Les listes en **Python**

I. Définition d'une liste

I.1. Les listes : qu'est-ce que c'est ?

- Pour mieux comprendre ce qu'est une liste, exposons l'introduction qui en est faite dans la documentation de **Python**.

  « **Python** connaît différents types de données combinés, utilisés pour regrouper plusieurs valeurs. Le plus souple est la liste, qui peut être écrit comme une suite, placée entre crochets, de valeurs (éléments) séparées par des virgules. Les éléments d'une liste ne sont pas obligatoirement tous du même type, bien qu'à l'usage ce soit souvent le cas. »

- On peut ainsi définir les listes `L1` et `L2` suivantes.

```
In [1] :   L1 = [1, 2, 5, 8, 9]
In [2] :   L2 = ["un mot", 1, [4, 4]]
```

Notons que `L1` est une liste d'entiers (objets de type `int`). La liste `L2`, quant à elle, est une liste qui contient 3 objets de types différents :

× `"un mot"` est une chaîne de caractère (objet de type `str`),

× `1` est un entier (objet de type `int`),

× `[4, 4]` est une liste (d'entiers en l'occurrence).

Pour autant, `L1` et `L2` sont des objets de même type `list`.

```
In [3] :   type(L1)
Out [3] :  list

In [4] :   type(L2)
Out [4] :  list
```

- Notons enfin qu'on peut accéder à la longueur d'une liste (son nombre d'éléments) à l'aide de l'opérateur `len`.

```
In [5] :   len(L1)
Out [5] :  5

In [6] :   len(L2)
Out [6] :  3
```

I.2. Accès aux éléments d'une liste

I.2.a) L'accès simple à un élément d'une liste

- Les éléments d'une liste sont indicés par des entiers qui démarrent en 0. Autrement dit :

× le 1er élément d'une liste est celui d'indice 0,

× le 2ème élément d'une liste est celui d'indice 1,

× le 3ème élément d'une liste est celui d'indice 2,

× ...

- La syntaxe pour accéder au $k^{\text{ème}}$ élément d'une liste utilise les crochets.

```
In [7] :   L1[0]
Out [7] :  1

In [8] :   L1[len(L1)-1]
Out [8] :  9

In [9] :   L2[0]
Out [9] :  "un mot"

In [10] :  L2[2]
Out [10] : [4, 4]
```

- Si une liste possède un nombre d'éléments noté n, son dernier élément est celui d'indice $n-1$. Demander à accéder aux éléments d'indice n ou plus provoque alors l'affichage d'un message d'erreur.

```
In [11] :        L1[12]
IndexError :     list index out of range

In [12] :        L2[3]
IndexError :     list index out of range
```

Les messages d'erreurs sont relativement explicites en **Python**. Il convient donc d'identifier les messages d'erreurs usuels car ils dévoilent le genre d'erreur commise dans le code, ce qui permet au programmeur de comprendre où l'erreur a été commise et d'alors corriger son code. Une erreur qui affiche le message « `out of range` » indique une demande d'accès à un élément d'indice trop élevé.

- Les éléments d'une liste sont aussi indicés du dernier au premier.
  Plus précisément :
  × le dernier élément d'une liste est celui d'indice -1,
  × l'avant-dernier élément d'une liste est celui d'indice -2,
  × l'avant-avant-dernier élément d'une liste est celui d'indice -3,
  × ...

  Là encore, on vérifiera à ne pas demander l'accès à un élément d'indice trop bas.

```
In [13] :        L1[-1]
Out [13] :       9

In [14] :        L2[-3]
Out [14] :       "un mot"

In [15] :        L2[-4]
IndexError :     list index out of range
```

I.2.b) L'accès multiple aux éléments d'une liste (slicing)

- Une liste est une structure de données linéaire : les éléments qu'elle contient apparaissent les uns à la suite des autres. On peut ainsi considérer qu'une liste a un début (ce qui apparaît après la parenthès ouvrante `[`) et une fin (ce qui apparaît avant la parenthèse fermante `]`). Il est alors possible de penser une liste comme une brioche pré-découpée en tranches égales. Chaque tranche correspond à un élément de la liste. Si la brioche n'est pas pré-découpée, on peut la trancher. Du point de vue des listes, découper une tranche (c'est l'opération appelée **slicing** en anglais) consiste à accéder à plusieurs éléments d'affilée de la liste. Comment demander l'accès à plusieurs éléments ? Il semble assez naturel d'indiquer à **Python** qu'on souhaite accéder aux éléments situés de l'indice `deb` à l'indice `fin`.
- Regardons comment cela se passe en pratique.

```
In [16] :        L1[1:4]
Out [16] :       [2, 5, 8]
```

Attention ! En écrivant `L1[1:4]`, on accède à `4-1 = 3` éléments. Ce sont ceux situés entre l'indice `1` inclus et l'indice `4` **exclu**.

- On peut demander à accéder à une tranche qui se situe en dehors de l'intervalle d'indices d'une liste. On peut en effet découper une tranche se situant au-delà de la fin de la brioche. Si on le fait, on récupère une tranche vide.

```
In [17] :        L1[10:17]
Out [17] :       []
```

- Il est possible de signifier à **Python** que l'on souhaite accéder à tous les éléments de la liste à partir de celui d'indice `deb` (inclus). Il suffit pour cela d'utiliser la même syntaxe qu'au-dessus tout en ne précisant pas l'indice de fin.

```
In [18] :        L1[1:]   # À partir de celui d'indice 1
Out [18] :       [2, 5, 8, 9]
```

Cet appel produit le même résultat que `L1[1:5]` : on accède à tous les éléments à partir de celui d'indice `1` jusqu'à celui d'indice `5` (exclu).

- Il est aussi possible d'accéder à tous les élements de la liste jusqu'à celui d'indice `fin` (exclu). Il suffit pour cela d'utiliser la même syntaxe qu'au-dessus tout en ne précisant pas l'indice de début.

```
In  [19] :  L1[:4]   # Jusqu'à celui d'indice 4 (exclu)
Out [19] :  [1, 2, 5, 8]
```

Cet appel produit le même résultat que `L1[0:4]` : on accède à tous les éléments à partir de celui d'indice `0` jusqu'à celui d'indice `4` (exclu).

- Il est aussi possible de demander à accéder à un élément sur 2 (par exemple).

```
In  [20] :  L1[0:4:2]   # Un sur 2 depuis celui d'indice 0
Out [20] :  [1, 5]
```

On demande l'accès à l'élément d'indice `0` (c'est `1`), puis à celui d'indice `0+2 = 2` (c'est `5`), puis à celui d'indice `2+2 = 4` (c'est `9`). En réalité, ce dernier élément n'est pas considéré puisqu'on a explicitement demandé l'accès jusqu'à l'élément d'indice `4` exclu.

