♦ TP3 – Aliasing, portée des variables et effet de bord

I. — Aliasing

En Python, lorsqu'on effectue l'affectation `A = v`, il se passe en fait différentes choses :

- la création d'un espace mémoire destiné à recevoir la valeur `v` : cet espace dispose d'un identifiant (une sorte d'adresse mémoire) prenant la forme d'un entier qu'on peut obtenir grâce à la fonction `id` ;
- le stockage de la valeur `v` dans l'espace mémoire dédié ;
- la création d'une étiquette (le nom de la variable), ici `A` ;
- la création d'un lien entre l'étiquette `A` et l'espace mémoire contenant `v`.

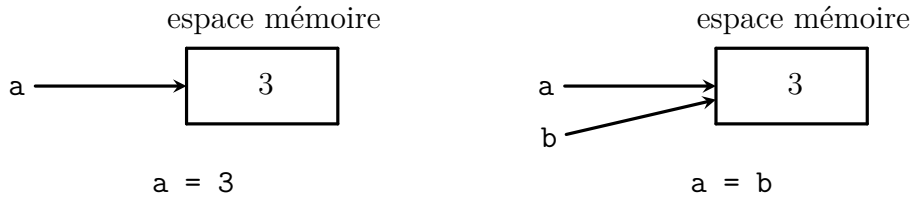
Ainsi, si on exécute le code suivant :

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

alors

– à la première ligne, Python va créer un espace mémoire, y stocker l'entier 3, créer un étiquette `a` puis en lien entre cette étiquette et l'espace mémoire contenant 3 ;

– à la seconde ligne, Python crée simplement une nouvelle étiquette `b` et un lien entre cette étiquette et l'espace mémoire contenant 3.

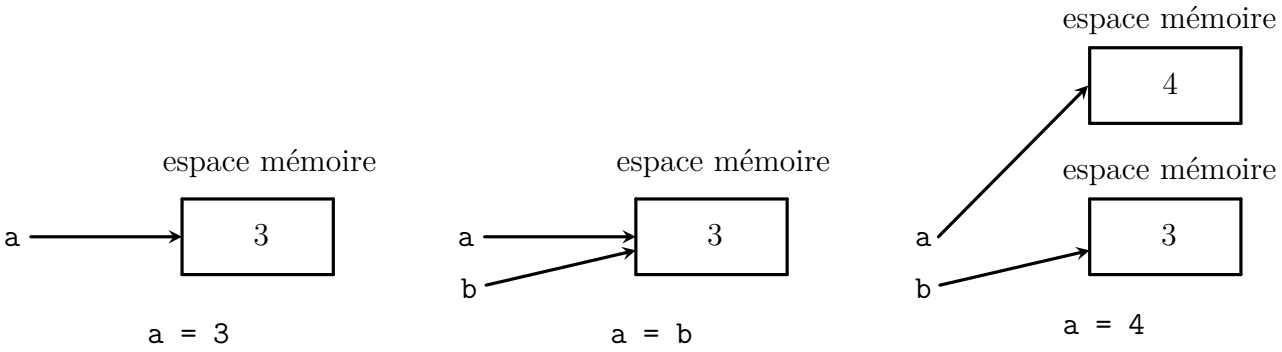


Ainsi, on dispose de deux étiquettes qui pointe vers le même espace mémoire, c'est ce qu'on appelle l'*aliasing*. On dit que `b` est un alias de `a`.

Pour les objets non mutables, l'aliasing ne pose pas de problème. Ainsi, si on exécute le code suivant :

```
a = 3
b = a
a = 4
```

alors Python effectue les mêmes actions que précédemment mais à la troisième ligne il crée un nouvel espace mémoire contenant l'entier 4, détruit le lien entre l'étiquette `a` et l'espace mémoire contenant 3 puis crée un nouveau lien entre `a` et l'espace mémoire contenant 4. Ainsi, `a` renvoie 4 et `b` renvoie toujours 3.



