Informatique et Calcul Scientifique

Listes Python

02.10.2024

La fois passée, on a vu...

- Les fonctions en Python
  - Définition
  - Paramètres (obligatoires et par défaut) VS arguments (positionnels et par mot-clé)

```
def ma_fonction(param1, param2, param3 = 1, param4 = 2):
    """documentation de ma fonction"""
    # Ensemble d'instructions composant le corps
    # de ma fonction et effectuant des operations
    # sur des variables *locales*
    return param1+param2, param3+param4
x = 50
a,b = ma_fonction(x,param2 = 10)
print(a,b) # retourne 60, 3
```

- La notion d'espace de travail et de portée des variables (locales et globales)
- Les modules en Python.

Aujourd'hui, on verra...

- Les listes Python
- Des opérations associées
- La notion de mutabilité
- Les copies superficielles.

Structures de données

Un programme informatique (en Python ou autre) **manipule des données**.

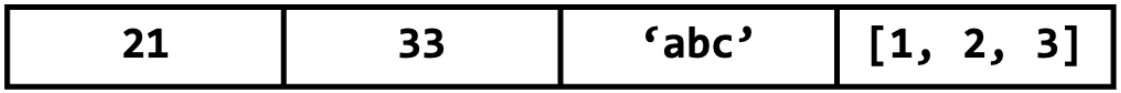
- Une partie essentielle de la programmation et de l'algorithmique est de choisir les bonnes **structures de données** : la manière d'organiser les données et de les stocker en mémoire afin de permettre à certaines opérations de manipulation des données de se dérouler de manière efficace (on quantifiera "efficace" plus tard...)
- Dans ce cours, on étudiera une des principales structures de données en Python : les **listes**.

Listes en Python

Une liste Python est une structure de données qui permet de regrouper diverses données de manière *ordonnée*.

- `L = ['a', 'b', 'c', 'd', 'e']`

- `L = [21, 33, "abc", [1, 2, 3] ]`



. La représentation des listes ci-dessus est schématique et ne reflète pas ce qui se passe vraiment en mémoire - plus d'infos plus tard.

Opérations sur les listes

On peut :

- Créer une liste, vide ou contenant déjà des éléments
- Effectuer des opérations de **lecture** :
  - Accéder à un élément d'une liste
  - Copier une liste
  - Effectuer des calculs sur une liste : trouver le min, le max, la longueur, ...
  - ...
- Effectuer des opérations d'**écriture** :
  - Modifier un élément d'une liste
  - Ajouter ou enlever un élément de la fin d'une liste
  - Insérer ou enlever un élément à un index quelconque de la liste
  - Etendre une liste avec les éléments d'une autre liste
  - ...

. Pour une liste complète des opérations possibles sur une liste en Python, voir la documentation Python.

Créer une liste

On déclare une liste par des valeurs séparées par des virgules et le tout encadré par des crochets droits.

- L'instruction `L = [0, 2, 4, 6]` crée une liste contenant les éléments 0, 2, 4 et 6 et l'affecte à la variable `L`.
- On peut créer une liste vide : `L = []`
- Les éléments d'une liste n'ont pas besoin d'être du même type... mais c'est souvent le cas en pratique.

```
L1 = [1, 2, 3]
print(f"{L1 = }")
L2 = ['a', 'b', 'c']
print(f"{L2 = }")
L3 = [1, 'a', 2, 'b', 3, 'c']
print(f"{L3 = }")
L4 = [[1, 2, 3], ['a', 'b', 'c']]
print(f"{L4 = }")
```

Output :

```
L1 = [1, 2, 3]
L2 = ['a', 'b', 'c']
L3 = [1, 'a', 2, 'b', 3, 'c']
L4 = [[1, 2, 3], ['a', 'b', 'c']]
```

Accéder à un élément d'une liste

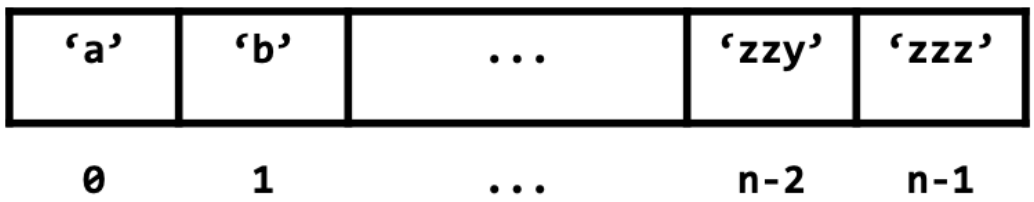
Comme les chaînes de caractères, les listes sont des **séquences** dont les éléments sont indexés. Ainsi, il s'agit d'objets **itérables**.

