Muable ou immuable

« Recopier et compléter sur la copie le code de la fonction reduire_triplet_au_sommet prenant une pile p en paramètre et qui la modifie en place. Cette fonction ne renvoie donc rien. »

Python : Objet muable ou immuable.
Fonction : Modifier en place ou renvoyer

Muable ou immuable

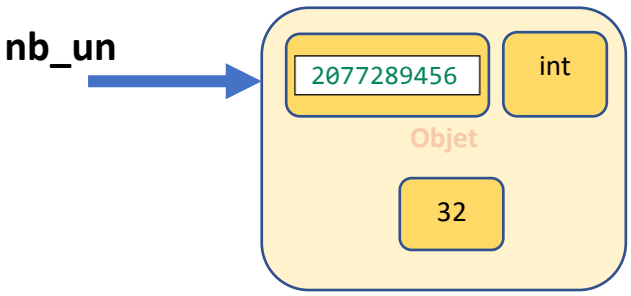
Immutable

« An immutable object is an object whose value cannot change. »

Plusieurs types d'objets python (booléens, entiers, flottants, chaînes et tuples) sont immuables. Cela signifie qu'après avoir créé l'objet et lui avoir attribué une valeur, vous ne pouvez pas modifier cette valeur.

Qu'est-ce que cela signifie dans les coulisses, dans la mémoire de l'ordinateur ? Un objet créé et doté d'une valeur se voit attribuer un espace en mémoire. Le nom de la variable liée à l'objet est un identifiant de cet emplacement en mémoire.

```
>>>  nb_un = 32
>>> type(nb_un)
<class 'int'>
>>> id(nb_un)
2077289456
```



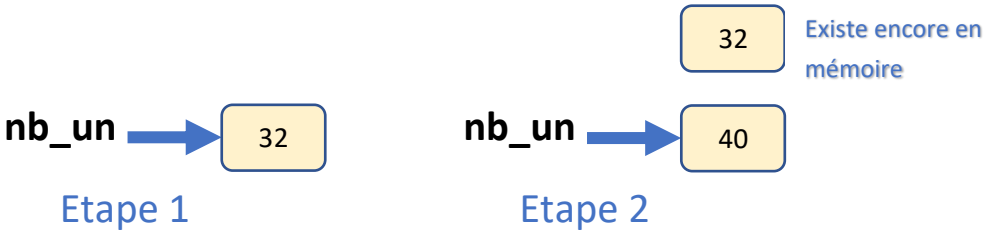
Les objets de type immuable ne peuvent pas être modifiés une fois créés, pourtant ce code ne provoque pas d'erreur :

```
>>> nb_un = 32
>>> nb_un = 40
```

En réalité, même si le nom est resté le même, ce n'est plus le même objet !

```
>>> nb_un = 32
>>> id(nb_un)
2088496112
>>> id(32)
2088496112
```

```
>>> nb_un = 40
>>> id(32)
2088496112
>>> id(nb_un)
2088496240
>>> id(40)
2088496240
```



Mutable

« In Python, 'mutable' is the ability of objects to change their values »

Contrairement aux types intégrés tels que int, float, bool, string, unicode, tuple qui sont des types d'objets immuables, les types list , dict , set sont des types d'objets muables. Les classes personnalisées sont aussi généralement modifiables.

```
>>> color = ["red", "blue", "green"]
>>> id(color)
87074096
>>> color[0] = "pink"
>>> id(color)
87074096
```

Etape 1 Etape 2

Ecriture des fonctions

Paramètres immuables

```
def addition(number_1, number_2):
    number_1 += 2
    number_2 += 3
    return number_1 + number_2
```

Dans le cas où vous transmettez des arguments tels que des nombres entiers, des chaînes ou des tuples à une fonction, le passage est comme un appel par valeur car vous ne pouvez pas modifier la valeur des objets immuables passés à la fonction.

```
>>> number_1 = 5
>>> number_2 = 10
>>> print(addition(number_1, number_2))
20
>>> print(number_1, number_2)
5 10
```

Paramètres muables

```
def duplicate_last(a_list):
    last_element = a_list[-1]
    a_list.append(last_element)
    return a_list
```

Le passage d'objets mutables peut être considéré comme un appel par référence, car lorsque leurs valeurs sont modifiées à l'intérieur de la fonction, cela se reflète également à l'extérieur de la fonction.

```
>>> initial_list = [1, 2, 3]
>>> new_list = duplicate_last(initial_list)
>>> print(new_list)
[1, 2, 3, 3]
>>> print(initial_list)
[1, 2, 3, 3]
```

Comme on peut le voir, la valeur globale de initial_list a été mise à jour alors que sa valeur n'a changé que dans la fonction !