- Il est aussi possible de combiner le slicing et l'indiçage négatif des éléments.

```
In  [21] :  L1[:-2]   # Jusqu'à l'avant dernier (exclu)
Out [21] :  [1, 2, 5]

In  [22] :  L1[-2:]   # Depuis l'avant dernier élément
Out [22] :  [8, 9]
```

Cela permet notamment d'obtenir très facilement le **miroir** (les éléments lus de droite à gauche) d'une liste.

```
In  [23] :  L1[-1::-1]   # Depuis le dernier jusqu'au 1er
Out [23] :  [9, 8, 5, 2, 1]
```

I.3. Modifications des éléments d'une liste

À chaque objet **Python**, est associé un nombre entier qui permet de l'**id**entifier. Il y a une bijection entre les objets et les identifiants de sorte qu'un objet possède un unique identifiant et que chaque identifiant représente un unique objet. Pour information, on peut accéder à l'identifiant d'un objet à l'aide de la fonction `id()`. On emploie ici le terme identifiant mais on pourrait aussi parler de l'**identité** d'un objet.
Les objets **Python** peuvent être séparés en deux grandes catégories :

× les objets **muables** (on dit **mutable** en anglais) dont on peut modifier le contenu sans que l'objet change d'identifiant.
C'est notamment le cas des listes.

× les objets **immuables** (on dit **immutable** en anglais). Ce sont les objets dont la valeur ne change pas. Les nombres, les chaînes et les n-uplets (appelés aussi tuples) sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée (source).

I.3.a) Modification d'un élément

- On a vu précédemment comment accéder à un élément d'une liste. La modification d'un élément d'une liste se fait par l'opérateur d'affectation.

```
In  [24] :  L1[0] = -6

In  [25] :  L1
Out [25] :  [-6, 2, 5, 8, 9]

In  [26] :  L1[-1] = [2, 4, 6]

In  [27] :  L1
Out [27] :  [-6, 2, 5, 8, [2, 4, 6]]
```

I.3.b) Modifications multiples en procédant par tranches (CULTURE)

- Il est aussi possible de modifer plusieurs éléments en une fois en utilisant le principe du découpage par tranches.

```
In [28] : L1[-2:] = [1, 0]

In [29] : L1
Out [29] : [-6, 2, 5, 1, 0]
```

- Il est à noter que ce mécanisme peut permettre de modifier la taille de la liste. On peut en effet remplacer une tranche par une tranche de taille plus petite ou par une tranche plus grande.

```
In [30] : L1[0:3] = [4, 3]

In [31] : L1
Out [31] : [4, 3, 1, 0]

In [32] : L1[0:1] = [18, 5, 9]

In [33] : L1
Out [33] : [18, 5, 9, 3, 1, 0]
```

Dans le dernier appel, `L1[0:1]` correspond à la tranche composée uniquement de l'élément d'indice `0`. Même si cette tranche est constituée d'un seul élément, il convient de comprendre qu'il s'agit d'une sous-liste de `L1` et pas d'un élément. Ainsi, les objets `L1[0:1]` (une liste constituée d'un seul élément) et `L1[0]` (un élément) sont radicalement différents.
En conséquence :

× l'affectation `L1[0:1] = [18, 5, 9]` remplace la tranche constitué de l'élément d'indice `0` par la tranche `[18, 5, 9]`.
Le résultat est la liste `[18, 5, 9, 3, 1, 0]`.

× l'affectation `L1[0] = [18, 5, 9]` modifie l'élément d'indice `0` de la lsite `L1` en le remplaçant par la liste `[18, 5, 9]`.
Le résultat est la liste `[[18, 5, 9], 3, 1, 0]`.

Il est à noter que ces différentes opérations ne modifient pas l'identifiant de l'objet (`L1` a même identifiant avant et après affectation).

- On a vu précédemment qu'on pouvait accéder à des tranches se situant au-delà de l'intervalle d'indices de la liste. Ce mécanisme peut notamment être utilisé pour ajouter un élément à la fin d'une liste.

```
In [34] : L1[6:] = [4]

In [35] : L1
Out [35] : [18, 5, 9, 3, 1, 0, 4]
```

Il est à noter que la tranche `L1[6:]` se situe en dehors de la liste `L1` qui ne contient que 6 éléments et dont le dernier élément est celui d'indice `5`.

L'affectation par tranches n'est pas un mécanisme au programme de la filière. En conséquence, il faut l'utiliser avec parcimonie.
Il est donné ici pour la culture informatique du lecteur.

I.4. Ajout et suppression d'éléments dans une liste

I.4.a) Ajout d'éléments en début / fin de listes par concaténation

- Concaténer deux listes, c'est regrouper les éléments de ces deux listes dans l'ordre d'apparition des éléments. La concaténation se fait à l'aide de l'opérateur `+`. Elle permet d'ajouter un élément en début / en fin de liste.

```
In [36] : L2 + [9]
Out [36] : ["un mot", 1, [4, 4], 9]

In [37] : [-2] + L2
Out [37] : [-2, "un mot", 1, [4, 4]]
```

- Évidemment, il est aussi possible d'ajouter plusieurs éléments en une fois.

```
In  [38] :  L2 + [7, 8, 9]
Out [38] :  ["un mot", 1, [4, 4], 7, 8, 9]

In  [39] :  [-2, 0, 5] + L2
Out [39] :  [-2, 0, 5, "un mot", 1, [4, 4]]
```

- Concaténer 2 listes ne modifie aucune des deux listes. Si on souhaite conserver le résultat obtenu, il faut le stocker dans une nouvelle liste.

```
In  [40] :  L = L2 + [9]

In  [41] :  L
Out [41] :  ["un mot", 1, [4, 4], 9]

In  [42] :  L = [-2] + L

In  [43] :  L
Out [43] :  [-2, "un mot", 1, [4, 4], 9]
```

Notons que la première affectation permet de définir l'objet `L`. On lui attribue alors un certain idenfitiant. La deuxième affectation redéfinit l'objet `L` et il se voit alors attribuer un autre identifiant.

- Il est possible d'utiliser la syntaxe `+=` pour ajouter des éléments en fin de liste. Plus précisément, la syntaxe `L += [2]` est sémantiquement équivalente à la syntaxe `L = L + [2]`.

```
In  [44] :  L += [2]

In  [45] :  L
Out [45] :  [-2, "un mot", 1, [4, 4], 9, 2]
```

- L'opération de concaténation a lieu entre deux listes. Tenter de concaténer une liste avec un objet autre qu'une liste provoquera une erreur de type.

```
In  [46] :  L + 4
TypeError :  can only concatenate list (not "int") to list
```

- Il est aussi possible d'ajouter, en début / fin de liste des éléments qui sont eux-mêmes des listes. Pour ce faire, il faut concaténer la liste initiale avec une liste de listes. Illustrons ce procédé.

```
In  [47] :  L = L + [ [1, 2] ]

In  [48] :  L
Out [48] :  [-2, "un mot", 1, [4, 4], 9, [1, 2]]

In  [49] :  [ [1, 2], 1, "truc"] + [ [[1, 1], 4], [2], 5]
Out [49] :  [ [1, 2], 1, "truc", [[1, 1], 4], [2], 5]
```

On pourra retenir qu'on obtient la liste résultat en remplaçant les symboles centraux « `] + [` » par une virgule.

