Listes et séquences

Structures de données

$\longrightarrow$ une façon de ranger et d'ordonner des objets.

Les listes sont les principales structures de données en Python. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Structures de données

$\longrightarrow$ une façon de ranger et d'ordonner des objets.

Les listes sont les principales structures de données en Python. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Structures de données

$\longrightarrow$ une façon de ranger et d'ordonner des objets.

Les listes sont les principales structures de données en Python. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Lorsque la quantité de mémoire allouée est fixée au moment de la création la structure de données est statique.
Lorsque l'attribution de la mémoire peut varier au cours du temps, la structure de données est dynamique.
Lorsque le contenu d'une structure de donnée est modifiable, la structure de donnée est mutable.

Structures de données

⟶ une façon de ranger et d'ordonner des objets.

Les listes sont les principales structures de données en Python. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

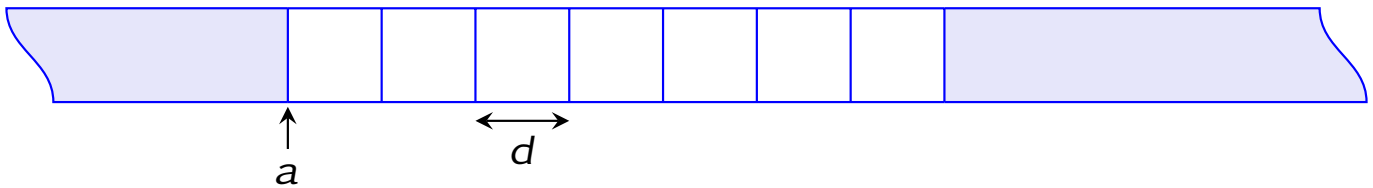
- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- les structures linéaires (tableaux, listes chaînées) ;
- les *matrices* ou *tableaux multidimensionnels* ;
- les *structures arborescentes* (arbres binaires) ;
- les *structures relationnelles* (bases de données ou graphes).

Tableaux et listes chaînées

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire :

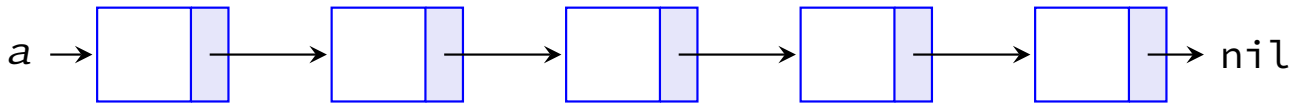


- un tableau est une structure de donnée statique mutable ;
- les éléments du tableau sont accessibles en lecture et en écriture en temps constant.

Les tableaux existent en Python : c'est la classe *array* fournie par la bibliothèque `Numpy`.

Tableaux et listes chaînées

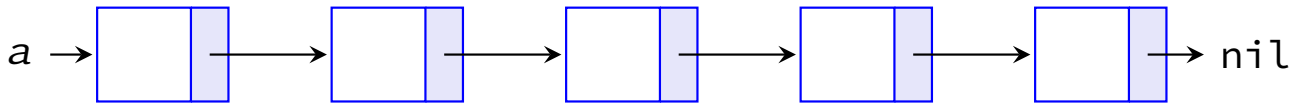
Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



- une liste est une structure de donnée dynamique ;
- le n^{e} élément d'une liste est accessible en temps proportionnel à n.

Tableaux et listes chaînées

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



- une liste est une structure de donnée dynamique ;
- le n^e élément d'une liste est accessible en temps proportionnel à n.

Les listes n'existent pas en Python : la classe *list* n'est pas une liste chaînée mais uns structure de données qui concilie les avantages des tableaux et des listes chaînées.

> *c'est une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.*

Construction d'une liste

Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

Construction d'une liste

Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

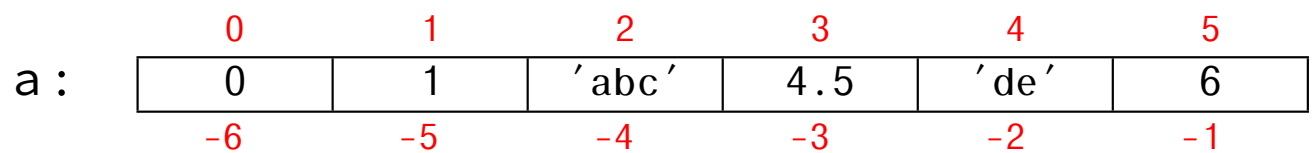
```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

On accède à chaque élément de la liste par son index, qui débute à 0.

