Représentation mémoire

Judicaël Courant

Lycée La Martinière-Monplaisir, Lyon

Le 18 novembre 2018

Introduction

Quelques subtilités de Python :

- Quelle est la différence entre `t += [1, 2]` et `t = t + [1, 2]` ?
- Que vaut `M` après les instructions `M = [[0] * 3] * 4 ; M[0][1] = 17` ?
- Que vaut `u` après `u = [1, 3, 2]; t = u; t.sort()` ?
- Et après `u = [1, 3, 2]; t = u; t = sorted(t)` ?

Objectif

Être capable de prédire (correctement) le comportement de Python.
Pour cela, besoin de comprendre :

- La représentation des données en mémoire.
- Le phénomène d'aliasing.

Représentation des données en mémoire

- Un modèle simple permet de comprendre l'essentiel.
- Comme tous les modèles, c'est une *représentation* de la réalité, non *la réalité elle-même*.

Notre modèle

Mémoire vive Suite de cases numérotées de façon consécutive. Aujourd'hui, 64 bits suffisent à représenter tous les numéros de cases possibles.

Adresse d'une case Son numéro.

Une case mémoire Contient 64 bits. Permet de coder l'adresse d'une autre case, ou quelques caractères.

NB : $2^{32} \approx 4 \times 10^9$, $2^{64} \approx 2 \times 10^{19}$.

Les objets

Objets Nom donné aux données manipulées par Python.

Tas Partie de la mémoire où les objets sont stockés.

Représentation des objets Cases contiguës du tas codant :

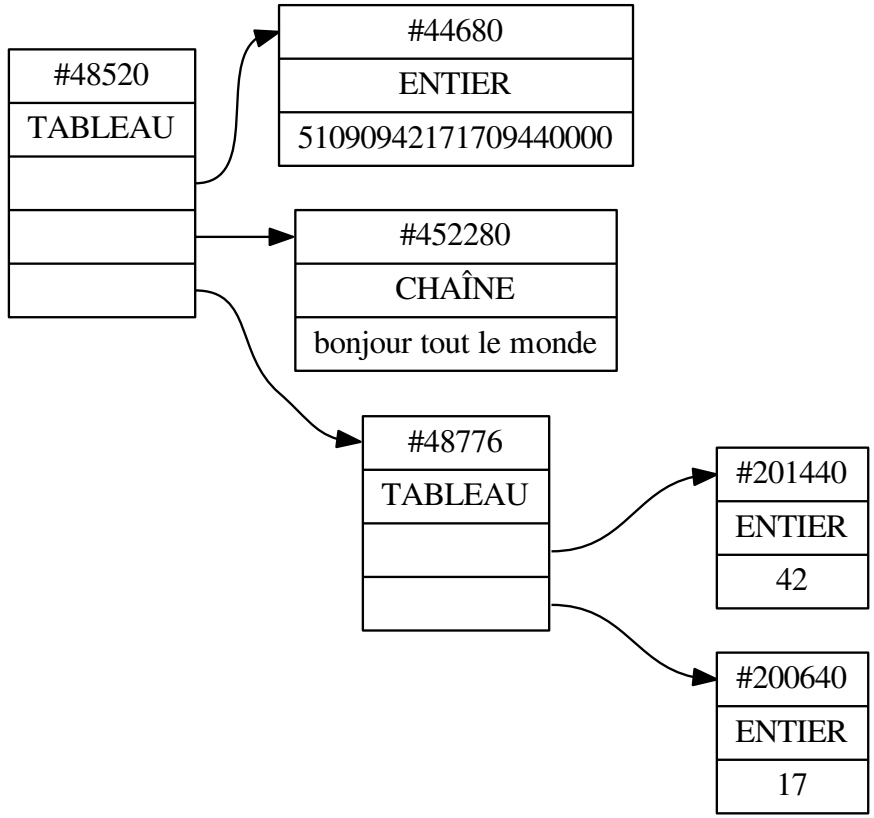
- le type de l'objet ;
- ses données propres ;
- les adresses des autres objets auxquels il fait référence.

***Adresse* d'un objet** Adresse de sa première case mémoire.

Exemple

Représentation de

```
1  [ 51090942171709440000,
2    'bonjour tout le monde',
3    [42, 17]
4    ]
```



Les variables

- Les *variables* permettent de désigner des objets.
- Elles sont stockées dans une partie de la mémoire appelée *pile*.
- Asymétrie : les *objets* ne peuvent pas désigner des *variables*.
- Chaque variable contient une *valeur*.

Qu'est-ce qu'une valeur ?

Cela dépend des langages de programmation.

Scilab Valeur d'un objet = *données* contenues dans l'objet.

Python Valeur d'un objet = *adresse* de l'objet.

Conséquences

On suppose qu'on a exécuté `u = [0] * (10**9)`
Que se passe t-il quand :

- On effectue l'affectation `t = u` ?
- On appelle `f(u)` avec `def f(t): return t[0]+t[1]` ?

Scilab

- Copie des données
- Coût : $O($`len(u)`$)$ en temps et en espace

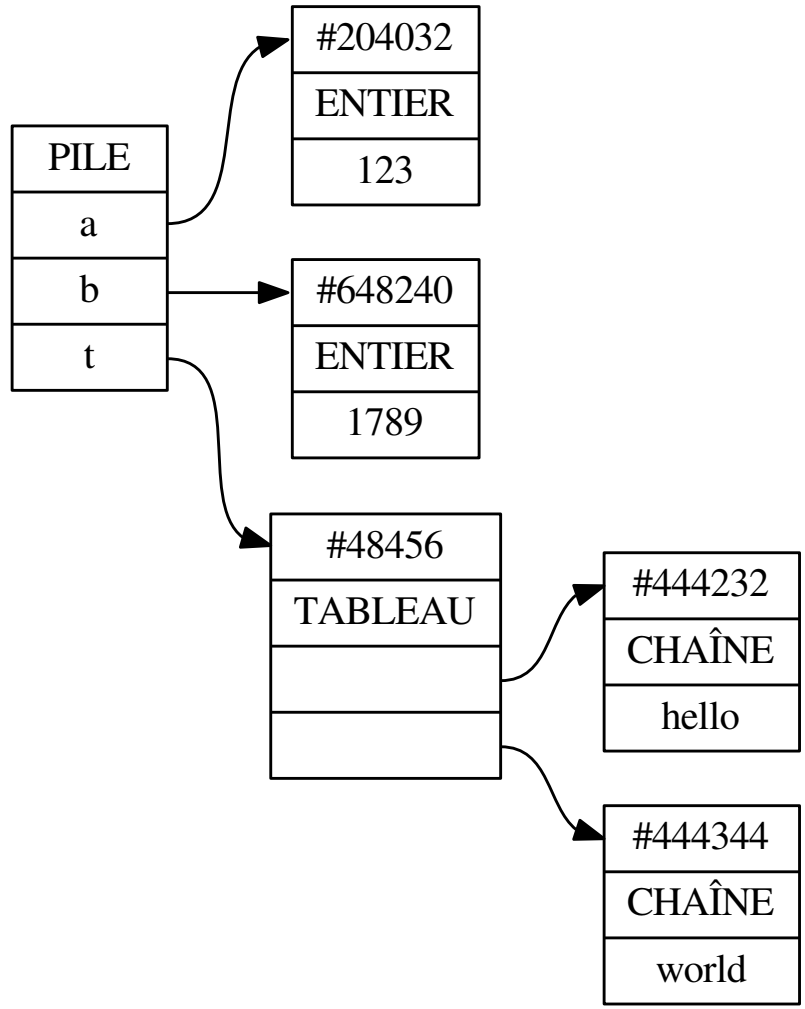
Python

- Copie d'une simple adresse
- Coût : $O(1)$ en temps et en espace.
- Aliasing : deux variables peuvent désigner le même objet. Pratique mais dangereux.

