

W3-S4-C3-les-differentes-copies

December 15, 2014

1 Les différentes copies

1.1 Complément - niveau basique

1.1.1 Deux types de copie

Pour résumer les deux grands types de copie que l'on a vues dans la vidéo: * La *shallow copy* - de l'anglais *shallow* qui signifie superficiel * La *deep copy* - de *deep* qui signifie profond

1.1.2 Le module copy

Pour réaliser une copie, la méthode la plus simple, en ceci qu'elle fonctionne avec tous les types de manière identique, consiste à utiliser [le module standard copy](#), et notamment * `copy.copy` pour une copie superficielle * `copy.deepcopy` pour une copie en profondeur

```
In []: import copy
      #help(copy.copy)
      #help(copy.deepcopy)
```

1.1.3 Un exemple

```
In []: # On se donne un objet de départ

      source = [
          [1, 2, 3], # le premier élément est une liste
          {1, 2, 3}, # un ensemble
          (1, 2, 3), # un tuple
          '123',     # un string
          123,       # un entier
      ]

      # et on en fait deux copies
      shallow_copy = copy.copy(source)
      deep_copy = copy.deepcopy(source)
```

1.2 Complément - niveau intermédiaire

1.2.1 Objets égaux au sens logique

Bien sûr ces trois objets se ressemblent si on fait une comparaison *logique*

```
In []: print 'source == shallow_copy:', source == shallow_copy
      print 'source == deep_copy:', source == deep_copy
```

1.2.2 Inspectons les objets de premier niveau

Mais par contre si on compare l'identité des objets de premier niveau, on voit que `source` et `shallow_copy` partagent leurs objets:

```
In []: for i in range(len(source)):
        print "source[{}] is shallow_copy[{}]".format(i, i),\
              source[i] is shallow_copy[i]
```

Alors que naturellement ce n'est pas le cas avec la copie en profondeur

```
In []: for i in range(len(source)):
        print "source[{}] is deep_copy[{}]".format(i, i),\
              source[i] is deep_copy[i]
```

On remarque tout de suite que les trois derniers objets n'ont pas été dupliqués comme on aurait pu s'y attendre; cela est dû, ici encore, à l'optimisation qui est mise en place dans python pour implémenter les types immuables comme des singletons lorsque c'est possible. Cela a été vu en détail dans le complément consacré à l'opérateur `is`.

1.2.3 On modifie la source

Il doit être clair à présent que, précisément parce que `deep_copy` est une copie en profondeur, on peut modifier `source` sans impacter du tout `deep_copy`.

S'agissant de `shallow_copy`, par contre, seuls les éléments de premier niveau ont été copiés. Aussi si on fait une modification par exemple à l'intérieur de la liste qui est le premier fils de `source`, cela sera répercuté dans `shallow_copy`

```
In []: print "avant, source      ", source
        print "avant, shallow_copy", shallow_copy
        source[0].append(4)
        print "après, source      ", source
        print "après, shallow_copy", shallow_copy
```

Si par contre on remplace complètement un élément de premier niveau dans la source, cela ne sera pas répercuté dans la copie superficielle

```
In []: print "avant, source      ", source
        print "avant, shallow_copy", shallow_copy
        source[0] = 'remplacement'
        print "après, source      ", source
        print "après, shallow_copy", shallow_copy
```

1.2.4 Copie et circularité

Le module `copy` est semble-t-il capable de copier - même en profondeur - des objets contenant des références circulaires.

```
In []: l = [None]
        l[0] = l
        l
```

```
In []: copy.copy(l)
```

```
In []: copy.deepcopy(l)
```

1.2.5 Pour en savoir plus

On peut se reporter à [la section sur le module `copy`](#) dans la documentation python.

Signalons également pythontutor.com qui est un site très utile pour comprendre comment python implémente les objets, les références et les partages. Toutefois `pythontutor.com` ne supporte pas le module `copy` ce qui est un peu dommage. On peut toutefois expérimenter avec des listes en utilisant le slicing `[:]` pour des copies superficielles.