

LES TABLEAUX

Lundi 3 novembre

Option Informatique
Ecole Alsacienne

PLAN

1. Les tableaux en Python
2. Références et portée des variables
3. Fin du TD "A la découverte de Python"
4. Mini-TD sur les tableaux

LES TABLEAUX EN PYTHON

DÉFINIR UN TABLEAU

- Case par case :
 - Syntaxe : `nom = [ e0 , e1 , ... , en ]`
 - Exemple : `t0 = [ 3 , 1 , 6 ]`
- Par sa taille et sa valeur de base :
 - Syntaxe : `nom = [valeur] * taille`
 - Exemple : `t1 = [0] * 1000`

FONCTIONS DE BASE

- Longueur du tableau :

```
longueur = len(tableau)
```

- Accéder à un élément : `t[i]`

- Modifier un élément : `t[i] = val`

- Que va afficher Python ?

```
v = [ 2 , 4 , 6 ]
```

```
print(v[3])
```

`IndexError: list index out of range`

TABLEAUX ET LISTES EN PYTHON

- Les tableaux en Python permettent un certain nombre de choses qui ne sont pas autorisées dans d'autres langages
 - Modifier la taille d'un tableau après sa déclaration
 - Ajouter un élément au début ou à la fin
 - Supprimer un élément au milieu du tableau
- C'est en fait la même structure qui est utilisée pour représenter les tableaux et les listes
- Le type correspondant en Python est d'ailleurs `list`

TABLEAUX ET TYPES EN PYTHON

- Python est plus permissif qu'OCaml : il est possible d'avoir plusieurs types d'objets dans le même tableau

- Exemple

```
t_mixte = [1, "hello", True]
print(t_mixte)
print(type(t_mixte))
```

```
[1, 'hello', True]
<class 'list'>
```

- **Remarque générale** : Ce n'est pas parce que Python permet beaucoup de choses qu'on doit en oublier la rigueur.

RÉFÉRENCES ET PORTÉE DES VARIABLES

COPIER UN TABLEAU

- **Question** : Que va afficher le programme suivant ?

```
t0 = [1, 2, 3]
```

```
t1 = t0
```

```
t1[0] = 4
```

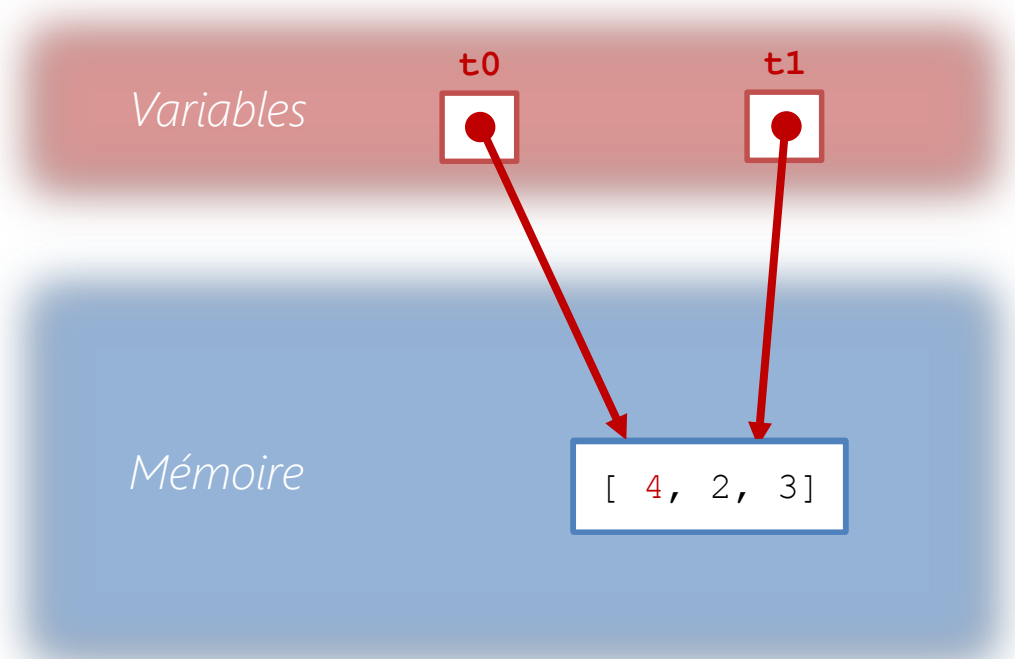
```
print(t0[0])
```

- **Réponse** : 4
- **Explication (version courte)** : Les deux variables t0 et t1 font référence au même tableau !