I.4.b) Ajout d'élément(s) en milieu de liste

- En combinant la technique de slicing et la concaténation de listes, il est possible d'ajouter un ou plusieurs éléments en milieu d'une liste. Plus précisément, on décrit ci-dessous comment insérer un élément à l'indice 3 dans la liste `L`.

```
In  [50] :  L = L[:3] + [0] + L[3:]

In  [51] :  L
Out [51] :  [ [1, 2], 1, "truc", 0, [[1, 1], 4], [2], 5]
```

- De manière générale, si l'on souhaite insérer `n` éléments `el1`, ..., `eln` , on écrira :

```
L = L[:i] + [ el1, ..., eln ] + L[i:]
```

- La liste `L` étant redéfinie, cette manière de procéder modifie l'identifiant de la liste `L`.

I.4.c) Suppression d'un élément dans une liste

- Il est possible de supprimer un élément présent dans une liste. Cela se fait via la fonction `del` en spécifiant l'élément de la liste que l'on souhaite supprimer

```
In [52] :   del L[4]

In [53] :   L
Out [53] :  [ [1, 2], 1, "truc", 0, [2], 5]

In [54] :   del L[4]

In [55] :   L
Out [55] :  [ [1, 2], 1, "truc", 0, 5]

In [56] :   del L[5]
IndexError : list assignment index out of range
```

- La fonction `del` est compatible avec le slicing.

```
In [57] :   del L[:3]

In [58] :   L
Out [58] :  [0, 5]
```

I.5. Un point sur la création d'une liste

I.5.a) Les procédés déjà rencontrés

- Il est possible de créer une **liste par extension**, c'est-à-dire en écrivant tous les éléments contenus dans cette liste les uns à la suite des autres et en les séparant par une virgule.
- Il est aussi possible de construire une liste par slicing d'une autre liste (une tranche d'une autre liste est une liste) ainsi qu'en combinant des listes par l'opérateur de concaténation.

I.5.b) À l'aide de l'opérateur *

- L'opérateur `*` est utilisé afin de créer une liste qui répète le contenu d'une liste. Illustrons son utilisation sur quelques exemples.

```
Out [59] :  [0]*5
In [59] :   [0, 0, 0, 0, 0]

Out [60] :  ["mot"]*3
In [60] :   ["mot", "mot", "mot"]

Out [61] :  [ [1, 2] ]*3
In [61] :   [ [1, 2], [1, 2], [1, 2] ]

Out [62] :  [ [1, 2], 4, "truc" ]*2
In [62] :   [ [1, 2], 4, "truc", [1, 2], 4, "truc" ]
```

- On peut aussi remarquer que :
  - × si on répète les éléments d'une liste vide, on obtient la liste vide.
  - × si on répète 0 fois les éléments d'une liste, on obtient la liste vide.

```
Out [63] :  []*5
In [63] :   []

Out [64] :  [1, 3, 4, 6, 7]*0
In [64] :   []
```

I.6. Les listes en compréhension

I.6.a) Un point sur la définition mathématique d'ensemble

- On a vu précédemment qu'on pouvait créer des listes en extension. Au lieu d'opter pour cette stratégie consistant à écrire un par un tous les éléments d'une liste, on peut construire cette liste en spécifiant les propriétés qui définissent ses valeurs. Cette différence entre l'écriture en compréhension et en extension est très classi que mathématiques lorsque l'on étudie des ensembles. Typiquement, l'ensemble des carrés d'entiers compris entre 1 et 5 peut s'écrire à l'aide des deux manières suivantes :
  × **en extension** : $\{1, 4, 9, 16, 25\}$.
  × **en compréhension** : $\{k^2 \mid k \in [\![1, 5]\!]\}$.

  En mathématiques, il est fréquent d'utiliser plutôt la deuxième manière de procéder. Elle est notamment beaucoup plus rigoureuse lorsque l'on s'intéresse à des ensembles contenant un nombre infini de valeurs. Typiquement, si on s'intéresse à l'ensemble des carrés d'entiers positifs, on écrira soit :
  × $\{0^2, 1^2, 2^2, 3^2, 4^2, \ldots\}$,
  × $\{k^2 \mid k \in \mathbb{N}\}$.

  On se sert ici de l'utilisation des points de suspension pour décrire tous les éléments de la liste en extension. Cette notation n'est pas très rigoureuse et on compte sur l'habitude du lecteur pour comprendre ce qui est signifié.
- En informatique, on ne s'intéresse pas à des listes de tailles infinies. Il n'en demeure pas moins qu'il pourrait être intéressant d'écrire des listes en compréhension. C'est l'une des fonctionnalités fantastiques (ne mâchons pas nos mots) offerte par **Python**.

I.6.b) Les listes en compréhension

- La syntaxe pour créer une liste en compréhension (ou parle aussi de compréhension de liste) est assez proche de celle utilisée en mathématique. Il s'agit de spécifier la propriété que doivent vérifier les éléments que l'on souhaite faire apparaître dans la liste. Typiquement, on peut demander de créer la liste contenant toutes les valeurs entières comprises entre `0` et `10` (exclu) ou encore la liste des cinq premiers carrés d'entiers. Pour ce faire, on procède généralement en demandant l'itération (à l'aide d'un `for`) des valeurs d'un intervalle (défini par un `range`).

```
Out [65] :   [k for k in range(10)]
In  [65] :   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Out [66] :   [k**2 for k in range(1, 6)]
In  [66] :   [1, 4, 9, 16, 25]
```

- Il est à noter que ce procédé peut aussi permettre de construire des listes contenant toujours la même valeur.

```
Out [67] :   [0 for k in range(10)]
In  [67] :   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

  On constate ici que le `k` ne joue pas de valeur particulière pour la détermination de la valeur `0`. On peut donc s'en passer.

```
Out [68] :   [0 for _ in range(10)]
In  [68] :   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Il est possible d'enchaîner les itérations. Cela permet par exemple d'obtenir tous les triplets d'éléments de $\{0, 1\}$.

```
Out [69] :   [(i, j, k) for i in range(2)
             for j in range(2) for k in range(2)]

In  [69] :   [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
             (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

- Il est aussi possible d'ajouter des conditions à l'aide de `if`. Il est par exemple possible de créer la liste de tous les carrés multiples de 6 d'entiers plus petits que 20.

```
Out [70] :   [k**2 for k in range(20) if k**2 % 6 == 0]
In  [70] :   [0, 36, 144, 324]
```

Il est aussi possible d'obtenir tous les triplets d'éléments différents de $\{1, 2, 3\}$ (autrement dit toutes les permutations de l'ensemble $\{1, 2, 3\}$).