Règle d'or

En Python, une valeur est juste l'adresse d'un objet.
Et les variables contiennent des valeurs.

```
1  a = 123
2  b = 1789
3  t = ['hello', 'world']
```



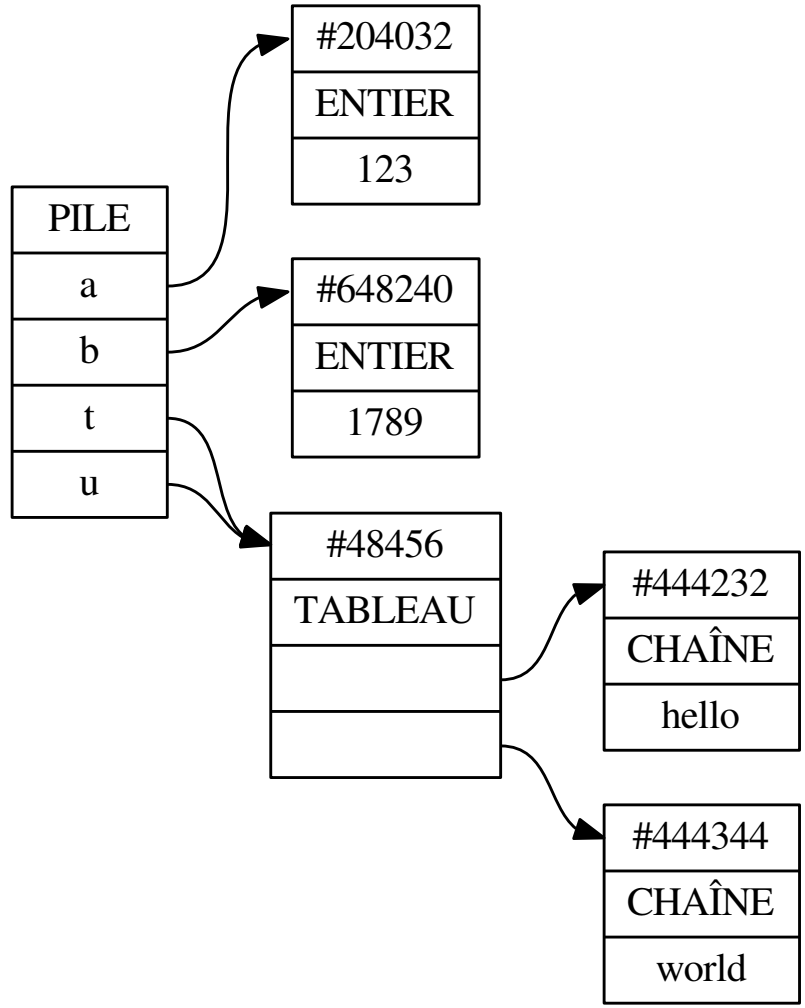
Aliasing

Deux variables peuvent désigner le même objet :

Après l'exécution de

```
1  u = t
```

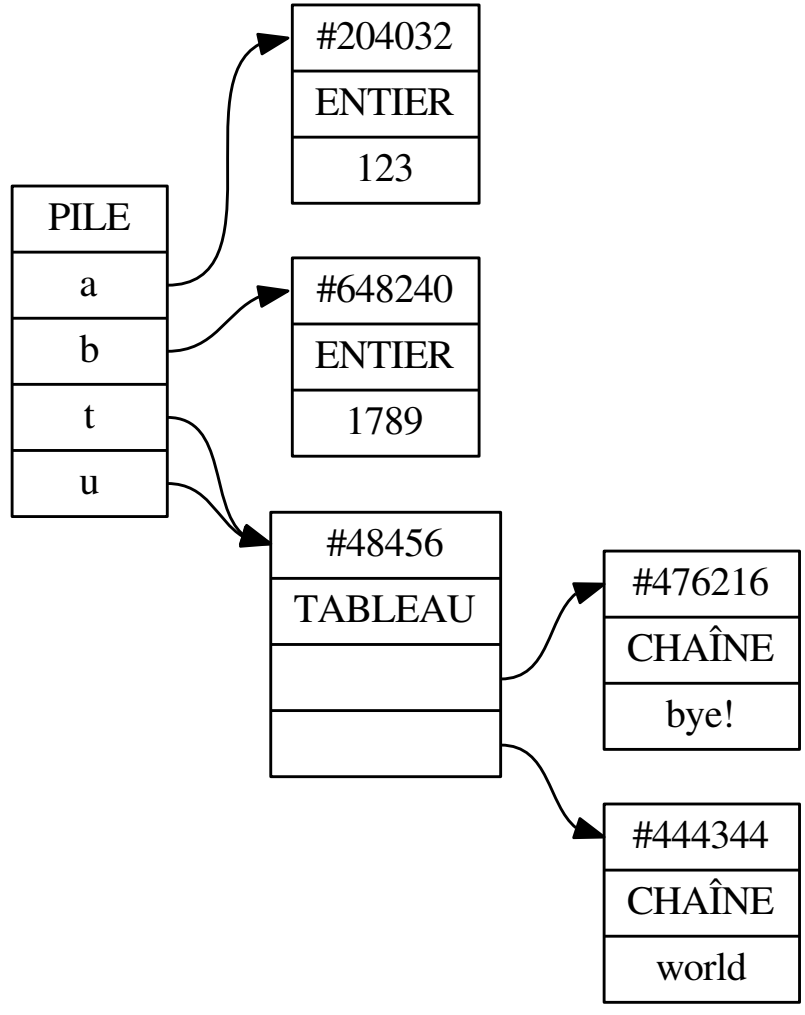
`u` et `t` sont alias l'un de l'autre.



Conséquence de l'aliasing

```
1   u[0] = 'bye!'
```

modifie l'objet désigné par `u`
donc l'objet désigné par `t`.



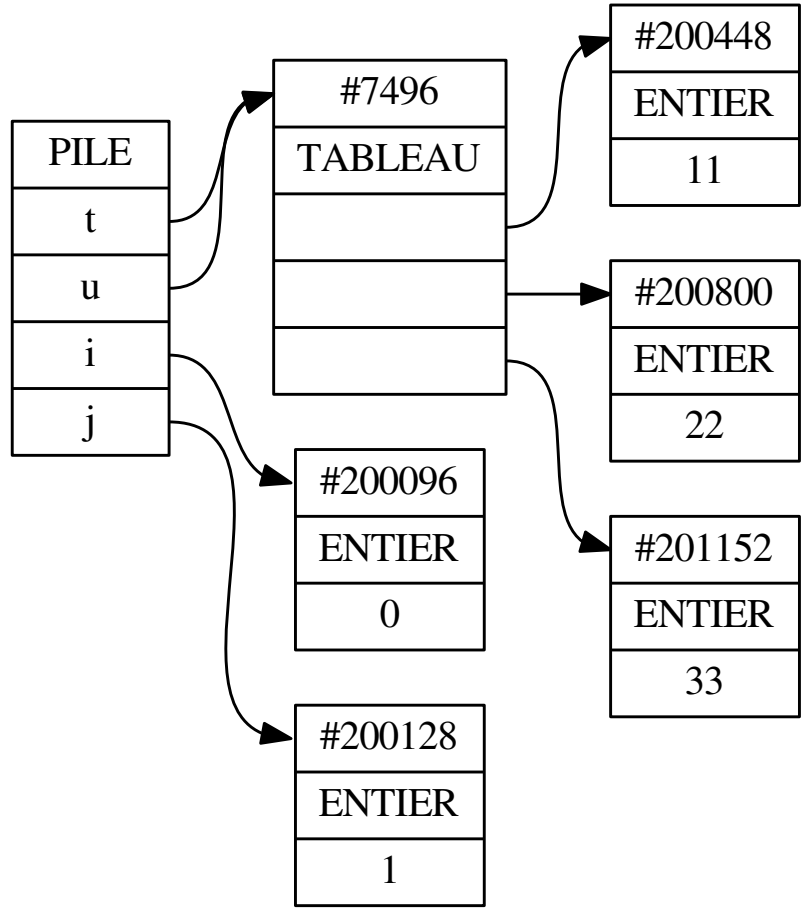
Intérêt de l'aliasing

On peut écrire une fonction qui échange deux éléments d'un tableau:

```
1  def swap(u, i, j):
2      u[i], u[j] = u[j], u[i]
3
4  t = [ 11, 22, 33 ]
5  swap(t, 0, 1)
```

(impossible à écrire en Scilab)

Avant ligne 2



Une distinction fondamentale

Ne pas confondre:

- Modifier une variable (faire qu'elle désigne un autre objet).
- Modifier un objet.

```
1  # Vélo : [ couleur, diam_roue_arr, retroviseur ]
2  peugeot = [ 'bleu', 30, False ]
3  azub = [ 'rouge', 26, True ]
4  monvelo = peugeot # en 1987
5  monvelo = azub # en 2009
6  monvelo[2] = False # en 2010
```

2009 j'ai changé de vélo (modifié la variable)

2010 j'ai changé mon vélo (modifié l'objet)