Dans des programmes plus complexes, nous pourrions utiliser fréquemment de nombreuses fonctions différentes. Si tous modifient les listes sur lesquels ils travaillent, il peut devenir assez difficile de garder trace de ce qui change.

De manière générale, nous ne voulons pas que nos fonctions changent les variables globales, même si elles contiennent des types de données modifiables comme des listes ou des dictionnaires.

Heureusement, il existe un moyen simple de contourner ce problème : nous pouvons faire une copie de l'objet.

```
def duplicate_last(a_list):
    c_list = a_list.copy()
    last_element = c_list[-1]
    c_list.append(last_element)
    return c_list
```

Cette fois c'est un autre objet qui a été modifié. Il n'y a plus d'effet de bord.

```
>>> initial_list = [1, 2, 3]
>>> new_list = duplicate_last(initial_list)
>>> print(new_list)
[1, 2, 3, 3]
>>> print(initial_list)
[1, 2, 3]
```

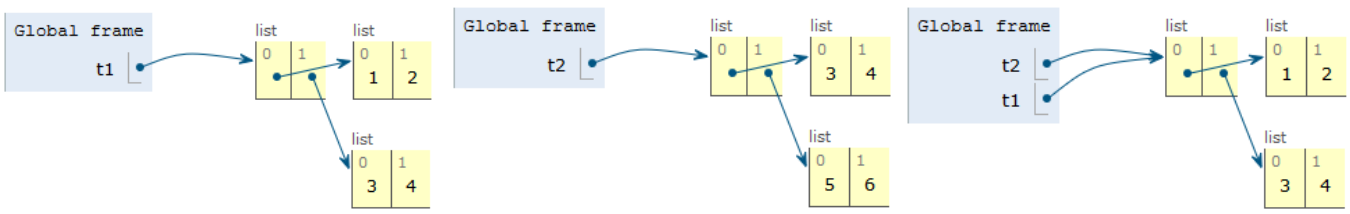
Remarque

```
def duplicate_last(a_list):
    '''Modification en place -> effet de bord'''
    last_element = a_list[-1]
    a_list.append(last_element)
    # return a_list ligne à éviter
```

Copie superficielle et copie profonde

Les instructions d'affectation en Python ne copient pas les objets, elles créent des liens entre la cible et l'objet.

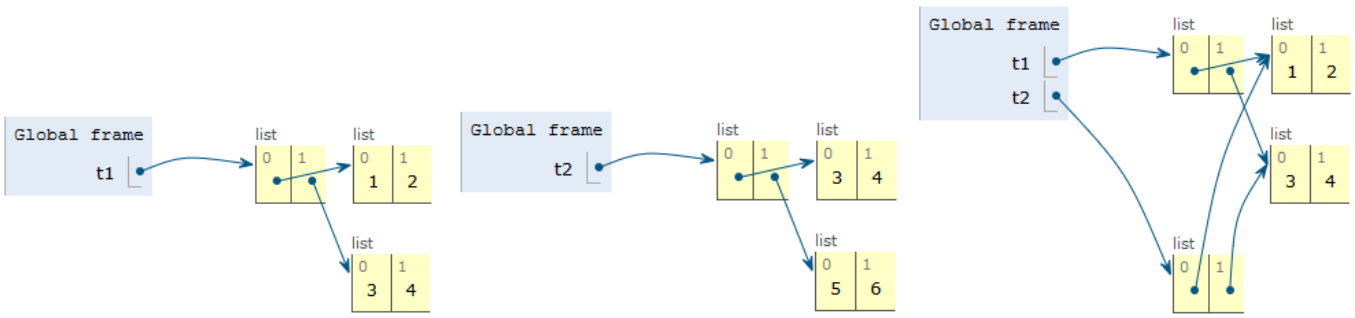
```
>>> t1 = [[1,2],[3,4]]
>>> t2 = [[3,4],[5,6]]
>>> print(id(t1), id(t1[0]), id(t1[1]))
87073536 87073576 85137088
>>> print(id(t2), id(t2[0]), id(t2[1]))
87073056 87073496 87075376
>>> t2 = t1
>>> print(id(t1), id(t1[0]), id(t1[1]))
87073536 87073576 85137088
>>>  print(id(t2), id(t2[0]), id(t2[1]))
87073536 87073576 85137088
```



Shallow copy

Une copie superficielle construit un nouvel objet composé puis insère dans l'objet composé des références aux objets trouvés dans l'original.