```
Out [71] :  [(i, j, k) for i in range(1, 3)
            for j in range(1, 3) for k in range(1, 3)
            if i != j and j != k and k !=i]

In [71] :   [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1),
            (3, 1, 2), (3, 2, 1)]
```

I.7. Une possible définition des matrices

I.7.a) Les matrices, des listes de listes

- En mathématiques, une matrice est un tableau de nombres. En informatique, une matrice peut être représentée par une liste de listes. Les listes internes définissent les lignes de la matrice. Par exemple, la matrice :

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

peut être représentée par la liste de listes suivante.

```
Out [72] :  M = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

- Les listes de listes sont elles-mêmes des listes. Ainsi, `M[0]` est elle-même une liste. On accède à ses éléments en écrivant `M[0][0]`, `M[0][1]` et `M[0][2]`.

```
In [72] :   M[1]
Out [72] :  [4, 5, 6]

In [73] :   M[1][2]
Out [73] :  6
```

On peut évidemment modifier la valeur d'un élément de la matrice.

```
In [74] :   M[1][2] = 0

In [75] :   M
Out [75] :  [ [1, 2, 3], [4, 5, 0], [7, 8, 9] ]
```

On peut aussi faire des accès multiples.

```
In [76] :   M[1:3]
Out [76] :  [ [4, 5, 0], [7, 8, 9] ]
```

I.7.b) Construction de matrices à l'aide de listes en compréhension

- Il est par exemple possible d'écrire la matrice nulle de $\mathscr{M}_3(\mathbb{R})$.

```
In [77] :   [[0 for i in range(3)] for j in range(3)]
Out [77] :  [ [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Il est aussi possible d'utiliser ce mécanisme pour définir la matrice M prise en exemple.

```
In [78] :   [[i + 3*j + 1 for i in range(3)] for j in
            range(3)]

Out [78] :  [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

On peut aussi obtenir, à l'aide d'une liste en compréhension, la transposée d'une matrice.

```
In [79] :   [ [M[j][i] for j in range(3)] for i in range(3)]
Out [79] :  [ [1, 4, 7], [2, 5, 8], [3, 0, 9]]
```

II. La classe `list` et ses méthodes

II.1. Une très brève introduction à la notion de classe

- Les listes que l'on crée en **Python** sont des instances (des représentants) de la classe `list`. Sans trop entrer dans les détails de la notion de classe, mettons toutefois en avant qu'une classe définit un certain nombre de méthodes (des fonctions) qui peuvent être utilisées sur les instances de la classe et qui ont pour but :
  - × de créer une instance de la classe (cette méthode est appelée un **constructeur** de la classe),
  - × de modifier des instances de la classe.

  Les méthodes d'une classe sont des fonctions inhérentes à la classe et elles s'utilisent directement « au sein de l'instance » étudiée. Autrement dit, ces méthodes « appartiennent » à l'instance créée.
- Dans le programme d'informatique de Mathématiques Appliquées, on demande de connaître seulement deux méthodes : `count` et `append`. Il existe beaucoup d'autres méthodes utilisables sur les listes (notamment `insert`, `remove`, `pop`, `clear`, `index`, `sort`).
  Ces éléments sont détaillés sur cette page de la documentation.

II.2. Quelques méthodes d'intérêt de la classe `list`

II.2.a) La méthode constructeur

- Comme toute classe, la classe `list` est munie d'un constructeur. Écrire `list()` permet de créer une liste vide.

```
In [80] :  list()
Out [80] : []
```

- Il est aussi possible de prendre en paramètre un objet itérable (c'est-à-dire un objet dont on peut parcourir les valeurs à l'aide d'une boucle `for` par exemple).

On peut notamment créer une liste à partir d'une chaîne de caractères ou d'un uplet de valeurs.

```
In [81] :  list("abc")
Out [81] : ['a', 'b', 'c']

In [82] :  list( (1, 2) )
Out [82] : [1, 2]

In [83] :  list( 1, (2, 3) )
Out [83] : [1, (2, 3)]

In [84] :  list( range(5) )
Out [84] : [1, (2, 3)]
```

- On peut aussi créer une liste à partir d'un intervalle `range(...)`. Profitons-en pour revoir cet itérable.

```
In [85] :  list( range(5) )
Out [85] : [0, 1, 2, 3, 4]

In [86] :  list( range(2, 5) )
Out [86] : [2, 3, 4]

In [87] :  list( range(2, 20, 3) )
Out [87] : [2, 5, 8, 11, 14, 17]

In [88] :  list( range(10, 3, -1) )
Out [88] : [10, 9, 8, 7, 6, 5, 4]
```

Pour rappel, un `range(n)` commence en `0` et se termine en `n-1`. Si on spécifie un début différent en écrivant `range( m, n)`, le premier élément sera `m` (inclus) et le dernier toujours `n-1` (car `n` est exclu). On peut enfin spécifier un pas d'itération (le pas par défaut est 1) ce qui peut par exemple permetttre de lister des éléments dans l'ordre décroissant.

II.2.b) La méthode `append`

- La méthode `append` permet d'ajouter un élément à la fin d'une liste. Comme dit au-dessus, les méthodes sont des fonctions qui s'utilisent « au sein d'une instance » de la classe `list`. Ainsi, on réalisera un `append` sur une liste (`L1` par exemple) pour ajouter un élément `x` pris en paramètre. La syntaxe est la suivante.

```
In  [89] :  L1.append(78)

In  [90] :  L1
Out [90] :  [18, 5, 9, 3, 1, 0, 4, 78]
```

  Évidemment, on peut ajouter en fin de liste des objets de n'importe quel type (des entiers, des listes, des chaînes de caractère ...).

```
In  [91] :  L1.append([1, 2, 1])

In  [92] :  L1
Out [92] :  [18, 5, 9, 3, 1, 0, 4, 78, [1, 2, 1]]

In  [93] :  L1.append("autre")

In  [94] :  L1
Out [94] :  [18, 5, 9, 3, 1, 0, 4, 78, [1, 2, 1], "autre"]
```

II.2.c) La méthode `pop` (CULTURE)

- En complément de la méthode `append`, on présente souvent la méthode `pop`. Pour comprendre le lien entre les deux, il faut faire l'analogie entre les listes **Python** et les piles d'assiettes. Dans une telle pile, les assiettes sont empilées les unes sur les autres. Lorsque l'on veut se servir d'une assiette, on retire (généralement) celle en haut de la pile. Ce mécanisme d'ajout / retrait est connu sous le nom de LIFO (**L**ast **I**n, **F**irst **O**ut). Comme on l'a vu dans le paragraphe précédent, la méthode `append` permet d'ajouter un élément en fin de liste (c'est-à-dire en haut de la pile).