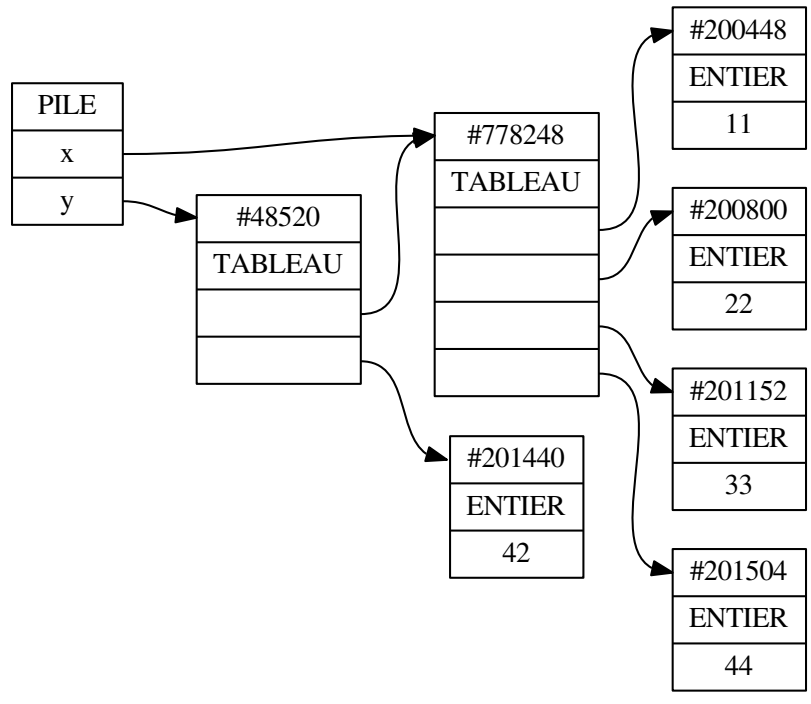
Des risques de l'aliasing

Modifier l'objet désigné par la variable `x`

- Change le contenu de l'objet désigné par `x`.
- Peut changer le contenu de l'objet désigné par une autre variable.

Exemple

```
1  x = [11, 22, 33]
2  y = [x, 42]
3  x.append(44)
```



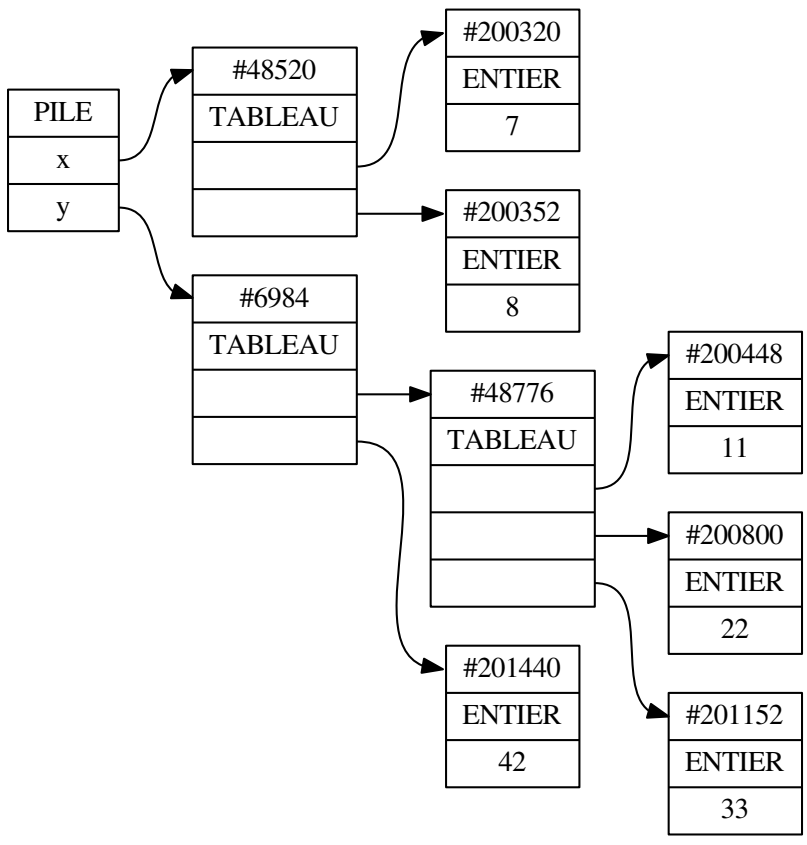
A contrario

Modifier la variable `x`

- Ne modifie pas les objets existants.
- Ne change pas le contenu des objets désignés par d'autres variables.

Exemple

```
1  x = [11, 22, 33]
2  y = [x, 42]
3  x = [7, 8]
```



À retenir

Distinction fondamentale

- Modifier une variable
- Modifier un objet

Comment créer / modifier une variable

Pour cela il n'y a que 2 instructions et demie :

Affectation de *variable* `x = e` crée/modifie la variable `x`

Passage de paramètre un appel à `f([1, 2, 3])` où

```
def f(x):
    ...
```

crée une variable *locale* `x` initialisée à `[1, 2, 3]`.

`+=, *=, ...` Elles modifient parfois une variable, parfois un objet.

Remarque

Un programme comportant une variable `x` mais pas d'instruction `x = ...` ou `x += ..., ...` :

- ne peut changer la valeur de `x`
- change peut-être le contenu de l'objet désigné par `x`

Comment modifier un objet

- Instructions innombrables.
- Dépendent des objets.
- Quelques classiques :

  Méthodes associées aux tableaux Exemples :
  - `x.append(17)`
  - `x.pop()`,
  - `x.insert(17, 28)`

  Opérateurs
  - `x[i] = e`
  - `x[i:j] = e`
  - `x[i][j] = e`
  - `x[i, j] = e` (matrices Numpy)
  - et parfois `x += e` et `x *= e`

Attention : ces opérateurs n'affectent pas la *variable* `x` mais *l'objet* désigné par `x`.

Les deux catégories d'objets en Python

Non-modifiables

- Les entiers.
- Les chaînes de caractères.
- Les n-uplets: `(1, 2, 3, 'soleil')`.

Modifiables

- Les tableaux.
- Les dictionnaires.

Une évidence

Si `x` désigne un objet *non-modifiable*, celui-ci ne peut pas être *modifié*.

Le cas de `+=` et `*=`

L'instruction `x = x + e` signifie :

- créer un objet `x + e`
- puis modifier la variable `x` pour qu'elle désigne ce nouvel objet

L'instruction `x += e` :

- signifie «modifier l'objet désigné par `x` pour l'augmenter de `e`».
- n'a de sens que si `x` désigne un objet modifiable.
- dans le cas contraire, est un raccourci pour `x = x + e`.

Exercices

Que vaut `y` après chacune de ces séquences d'instructions ?

```
1  x = 24
2  y = x
3  x += 3
```

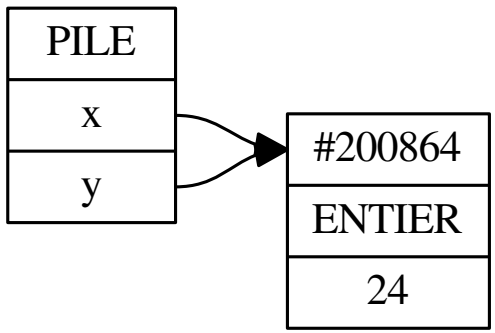
```
1  x = [1, 2, 3]
2  y = x
3  x += [7, 8]
```

```
1  x = 'hello'
2  y = x
3  x += ' world'
```

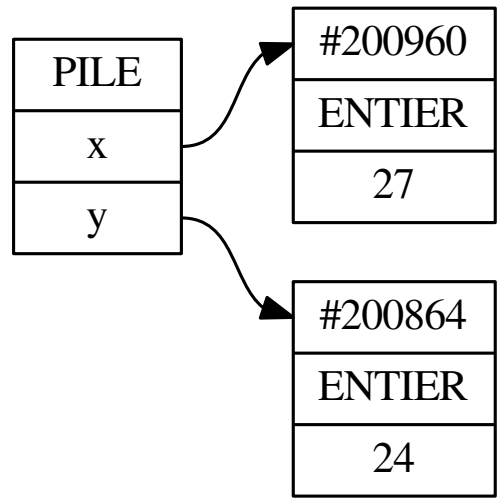
Solution exercice 1

```
1  x = 24
2  y = x
3  x += 3
```

Avant la ligne 3



Après



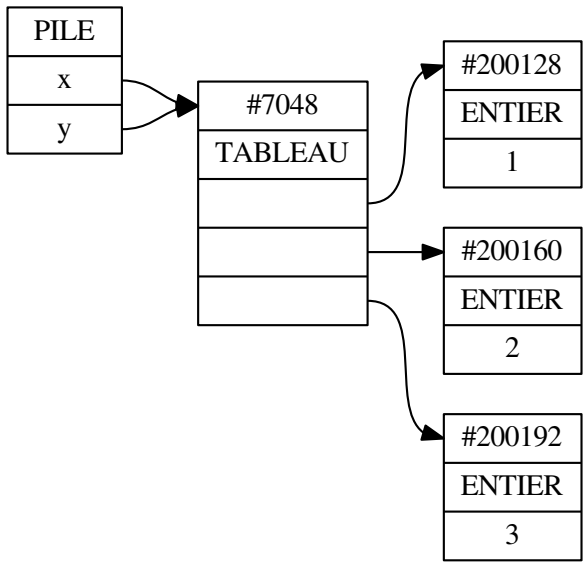
Remarque

`y` désignait un objet non-modifiable, qui ne pouvait donc être modifié.

