On découvre

- la vraie nature des variables
- la notion de référence vers une valeur
- le phénomène d'aliasing
- les conséquences de l'aliasing lors de la manipulation de listes Python qui sont mutables
- l'utilisation de Python Tutor pour visualiser l'état mémoire

Jusqu'à présent nous avons considéré les variables comme des cases dans un tableau (de la mémoire) contenant une valeur. Cette représentation n'est pas fausse et est suffisante pour bien des situations. Mais nous allons voir ici qu'elle peut être insuffisante.

Jusqu'ici nous avons dit qu'une variable est une association entre un nom et une valeur. En réalité : une variable est une association entre un nom et une *référence* à une valeur. Ce qu'on peut écrire rapidement par : une variable *référence* une valeur.

1 Notion de référence illustrée par Python Tutor

L'état de la mémoire est l'ensemble des associations variable-valeur. Suite à l'affectation suivante :

```
a = 3
```

nous avons estimé que la mémoire était de la forme :

a	3

Python Tutor `pythontutor.com/` génère un schéma[1] différent pour illustrer cet état de la mémoire, basé sur des flèches.

Le fait que la variable `a` *référence* la valeur `3` (plus exactement associe au nom `a` une *référence* vers la valeur `3`), est représenté par une flèche qui lie la variable `a` à l'objet `3` en mémoire.

NB: La référence n'est pas la valeur en mémoire, mais le moyen d'accéder à cette valeur soit pour la lire soit pour la modifier. Graphiquement on peut représenter la référence comme une flèche, comme le fait Python Tutor. Si cette notion ne vous paraît pas très claire ce n'est pas très grave, vous comprendrez dans la suite de vos études d'informatique. L'important est que vous sachiez éviter les problèmes décrits par la suite de ce cours.

2 Partager des valeurs

On considère les instructions suivantes :

```
a = 3
b = a
```

dont l'exécution peut être décrite par le tableau suivant :

a	3	3
b	/	3

[1] Ce n'est important de comprendre ce que signifient *Frames* et *Objects* dans l'affichage de Python Tutor. Les variables apparaissent dans la partie *Frames* et les valeurs en mémoire apparaissent dans la partie *Objects* .

Les instructions font intervenir une seule valeur : `3`.

L'affichage de Python Tutor met en valeur le fait que les variables `a` et `b` référencent la même valeur `3` en mémoire[2] : les 2 flèches désignent le même objet.

Les références permettent donc le *partage de valeurs*, ce qui peut poser problème dans certains cas vus plus bas.

3 Créer de nouvelles valeurs en mémoire

On ajoute une nouvelle affectation de `a` avec les instructions suivantes :

```
a = 3
b = a
a = 5
```

dont l'exécution peut être décrite par le tableau suivant :

a	3	3	5
b	/	3	3

L'affichage de Python Tutor montre clairement qu'à l'issue de l'exécution `a` et `b` référencent 2 valeurs différentes [3]:

On obtient le même résultat si on remplace `a = 5` par l'affectation `a = a + 2`, qui est exécutée ainsi :

- l'expression `a + 2` est évaluée
  - elle vaut `5`
  - c'est une *nouvelle* valeur, elle apparaît comme telle
- l'affectation modifie l'association entre le nom `a` et une référence à sa valeur
  - la variable référence maintenant cette nouvelle valeur `5`

Les affectations impliquant d'autres types de données (booléens, chaînes de caractères, listes…) suivent le même principe du moment que lors de l'affectation une *nouvelle valeur* est créée et référencée par la variable.

L'affichage de Python Tutor est néanmoins un peu différent dans le cas d'une liste : il fait apparaître les éléments de la liste ainsi que leur indice dans la liste[4].

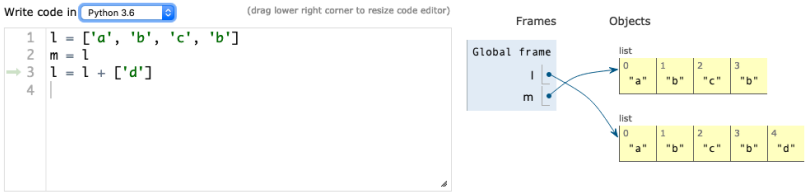
[2] Dit autrement : les deux références associées à `a` et à `b` sont égales : elles désignent la même valeur `3`.