- Les éléments d'une liste de longueur `n` sont indexés par `0, 1, ..., n-1`. La valeur de l'élément `i` est donné par `L[i]`.

```
L = ['a', 'b', 'c', 'd']
print(f"{L[0] = }")
print(f"{L[1] = }")
print(f"{L[2] = }")
print(f"{L[3] = }")
```

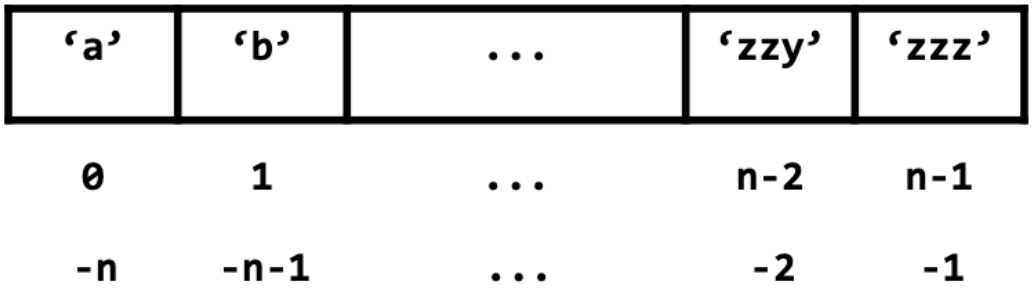
Output :

```
L[0] = 'a'
L[1] = 'b'
L[2] = 'c'
L[3] = 'd'
```



Accéder à un élément d'une liste

- On peut utiliser les indices négatifs `-1, -2, ..., -n` pour accéder aux éléments de la liste à partir de la fin :



- Si on essaie d'accéder à un index `i` trop grand ou trop petit, une exception `IndexError` est générée :

```
L = ['a',3,[1,2]]
print(f"L[-2] = {L[-2]}")
print(f"L[-5] = {L[-5]}")
print(f"L[3] = {L[3]}")
```

```
Output :
L[-2] = 3
IndexError: list index
out of range
IndexError: list index
out of range
```

Accéder à une tranche de liste

Comme pour les chaînes de caractères, on peut accéder à une sous-liste de la liste.

- Pour une liste `L`, `L[i:j]` est une liste qui contient les éléments de la liste `L` entre les indices `i` (inclus) et `j` (exclus).
- Si `i` est omis, il est pris à 0 par défaut. Si `j` est omis, il est pris à `len(L)` par défaut. `L[:]` est donc une copie superficielle[1] de la liste `L`.

```
L = [10, 30, 50, 70, 90]
print(L[2:4])
print(L[2:5])
L1 = L[:]
print(L1)
print(L1 is L)
```

```
Output :
[50, 70]
[50, 70, 90]
[10, 30, 50, 70, 90]
False
```

1. Nous reviendrons sur ce point **très** important dans la suite de ce cours

Accéder à une tranche de liste

- De même que pour les `str`, on peut définir un pas pour le slicing : `L[i:j:k]` est une liste contenant les éléments de `L` d'indices `i`, `i + k`, `i + 2k`, `...` jusqu'à `j` exclus.
- Attention, comme pour `range()`, le pas peut être négatif !
- Le pas est pris à 1 par défaut.

```
L = [10, 20, 30, 40, 50, 60, 70]
print(L[2:6:2])
print(L[6:1:-2])
print(L[2::])
print(L[:2:])
print(L[::-2])
```

Output :

```
[30, 50]
[70, 50, 30]
[30, 40, 50, 60, 70]
[10, 20]
[70, 50, 30, 10]
```

Parcourir une liste

Tout comme on peut itérer sur un `range` ou un `str` à l'aide d'une boucle `for` et du mot-clé `in`, on peut itérer sur une liste.

- La variable `val` ci-dessous prend les valeurs successives de la liste à chaque itération.

```
L = [1, 3, 5, 7]
for val in L:
    print(val)
```

```
Output :

1
3
5
7
```

Créer une liste à partir d'un itérable

On peut convertir n'importe quel objet **itérable** en une liste.

```
L1 = list(range(2, 9, 3))
print(L1)

s = "salut"
L2 = list(s)
print(L2)

n = 3
L3 = list(n)
```

Output :

```
[2, 5, 8]
['s', 'a', 'l', 'u', 't']
TypeError: 'int' object is not
iterable
```

Modifier un élément d'une liste

- On peut modifier l'élément d'indice `i` d'une liste `L` avec l'instruction `L[i] = nouvelle_valeur`.