  Il existe une méthode permettant de « dépiler » une liste, c'est-à-dire de retirer le dernier élément de la liste : c'est la méthode `pop`.

```
In  [95] :  L1.pop()
Out [95] :  "autre"

In  [96] :  L1.pop()
Out [96] :  [1, 2, 1]

In  [97] :  L1
Out [97] :  [18, 5, 9, 3, 1, 0, 4, 78]

In  [98] :  L1.pop()
Out [98] :  78

In  [99] :  L1
Out [99] :  [18, 5, 9, 3, 1, 0, 4]
```

  Notons au passage que la méthode `pop` renvoie pour résultat l'élément qui est dépilé de la liste. Ce dernier élément étant supprimé, la liste résultante contient un élément de moins.

- Du fait de l'implémentation des listes **Python**, les méthodes d'ajout (`append`) et de retrait en fin de liste (`pop`) sont efficaces.
- Il est aussi possible de « dépiler » un élément qui n'est pas le dernier de la liste. Il suffit pour cela de préciser l'indice de l'élément qu'on souhaite dépiler.

```
In  [100] :  L1.pop(2)
Out [100] :  9

In  [101] :  L1
Out [101] :  [18, 5, 3, 1, 0, 4]
```

II.2.d) La méthode `count`

- La méthode `count`, comme son nom l'indique, permet de compter le nombre d'occurrences d'un objet (pris en paramètre) dans une liste.
  Afin d'illustrer son utilisation, commençons par modifier `L1`.

```
In  [102] :  L1[0:3] = [1, 1, 1]

In  [103] :  L1
Out [103] :  [1, 1, 1, 3, 1, 0, 4, 78]
```

  On a ajouté dans `L1` des valeurs `1` et on demande maintenant de compter combien `L1` en contient.

```
In  [104] :  L1.count(1)
Out [104] :  4
```

- Attention cependant : lorsque l'on compte le nombre de `1` contenus dans `L1`, il s'agit précisément de compter le nombre d'éléments de `L1` qui sont des `1`. On ne doit pas compter les `1` qui seraient contenus dans un élément de `L1`.

```
In  [105] :  L1[-1] = [1, 1]

In  [106] :  L1
Out [106] :  [1, 1, 1, 3, 1, 0, 4, [1, 1]]

In  [107] :  L1.count(1)
Out [107] :  4

In  [108] :  L1.count([1, 1])
Out [108] :  1
```

  La liste `[1, 1]` est différent de l'objet `1` et n'est donc pas comptabilisée lorsqu'on demande le nombre de `1` dans `L1`.

III. Itération sur les listes

III.1. Itération sur les indices d'une liste

- Lorsque l'on souhaite parcourir les éléments d'une liste, on peut itérer sur les indices de la liste.

```
In  [109] :  L = [1, 2, 4, 5]
In  [110] :  for i in range(len(L)) :
       ... :     print(L[i])
       ... :

   1
   2
   4
   5
```

  Ce type d'itération est particulièrement utilisé pour les questions où l'on demande de trouver « l'indice de l'élément qui vérifie la condition … ».
  Typiquement, on peut boucler sur les indices si on cherche l'indice du maximum / minimum d'une liste.

III.2. Itération sur les éléments d'une liste

- Une liste étant un objet itérable (c'est une notion très propre à **Python**), il est aussi possible d'itérer sur les éléments d'une liste.

```
In  [111] :  for e in L :
       ... :     print(e)
   1
   2
   4
   5
```

  Ce type d'itération est particulièrement utilisé pour les questions où l'on demande de trouver « l'élément qui vérifie la condition … ».
  Typiquement, on peut boucler sur les éléments si on cherche la valeur du maximum / minimum d'une liste.

- Au passage, il convient de noter que :
  - × pour représenter un indice (qui varie), on utilisera la notation `i` (ou `j` ou `k` par exemple).
  - × pour représenter un élément, on utilisera la notation `e` (ou `elmt` ou une notation qui est adaptée au type d'éléments présents dans la liste).

  D'un point de vue syntaxique, il est tout à fait possible d'appeler un indice `x` ou encore `f`. On peut aussi, en mathématique, appeler une fonction x et lui donner en entrée une variable f ($x : f \mapsto f^2$ pour représenter l'élévation au carré de $\mathbb{R}$ dans $\mathbb{R}$). Mais on préfère s'en tenir à des notations plus usuelles qui permettent d'éviter de commettre des erreurs. En informatique, on estime qu'il est nécessaire de faire un effort dans le nommage des fonctions et des variables. Cela fait partie des principe de la bonne programmation.

III.3. Itération mixte (CULTURE)

- Cela ne fait pas partie du programme officiel mais il est aussi possible de faire une itération mixte permettant à la fois de considérer l'indice et la valeur de l'élément considéré. Pour cela, il faut utiliser la commande `enumerate`.

```
In [112]: for (i, e) in enumerate(L) :
     ...:     print(i, e)
     ...:
0 1
1 2
2 4
3 5
```

  Ce type d'itération est particulièrement utilisé pour les questions où l'on demande de trouver « l'indice et la valeur de l'élément qui vérifie la condition ... ». Typiquement, on peut procéder à une itération mixte (ou une itération seulement sur les indices) si l'on souhaite trouver l'indice et la valeur du plus grand / petit élément d'une liste.

IV. Prédicats sur les listes

IV.1. Notion de prédicat

- On appelle prédicat une fonction qui a un résultat booléen. Il s'agit de toutes les fonctions qui consistent à vérifier qu'une condition donnée est vérifiée. Par exemple :
  - × vérifier qu'une liste contient des nombres rangés dans l'ordre croissant.
  - × vérifier que deux listes contiennent les mêmes éléments.
  - × vérifier qu'une liste contient plus de 1 que de 0.
  - × ...

  Ces fonctions renvoient `True` si la condition est vérifiée et `False` sinon.

IV.2. Identification de la structure itérative à utiliser

IV.2.a) Cas où il faut forcément parcourir toute la liste

- Pour vérifier si certaines conditions sont vérifiées, il peut être nécessaire de parcourir tous les éléments de la liste. Lorsque c'est le cas, la structure itérative `for` est la plus adaptée.
- Illustrons ce point avec la fonction `plus_de_un` qui vérifie si une liste contient plus de 1 que de 0.

```
1  def plus_de_un(L) :
2      c_un = 0 # on initialise un compteur de 1
3      c_zero = 0 # on initialise un compteur de 0
4      for i in range(len(L)) :
5          if L[i] == 0 :
6              c_zero += 1
7          elif L[i] == 1 :
8              c_un += 1
9      if c_un >= c_zero : # remarque à suivre sur ce bloc
10         return True
11     else :
12         return False
```

- Il y a quelques remarques à faire sur la question précédente :
  - × le test d'égalité se note ==. L'opérateur = (qu'on peut lire « reçoit » plutôt que « égal ») sert uniquement à l'affectation.
  - × on a utilisé la syntaxe `c_zero += 1` afin d'incrémenter le compteur de 0. Cette syntaxe est sémantiquement équivalente à la syntaxe :

    `c_zero = c_zero + 1`

    Cette dernière syntaxe est généralement préférée lorsque l'on débute. La première est appréciée par les codeurs qui ont plus d'expérience.
  - × les lignes 9 à 12 témoignent d'une incompréhension de la notion de booléen. On s'intéresse au booléen `c_un >= c_zero`.
    Plus précisément, on souhaite renvoyer :
    - ▸ `True` si ce booléen est évalué à `True`.
    - ▸ `True` si ce booléen est évalué à `False`.