Si on exécute le programme suivant :

```
a = 3
print(id(a))
b = a
print(id(b))
a = 4
print(id(a))
```

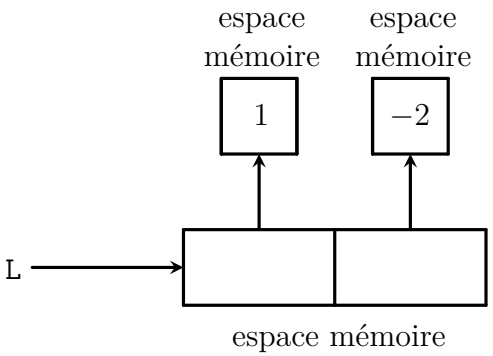
on obtient à l'affichage

```
139841127776560
139841127776560
139841127776592
```

ce qui est cohérent puisque dans les deux premières affectations `a` et `b` sont liés au même espace mémoire mais, à la troisième affectation, l'espace mémoire associé à `a` change.

Pour les objets mutables comme les listes, les choses sont plus compliquées. Considérons, par exemple, l'affectation `L = [1,-2]`. Dans ce cas, Python va

- créer un espace mémoire pour chacun des éléments de la liste donc, ici, un espace mémoire pour 1 et un espace mémoire pour -2 ;
- stocker chaque valeur dans l'espace mémoire dédié ;
- créer un espace mémoire pour la liste elle-même avec des référence vers les espaces mémoires des éléments créés précédemment ;
- créer une étiquette `L` liée à l'espace mémoire de la liste.



Ceci a un impact important sur l'aliasing. En effet, si on exécute le code suivant :

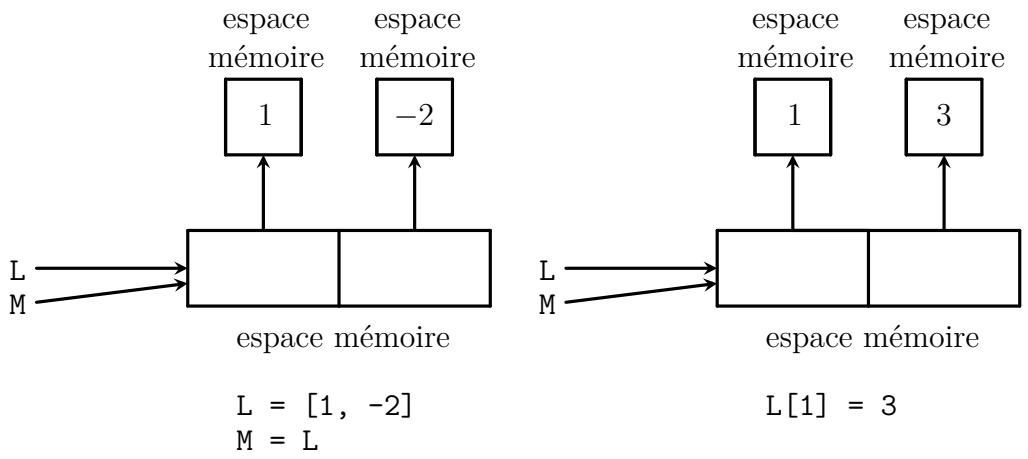
```
L = [1, -2]
M = L
L = [1, 3]
```

on obtient que le même type de schéma que pour les variables non mutables.

En revanche, si on exécute le code suivant :

```
L = [1, -2]
M = L
L[1] = 3
```

alors on ne modifie pas le fait que `L` et `M` sont liés au même espace mémoire de liste mais on modifie la valeur stockée dans l'espace mémoire de second élément, ce qui a pour effet de modifier `L` mais **également de modifier** `M`.



Exercice 1. Déterminer le ou les affichages obtenu(s) lors de l'exécution de chacun des programmes suivants.

```
# Programme 1
a = 3
b = 3
c = a
a = 5
print(a,b,c)
```

```
# Programme 2
L = [1, 2, 3]
M = [1, 2, 3]
L[0] = 'a'
print(L)
print(M)
```

```
# Programme 3
L = [4, 'c', True]
N = L
N[1] = 3.14
print(L)
print(N)
```

```
# Programme 4
M = [0, 0, 0]
N = 4*[M]
print(M, N)
N[1][0] = 1
print(M, N)
N[2] = [2, 3, 4]
print(M, N)
N[3][2] = 11
print(M, N)
N[2][1] = 'n'
print(M, N)
```

```
# Programme 5
N = 4*[3*[0]]
print(N)
N[1][0] = 1
print(N)
N[2] = [2, 3, 4]
print(N)
N[3][2] = 11
print(N)
N[2][1]= 'n'
print(N)
```

```
# Programme 6
N = [[0 for i in
   range(3)] for j
   in range(4)]
print(N)
N[1][0] = 1
print(N)
N[2] = [2, 3, 4]
print(N)
N[3][2] = 11
print(N)
N[2][1] = 'n'
print(N)
```

II. — Effet de bord

Dans le script d'une fonction, les variables ont une portée locale ce qui signifie qu'elles n'existent qu'au cours de l'exécution de la fonction et qu'elles ne sont plus utilisables en dehors de la fonction.