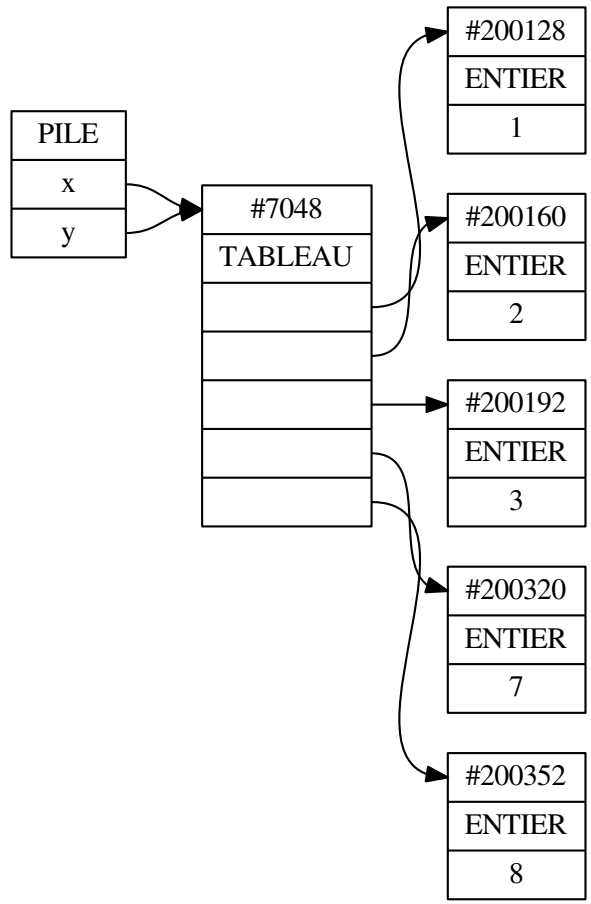
Solution exercice 2

```
1  x = [1, 2, 3]
2  y = x
3  x += [7, 8]
```

Avant la ligne 3



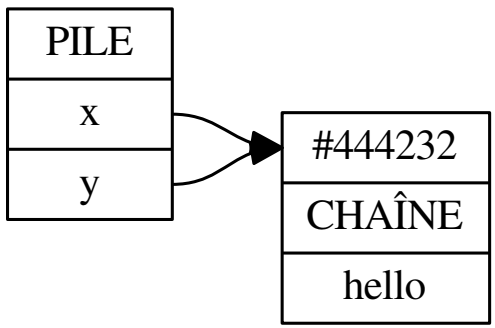
Après



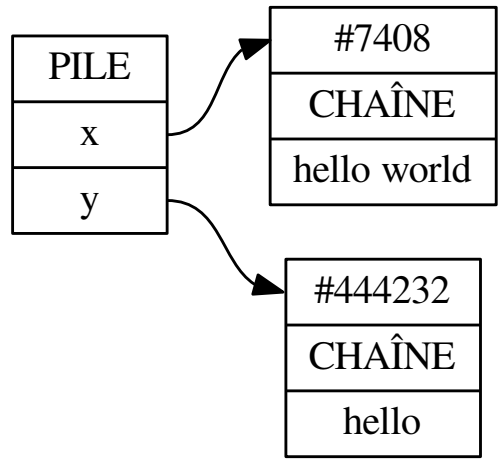
Solution exercice 3

```
1  x = 'hello'
2  y = x
3  x += ' world'
```

Avant la ligne 3



Après



Remarque

`y` désignait un objet non-modifiable

Cas des appels de fonctions

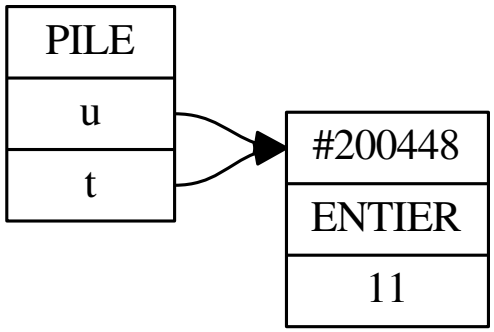
Lorsqu'on a une fonction `def f(x): ...` et qu'on appelle `f(e)`, Python:

- évalue l'expression `e` ;
- obtient ainsi une *valeur* (rappel : règle d'or) ;
- crée une nouvelle variable `x` sur la pile qu'il met à cette valeur ;
- exécute le corps de la fonction ;
- supprime la variable de la pile à la fin de la fonction.

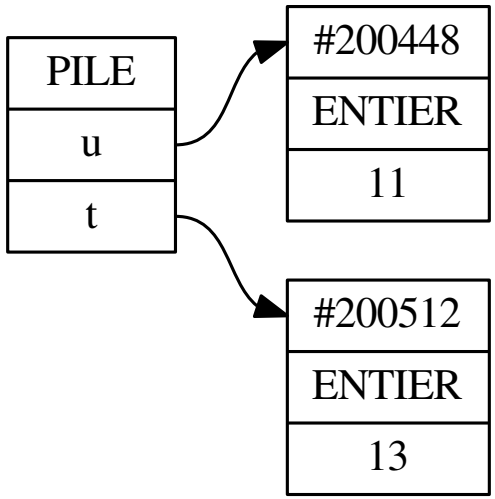
Exemple

```
1  def f(t):
2      t = t + 2
3      return t
4
5  u = 11
6  v = f(u)
```

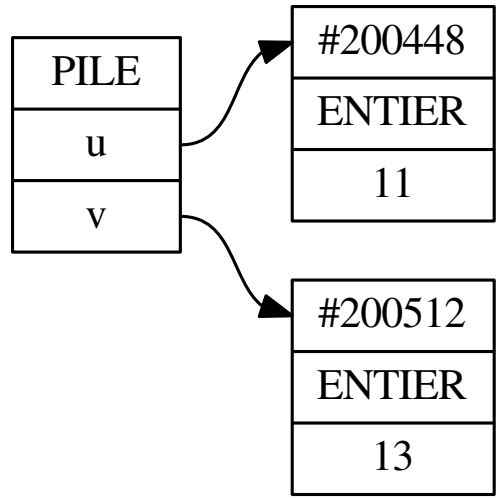
Avant ligne 2



Après ligne 2



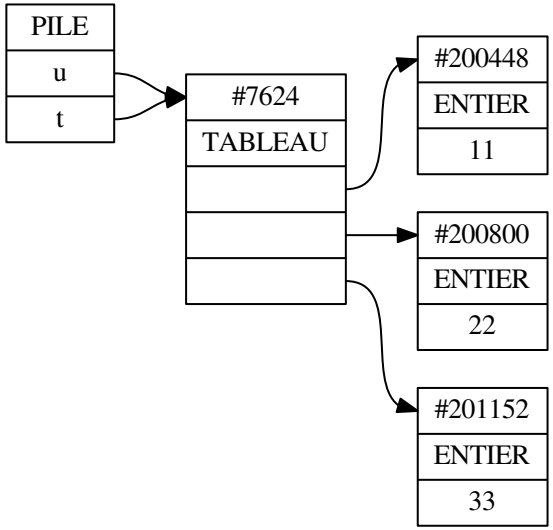
Après ligne 6



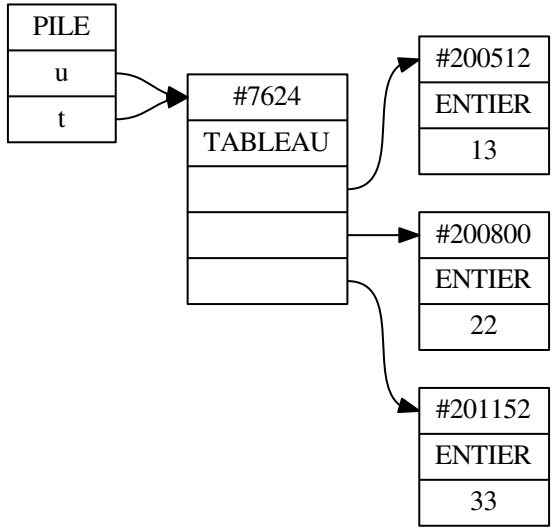
Exemple

```
1  def f(t):
2      t[0] += 2
3
4  u = [11, 22, 33]
5  f(u)
```