- Si l'élément à remplacer est une tranche de liste, donc un objet **itérable** même s'il ne contient qu'un seul élément, il doit être remplacé par un autre autre objet itérable (liste ou string).
- Les deux itérables n'ont pas besoin d'être de même taille !

```
ma_liste = [1, 3, 5, 7, 9, 11]
ma_liste[2] = 'A'
print(ma_liste)
ma_liste[3:6] = 'B'
print(ma_liste)
ma_liste[1:2] = ['c','d','e']
print(ma_liste)
ma_liste[3:4] = 1
```

```
Output :
[1, 3, 'A', 7, 9, 11]
[1, 3, 'A', 'B']
[1, 'c', 'd', 'e', 'A', 'B']
TypeError: can only assign
an iterable
```

Un exemple : liste de listes

Les éléments d'une liste peuvent être de n'importe quel type ; en particulier, ils peuvent être à leur tour des listes.

```
L = [[10, 20, 30], [1, 3, 5]]
print(L[0])
print(L[1])
for i in L[0]:
    print(i, end = " ")
print()
print(L[1][0], L[1][1], L[1][2])
```

Output :

```
[10, 20, 30]
[1, 3, 5]
10 20 30
1 3 5
```

Un exemple : liste de listes

Une liste de listes est utile pour représenter une matrice `m` $\times$ `n`.

- On utilise une liste contenant `m` listes, chacune contenant `n` éléments.
- La matrice ci-dessous peut être représentée par `M = [[1, 2], [3, 4], [5, 6]]`

	0	1
0	1	2
1	3	4
2	5	6

- `M[i][j]` est l'élément de la ligne `i` et de la colonne `j` :

```
M = [[1, 2], [3, 4], [5, 6]]
print(f"{M[2][1]=}, {M[1][0]=}")
```

```
Output :
M[2][1]=6, M[1][0]=3
```

. On peut aussi créer des "vecteurs de matrices" ! Cet objet mathématique s'appelle un *tenseur*. On crée pour cela une liste, dans une liste, dans une liste...

Insérer un élément à la fin d'une liste

La méthode[2] `append()` permet de rajouter un élément à la fin d'une liste.

- S'agissant d'une méthode, son appel est différent des fonctions habituelles.
  Pour appliquer la méthode `append` sur une liste `L`, on utilise la sysntaxe : `L.append(valeur)` .

```
L = [0, 1]
print(L, "est de longueur", len(L))
L.append(10)
print(L, "est de longueur", len(L))
L.append(13)
print(L, "est de longueur", len(L))
```

Output :

```
[0, 1] est de longueur 2
[0, 1, 10] est de
longueur 3
[0, 1, 10, 13] est de
longueur 4
```

- Elle **modifie** donc la liste (important !).

2. Une méthode est une fonction définie pour une *classe* particulière d'objets

Exemple : Construction de matrices

On peut utiliser la méthode `append()` pour remplir une matrice de dimension `m` $\times$ `n`.

```
m,n = 3,4
M = []
co = 1
for i in range(m):          # row
    line = []
    for j in range(n):      # column
        line.append(co)
        co += 1
    M.append(line)
print(M)
```

Output :

```
[[1, 2, 3, 4], [5, 6, 7, 8]
, [9, 10, 11, 12]]
```

Il faut pour cela remplir les `n` colonnes d'une ligne individuellement et les stocker dans une liste (boucle `j`), puis ajouter chaque ligne précédemment créée à la liste représentant la matrice `M` (boucle `i`).

Insérer un élément en milieu de liste

La méthode `insert(i, x)` appliquée sur la liste `L` permet d'y insérer l'élément `x` à l'indice `i` : `L.insert(i,x)`

- Tous les éléments après `i` sont décalés "vers la droite". L'élément qui était à l'indice `i` est maintenant à l'indice `i+1`.
- Elle **modifie** donc la liste `L`.

```
L = [1, 3, 5, 7]
L.insert(2, 'a')
print(L)
L.insert(-2,'b')
print(L)
```

Output :

```
[1, 3, 'a', 5, 7]
[1, 3, 'a', 'b', 5, 7]
```

A explorer :

- Que se passe-t-il si on appelle `insert()` avec un index trop grand ? ou avec un index trop négatif ?
- Avec la méthode `insert()`, comment insère-t-on un élément en tête de liste ? en fin de liste ?

Enlever un élément de la fin d'une liste

La méthode `pop()` permet d'enlever un élément de la fin d'une liste :

```
L = [3, 6, 9]
L.pop()
print(L)
```

Output :

```
[3, 6]
```

- `pop()` modifie la liste `L` et **retourne** l'élément enlevé de la liste, qu'on peut par exemple sauvegarder dans une variable :

```
L = [3, 6, 9]
x = L.pop()
print(f"on a pop l'element {x}")
print(f"{L = }")
print(f"{x = }")
```

Output :

```
on a pop l'element 9
L = [3, 6]
x = 9
```

- Testez : que se passe-t-il si on essaie d'appliquer `pop()` à une liste vide ?