```
In [1]: a[2]
Out[1]: 'abc'

In [2]: b[3][1]
Out[2]: 2
```



Construction d'une liste

Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

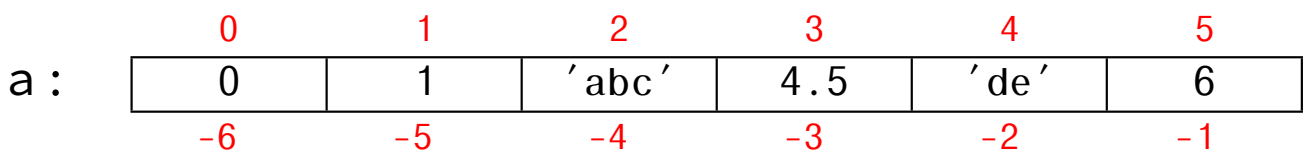
```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

Lorsque l'index est négatif, le décompte est pris en partant de la fin. Ainsi, les éléments d'indices $-k$ et $\ell - k$ sont les mêmes.

```
In [4]: a[-2]
Out[4]: 'de'
```

On calcule la longueur d'une liste à l'aide de la fonction **len**.

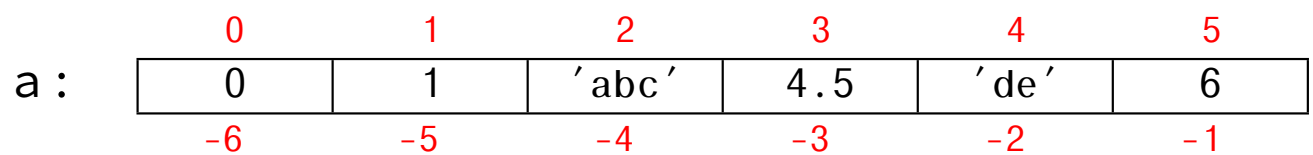


Slicing

Python permet le « découpage en tranches » : si `l` est une liste, alors `l[i:j]` est une nouvelle liste constituée des éléments dont les index sont compris entre i et $j-1$:

```
In [5]: a[1:5]
Out[5]: [1, 'abc', 4.5, 'de']

In [6]: a[1:-1]
Out[6]: [1, 'abc', 4.5, 'de']
```



Slicing

Python permet le « découpage en tranches » : si `l` est une liste, alors `l[i:j]` est une nouvelle liste constituée des éléments dont les index sont compris entre i et $j - 1$:

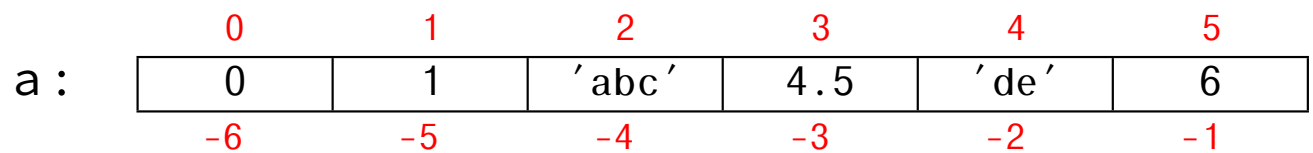
```
In [5]: a[1:5]
Out[5]: [1, 'abc', 4.5, 'de']

In [6]: a[1:-1]
Out[6]: [1, 'abc', 4.5, 'de']
```

Lorsque i est absent, il est pris par défaut égal à 0 ; lorsque j est absent, il est pris par défaut égal à la longueur de la liste.

```
In [7]: a[:4]
Out[7]: [0, 1, 'abc', 4.5]

In [8]: a[-3:]
Out[8]: [4.5, 'de', 6]
```



Slicing

Python permet le « découpage en tranches » : si `l` est une liste, alors `l[i:j]` est une nouvelle liste constituée des éléments dont les index sont compris entre i et $j-1$:

```
In [5]: a[1:5]
Out[5]: [1, 'abc', 4.5, 'de']

In [6]: a[1:-1]
Out[6]: [1, 'abc', 4.5, 'de']
```

Lorsque i est absent, il est pris par défaut égal à 0 ; lorsque j est absent, il est pris par défaut égal à la longueur de la liste.

```
In [7]: a[:4]
Out[7]: [0, 1, 'abc', 4.5]

In [8]: a[-3:]
Out[8]: [4.5, 'de', 6]
```

À retenir `l[:n]` calcule la liste des n premiers éléments et `l[-n:]` la liste des n derniers.