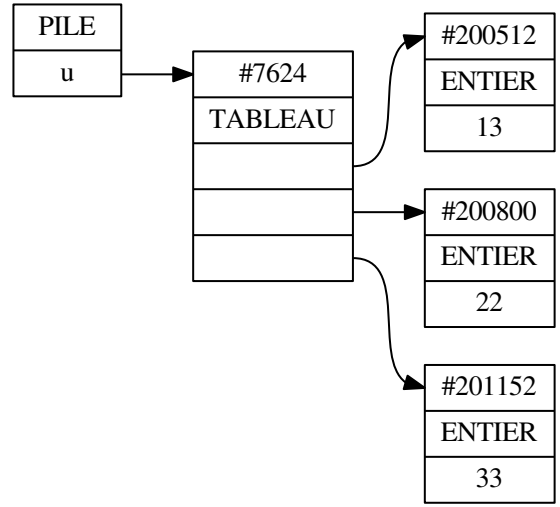
Avant ligne 2



Après ligne 2



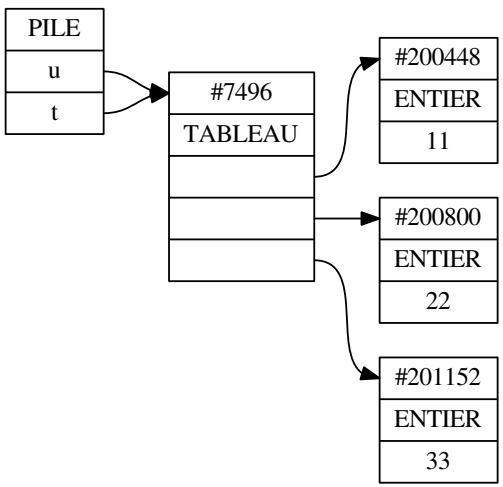
Après ligne 5



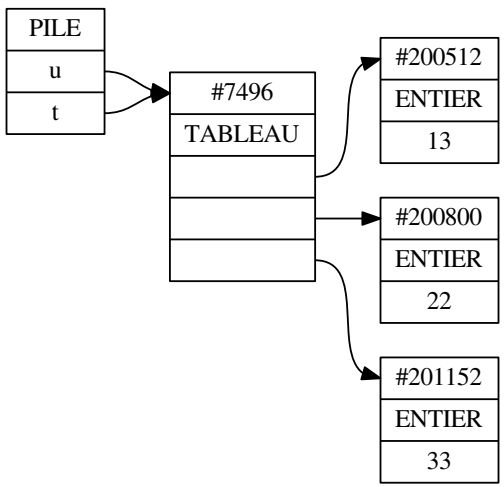
Exemple

```
1  def f(t):
2      t[0] += 2
3      return t # là, on cherche les ennuis...
4
5  u = [11, 22, 33]
6  u = f(u)
```

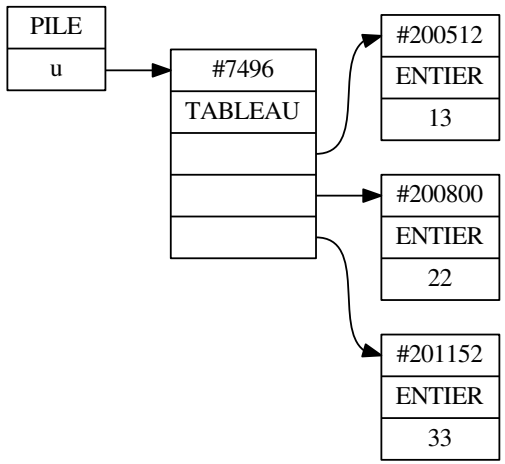
Avant ligne 2



Après ligne 2



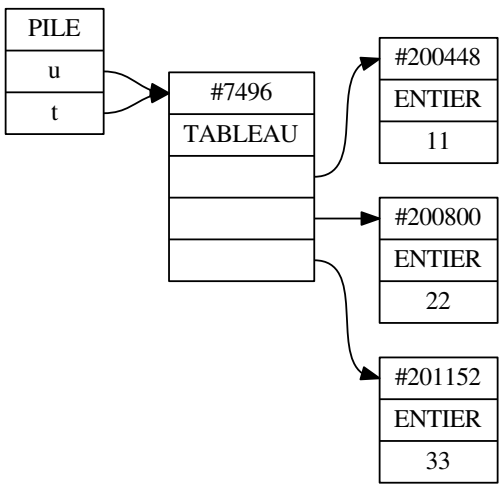
Après ligne 6



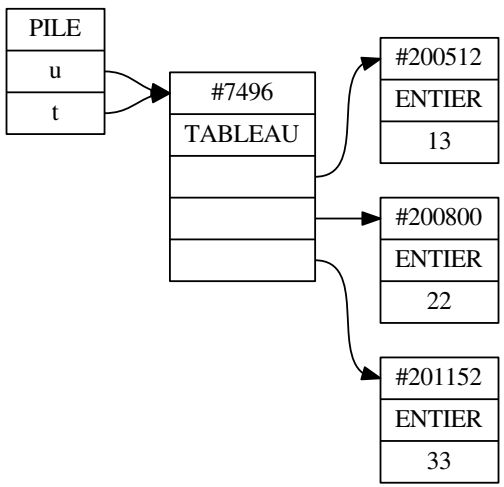
Exemple

```
1  def f(t):
2      t[0] += 2
3      return t # là, on cherche les ennuis...
4
5  u = [11, 22, 33]
6  v = f(u)
```

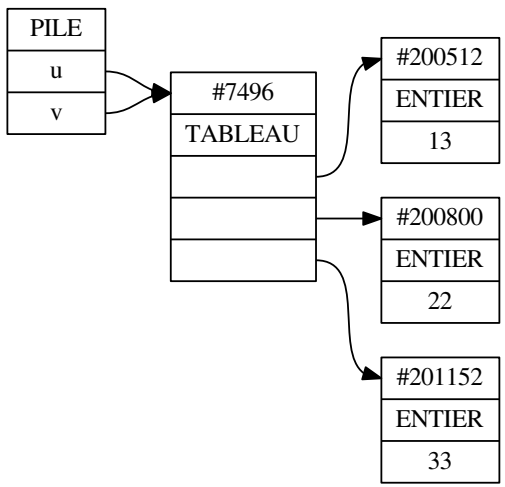
Avant ligne 2



Après ligne 2



Après ligne 6



Remarque sur l'écriture des fonctions

Renvoyer un argument (éventuellement modifié) est :

Inutile L'utilisateur n'a pas besoin du résultat (il l'a déjà)

Dangereux Cela laisse penser à l'utilisateur que :

- l'objet renvoyé est un nouvel objet
- la fonction ne modifie pas ses arguments

Autant que possible, on choisira donc d'écrire :

- soit une fonction qui ne renvoie rien.
- soit une fonction qui ne modifie pas ses arguments.

Exercice

Que vaut `u` après ces instructions ?

```
1  u = [1, 3, 2]
2  t = u
3  t.sort()
```

Et après celles-ci ?

```
1  u = [1, 3, 2]
2  t = u
3  t = sorted(t)
```

Solution : cas de la méthode `sort`

```
1  u = [1, 3, 2]
2  t = u
3  t.sort()
```

- L'exécution de la méthode `sort`
  - peut modifier l'objet désigné par `t`
  - ne peut pas modifier la variable `t`
- Donc après la ligne 3, `u` et `t` désignent encore le même objet.
- On peut penser que `t.sort()` trie le tableau `t`.

Donc `t` et `u` valent `[1, 2, 3]`.

Solution : cas de la fonction `sorted`

```
1  u = [1, 3, 2]
2  t = u
3  t = sorted(t)
```

- L'exécution de `sorted(t)` renvoie un résultat.
- Cela laisse penser qu'elle ne modifie pas `t`.

Donc `u` vaut `[1, 3, 2]` (et `t` `[1, 2, 3]`).