Enlever un élément du milieu d'une liste

La méthode `pop()` peut aussi prendre un indice comme argument optionnel.

- Dans ce cas `L.pop(i)` enlève l'élément à l'indice `i` de la liste, et décale tous les éléments qui suivent "vers la gauche" :

```
L = [1, 3, 5, 7, 9]
x = L.pop(2)
print(f"on a pop l'element {x}")
print(L)
```

```
Output :
on a pop l'element 5
[1, 3, 7, 9]
```

- Testez : que se passe-t-il si on appelle `L.pop(i)` avec `i` trop grand ou trop négatif ?

Enlever un élément du milieu d'une liste

On peut enlever un élément de la liste selon sa **valeur** au lieu de son indice.

- La méthode `L.remove(x)` enlève le premier élément de valeur `x` dans la liste `L`, si un tel élément existe (elle **modifie** donc la liste).

```
L = [1, 3, 5, 7, 3]
L.remove(3)
print(L)
L.remove(3)
print(L)
```

Output :

```
[1, 5, 7, 3]
[1, 5, 7]
```

- Testez : que se passe-t-il si on appelle `remove()` sur une liste `L` avec comme argument une valeur `x` qui n'est pas présente dans la liste ? Quelle est la différence avec `L.pop(i)`, où `i` n'existe pas ?

Etendre une liste, concaténer des listes

- Pour des listes `L1` et `L2`, `L1.extend(L2)` ajoute tous les éléments de `L2` à la fin de `L1`. Elle modifie donc `L1` et laisse `L2` telle quelle.

```
L1 = [1, 3]
L2 = [2, 4]
L1.extend(L2)
print(L1)
print(L2)
```

```
Output :
[1, 3, 2, 4]
[2, 4]
```

- Comme les strings, on peut concaténer deux listes avec l'opérateur `+`. La concaténation de listes crée une nouvelle liste concaténée et laisse `L1` et `L2` telles quelles.

```
L1 = [1, 3]
L2 = [2, 4]
L = L1 + L2
print(L)
print(L1, L2)
```

```
Output :
[1, 3, 2, 4]
[1, 3] [2, 4]
```

Concaténer des listes

- On peut aussi concaténer une liste avec elle-même avec l'opérateur `*` . Cette opération crée une nouvelle liste et laisse la liste originale telle quelle.

```
L1 = [1, 3]
L = L1 * 3
print(L)
print(L1)
```

```
Output :
[1, 3, 1, 3, 1, 3]
[1, 3]
```

- On peut réaffecter le résultat d'une opération de concaténation sur une liste à la même variable. L'objet liste n'est pas modifé par l'opération, mais la variable a été réaffectée.

```
L1 = [1, 3]
L2 = [2, 4]
L1 = L1 + L2
print("L1 =", L1)
L3 = [1, 3]
L3 = L3 * 2
print("L3 =", L3)
```

```
Output :
L1 = [1, 3, 2, 4]
L3 = [1, 3, 1, 3]
```

Faire des calculs sur une liste

- `max(L)` et `min(L)` donnent le maximum et le minimum des valeurs de `L` (s'il existe un ordre sur les valeurs de `L` !)
- `L.count(x)` retourne le nombre d'occurences de la valeur `x` dans `L`
- `L.index(x)` retourne le premier indice de `L` où `x` apparaît si un tel indice existe
- `L.sort()` trie la liste `L` (et donc modifie `L`)
- `L.reverse()` inverse l'ordre des éléments de `L` (et donc modifie `L`)
- La fonction `len(L)` retourne la longueur (le nombre d'éléments) de cette liste
- ...

Faire des calculs sur une liste : exemples

```
L = [2, 6, 6, 4, 8, 10]
m, M = min(L), max(L)
print(f"{m = }, {M = }")
print(L.count(6))
print(L.index(3))
print(L.index(4))
L.sort()
print(f"La liste est modifiee :
                {L = }")
L.reverse()
print(f"La liste est modifiee :
                {L = }")
print(f"Longueur : {len(L)}")
```

Output :

```
m = 2, M = 10
2
ValueError : 3 is not in
list
3
La liste est modifiée :
L = [2, 4, 6, 6, 8, 10]
La liste est modifiée :
L = [10, 8, 6, 6, 4, 2]
Longueur : 6
```

Appartenance

On peut vérifier l'appartenance d'un élément de valeur `x` à une liste `L` avec le mot-clé `in` qui nous permet d'exprimer la condition booléenne `x in L`.

