbis = L.copy() # On commence par une copie profonde  
    ### Corps de la fonction ###  
    # On ne modifie QUE Lbis (locale), JAMAIS L  
    return Lbis # à la fin on renvoie la liste modifiée  
                # ... qui est supprimée (locale) après l'appel
```

➔ Voyons un exemple ensemble : on veut coder une fonction qui remplace la 1^{ère} valeur de la liste par le str 'zéro'

Code 5.2 : fonction « en place » (procédurale)

```
def change_1_terme(L) :
```

```
# Test :  
Ltst = [1,3,5]
```

Code 5.3 : fonction avec copie

```
def change_1_terme(L) :
```

```
# Test :  
Ltst = [1,3,5]
```

Remarque : Après le TD, et sauf indication contraire du sujet, il sera recommandé de préférer l'utilisation de la copie pour le moment (tant que vous ne maîtrisez pas bien ce que vous faites).