Exercice

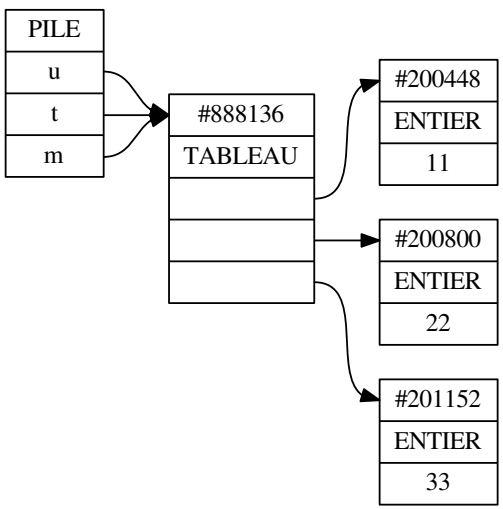
Que pensez-vous de la fonction suivante ?

```
1  def f(t):
2      m = t
3      m[0] += 2
4      return m
5
6  u = [11, 22, 33]
7  v = f(u)
```

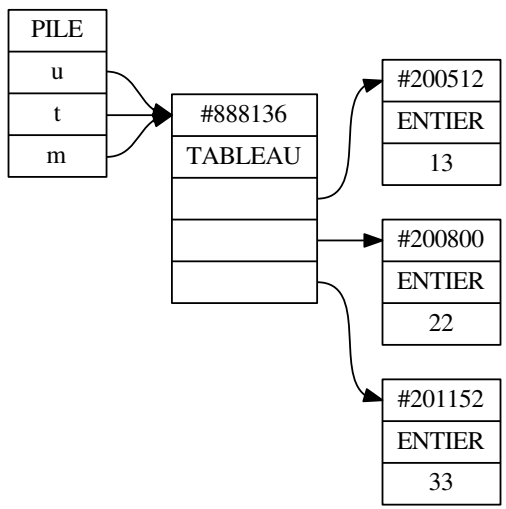
Solution de l'exercice

```
1  def f(t):
2      m = t
3      m[0] += 2
4      return m
5
6  u = [11, 22, 33]
7  v = f(u)
```

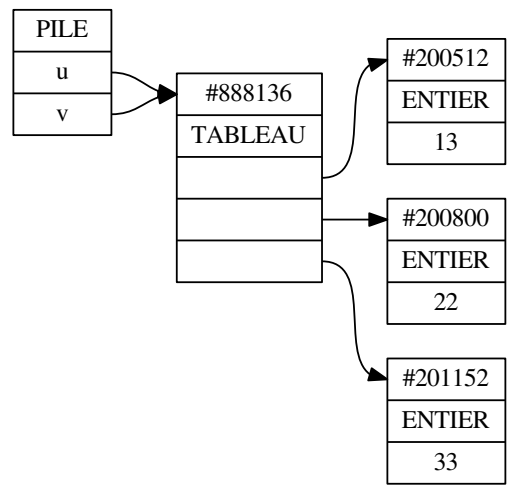
Avant ligne 3



Après ligne 3



Après ligne 7



Exercice

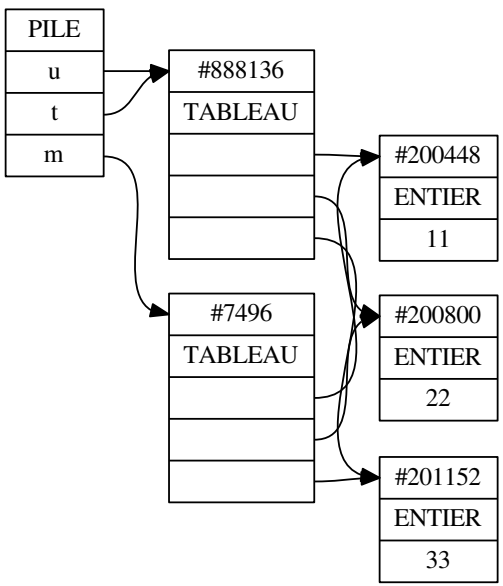
Récrire la fonction `f` pour qu'elle ne modifie pas `t` :

```
1  def f(t):
2      m = t
3      m[0] += 2
4      return m
5
6  u = [11, 22, 33]
7  v = f(u)
```

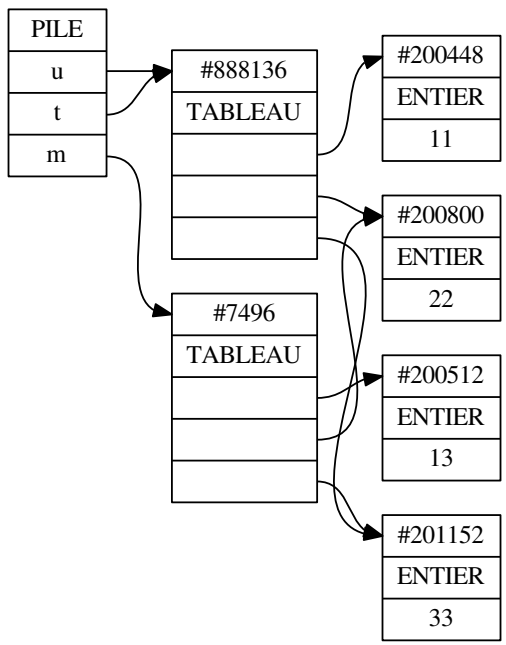
Solution

```
1  def f(t):
2      m = t.copy()
3      m[0] += 2
4      return m
5
6  u = [11, 22, 33]
7  v = f(u)
```

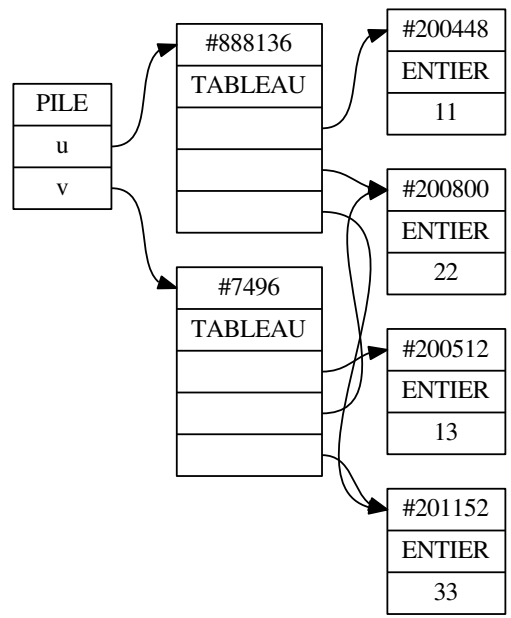
Avant ligne 3



Après ligne 3



Après ligne 7



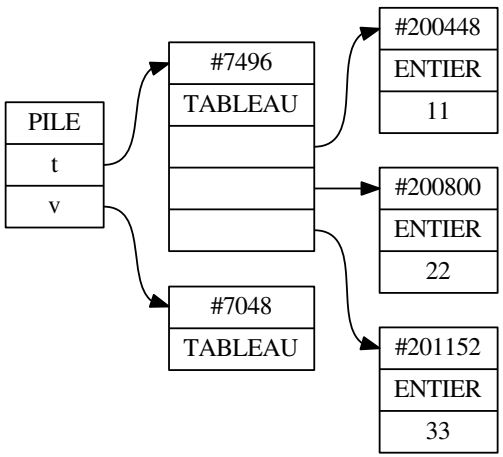
Copie de tableaux

`m = t.copy()` équivaut à `m = copie(t)`, avec la définition

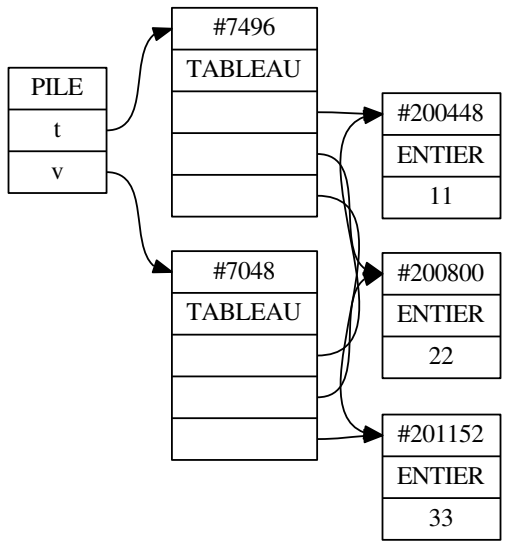
```
1  def copie(t):
2      v = []
3      for x in t:
4          v.append(x)
5      return v
```

```
6  t = [11, 22, 33]
7  m = copie(t)
```

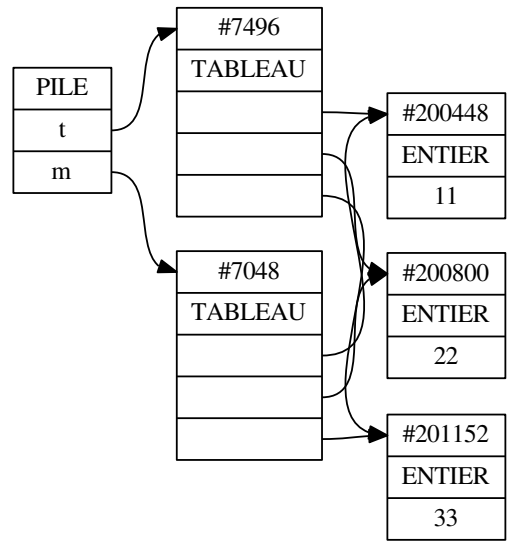
Avant ligne 3



Avant ligne 5



Après ligne 7



Remarques

- Différentes façons de copier `t` dans `u` :
  - `u = t.copy()`
  - `u = copie(t)`
  - `u = [x for x in t]`
  - `u = t[:]`
  - `u = list(t)`

  Façon canonique : la première
- Complexité (temps et mémoire) :

$$O(\texttt{len(t)})$$

Copier ou ne pas copier ?