Ainsi, si on exécute le code suivant :

```
a = 3

def plus_2(x):
  a = x+2
  return a

print(plus_2(a))
print(a)
```

on obtient à l'affichage

```
5
3
```

La variable `a` vaut initialement 3. Lors de l'appel `plus_2(a)`, on va donc exécuter la fonction en prenant 3 en argument. À l'intérieur de la fonction, une variable `a` est crée - différente de la précédente, on lui affecte la valeur $3 + 2 = 5$ puis on renvoie 5. Ainsi, le premier affichage est bien 5 et, ensuite, le second affichage fait référence à la « première variable `a` » (puisque la « seconde variable `a` » a été supprimée avec la fin de l'exécutation de la fonction).

Dans le cas où une variable à une valeur mutable, comme une liste, il est cependant possible que la fonction modifie la valeur de cette variable : on dit alors que la fonction a un effet de bord.

C'est le cas de la fonction suivante :

```
def plus_2_liste(liste) :
  for k in range(len(liste)) :
    liste[k] += 2
  return liste
```

En effet, lors de l'exécution de la fonction, la liste passée en argument va être modifiée au niveau des espaces mémoire de ses éléments et cette modification s'étend au-delà de la fonction. Une façon d'éviter ce problème est de créer une nouvelle liste. Attention, cependant, le script suivant possède le même effet de bord :

```
def plus_2_liste(liste) :
  M = liste
  for k in range(len(M)) :
    M[k] += 2
  return liste
```

puisque `L` et `M` sont liées au même espace mémoire. Il faut créer une nouvelle liste avec un nouvel espace mémoire. On peut procéder ainsi :

```
def plus_2_liste(liste) :
  M = [e for e in liste]
  for k in range(len(M)) :
    M[k] += 2
  return M
```

On peut remplacer la ligne `M = [e for e in liste]` par l'extraction `M = liste[:]`. Cependant, avec l'une ou l'autre de ces deux syntaxes, on aura à nouveau un problème d'aliasing si des éléments de `liste` sont eux-mêmes des listes. Dans ce cas, il faut faire appel à la fonction `deepcopy` du module `copy` en écrivant `M = copy.deepcopy(liste)`.

Exercice 2. Déterminer la valeur renvoyée par la fonction ainsi que la valeur de la variable `a` à la fin de l'exécution de chacun des programmes suivants.

```
# Programme 1
a = 2
def carre(x):
  a = x**2
  return a
```

```
# Programme 2
a = 2
def puissance_a(x):
  b = x**a
  return b
```

```
# Programme 3
a = 2
def puissance_a(x):
  a = x**a
  return a
```

Exercice 3. Réécrire le programme 4 de l'exercice 1 afin d'obtenir les mêmes affichages pour la liste `N` mais sans modifier la liste `M` initiale. (Le programme doit fonctionner quelle que soit la liste `M` initiale.)

Exercice 4. On reprend le programme 3 de l'exercice 1.

1. On remplace, dans `L`, `'c'` par la liste `[1,2,3]` et on remplace `N[1] = 3.14` par `N[1][1] = 3.14`. Qu'obtient-on à l'affichage ?
2. Réécrire le programme ainsi modifié pour qu'il affiche la même valeur de `N` mais sans modifier la liste `L`.

Exercice 5. La fonction de l'exercice 26 du TP2 a-t-elle un effet de bord ? Si oui, écrire une fonction `valeur_absolue_liste_2` effectuant le même travail mais sans effet de bord.

Exercice 6. Écrire, sans effet de bord,

1. une fonction `ajout` qui prend en argument une liste `L` et un objet `o` et qui ajoute `o` à la liste `L` en dernière position.
2. une fonction `suppr` qui prend en argument une liste `L` et un objet `o` et qui supprime toutes les occurrences de `o` dans la liste `L`.