

SEMESTRE 1 / COURS 3 - LES LISTES

ITC MPSI & PCSI – Année 2025-2026



1. Notion de liste
2. Opérations élémentaires sur une liste
3. Opérations plus avancées sur les listes
4. Copie d'une liste

NOTION DE LISTE

QU'EST CE QU'UNE LISTE ?

- C'est une séquence (suite) d'objets de types éventuellement différents.

QU'EST CE QU'UNE LISTE ?

- C'est une séquence (suite) d'objets de types éventuellement différents.
- Chaque élément d'une liste possède un indice.

QU'EST CE QU'UNE LISTE ?

- C'est une séquence (suite) d'objets de types éventuellement différents.
- Chaque élément d'une liste possède un indice.
- Le premier élément porte l'indice 0, le suivant a l'indice 1, etc.

QU'EST CE QU'UNE LISTE ?

- C'est une séquence (suite) d'objets de types éventuellement différents.
- Chaque élément d'une liste possède un indice.
- Le premier élément porte l'indice 0, le suivant a l'indice 1, etc.
- Une liste s'écrit entre deux crochets et ses éléments sont séparés d'une virgule.

- En **extension** en écrivant tous les éléments.

- En **extension** en écrivant tous les éléments.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> type(liste)  
<class 'list'>
```

- En **extension** en écrivant tous les éléments.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> type(liste)  
<class 'list'>
```

- En **compréhension** en donnant une « formule » construisant la liste.

- En **extension** en écrivant tous les éléments.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> type(liste)  
<class 'list'>
```

- En **compréhension** en donnant une « formule » construisant la liste.

```
>>> liste = [i**2 for i in range(10)] # carrés \  
↪ des entiers <10  
>>> liste  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

CRÉATION D'UNE LISTE

- En **extension** en écrivant tous les éléments.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> type(liste)  
<class 'list'>
```

- En **compréhension** en donnant une « formule » construisant la liste.

```
>>> liste = [i**2 for i in range(10)] # carrés \  
↪ des entiers <10  
>>> liste  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Créer une liste vide : `L = []` ou `L = list()`.

- La fonction `list(<objet>)` peut être utilisée pour créer une liste d'éléments à partir d'un objet itérable.

- La fonction `list(<objet>)` peut être utilisée pour créer une liste d'éléments à partir d'un objet itérable.

```
>>> L1 = list('Bonjour')
>>> L1
['B', 'o', 'n', 'j', 'o', 'u', 'r']
>>> L2 = list(range(1,10,2))
>>> L2
[1, 3, 5, 7, 9]
```

OPÉRATIONS ÉLÉMENTAIRES SUR UNE LISTE

La fonction `len(<liste>)` renvoie le nombre d'objets dans la liste.

La fonction `len(<liste>)` renvoie le nombre d'objets dans la liste.

```
>>> liste = [1, 2.5 , [1, 2], 'A']  
>>> len(liste)  
4
```

- Le test d'inégalité entre listes n'est pas très employé (ordre lexicographique).

- Le test d'inégalité entre listes n'est pas très employé (ordre lexicographique).
- Le test d'égalité (==) est courant, il donne **True** si les deux listes ont les mêmes éléments dans le même ordre, **False** sinon.

- Le test d'inégalité entre listes n'est pas très employé (ordre lexicographique).
- Le test d'égalité (==) est courant, il donne **True** si les deux listes ont les mêmes éléments dans le même ordre, **False** sinon.

COMPARAISON

```
>>> [1, 5] < [3, 4]  # ordre lexicographique
True
>>> [1, 2, 8] < [3]
True
>>> [1, 2] < ['3', 4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '<' not supported between instances of \
↳ 'int' and 'str'
>>> [1, 2] == ['3', 4]
False
>>> L = [1,2]
>>> L == [1,2]
True
```

Pour savoir si une valeur figure dans une liste on utilise l'opérateur **in**.

Pour savoir si une valeur figure dans une liste on utilise l'opérateur **in**.

```
>>> liste = [1, 2.5, 'A', [1, 2]]
```

```
>>> 1 in liste
```

```
True
```

```
>>> 2 in liste
```

```
False
```

```
>>> [1, 2] in liste
```

```
True
```

- La concaténation (+) permet de mettre deux listes bout à bout pour en faire une nouvelle.