- On peut similairement formuler la condition booléenne `x not in L`.

```
L = [1, 3, 5, 7]
print(2 in L)
print(3 in L)
print(4 not in L)

if 2 in L:
    print("?!#@$??")
else:
    print("ouf")
```

Output :

```
False
True
True
ouf
```

Compréhension de listes

On peut créer une liste de manière concise en utilisant la *compréhension de listes*. On utilise la syntaxe suivante :

`[opérations for élément in itérable]`

- Les deux programmes ci-dessous créent la même liste `L = [0, 1, 2, 3, 4]` :

```
L = []
for x in range(5):
    L.append(x)
```

```
L = [x for x in range(5)]
```

- Les deux programmes ci-dessous créent la même liste `L = [0, 2, 4, 6, 8]` :

```
L = []
for x in range(5):
    L.append(2*x)
```

```
L = [2*x for x in range(5)]
```

Compréhension de listes

Dans les exemples précédents, on a construit une liste où les éléments successifs résultent d'une même opération appliquée aux élément successif d'un range.

- Au lieu d'un range, on peut parcourir n'importe quel itérable : une autre liste...

```
L1 = [1, 2, 3, 4, 5]
L2 = [x**2 for x in L1]
print(L2)
```

Output :

```
[1, 4, 9, 16, 25]
```

- ... un string ...

```
s = "salut"
L = [c for c in s]
print(L)
```

Output :

```
['s', 'a', 'l', 'u', 't']
```

Compréhension de listes

- On peut même ajouter des conditions avec une instruction `if` :

```
L = [1, 3, 5, 7, 9]
L1 = [x for x in L if x > 4]
print(L1)
```

Output :

[5, 7, 9]

- Ou itérer sur plusieurs boucles :

```
L = [x + y for x in "ah" for y in "non"]
print(L)
```

Output :

['an', 'ao', 'an', 'hn', 'ho', 'hn']

Copier une liste

Pour une liste `L1`, l'instruction `L2 = L1.copy()` crée une **nouvelle liste** contenant exactement les mêmes valeurs que `L1` et affecte cette nouvelle liste à la variable `L2` [3].

```
L1 = [2, 4, 6, 8]
L2 = L1.copy()
print("L1 et L2 contiennent \
les memes valeurs:")
print(L1, L2)
print(L1 == L2)
print("L1 et L2 ne sont pas \
le meme objet:")
print(id(L1), id(L2))
print( L1 is L2)
```

```
Output :
L1 et L2 contiennent les
mêmes valeurs:
[2, 4, 6, 8] [2, 4, 6, 8]
True
L1 et L2 ne sont pas le
même objet:
4382652544 4382350656
False
```

3. Note : cette opération est équivalente à `L2 = L1[:]` vue précédemment.

Copier une liste

Comme les listes `L1` et `L2` ne sont pas le même objet, elles évoluent de manière indépendente.

```
L1 = [2, 4, 6]
L2 = L1.copy()
L1.append(8)
print(f"L1 = {L1}, L2 = {L2}")
L2[1] = 42
print(f"L1 = {L1}, L2 = {L2}")
```

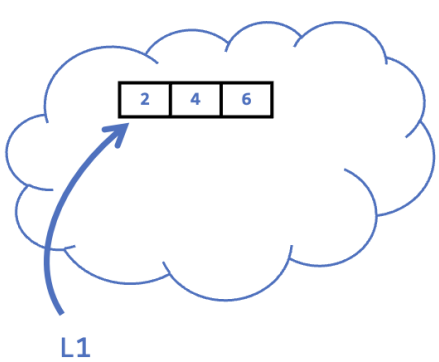
Output :

```
L1 = [2, 4, 6, 8],
L2 = [2, 4, 6]
L1 = [2, 4, 6, 8],
L2 = [2, 42, 6]
```

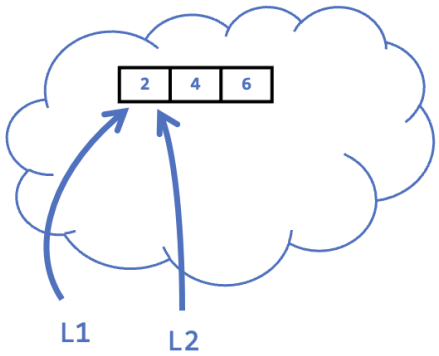
Copier une liste

Que se passe-t-il en mémoire lorsqu'on exécute le code suivant ?