[3] Dit autrement : le nom `b` reste associé à une référence qui désigne la valeur `3`. Le nom `a` est associé à une nouvelle référence qui désigne la valeur `5`.

[4] On peut se demander pourquoi les éléments de la liste apparaissent dans des cases alors qu'on pourrait s'attendre à ce qu'une flèche parte de chaque case pour désigner la valeur référencée… Python Tutor procède ainsi et même si ça ne semble pas forcément très cohérent on est bien content de pouvoir copier ses schémas !

L'exemple suivant illustre l'apparition d'une nouvelle valeur de type liste en mémoire, créée par l'opérateur de concaténation de liste. L'affectation `l = l + ['d']` de la variable `l` est exécutée en deux étapes :

- l'expression `l + ['d']` est évaluée
  - elle vaut `['a', 'b', 'c', 'b', 'd']`
  - c'est une *nouvelle* valeur, elle apparaît comme telle
- l'affectation modifie l'association entre le nom `l` et une référence à sa valeur
  - la variable référence maintenant cette nouvelle valeur `['a', 'b', 'c', 'b', 'd']`.



4 Modifier des valeurs partagées, aliasing

Nous avons vu jusqu'à présent 2 sortes de types de données : les types non mutables (`int`, `float`, `bool`, `str` et `range`) et le type liste (`list`) qui lui est mutable.

Les nombres, les booléens, les chaînes et les intervalles, appartenant à des types non mutables, sont des valeurs *qui ne peuvent être modifiées* :

- on ne peut que les utiliser dans des expressions (pas à gauche d'une affectation)
- ces expressions produisent de *nouvelles* valeurs.

Au contraire les listes sont mutables. Si `liste` est de type `list` et référence une liste non vide, on peut modifier `liste` en affectant une valeur à `liste[0]`. On peut aussi utiliser la méthode `append` qui permet d'ajouter un élément à la fin de la liste.

Et ça peut être le début des ennuis dans le cas d'un partage de valeurs…

On repart de l'exemple précédent dans lequel les deux variables `l` et `m` référencent la même *valeur partagée* de type `list`.

On modifie cette valeur référencée en remplaçant l'élément d'indice 2 par la valeur `'e'` par `l[2]= 'e'` :

On constate que :

- les deux variables `l` et `m` référencent toujours cette même liste partagée, qui a été modifiée.
- La valeur référencée par la variable `m` a donc été modifiée, alors qu'aucune affectation à `m` n'apparaît dans le programme après l'initialisation de `m` ! `m[2]` vaut `'e'` alors que c'est `l[2]` qui a été affecté. On peut avoir l'impression désagréable que la valeur de `m` a bougé "toute seule".

Ce phénomène est appelé *aliasing* : plusieurs références permettent d'accéder à une même valeur partagée mutable, dont la modification se répercute sur toutes les références.

La représentation des références par des flèches permet de mieux comprendre le problème que le traditionnel tableau de la mémoire :

`l`	`['a', 'b', 'c', 'b']`	`['a', 'b', 'c', 'b']`	`['a', 'b', 'e', 'b']`
`m`	/	`['a', 'b', 'c', 'b']`	`['a', 'b', 'e', 'b']`

Tout moyen de modifier une valeur partagée déclenchera le problème. Supposons qu'on ajoute un élément à la liste par `m.append('z')` :

Cette fois on peut avoir l'impression bizarre que la valeur référencée par `l` a changé "toute seule" en récupérant un nouvel élément, sans affectation à `l` ni modification via le nom `l`.

NB : on fera bien la différence entre l'opérateur de concaténation de listes `+` qui produit *une nouvelle liste* et la procédure `append` qui *modifie* la liste à laquelle elle s'applique, et ne renvoie rien (`None`).

5 Memento

- une variable associe un nom et une *référence* à une valeur
- des variables différentes peuvent référencer une même valeur. C'est un phénomène d'*aliasing*
- l'aliasing de valeurs mutables – les listes – doit être maîtrisé lors de la modification de telles valeurs

On découvre :

- pourquoi la copie de valeurs de type `list` est indispensable
- comment réaliser une copie d'une liste avec Python
- le cas des listes de listes

À partir de cette semaine nous allons manipuler intensivement des listes dans les gros TPs qui terminent le semestre.