CONCATÉINATION, RÉPÉTITION

- La concaténation (+) permet de mettre deux listes bout à bout pour en faire une nouvelle.
- La répétition (*) permet de créer une liste en répétant un certain nombre de fois un ou plusieurs éléments.

CONCATÉINATION, RÉPÉTITION

- La concaténation (+) permet de mettre deux listes bout à bout pour en faire une nouvelle.
- La répétition (*) permet de créer une liste en répétant un certain nombre de fois un ou plusieurs éléments.

```
>>> liste1 = ['lundi', 'mardi', 'mercredi']
>>> liste2 = ['jeudi', 'vendredi']
>>> liste3 = liste1 + liste2  # concaténation
>>> liste3
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> liste4 = [0] * 7  # répétition
>>> liste4
[0, 0, 0, 0, 0, 0, 0]
```

- Les éléments d'une liste sont indicés à partir de 0.

- Les éléments d'une liste sont indicés à partir de 0.
- L'élément ℓ_i d'indice i d'une liste L s'écrit $L[i]$.

- Les éléments d'une liste sont indicés à partir de 0.
- L'élément ℓ_i d'indice i d'une liste L s'écrit $L[i]$.
- On peut représenter la liste L , contenant n éléments, par :

$$L = [\ell_0, \ell_1, \dots, \ell_{n-1}].$$

- Les éléments d'une liste sont indicés à partir de 0.
- L'élément ℓ_i d'indice i d'une liste L s'écrit $L[i]$.
- On peut représenter la liste L , contenant n éléments, par :

$$L = [\ell_0, \ell_1, \dots, \ell_{n-1}].$$

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> liste[1]  
2.5  
>>> liste[2]  
[1, 2]
```

Il est possible d'utiliser des indices négatifs.

Il est possible d'utiliser des indices négatifs.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> liste[-1] # accès au dernier élément  
'A'  
>>> liste[-2] # l'avant-dernier  
[1, 2]
```


- Les listes sont des objets **mutables** : on peut modifier les éléments individuellement.

- Les listes sont des objets **mutables** : on peut modifier les éléments individuellement.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> liste[1] = 3  
>>> liste  
[1, 3, [1, 2], 'A']
```

- Les listes sont des objets **mutables** : on peut modifier les éléments individuellement.

```
>>> liste = [1, 2.5, [1, 2], 'A']  
>>> liste[1] = 3  
>>> liste  
[1, 3, [1, 2], 'A']
```

- Si un élément d'une liste est une liste, on peut modifier de même ses éléments.

- Les listes sont des objets **mutables** : on peut modifier les éléments individuellement.

```
>>> liste = [1, 2.5 , [1, 2], 'A']  
>>> liste[1] = 3  
>>> liste  
[1, 3, [1, 2], 'A']
```

- Si un élément d'une liste est une liste, on peut modifier de même ses éléments.

```
>>> liste = [1, 2.5 , [1, 2], 'A']  
>>> liste[2][0] = 'un'  
>>> liste  
[1, 2.5, ['un', 2], 'A']
```

- Parcours par **indice** avec une boucle **for** (la variable représente un indice).

- Parcours par **indice** avec une boucle **for** (la variable représente un indice).

```
>>> liste = [1, 2.5, 'A']  
>>> for i in range(len(liste)): print(i, liste[i])  
...  
0 1  
1 2.5  
2 A
```

- Parcours par élément avec une boucle **for** (la variable représente un élément).

- Parcours par élément avec une boucle **for** (la variable représente un élément).

```
>>> liste = [1, 2.5, 'A']  
>>> for el in liste: print(el)  
...  
1  
2.5  
A
```


- On peut également utiliser `enumerate`.

- On peut également utiliser `enumerate`.