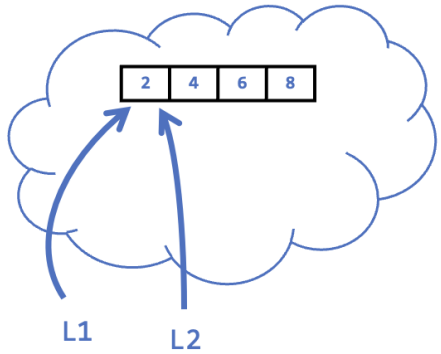
```
L1 = [2, 4, 6]
L2 = L1
L1.append(8)
for i in L2:
    print(i)
```



L1 = [2, 4, 6]



L2 = L1

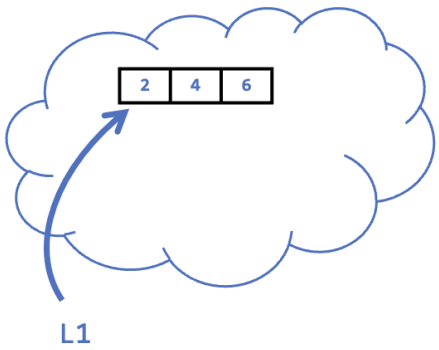


L1.append(8)

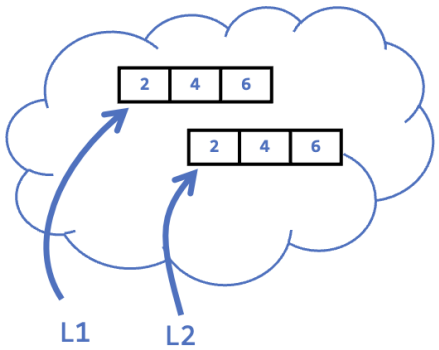
Copier une liste

Que se passe-t-il en mémoire lorsqu'on exécute le code suivant ?

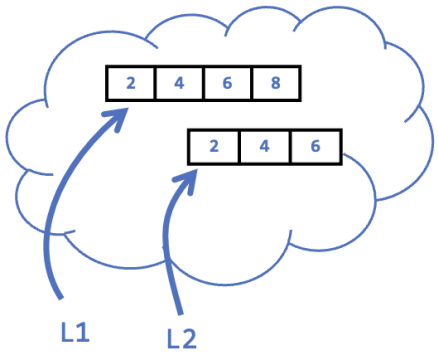
```
L1 = [2, 4, 6]
L2 = L1.copy()
L1.append(8)
for i in L2:
    print(i)
```



L1 = [2, 4, 6]



L2 = L1.copy()



L1.append(8)

Représentation de listes en mémoire

En utilisant la syntaxe `L2 = L1.copy()` ou `L2 = L1[:]`, Python crée un nouvel objet référencé par `L2`.

- Un comportement prévisible :

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0] = [1, 3, 5]
print(L1)
print(L2)
```

```
Output :
[[1, 3, 5], [2, 4]]
[[1, 3], [2, 4]]
```

- Un comportement bizarre ?

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0].append(5)
print(L1)
print(L2)
```

```
Output :
[[1, 3, 5], [2, 4]]
[[1, 3, 5], [2, 4]]
```

Représentation de listes en mémoire

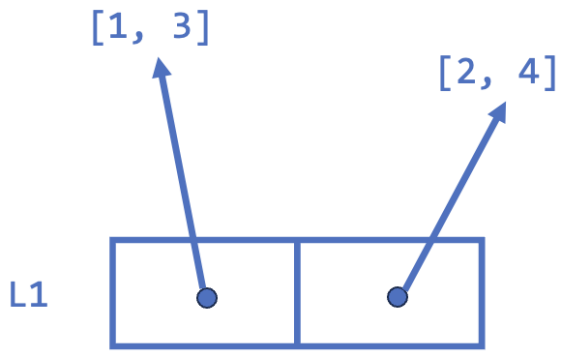
- Pour comprendre ce que fait ce programme, il faut comprendre comment les listes Python sont implémentées en mémoire.
- Une liste Python [4] est stockée en mémoire comme un bloc contigu de **références** aux éléments de la liste.
  - Une référence est l'adresse en mémoire de l'élément auquel elle se réfère.
  - Chaque référence occupe un bloc de taille constante en mémoire.
- Ainsi, l'interpréteur Python peut allouer efficacement l'espace de stockage d'une liste, même si les éléments de la liste sont arbitrairement grands (par exemple, dans le cas d'une liste de listes).