On a vu dans le support précédent que le partage de référence et la modification de listes partagées pouvaient déclencher des phénomènes d'aliasing désagréables.

Pour éviter ces phénomènes, on vous demandera d'éviter tout partage de valeur de type liste.

1 Ce qu'on a fait jusqu'à présent : style fonctionnel

Jusqu'à présent on a codé des fonctions qui prennent en paramètre des listes et qui renvoient des listes, dans un style appelé "fonctionnel".

Par exemple, pour coder une fonction qui remplace les valeurs négatives d'une liste d'entier par leur valeur absolue nous avons écrit :

```
def positive(liste:list[int]) -> list[int]:
    """Renvoie une nouvelle liste dans laquelle les valeurs négatives
    de `liste` sont remplacées par leur valeur absolue.

    precondition : /
    Exemples :
    $$$ positive([0, 3, -6, 12, -7])
    [0, 3, 6, 12, 7]
    $$$ positive([0, 3, 6, 12, 7])
    [0, 3, 6, 12, 7]
    $$$ positive([-3, -6, -12])
    [3, 6, 12]
    $$$ positive([])
    []
    """
    l = []
    for e in liste:
        if e >= 0:
            l.append(e)
        else:
            l.append(-e)
    return l
```

Le code crée une liste vide dans laquelle on copie élément par élément la liste paramètre, éventuellement en changeant le signe de l'élément. La liste renvoyée est un nouvel objet qui ne partage aucune valeur avec la liste passée en paramètre et il n'y a pas d'aliasing possible. On peut le vérifier :

```
>>> l1 = [0, 3, -6, 12, -7]
>>> l2 = l1
>>> l3 = positive(l1)
>>> l3
[0, 3, 6, 12, 7]
>>> l2 # n'a pas changé
[0, 3, -6, 12, -7]
>>> l1 # non plus
[0, 3, -6, 12, -7]
```

Une autre approche possible (style "procédural") serait d'écrire une procédure qui prend en paramètre une liste et qui modifie sa valeur, sans rien renvoyer.


```
def positivee(liste:list[int]) -> None:
    """Modifie la liste `liste` en remplaçant  les valeurs négatives par leur valeur absolue.

    precondition : /
    Exemples :
    $$$ l = [0, 3, -6, 12, -7] #1
    $$$ positivee(l)           #2
    $$$ l                      #3
    [0, 3, 6, 12, 7]           #4
    $$$ l = [0, 3, 6, 12, 7]   #5
    $$$ positivee(l)           #6
    $$$ l                      #7
    [0, 3, 6, 12, 7]           #8
    $$$ l = [-3, -6, -12]      #9
    $$$ positivee(l)           #10
    $$$ l                      #11
    [3, 6, 12]                 #12
    $$$ l = []                 #13
    $$$ positivee(l)           #14
    $$$ l                      #15
    []                         #16
    """
    for i in range(len(liste)):
        if liste[i] < 0:
            liste[i] = -liste[i]
```

Observons les tests de cette fonction. Pour établir qu'un appel à `positivee(l)` modifie la liste paramètre `l`, on ne peut pas comme d'habitude utiliser la valeur renvoyée par la fonction, puisque cette fonction ne renvoie rien. Il faut donc pouvoir dire dans le test :

- avant appel : `l` a une valeur, par exemple `[0, 3, -6, 12, -7]`
- après appel : `l` a été modifiée et vaut maintenant `[0, 3, 6, 12, 7]`

Pour ce faire on doit utiliser une variable locale au test, la liste `l`, qui est la liste à modifier. Cette approche est classique en test et vous l'utiliserez abondamment dès la L2 Info.