```
>>> liste = [1, 2.5, 'A']
>>> for (i,el) in enumerate(liste): print(i,el)  # i \
↪ est l'indice
...
0 1
1 2.5
2 A
```

OPÉRATIONS PLUS AVANCÉES SUR LES LISTES

- Pour extraire une sous-liste d'une liste L :

`L[dep:fin:pas]`

On extrait les éléments de l'indice **dep (inclus)** à l'indice **fin (exclu)**, de **pas** en **pas**.

- Pour extraire une sous-liste d'une liste L :

$L[\text{dep}:\text{fin}:\text{pas}]$

On extrait les éléments de l'indice **dep (inclus)** à l'indice **fin (exclu)**, de **pas** en **pas**.

- La partie $[\text{:pas}]$ est facultative et si elle est omise, le pas vaut 1 par défaut

- Pour extraire une sous-liste d'une liste L :

`L[dep:fin:pas]`

On extrait les éléments de l'indice **dep** (**inclus**) à l'indice **fin** (**exclu**), de **pas** en **pas**.

- La partie `[ :pas ]` est facultative et si elle est omise, le pas vaut 1 par défaut
- Si l'argument **dep** est omis, alors c'est 0 par défaut (début de la liste).

- Pour extraire une sous-liste d'une liste L :

$L[\text{dep}:\text{fin}:\text{pas}]$

On extrait les éléments de l'indice **dep** (**inclus**) à l'indice **fin** (**exclu**), de **pas** en **pas**.

- La partie $[\text{:pas}]$ est facultative et si elle est omise, le pas vaut 1 par défaut
- Si l'argument **dep** est omis, alors c'est 0 par défaut (début de la liste).
- Si l'argument **fin** est omis, alors c'est jusqu'à la fin de la liste.

- Pour extraire une sous-liste d'une liste L :

$L[\text{dep}:\text{fin}:\text{pas}]$

On extrait les éléments de l'indice **dep** (**inclus**) à l'indice **fin** (**exclu**), de **pas** en **pas**.

- La partie $[\text{:pas}]$ est facultative et si elle est omise, le pas vaut 1 par défaut
- Si l'argument **dep** est omis, alors c'est 0 par défaut (début de la liste).
- Si l'argument **fin** est omis, alors c'est jusqu'à la fin de la liste.
- Si le **pas** est négatif il faut raisonner de droite à gauche.

SLICING – EXEMPLE

```
>>> L = [1, 2.5 , [1, 2], 'A','B'] # définition de \
↪ la liste
>>> L[1:4:2] # de L[1] à L[3] par pas de 2
[2.5, 'A']
>>> L[:3] # les 3 premiers
[1, 2.5, [1, 2]]
>>> L[1:] # tout à partir du deuxième
[2.5, [1, 2], 'A', 'B']
```

SLICING – EXEMPLE

```
>>> liste = [1, 2.5 , [1, 2], 'A', 'B']
>>> L[1::2]  # de 2 en 2 à partir du deuxième
[2.5, 'A']
>>> L[-3:]   # les 3 derniers
[[1, 2], 'A', 'B']
>>> L[::]    # toute la liste
[1, 2.5, [1, 2], 'A', 'B']
>>> L[3:0:-1] # du 4ième au 2ième en sens inverse
['A', [1, 2], 2.5]
```

SLICING – EXEMPLE

- Si `pas` est positif, alors il faut que `dep` soit strictement inférieur à `fin`, sinon la liste renvoyée est vide,

SLICING – EXEMPLE

- Si **pas** est positif, alors il faut que **dep** soit strictement inférieur à **fin**, sinon la liste renvoyée est vide,
- si **pas** est négatif, alors il faut que **dep** soit strictement supérieur à **fin**, sinon la liste renvoyée est vide.

SLICING – EXAMPLE

- Si **pas** est positif, alors il faut que **dep** soit strictement inférieur à **fin**, sinon la liste renvoyée est vide,
- si **pas** est négatif, alors il faut que **dep** soit strictement supérieur à **fin**, sinon la liste renvoyée est vide.

```
>>> L = [1, 2.5 , [1, 2], 'A', 'B'] # définition de \
↪ la liste
>>> L[3:4] # seul l'élément L[3]
['A']
>>> L[3:3] # liste vide
[]
>>> L[3:2:-1] # seul l'élément L[3]
['A']
>>> L[3:3:-1] # liste vide
[]
```

- L'instruction `del` permet la suppression.

- L'instruction **del** permet la suppression.

```
>>> liste = [0, 1, '3/2', 2, 3, 4]
>>> del liste[5] # suppression d'un élément
>>> liste
[0, 1, '3/2', 2, 3]
>>> del liste[:2] # suppression des 2 premiers
>>> liste
['3/2', 2, 3]
```

- Le slicing permet aussi la suppression.

- Le slicing permet aussi la suppression.