4. Dans CPython, l'implémentation par défaut et la plus courante de Python

Le premier exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0] = [1, 3, 5]
```

1. `L1 = [[1, 3], [2, 4]]`
   → `L1` contient deux références vers les listes `[1, 3]` et `[2, 4]` stockées en mémoire.



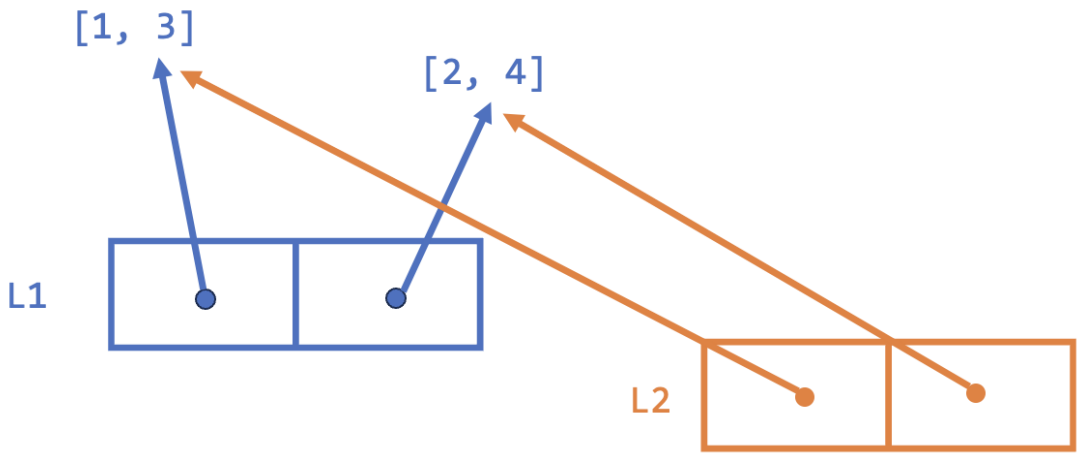
Note : Dans cet exemple et le suivant, vous pouvez afficher les `id` des éléments de `L1` et `L2` à chaque étape pour comprendre ce qui se passe.

Le premier exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0] = [1, 3, 5]
```

2. `L2 = L1.copy()`

   → `L2` est une copie des références vers lesquelles pointent les éléments de la liste `L1`. Les éléments de `L1` et `L2` pointent ainsi vers les mêmes objets.



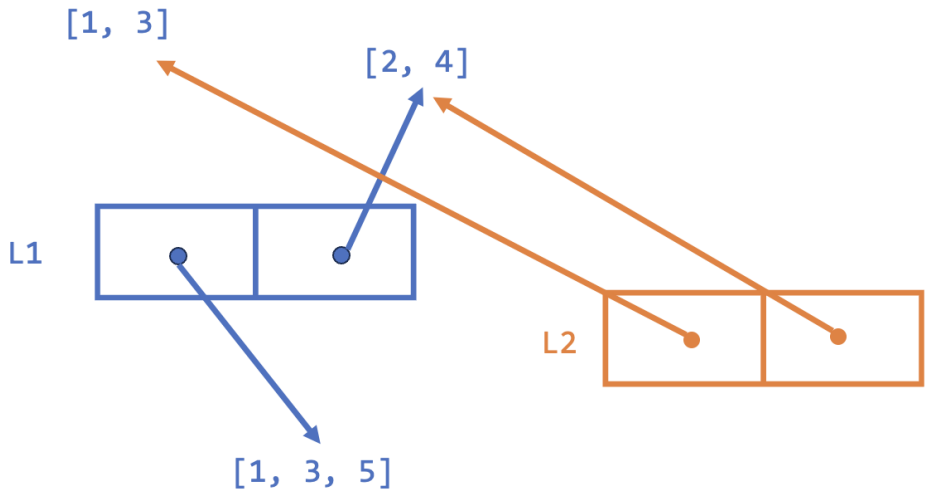
Le premier exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0] = [1, 3, 5]
```

3. `L1[0] = [1, 3, 5]`

   $\rightarrow$ Le 0^{eme} élément de `L1` est remplacé par une nouvelle référence vers la liste `[1, 3, 5]` qui est créée ailleurs dans la mémoire.

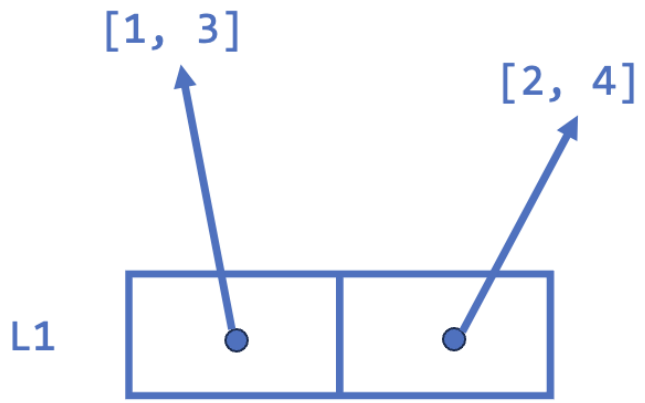
   $\rightarrow$ Le 0^{eme} élément de `L2` reste inchangé.



Le second exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0].append(5)
```