Contexte : écriture d'une fonction qui modifie un tableau `t`

La copie
- Permet de garder les données originales de `t`
- Coûte cher en temps et mémoire ($O($`len(t)`$)$)

Question Ai-je *besoin* de garder la version précédente de `t` ?

Si oui Faire une copie `u` de `t`, modifier `u` et le renvoyer.

Si non Modifier `t` et ne rien renvoyer.

Structures de données profondes

- Exemple : tableaux de tableaux.
- Les principes précédents restent valides.
- Réfléchir en ayant en tête :

  La règle d'or Une valeur est juste l'adresse d'un objet.
  La distinction fondamentale Modifier une variable/un objet.

Exercice : copie superficielle

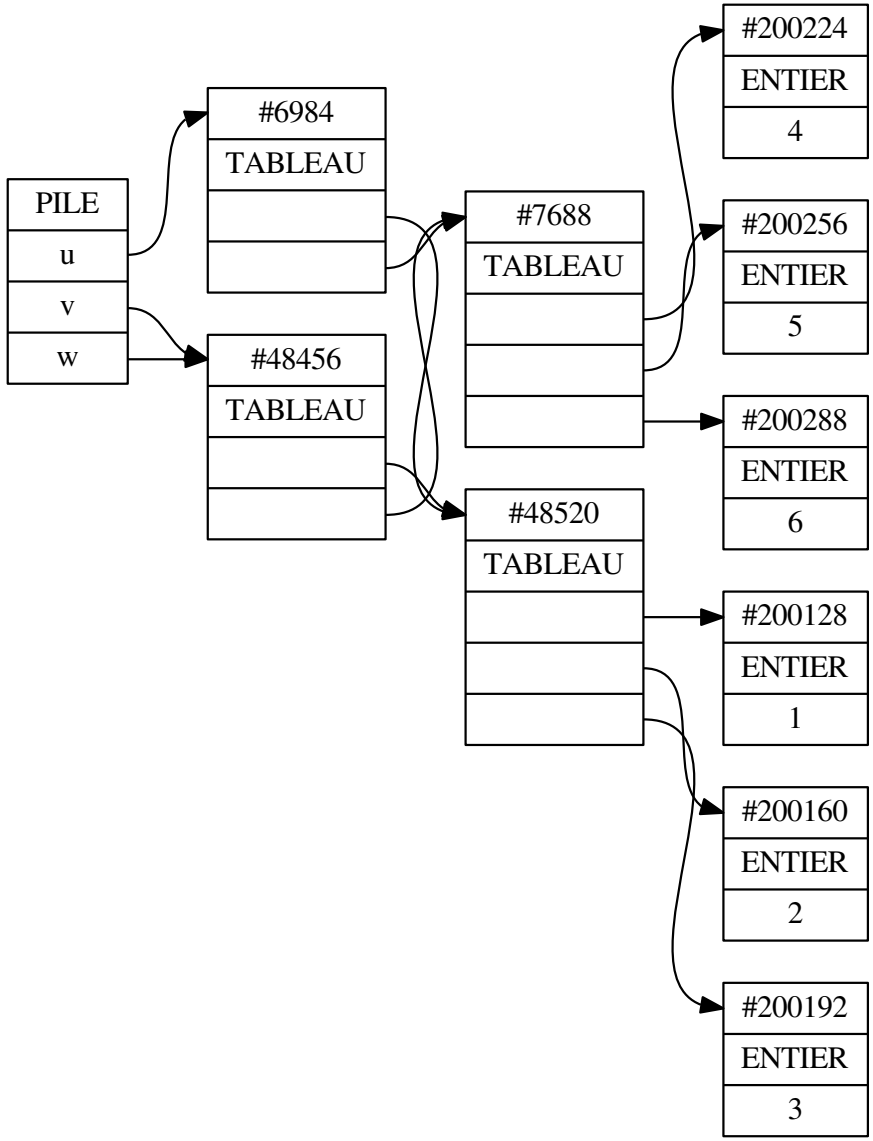
Que valent `u`, `v` et `w` après ces instructions ?

```
1  u = [[1, 2, 3], [4, 5, 6]]
2  v = u.copy()
3  w = v
4  u[0] = [7, 8, 9]
5  u[1][0] = 17
```

Solution

Après les 3 premières lignes :

```
1  u = [[1, 2, 3], [4, 5, 6]]
2  v = u.copy()
3  w = v
```



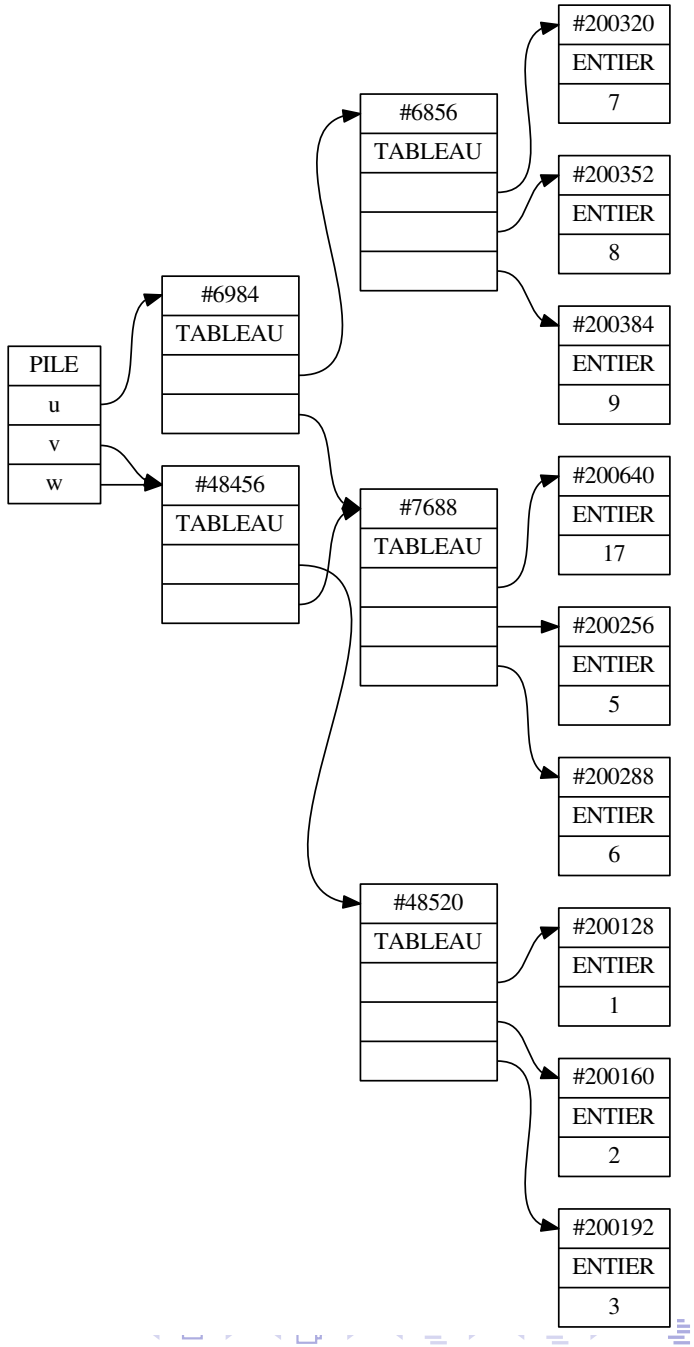
Solution (suite)

Et après les deux dernières :

```
4  u[0] = [7, 8, 9]
5  u[1][0] = 17
```

Remarque

- `u` et `v` ne sont pas les mêmes objets
- mais ils partagent leurs éléments
- en particulier `u[1]` et `v[1]` sont le *même* tableau...