```
>>> liste = [0, 1, '3/2', 2, 3, 4]
>>> liste[5:] = []
>>> liste
[0, 1, '3/2', 2, 3]
>>> liste[:2] = [] # suppression des 2 premiers
>>> liste
['3/2', 2, 3]
```

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$?

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1,2,3]$? ($L = L + [4,5]$ ou $L += [4,5]$)

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$?

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$?

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$?

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$? ($L[5:] = []$ ou `del` $L[5:]$)

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$? ($L[5:] = []$ ou `del` $L[5:]$)
- $L = [-1, 0, 1, 2, 3, 4, 5]$?

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$? ($L[5:] = []$ ou `del` $L[5:]$)
- $L = [-1, 0, 1, 2, 3, 4, 5]$? ($L[:2] = []$ ou `del` $L[:2]$)

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$? ($L[5:] = []$ ou `del` $L[5:]$)
- $L = [-1, 0, 1, 2, 3, 4, 5]$? ($L[:2] = []$ ou `del` $L[:2]$)
- $L = [1, 2, 3, 3.5, 4, 5]$?

EXERCICE

Comment obtenir $L = [1,2,3,4,5]$, en partant de :

- $L = [1, 2, 3]$? ($L = L + [4, 5]$ ou $L += [4, 5]$)
- $L = [3, 4, 5]$? ($L = [1, 2] + L$)
- $L = [1, 2, 5]$? ($L[2:2] = [3, 4]$ ou $L = L[0:2] + [3, 4] + L[2:]$)
- $L = [1, 2, 3, 4, 5, 6, 7]$? ($L[5:] = []$ ou `del L[5:]`)
- $L = [-1, 0, 1, 2, 3, 4, 5]$? ($L[:2] = []$ ou `del L[:2]`)
- $L = [1, 2, 3, 3.5, 4, 5]$? ($L[3:4] = []$ ou `del L[3:4]`).

On les trouve en tapant dans la console `dir(list)`

MÉTHODES ASSOCIÉES AUX LISTES

On les trouve en tapant dans la console `dir(list)`, ce qui donne :

```
>>> dir(list)
['_add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', \
↳ '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', \
↳ '__getattr__', '__getitem__', '__getstate__', '__gt__', '__hash__', \
↳ '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', \
↳ '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', \
↳ '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', \
↳ '__setattr__', '__setitem__', '__sizeof__', '__str__', \
↳ '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', \
↳ 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

On obtient ainsi la liste des méthodes applicables aux listes.

MÉTHODES `append` ET `insert`

- Ajouter un objet en fin de liste : méthode `append(<objet>)`.

```
>>> liste = [1, 2, 3]
>>> liste.append(4)
>>> liste
[1, 2, 3, 4]
```


MÉTHODES append ET insert

- Insérer un objet à l'indice i : `insert(<indice>, <objet>)`.

```
>>> liste = [1, 2, 5]
>>> liste.insert(2, 4)
>>> liste
[1, 2, 4, 5]
>>> liste.insert(2, 3)
>>> liste
[1, 2, 3, 4, 5]
```

MÉTHODES append ET insert

- Pour insérer une **sous-liste** à l'indice i on utilise le « slicing ».

```
>>> liste = [1, 2, 5]
>>> liste[2:2] = [3, 4]  # insertion à l'indice 2
>>> liste
[1, 2, 3, 4, 5]
```

- On notera la différence avec `liste.insert(2,[3,4])`, qui donne la liste `[1,2,[3,4],5]`.

MÉTHODES append ET insert

- Le « slicing » permet de faire des insertions/suppressions en remplaçant la sous-liste extraite par une autre sous-liste.

```
>>> liste = [1, 2, 2.5, '11/4', 4, 5]
>>> liste[2:4] = [3]
>>> liste
[1, 2, 3, 4, 5]
```

MÉTHODES pop ET remove

- La méthode `pop()` renvoie le dernier élément d'une liste, cet élément est alors **supprimé** de la liste.

```
>>> liste = [1, 2, 3, 4]
>>> liste.pop()
4
>>> liste
[1, 2, 3]
```

MÉTHODES pop ET remove

- La méthode `remove(<valeur>)` supprime la première occurrence d'une <valeur> dans une liste.

```
>>> liste = [1, 2, 'trois', 3, 4, 'trois']  
>>> liste.remove('trois') # suppression valeur \  
↪ d'indice 2  
>>> liste  
[1, 2, 3, 4, 'trois']
```

COPIE D'UNE LISTE

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.