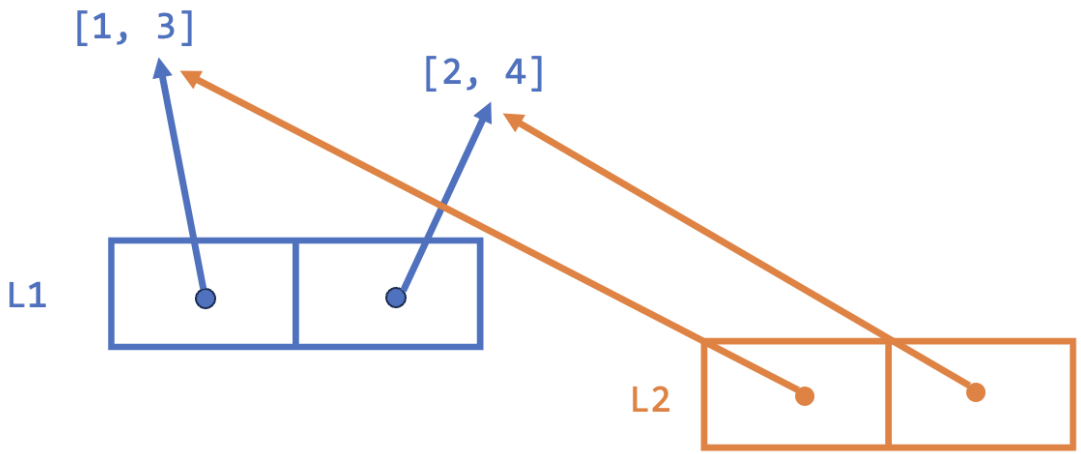
1. `L1 = [[1, 3], [2, 4]]`



Le second exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0].append(5)
```

2. `L2 = L1.copy()`

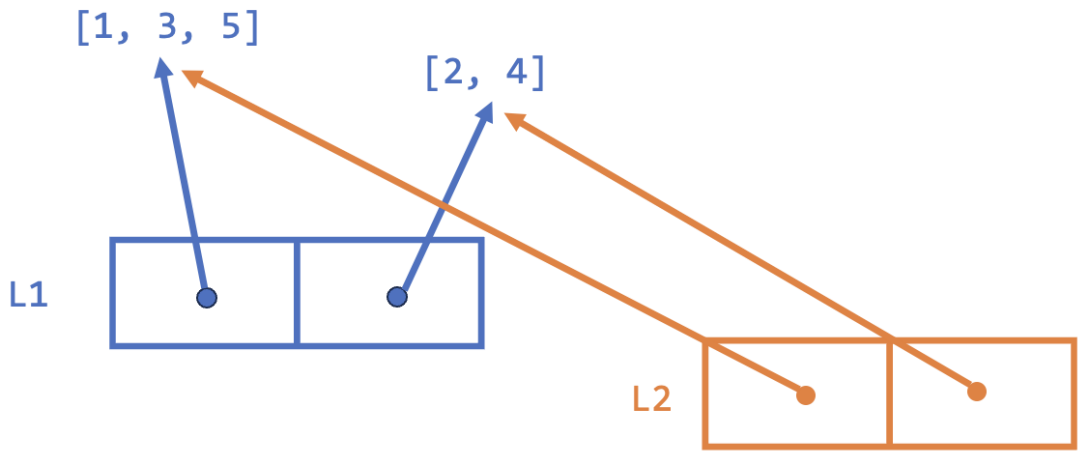


Le second exemple

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0].append(5)
```

3. `L1[0].append(5)`

   $\rightarrow$ La liste `[1,3]` à laquelle le 0^e élément de la liste `L1` fait référence est modifiée. Comme `L2[0]` fait référence au **même** objet, son affichage affichera la même valeur.



Moralité

1. Les listes ne contiennent pas des objets, mais des **références** à des objets existant en mémoire.
2. Les listes sont des objets **mutables** (c'est-à-dire des objets qu'on peut modifier après les avoir créés).
3. Lorsqu'on exécute `L2 = L1.copy()` ou `L2 = L1[:]` on crée en réalité une **copie superficielle** de la liste `L1`. C'est-à-dire qu'on ne copie que les références vers les objets originaux.

Remarque : Pour faire une "vraie" copie, ou **copie profonde** d'une liste de listes (qui va copier chacune des sous-listes au lieu de pointer vers les mêmes sous-listes) on utilise la fonction `deepcopy()` du module `copy`.

Moralité

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0] = [1, 3, 5]
```

```
L1 = [[1, 3], [2, 4]]
L2 = L1.copy()
L1[0].append(5)
```

- Donc le comportement "bizarre" dans le second exemple est dû au fait que L1 et L2 satisfont la propriété (1) et que leurs éléments, étant aussi des listes, satisfont la propriété (2).
- La mutabilité est une propriété en Python qui offre une certaine puissance de programmation mais qui rend les programmes plus vulnérables à des erreurs.
- Certains langages de programmation ne permettent aucun objet mutable !