COPIER UN TABLEAU

```
t0 = [1, 2, 3]
t1 = t0
t1[0] = 4
print(t0[0])
```

- Explication (version longue) :
 - La première instruction effectue les opérations suivantes
 - Allocation de l'espace mémoire pour stocker le tableau
 - Stockage des valeurs du tableau
 - Création d'une variable faisant référence à ce tableau
 - La deuxième instruction crée une autre variable, qui "pointe" sur le même tableau
 - La troisième instruction modifie la première case du tableau, et donc les deux variables
 - La quatrième instruction lit le contenu en mémoire



COPIER UN TABLEAU – TABLEAUX INDÉPENDANTS

- Pour copier un tableau en Python et obtenir deux tableaux indépendants, on utilise la fonction `list` :

```
t0 = [1, 2, 3]
```

```
t1 = list(t0)
```

```
t1[0] = 4
```

```
print(t0[0])
```

- Résultat : 1
- Remarque : On utilise ici ce qu'on appelle un "constructeur de copie"

COPIER UN TABLEAU – TABLEAUX INDÉPENDANTS

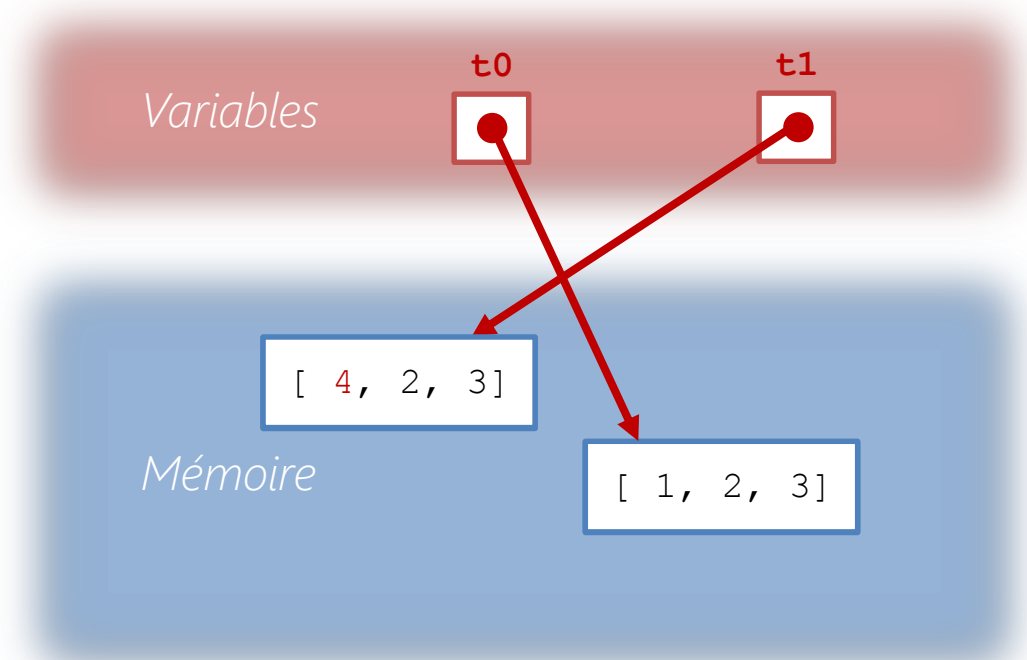
- Grâce à l'appel à la fonction `list`, un deuxième emplacement est alloué dans la mémoire :

```
t0 = [1, 2, 3]
```

```
t1 = list(t0)
```

```
t1[0] = 4
```

```
print(t0[0])
```



TESTS D'ÉGALITÉ

- Python propose plusieurs opérateurs pour tester l'égalité entre deux variables
 - Le double égal permet de tester si les valeurs sont identiques
 - Le mot-clef `is` permet de tester si les références sont identiques

- Exemple

```
t0 = [1, 2, 3]
t1 = [1, 2, 3]
print(t0 == t1)
print(t0 is t1)
```

True

False

PORTÉE DES VARIABLES

- La **portée des variables** est une notion importante en programmation.
- Il s'agit de répondre à la question "*Quand puis-je accéder à une variable x ?*"
- La réponse est moins triviale qu'on ne le croit

VARIABLE PASSÉE COMME ARGUMENT D'UNE FONCTION

- On a vu qu'il est possible de passer une variable comme argument d'une fonction

```
def afficher_variable(v):  
    print("La variable passee en argument vaut", v)
```

```
y = 3  
afficher_variable(y)
```

La variable passee en argument vaut 3

- On peut donc récupérer la valeur d'une variable définie à l'extérieur d'une fonction, à condition de la passer comme argument.

VARIABLE CRÉÉE DANS UNE FONCTION

- Il est également possible de créer des variables à l'intérieur d'une fonction :

```
def afficher_variable_et_creer_double(v):  
    print("La variable passee en argument vaut", v)  
    double = v*2  
    print("Le double de cette variable vaut", double)
```

```
z = 4  
afficher_variable_et_creer_double(z)  
print(double)
```

La variable passee en argument vaut 4

Le double de cette variable vaut 8

print(double)

NameError: name 'double' is not defined

- Les variables définies à l'intérieur d'une fonction ne sont disponibles que dans cette fonction !
- On parle de l' "espace local" d'une fonction

VARIABLE DÉCLARÉE À L'EXTÉRIEUR D'UNE FONCTION

- De façon plus surprenante, il est possible d'accéder à la valeur d'une variable dans une fonction, même passer cette variable come argument

```
def afficher_variable_w():  
    print("La variable w vaut", w)
```

```
w = 5  
afficher_variable_w()
```

La variable w vaut 5

- Dans la mesure du possible, on évitera ce type de construction, car elles sont sources d'erreurs.