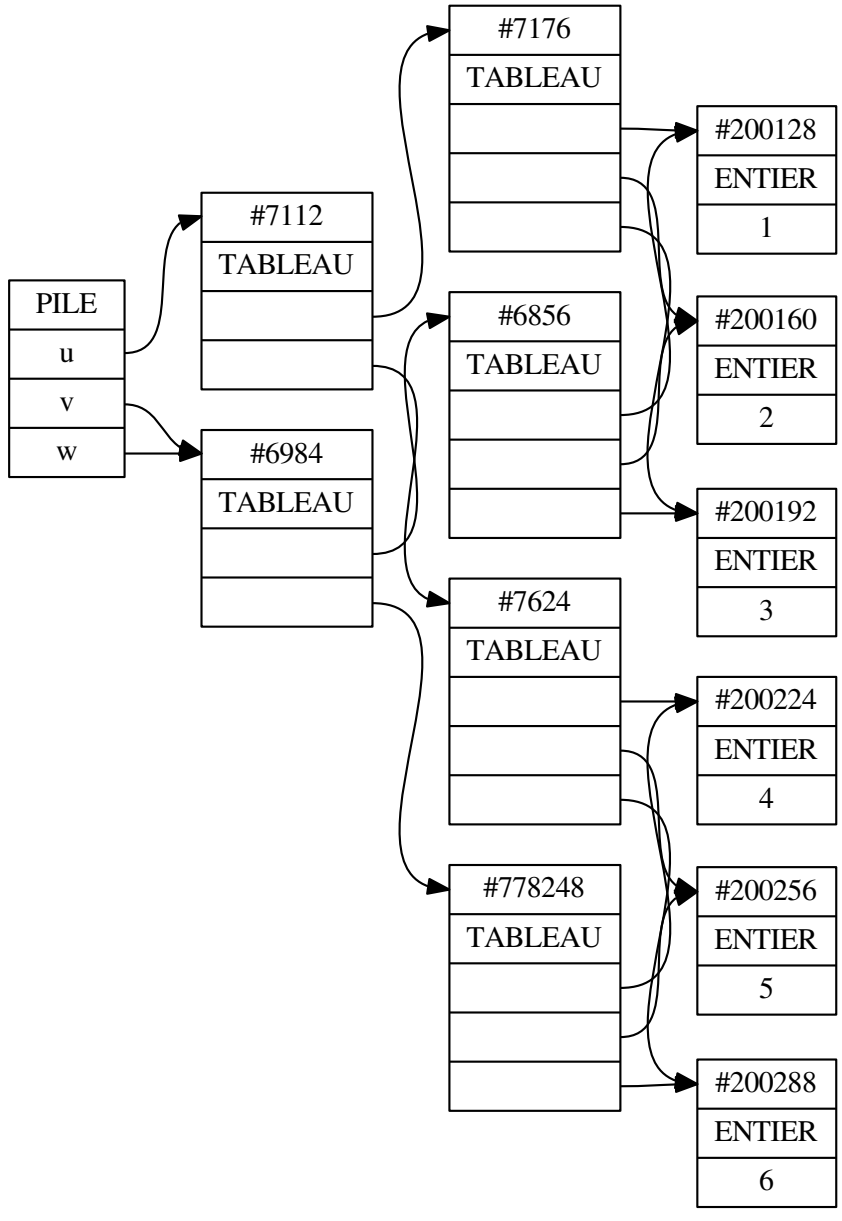
Copie profonde

La fonction `deepcopy` du module `copy` brise le partage aussi profondément que nécessaire :

```
1  u = [[1, 2, 3], [4, 5, 6]]
2  v = deepcopy(u)
3  w = v
```

Remarque

- Coûte évidemment cher (temps et espace)
- À utiliser avec discernement



Quelques opérations classiques sur les tableaux Python

Répétition d'un tableau

`u * n` avec `u` tableau équivaut à `tableau(u, n)` où

```
1  def tableau(u, n):
2      t = []
3      for i in range(n):
4          t += u
5      return t
```

Exercice

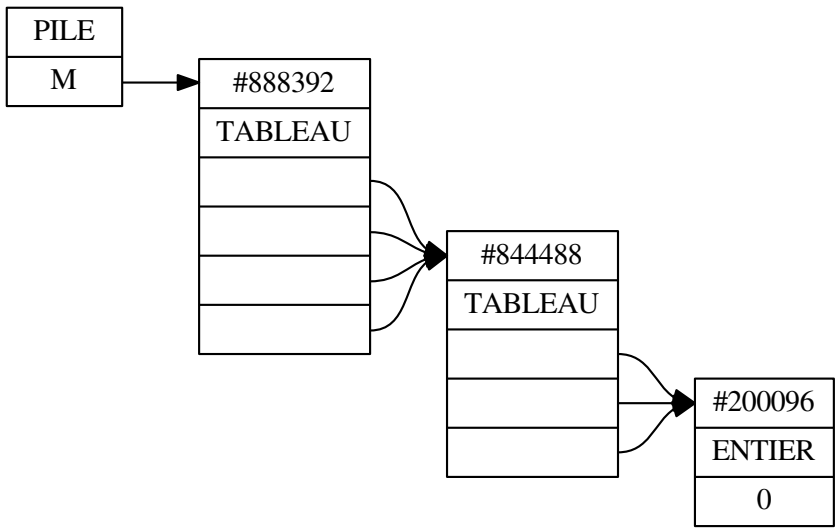
Que vaut `M` après les instructions

`M = [[0] * 3] * 4 ; M[0][1] = 17` ?

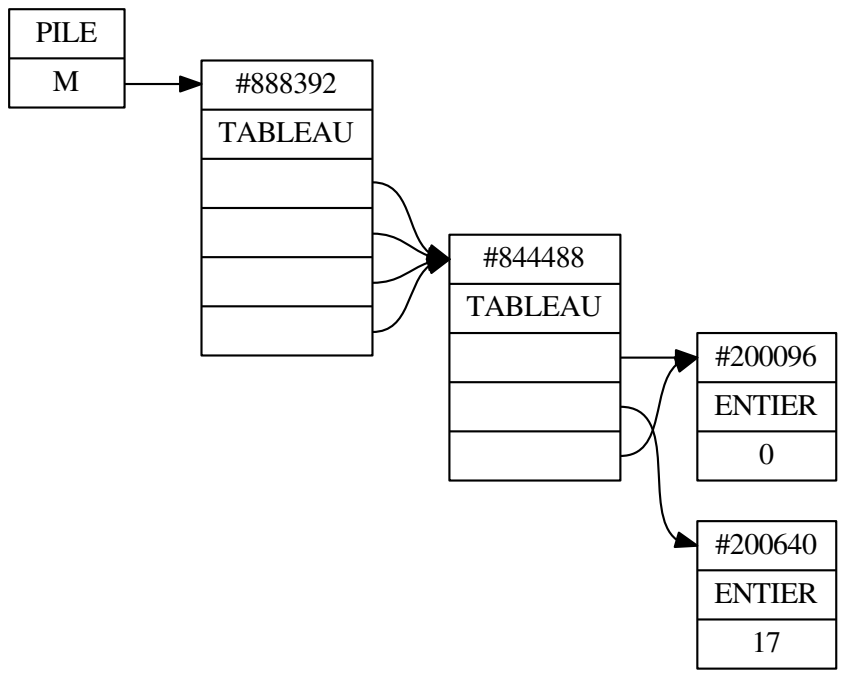
Solution

```
1  M = [[0] * 3] * 4
2  M[0][1] = 17
```

Entre les deux lignes



Après la ligne 2



Slicing

Sur les tableaux Python `t[i:j]` équivaut à `slice(t, i, j)` où

```
1  def slice(t, i, j):
2      u = []
3      for k in range(i, j):
4          u.append(t[k])
5      return u
```

En particulier

- `t[i:j]` est un nouveau tableau, différent de `t`.
- ses éléments sont les mêmes que ceux de `t` d'indices `i` (inclus) à `j` (exclu).
- `t[:]` (abréviation de `t[0:len(t)]`) est une copie de `t`.

Exemple

- Après l'exécution de

```
t = [0, 11, 22, 33, 44, 55]
v = t[2:4]
v[1] = 17
```

  - v vaut [**22, 17**]
  - t vaut [**0, 11, 22, 33, 44, 55**]

- Cas des tableaux Numpy : après l'exécution de

```
import numpy as np
t = np.array([0, 11, 22, 33, 44, 55])
v = t[2:4]
v[1] = 17
```

  - v vaut array([**22, 17**])
  - t vaut array([**0, 11, 22, 17, 44, 55**]).

Tableaux numpy

Avec un tableau numpy `t`, `v = t[i:j]` construit une vue sur `t`.

- Cette *vue* ressemble à un tableau : opérations usuelles.
- `v[k]` renvoie la valeur `t[h]` où `h` est calculé à partir de `k` et des paramètres de la vue (ici `h` vaut `k+i`).
- `v[k] = v` a le même effet que `t[h] = v`:

Pour une matrice `m` (tableau bidimensionnel) :

- De même, `m[i1:j1, i2:j2]` est une vue sur la matrice `m`.

Exercice

Écrire une fonction `echange(t, k)` échangeant les `k` premiers éléments de `t` avec les `k` derniers, en supposant

1. `t` est un tableau Python.
2. `t` est un tableau Numpy.

Conclusion/résumé

La règle d'or Une valeur est juste l'adresse d'un objet.

La distinction fondamentale Modifier une variable/un objet.

Deux catégories d'objets Modifiables / non-modifiables.

Copyleft