- La liste est initialisée à la ligne 1 par une affectation avec la valeur `[0, 3, -6, 12, -7]`. Cette affectation modifie la mémoire locale associée à toutes les lignes de test contenues dans la docstring. Cette ligne ne contient pas de comparaison entre une valeur calculée et une valeur attendue, son exécution par L1Test ne produit pas d'effet dans la fenêtre de test.
- La ligne 2 effectue l'appel à `positivee(l)`, qui modifie la valeur de `l`. De même cette ligne ne contient pas de comparaison entre une valeur calculée et une valeur attendue, son exécution par L1Test ne produit pas d'effet dans la fenêtre de test.
- Les lignes 3 et 4 contiennent un test classique qui vérifie que la nouvelle valeur de `l` est bien `[0, 3, 6, 12, 7]`. L'exécution de ces 2 lignes produira un affichage classique en vert ou en rouge dans la fenêtre de test, selon la valeur de `l`.

Les lignes 5 à 8 puis 9 à 12 puis 13 à 16 contiennent de nouveaux tests sur le même schéma avec d'autres valeurs pour `l` :

- `l` valant `[0, 3, 6, 12, 7]`, lignes 5 à 8
- `l` valant `[-3, -6, -12]`, lignes 9 à 12
- `l` valant la liste vide, lignes 13 à 16.

La fonction `positivee` convient donc au final 4 tests qui vérifient chacun la valeur de `l` après appel de la fonction.

On commence à se rendre compte avec cet court exemple qu'écrire des tests c'est avant tout écrire du code (de test) qui vérifie que du code (la fonction à tester) se comporte correctement. Ce sera flagrant en L2, quand vous utiliserez des bibliothèques de test standard.

Avec une telle fonction `positivee` qui modifie son paramètre, le problème d'aliasing peut arriver rapidement :

```
>>> l1 = [0, 3, -6, 12, -7]
>>> l2 = l1
>>> positivee(l1)
>>> l1
[0, 3, 6, 12, 7]
>>> l2
[0, 3, 6, 12, 7]
```

On remarque que, comme `l1` et `l2` référencent la même valeur partagée, la modification de la liste référencée par `l1` a eu une action sur la liste référencée par `l2`.

On vous demande donc de continuer à utiliser un style fonctionnel. Par exemple, pour modifier la grille du jeu (parce qu'un joueur y a placé un pion par exemple), on écrira une fonction avec une signature de ce genre :

```
def modification_grille(grille:list[list[...]], ...) -> list[list[...]]:
    """Renvoie une nouvelle grille telle que...
    """
    ...
```

Cette fonction prend en paramètre la grille courante (une liste de liste) et renvoie un nouvel objet de type liste de liste, qui ne partage aucune valeur avec la liste passée en paramètre (pour éviter les problèmes d'aliasing). La valeur renvoyée tient compte de la modification (par exemple : un nouveau pion est apparu dans la grille).

Cela dit, quand on veut juste modifier un élément d'une liste de liste, est-on obligé de recopier tous les éléments un par un ? Python fournit des opérateurs qui permettent de copier une liste en créant une nouvelle liste.

2 Différentes manières de copier une liste

Les manières de créer une nouvelle liste, copie d'une liste donnée, sont nombreuses. On a déjà parlé de copier la liste éléments par éléments dans une liste vide. On n'y pense pas car l'idée de copie n'apparaît pas, mais créer une tranche de liste entière par `liste[:]` crée aussi une *nouvelle liste* identique à `liste`.

Python fournit par ailleurs des fonctions dédiées à la copie.

2.1 Méthode `list.copy()`

Python fournit une méthode `liste.copy()` sur les valeurs de type `list` qui crée une copie de `liste` :

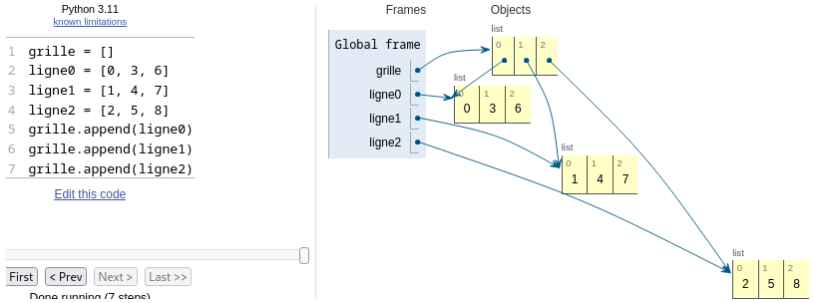
2.2 Le cas des listes de listes, la copie profonde

On représente une grille par une liste de listes, au choix la liste de ses lignes ou la liste de ses colonnes.