MODIFIER UNE VARIABLE DANS UNE FONCTION

- Il est possible de modifier une variable passée en argument d'une fonction :

```
def doubler_variable(x):  
    print("La variable passee en argument vaut", x)  
    x = x*2  
    print("A la fin de la fonction, elle vaut", x)  
  
x = 6  
doubler_variable(x)  
print("Une fonction la fonction terminee, la variable vaut ", x)
```

```
La variable passee en argument vaut 6  
A la fin de la fonction, elle vaut 12  
Une fonction la fonction terminee, la variable vaut  6
```

- Cependant, ces modifications ne sont valables qu'à l'intérieur de la fonction : dès qu'on sort de la fonction, la variable retrouve son état précédent !
- Une copie de la variable passée en argument est en fait créée dans l'"espace local" de la fonction
 - C'est cette copie qu'on manipule tant qu'on est dans la fonction
 - Cette copie est détruite lorsqu'on sort de la fonction

MODIFIER UNE VARIABLE DANS UNE FONCTION

- Si une variable est définie à l'extérieur d'une fonction, il n'est pas possible de la modifier dans la fonction

```
def doubler_variable_y():  
    print("La variable y vaut", y)  
    y = y*2  
    print("A la fin de la fonction, elle vaut", y)
```

```
y = 7  
doubler_variable_y()  
print("Une fonction la fonction terminee, la variable y vaut ", y)
```

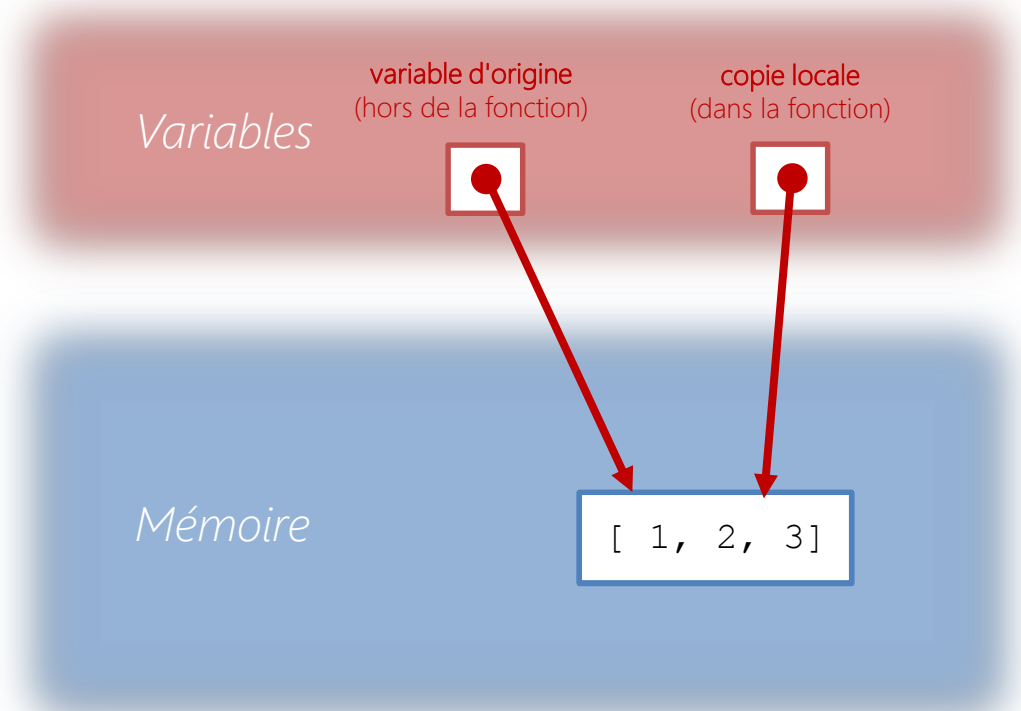
```
print("La variable y vaut", y)
```

```
UnboundLocalError: local variable 'y' referenced before assignment
```

- Pour réaliser ce type d'opérations, il faudrait utiliser des variables globales (que nous essayerons d'éviter)

MODIFIER UN TABLEAU DANS UNE FONCTION

- Comme on l'a vu un peu plus tôt, une variable de type `list` est en fait un pointeur sur un emplacement mémoire.
- **Conséquence 1** : La copie dans l'espace local de la fonction pointe sur le même espace mémoire que la variable d'origine
- **Conséquence 2** : Il est possible de modifier le tableau depuis l'intérieur de la fonction



MODIFIER UN TABLEAU DANS UNE FONCTION

- Plus exactement, il est possible de modifier le tableau, mais pas de le remplacer intégralement

```
def modifier_tab(t):
```

```
    t[0] = 4
```

```
    t[1] = 5
```

```
    t[2] = 6
```

```
t = [1,2,3]
```

```
modifier_tab(t)
```

```
print(t)
```

```
[4, 5, 6]
```

```
def remplacer_tab(t):
```

```
    t = [4, 5, 6]
```

```
t = [1,2,3]
```

```
remplacer_tab(t)
```

```
print(t)
```

```
[1, 2, 3]
```

RETOUR AUX TD