    Autrement dit, on souhaite renvoyer la valeur à laquelle est évaluer ce booléen. On préférera donc (et il faut considérer que c'est important) l'écriture ci-dessous.

```
1  def plus_de_un(L) :
2      c_un = 0 # on initialise un compteur de 1
3      c_zero = 0 # on initialise un compteur de 0
4      for i in range(len(L)) :
5          if L[i] == 0 :
6              c_zero += 1
7          elif L[i] == 1 :
8              c_un += 1
9      return c_un >= c_zero
```

  - × le fait que les indices commencent en 0 et que les intervalles `range` exluent le dernier terme amène parfois à des difficultés pour les programmeurs débutant. Profitons-en pout noter que lorsqu'on itère sur les listes, il suffit d'écrire : `for i in range(len(L))`. Il n'y a donc pas lieu de s'effrayer trop avant de ces considérations d'indices.

IV.2.b) Cas où un parcours partiel peut suffire

- Dans le cas où l'on souhaite vérifier que tous les éléments d'une liste vérifient une condition donnée, un parcours partiel de celle-ci peut suffire. En effet, dès qu'on tombe sur un élément qui ne vérifie pas la condition, on sait qu'elle n'est pas vérifiée pour **tous** les éléments. Dans ce cas, on n'a plus à contrôler les éléments restants. Finalement, on peut itérer **tant que** la condition est vérifiée.
- Illustrons le point précédent avec la fonction `tous_positifs` qui consiste à vérifier que tous les éléments d'une liste sont positifs.

```
1  def tous_positifs(L) :
2      n = len(L) # nom pertinent pour la longueur de L
3      bool = True # on initialise un booléen
4      i = 0 # on initialise une variable d'itération
5      while bool and i < n :
6          if L[i] < 0 :
7              bool = False
8          i += 1
9      return bool
```

- Il y a quelques remarques à faire sur la question précédente :
  - × comme on n'utilise pas de boucle `for`, il convient d'initialiser et de mettre à jour une variable d'itération (nommée `i`) permettant de parcourir un par un les éléments de la liste. Comme le dernier indice d'une liste est `n-1`, il convient de vérifier qu'on ne demandera pas l'accès à un élément d'indice plus grand. Cela aurait pour effet de provoquer l'erreur d'indice « `list index out of range` ».
  - × on a ici encore utilisé la syntaxe `i += 1` en lieu et place de `i = i+1`.
  - × on a aussi bien fait attention à appliquer la remarque du paragraphe précédent pour renvoyer l'évaluation du booléen et éviter d'écrire qu'on renvoie `True` s'il est évaluer à vrai et `False` sinon.

× le booléen est initialisé à `True`. Cela a du sens car au moment où on a considéré aucun élément de la liste, on peut signaler que tous les éléments considérés (aucun en l'occurrence) sont bien positifs. En réalité, on peut établir qu'à l'entrée du $(\texttt{i} + 1)^{\text{ème}}$ tour de boucle (le décalage de 1 est dû au fait qu'à l'entrée du 1er tour de boucle `i` vaut 0), le booléen `bool` prend la valeur `True` si les `i` premiers éléments de la liste sont positifs et prend la valeur `False` sinon. On peut démontrer que cette propriété est vérifiée à chaque tour de boucle. C'est pourquoi on l'appelle un **invariant de boucle**. Mettre en avant cet invariant permet de démontrer rigoureusement que la fonction réalise ce qu'on attend d'elle. En l'occurrence, si on sort de la boucle car `i` est supérieur ou égal à `n`, alors les `n` premiers éléments de la liste sont positifs et par invariant de boucle, `bool` vaut `True`.

× les lignes 6 et 7 peuvent être remplacées par une mise à jour du booléen. On peut alors écrire la fonction sous cette la forme ci-dessous.

```
5     while bool and i < n :
6         bool = L[i] >= 0
7         i += 1
8     return bool
```

× avec l'écriture précédente, on peut se demander à quoi sert l'introduction de la variable `bool`. Il serait en effet plus compact de la faire apparaître dans la condition de l'itération.

```
1 def tous_positifs(L) :
2     n = len(L) # nom pertinent pour la longueur de L
3     i = 0 # on initialise une variable d'itération
4     while L[i] >= 0 and i < n : # remarque à suivre
5         i += 1
6     return i == n
```

Le code apparaît plus simple et semble donc plus pertinent. Il est à noter que la valeur renvoyée est un peu différente. Il s'agit de vérifier si on est sorti de l'itération car on a trouvé un indice `i` tel que `L[i] < 0` ou si on a testé tous les éléments de la liste et qu'on sort de la boucle car la propriété `i < n` n'est plus vérifiée. Dans ce dernier cas, en fin de boucle, la variable `i` prend la valeur `n` car, à l'entrée du dernier tour de boucle, `i` vaut `n-1` et que cette variable est incrémentée de 1 lors du dernier tour de boucle. Si en sortie de boucle `i < n`, on est assuré que l'on est sorti avant d'avoir testé tous les éléments de la liste. Ainsi, tous les éléments de la liste sont positifs si et seulement si `i` prend la valeur `n` en sortie de boucle. L'invariant de boucle est ici : à l'entrée du $(\texttt{i}+1)^{\text{ème}}$ tour de boucle, les `i` premiers éléments de la liste sont positifs.

× la ligne 4 est en fait fausse ! Elle a été présentée ainsi pour insister sur la manière dont est évaluée une conjonction. Lorsqu'on évalue `cond1 and cond2`, on évalue tout d'abord la valeur de la condition `cond1`. Deux cas se présentent alors. Soit `cond1` prend la valeur :

- `False`. Dans ce cas, on peut conclure que la conjonction est fausse sans avoir à évaluer `cond2`.
- `True`. Dans ce cas, il faut évaluer `cond2` pour connaître la valeur de la conjonction.