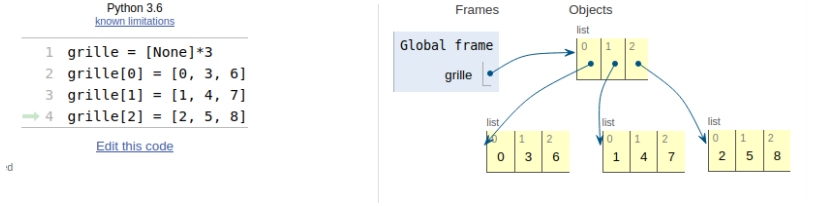
Pour représenter le tableau suivant :

0	3	6
1	4	7
2	5	8

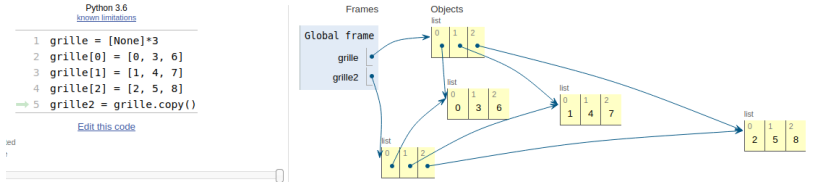
on utilisera par exemple la liste de ses lignes de la manière suivante :



On remarquera dans le cas de ce code le partage de valeur par exemple entre la liste référencée par `ligne0` et la liste référencée par `grille[0]`. Alors que dans le cas du code suivant, il n'y aucun partage de valeur :

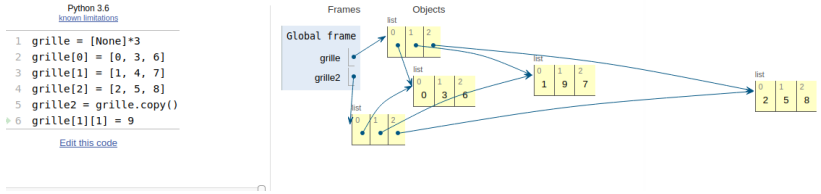


On essaie de copier cette liste de liste avec l'opérateur `copy` fourni par la classe `list` vu précédemment :



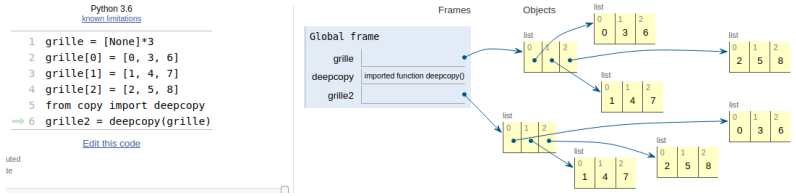
Comme le montre le schéma ci-dessus, `copy` effectue une *copie superficielle* de la liste principale : les listes représentant les lignes/colonnes ne sont pas copiées. Les listes représentant les lignes/colonnes sont partagées entre la liste `grille` et sa copie `grille2`.

Modifier un élément du tableau représenté par `grille` modifiera celui représenté par `grille2`. On a donc à nouveau un phénomène d'aliasing.



`grille[1][1]` et `grille2[1][1]` référencent tout deux la valeur `9` suite à la modification de `grille`.

On utilisera alors la fonction `deepcopy` fournie par le module `copy`. Il n'existe pas d'équivalent dans la classe `list`. `deepcopy` effectue une *copie profonde*, comme son nom l'indique.



Vous reverrez dans la suite de vos études d'informatique la notion de copie superficielle/profonde. Pour ce semestre, on retiendra juste comment faire pour ne pas être embêté · e par des histoires d'aliasing…

3 Application aux grilles : erreurs à ne pas faire

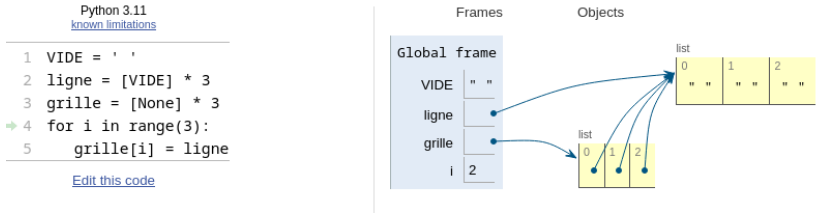
Les opérations classiques sur les grilles sont l'initialisation et la modification.