ORIGINE DU PROBLÈME

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.
- Mais elle crée une deuxième étiquette sur cet objet.

ORIGINE DU PROBLÈME

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.
- Mais elle crée une deuxième étiquette sur cet objet.
- Les listes étant des objets mutables, ceci peut donner des surprises.

ORIGINE DU PROBLÈME

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.
- Mais elle crée une deuxième étiquette sur cet objet.
- Les listes étant des objets mutables, ceci peut donner des surprises.

```
>>> L1 = [1, 2, 3, [1, 2, 3]]
```

```
>>> L2 = L1
```

```
>>> L1[0] = 0
```

```
>>> L1[3][0] = 0
```

```
>>> L1
```

```
[0, 2, 3, [0, 2, 3]]
```

```
>>> L2
```

```
[0, 2, 3, [0, 2, 3]]
```

ORIGINE DU PROBLÈME

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.
- Mais elle crée une deuxième étiquette sur cet objet.
- Les listes étant des objets mutables, ceci peut donner des surprises.

```
>>> L1 = [1, 2, 3, [1, 2, 3]]
```

```
>>> L2 = L1
```

```
>>> L1[0] = 0
```

```
>>> L1[3][0] = 0
```

```
>>> L1
```

```
[0, 2, 3, [0, 2, 3]]
```

```
>>> L2
```

```
[0, 2, 3, [0, 2, 3]]
```

- La modification de L1 a été répercutée sur la variable L2.

ORIGINE DU PROBLÈME

- Une affectation `<var> = <list>` ne duplique pas l'objet `<list>`.
- Mais elle crée une deuxième étiquette sur cet objet.
- Les listes étant des objets mutables, ceci peut donner des surprises.

```
>>> L1 = [1, 2, 3, [1, 2, 3]]
```

```
>>> L2 = L1
```

```
>>> L1[0] = 0
```

```
>>> L1[3][0] = 0
```

```
>>> L1
```

```
[0, 2, 3, [0, 2, 3]]
```

```
>>> L2
```

```
[0, 2, 3, [0, 2, 3]]
```

- La modification de L1 a été répercutée sur la variable L2.
- Car L1 et L2 pointent vers un même contenu en mémoire.

COMMENT DUPLIQUER UNE LISTE ?

Extraire les éléments de la liste ne suffit pas. On utilise **deepcopy** du module **copy**.

COMMENT DUPLIQUER UNE LISTE ?

Extraire les éléments de la liste ne suffit pas. On utilise **deepcopy** du module **copy**.

```
>>> L1 = [1, 2, 3, [1, \
↪ 2, 3]]
```

```
>>> L2 = L1
```

```
>>> L3 = L1[:]
```

```
>>> from copy import \
↪ deepcopy
```

```
>>> L4 = deepcopy(L1)
```

```
>>> L1[0] = 0
```

```
>>> L1[3][0] = 0
```

```
>>> L1
```

```
[0, 2, 3, [0, 2, 3]]
```

```
>>> L2
```

```
[0, 2, 3, [0, 2, 3]]
```

```
>>> L3
```

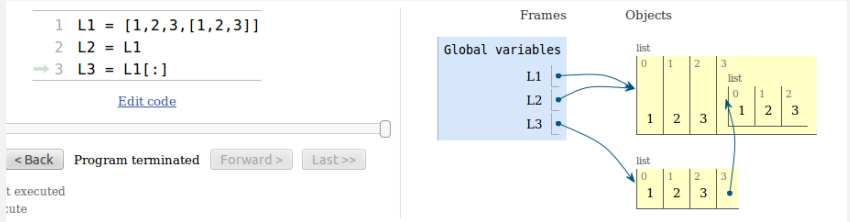
```
[1, 2, 3, [0, 2, 3]]
```

```
>>> L4
```

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

COMMENT DUPLIQUER UNE LISTE ?

● On peut visualiser la gestion mémoire lors des affectations à l'aide de la figure ci-dessous :



Visualisation d'affectations sur le site www.pythontutor.com