En écrivant `L[i] >= 0 and i < n`, on teste d'abord si `L[i] >= 0` avant de tester si `i < n`. Cela n'a pas de sens : il faut toujours s'assurer que `i` est un indice acceptable avant d'évaluer `L[i]`. Si ce n'est pas le cas (`i` peut en effet prendre la valeur `n`), on va déclencher l'erreur d'indice habituelle « `list index out of range` ».
Finalement, la bonne écriture est la suivante.

```
1 def tous_positifs(L) :
2     n = len(L) # nom pertinent pour la longueur de L
3     i = 0 # on initialise une variable d'itération
4     while i < n and L[i] >= 0 :
5         i += 1
6     return i == n
```

IV.2.c) De l'utilisation d'une boucle `for` pour un parcours partiel

- Dans le paragraphe précédent, on a codé une fonction permettant de vérifier que tous les éléments d'une liste sont positifs. Pour ce faire, on inspecte les éléments tant qu'ils sont positifs et on sort de la boucle dès qu'on en recontre un strictement négatifs. En fait, si on rencontre un tel élément, on peut renvoyer `False` directement. L'idée que la fonction pourrait faire un renvoi dès que la condition n'est pas vérifiée est possible à mettre en place avec **Python**. Écrivons le code correspondant avant de le commenter.

```
1  def tous_positifs(L) :
2      for i in range(len(L)) :
3          if L[i] < 0 :
4              return False
5      return True
```

- De manière totalement non intuitive, on utilise une boucle `for` alors qu'on n'inspecte pas (forcément) tous les éléments de la liste considérée. Il faut comprendre que lorsqu'on rencontre un `return` la fonction s'arrête et renvoie la valeur mentionnée. Dans ce code, on inspecte les éléments de liste un par un et la découverte d'un élement strictement négatif provoque un renvoi de la fonction. Si un tel élément n'est pas rencontré, alors la boucle se termine. On arrive en ligne 5 seulement si c'est le cas. La fonction doit alors renvoyer `True` pour signaler que la liste ne contient que des éléments positifs.
- Il est possible de faire une dernière petite amélioration à ce code. En effet, l'indice `i` ne nous est guère utile ici.
  On peut donc écrire la dernière version suivante.

```
1  def tous_positifs(L) :
2      for e in L :
3          if e < 0 :
4              return False
5      return True
```

- Ce dernier code est le plus condensé et peut-être le plus simple à écrire car il ne nécessite pas de création et mise à jour d'une variable booléenne. C'est celui qui correspond le mieux à la philosophie du langage **Python**.
- Il est difficile de savoir quelle solution sera retenue aux concours des filières commerciales. Évidemment, tout programme répondant correctement à la question permet d'obtenir la totalité des points alloués à la question. Cependant, en informatique, beaucoup de questions consistent en un code à trou qu'il s'agit de compléter. C'est un exercice difficile qui impose de comprendre l'état d'esprit du concepteur. Il est à noter que les concepteurs ont des maîtrises variés des outils et langages informatiques. Lors des dernières sessions, des attendus de base (nommage cohérent des variables / fonctions, indentation correcte) n'étaient pas respectés. Il est à espérer que le passage à **Python** permettra une montée en gamme en terme de présentation des exercices.

`À RETENIR`

On pourra retenir la possibilité, pour coder certains prédicats, d'un « renvoi sur `False` ». Plus précisément, lorsque le prédicat consiste à vérifier que tous les éléments d'une liste vérifient une propriété donnée, il est possible d'utiliser une boucle `for` et d'en sortir dès la découverte d'un élément ne vérifiant pas cette propriété.

V. Des programmes classiques

V.1. Recherche du maximum dans une liste

V.1.a) Implémentation

- Afin de trouver le maximum d'une liste de nombres :
  - × on crée une variable M qu'on initialise au premier élément de la liste,
  - × on itère sur les éléments de la liste et on met à jour M dès que l'on trouve une valeur plus grande.

  En procédant ainsi, l'invariant de boucle est le suivant : à l'entrée du i$^{\text{ème}}$ tour de boucle, la variable M contient la plus grande des i premières valeurs de la liste. Cela permet de conclure qu'à la fin du parcours de la liste, la variable M contient bien la plus grande valeur de la liste.
- Voici une implémentation de cette fonction.

```
1 def max(L) :
2     M = L[0]
3     for i in range(1, len(L)) :
4         if L[i] > M :
5             M = L[i]
6     return M
```

  On remarque que l'on débute l'itération à l'indice 1. En effet, comme M contient initialement la valeur de L[0], il est inutile de vérifier si L[0] est strictement plus grand que M.
- Comme on cherche une valeur et non un indice, il était aussi possible (souhaitable ?) d'itérer sur les éléments de la liste.

```
1 def max(L) :
2     M = L[0]
3     for e in L :
4         if e > M :
5             M = e
6     return M
```

  En agissant ainsi, on teste si le premier élément de la liste est strictement supérieur à lui-même (premier tour de boucle). Pour éviter cette vérification initule, on peut itérer sur les éléments de la queue de la liste (liste privée de son premier élément).

```
1 def max(L) :
2     M = L[0]
3     for e in L[1:] :
4         if e > M :
5             M = e
6     return M
```

V.1.b) Recherche de l'indice du maximum d'une liste

- Afin de trouver l'indice du maximum d'une liste, on reprend le tout premier programme. Pour découvrir cet indice, il est nécessaire de trouver, au passage, la valeur du maximum de la liste concernée. Cela peut se faire en un parcours de liste en créant une variable pour garder en mémoire la valeur maximale rencontrée lors du parcours ainsi que l'indice de cette valeur. L'invariant de boucle est le suivant : à l'entrée du i$^{\text{ème}}$ tour de boucle, la variable M contient la plus grande des i premières valeurs de la liste et i_M contient l'indice de cet élément.

```
1 def indice_max(L) :
2     M = L[0]
3     i_M = 0
4     for i in range(1, len(L)) :
5         if L[i] > M :
6             M = L[i]
7             i_M = i
8     return i_M
```

- On peut adapter le programme précédent afin de renvoyer l'indice du maximum ainsi que sa valeur. Pour ce faire, il suffit de modifier l'instruction de renvoi : `return (i_M, M)`.

VI. Listes et gestions de la mémoire (CULTURE)

VI.1. Alias d'une liste

- Pour comprendre les différentes notions de copies d'une liste, on s'intéresse tout d'abord au programme suivant.

```
1 a = [1, 2]
2 b = a
3 a[0] = 5
4 print(a)
5 print(b)
```

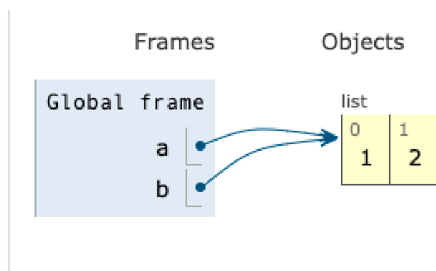
Ce programme permet d'afficher le contenu des listes `a` et `b`. Notons que :

× la liste `a` est initialisée à `[1, 2]`. Son premier terme est ensuite mis à jour en ligne 3. Ainsi, en ligne 4, on s'attend à l'affichage : `[5, 2]`.
C'est bien le cas.