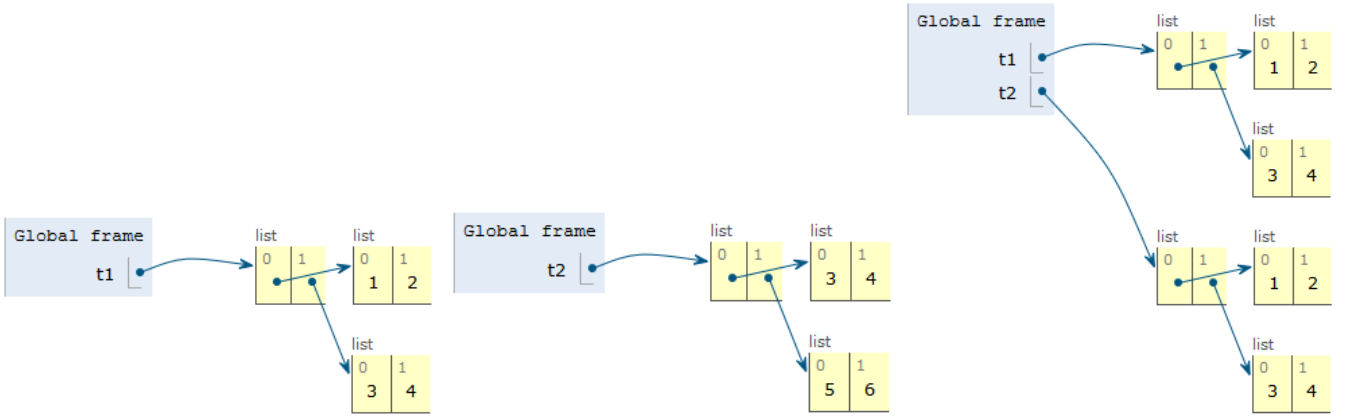
```
>>> t1 = [[1,2],[3,4]]
>>> print(id(t1), id(t1[0]), id(t1[1]))
87072936 87072816 87075576
>>> t2 = [[3,4],[5,6]]
>>> print(id(t2), id(t2[0]), id(t2[1]))
87193480 87193280 84985856
>>> t2 = t1.copy()
>>> print(id(t2), id(t2[0]), id(t2[1]))
87195400 87072816 87075576
```



Deep copy

Une copie récursive (ou profonde) construit un nouvel objet composé puis, récursivement, insère dans l'objet composé des copies des objets trouvés dans l'objet original.

```
>>> t1 = [[1,2],[3,4]]
>>> print(id(t1), id(t1[0]), id(t1[1]))
85126920 75583200 85857224
>>> t2 = [[3,4],[5,6]]
>>>  print(id(t2), id(t2[0]), id(t2[1]))
19984672 74518480 87193680
>>> import copy
>>> t2 = copy.deepcopy(t1)
>>>  print(id(t2), id(t2[0]), id(t2[1]))
75627856 84984376 74840344
```



Valeur par défaut dans les fonctions

La déclaration de la fonction est faite au moment de la lecture du code. Les paramètres par défauts sont créés à ce moment-là, c'est-à-dire sont créés une fois pour toute.

```
>>> def fonction(n_uplet=()):
...     pass  # Suite du code
```

Modifier un objet immuable, c'est créer un nouvel objet et changer le pointeur de la variable, il n'y a donc pas de soucis.

```
>>> def fonction(liste=[]):
...     pass  # Suite du code
```

Modifier un objet muable, c'est avoir un objet altéré lors de l'appel suivant. Il faut donc faire :

```
>>> def fonction(liste=None):
...     if liste is None:
...         liste = []
...     # Suite du code
```