3.1 Initialisation d'une grille à une valeur constante

Supposons que dans le cadre d'un jeu on souhaite initialiser une grille carrée de taille 3 de telle manière que chaque case soit vide. On représente la grille par la liste de ses lignes. On représente une case vide par le caractère espace. On pourrait être tenté de construire une ligne de longueur 3 contenant des caractères espaces, puis d'utiliser 3 fois cette ligne comme suit :

```
VIDE = ' '
def init_grille_incorrect() -> list[list[str]]:
    """
    $$$ init_grille_incorrect()
    [[VIDE, VIDE, VIDE], [VIDE, VIDE, VIDE], [VIDE, VIDE, VIDE]]
    """
    ligne = [VIDE] * 3
    grille = [None] * 3
    for i in range(3):
        grille[i] = ligne
    return grille
```

L'aliasing qui en résulte apparaîtra quand on voudra modifier par exemple la case d'indices (0, 0) : les cases d'indices (1, 0) et (2, 0) subiront la même modification !




```
>>> grille = init_grille_incorrect()
>>> grille[0][0] = 'X'
>>> grille
[['X', ' ', ' '], ['X', ' ', ' '], ['X', ' ', ' ']]
```

Pour éviter l'aliasing, il suffit :

- soit de supprimer la variable locale `ligne` et d'écrire directement `grille[i] = [VIDE]*3`
- soit d'utiliser des copies de `ligne` et d'écrire `grille[i] = ligne.copy()`

3.2 Modification d'une grille

Supposons qu'on souhaite modifier la valeur de la case d'indice (i, j) d'une grille. Si on bien compris ce qui précède, on écrira une fonction `modif_grille` qui prend en paramètre une grille et les données nécessaires à sa modification et renvoie une nouvelle grille. De plus, pour ne pas être obligé de recopier tous les éléments de la grille paramètre, on utilisera un opérateur de copie. Mais si on se trompe d'opérateur…

```
def modif_grille_incorrect(grille:list[list[int]], i:int, j:int, val:int) -> list[list[int]]:
    """
    $$$ modif_grille_incorrect([[0, 0], [0, 0]], 0, 0, 4)
    [[4, 0], [0, 0]]
    """
    grille2 = grille.copy()
    grille2[i][j] = val
    return grille2
```

… on s'aperçoit que la modification a impacté aussi la grille passée en paramètre !

```
>>> g1 = [[0, 0], [0, 0]]
>>> g2 = modif_grille(g1, 0, 0, 4) # plus lisible
>>> g2
[[4, 0], [0, 0]]
>>> g1
[[4, 0], [0, 0]] # !!!!
```

On pourrait d'ailleurs écrire un test qui échoue, qui utilise des variables locales au test comme expliqué précédemment :

```
def modif_grille_incorrect(grille:list[list[int]], i:int, j:int, val:int) -> list[list[int]]:
    """
    $$$ g1 = [[0, 0], [0, 0]]
    $$$ g2 = modif_grille_incorrect2(g1, 0, 0, 4)
    $$$ g2
    [[4, 0], [0, 0]]
    $$$ g1 # ce test échoue !
    [[0, 0], [0, 0]]
    """
    grille2 = grille.copy()
    grille2[i][j] = val
    return grille2
```

Pour éviter l'aliasing, il suffit d'utiliser l'opérateur `deepcopy`.

4 Memento

- les grilles, représentées par des listes de listes, sont sujettes aux problèmes d'aliasing
- on adoptera en TP un style fonctionnel : des fonctions qui prennent en paramètre une grille (liste de liste) et renvoie une nouvelle liste
- Python fournit de quoi créer une nouvelle liste par copie d'une autre liste
- l'opérateur `copy` de la classe `list` effectue une copie superficielle. Dans le cas d'une liste de liste, les lignes ou colonnes de la grille restent partagées
- l'opérateur `deepcopy` du module `copy` effectue une copie profonde : plus de pb d'aliasing.