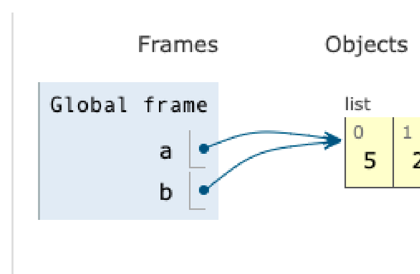
× la liste `b` est initialisée en ligne 2. On lui affecte la valeur de `a`, à savoir `[1, 2]`. Elle n'est pas mise à jour ensuite. Ainsi, en ligne 5, on s'attend à l'affichage : `[1, 2]`.
Ce n'est pas le cas ! En réalité, c'est `[5, 2]` qui est encore affiché.

Que s'est-il passé ? Pour répondre à cette question, il faut s'interroger sur la gestion de la mémoire.

- Pour visualiser plus précisément ce que réalise le code précédent, on va utiliser **Python Tutor**. Il suffit alors de recopier le code est de cliquer sur *Visualize Execution*. Après exécution de la ligne 3 du code, on obtient le graphique variables / objets suivant :



On s'aperçoit alors que `a` et `b` sont 2 noms différents mais qu'ils « pointent » vers le même objet. Cela permet de comprendre le fonctionnement du code précédent. Après exécution de la ligne 5, voici ce qu'on obtient :



On dit alors que `b` est un **alias** de la liste `a`.

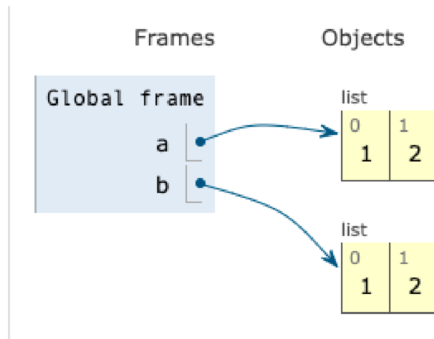
VI.2. Copie superficielle d'une liste

- Dans l'exemple précédent, on a vu que créer un alias ne correspond pas à créer une nouvelle liste mais simplement créer un nouveau nom qui pointe vers le même objet liste. Si on veut créer une nouvelle liste, on peut procéder comme suit.

```
1 a = [1, 2]
2 b = a[:]
3 a[0] = 5
4 print(a)
5 print(b)
```

- En ligne 2, on crée une variable `b`. Celle-ci est affectée à la tranche `a[:]` qui correspond à tout le contenu de la liste `a`. On l'a vu au moment du **slicing** : les tranches (= slices) créées sont de nouveaux objets. On a donc apparemment bien réussi à faire ce que l'on souhaitait, à savoir créer une copie de la liste initiale qui est un objet différent de la liste initiale.

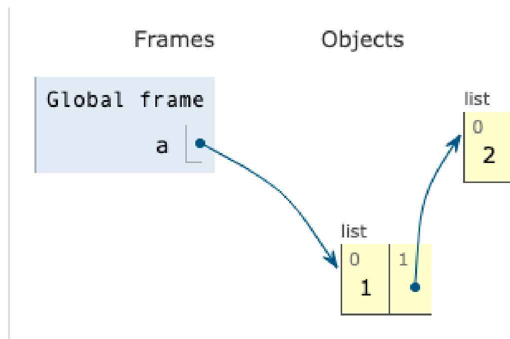
Après exécution de la ligne 2, voici le schéma variables / objets :



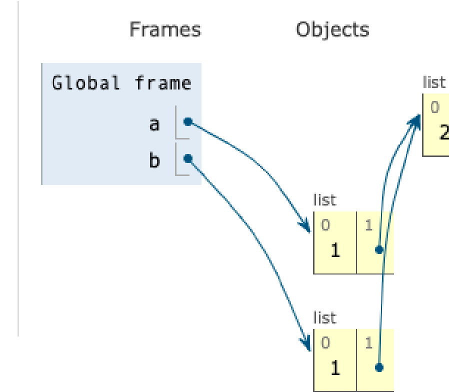
- Dans le point précédent on a créé une **copie superficielle** de la liste `a` (on parle de **shallow copy** en anglais). Le qualificatif **superficielle** peut sembler étonnant. Pour mieux le comprendre, considérons le programme suivant.

```
1 a = [1, [2]]
2 b = a[:]
3 a[1][0] = 5
4 print(a)
5 print(b)
```

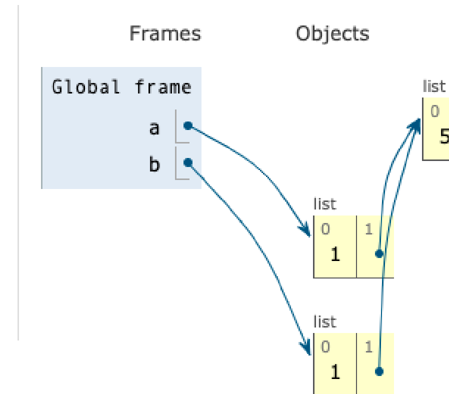
Après exécution de la ligne 1, on a le schéma variables / objets suivant :



Après exécution de la ligne 2, ce schéma est mis à jour comme suit.



On comprend mieux le qualificatif « superficielle ». Une nouvelle liste est bien créée. Les objets simples (entiers et flottants) sont bien copiés dans cette nouvelle liste. Le problème survient avec les objets composés. On ne crée pas un nouvel objet pour la liste `[2]` avec une référence qui pointerait vers ce nouvel objet. On crée au contraire, dans la liste `b`, une référence vers l'objet initial `[2]`. De sorte qu'en sortie de ligne 3, on obtient le schéma suivant :



VI.3. Copie profonde d'une liste

- On a vu dans le paragraphe précédent que la notion de copie superficielle crée une copie qui « s'arrête au premier niveau ». Si l'on souhaite que la liste copiée ne partage aucune référence avec la liste initiale, il faudrait que les objets composés de `a` donnent naissance à de nouveaux objets lors de la copie. Attention, car ces objets composés (il s'agissait de `[2]` dans l'exemple précédent) peuvent eux-mêmes contenir des objets composés qui peuvent eux-mêmes contenir des objets composés et ainsi de suite (par exemple `[2, [1, [3, 5]]]`). Si l'on veut empêcher qu'il y ait des références partagées par `a` et `b`, il faut que la copie se fasse à tous les niveaux de profondeur. C'est pourquoi l'on parle de `copie profonde` (on parle de **deepcopy** en anglais). Si l'on effectue une copie profonde de `a`, un nouvel objet est créé. On insère alors récursivement dans ce nouvel objet des copies des objets trouvés dans la liste original. C'est pourquoi on parle aussi de **copie récursive**.
- Afin de réaliser une copie profonde, on utilise généralement la bibliothèque `copy` de **Python**.

```
1 import copy
2 a = [1, [2]]
3 b = copy.deepcopy(a)
4 a[1][0] = 5
5 print(a)
6 print(b)
```

Voici le schéma après exécution de la ligne 3.