Slicing

Sélection partielle

La syntaxe `lst[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection :

```
In [9]: l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [10]: l[2:9:3]
Out[10]: [2, 5, 8]

In [11]: l[3::2]
Out[11]: [3, 5, 7, 9]

In [12]: l[:-1:2]
Out[12]: [0, 2, 4, 6, 8]
```

Slicing

Sélection partielle

La syntaxe `lst[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection :

```
In [9]: l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [10]: l[2:9:3]
Out[10]: [2, 5, 8]

In [11]: l[3::2]
Out[11]: [3, 5, 7, 9]

In [12]: l[:-1:2]
Out[12]: [0, 2, 4, 6, 8]
```

Il est possible de choisir un pas négatif :

```
In [13]: l[-1:0:-1]
Out[13]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Création d'une liste

par énumération

Les listes sont de type *list* et la fonction **`list`** convertit lorsque c'est possible un objet d'un certain type vers le type *list*. C'est le cas en particulier des énumérations produites par la fonction **`range`** :

```
In [14]: list(range(11))
Out[14]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [15]: list(range(13, 2, -3))
Out[15]: [13, 10, 7, 4]
```

Création d'une liste

par énumération

Les listes sont de type *list* et la fonction **list** convertit lorsque c'est possible un objet d'un certain type vers le type *list*. C'est le cas en particulier des énumérations produites par la fonction **range** :

```
In [14]: list(range(11))
Out[14]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [15]: list(range(13, 2, -3))
Out[15]: [13, 10, 7, 4]
```

La fonction **list** appliquée à une chaîne de caractères renvoie une liste de caractères :

```
In [16]: list("Louis-Le-Grand")
Out[16]: ['L', 'o', 'u', 'i', 's', '-', 'L', 'e', '-', 'G', 'r',
'a', 'n', 'd']
```

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$$\left\{x \in [\![0, 10]\!] \;\middle|\; x^2 \leqslant 50\right\}$$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$$\left\{x \in [\![0,10]\!] \;\middle|\; x^2 \leqslant 50\right\}$$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Produit cartésien de deux listes :

```
In [18]: a, b = [1, 3, 5], [2, 4, 6]

In [19]: [(x, y) for x in a for y in b]
Out[19]: [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6), (5, 2), (5, 4),
(5, 6)]
```

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$$\left\{x \in [\![0,10]\!] \;\middle|\; x^2 \leqslant 50\right\}$$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Produit cartésien de deux listes :

```
In [18]: a, b = [1, 3, 5], [2, 4, 6]

In [19]: [(x, y) for x in a for y in b]
Out[19]: [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6), (5, 2), (5, 4),
(5, 6)]
```

Un triplet pythagoricien est un triplet $(x,y,z) \in (\mathbb{N}^*)^3$ tels que $x^2+y^2=z^2$.

```
In [20]: [(x, y, z) for x in range(1, 20) for y in range(x, 20) for z in
range(y, 20) if x * x + y * y == z * z]
Out[20]: [(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
```

$$\left\{(x,y,z) \in [\![1,19]\!]^3 \;\middle|\; x \leqslant y \leqslant z \text{ et } x^2+y^2=z^2\right\}.$$

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$$\left\{x \in [\![0,10]\!] \;\middle|\; x^2 \leqslant 50\right\}$$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Produit cartésien de deux listes :

```
In [18]: a, b = [1, 3, 5], [2, 4, 6]

In [19]: [(x, y) for x in a for y in b]
Out[19]: [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6), (5, 2), (5, 4),
(5, 6)]
```

Un triplet pythagoricien est un triplet $(x,y,z) \in (\mathbb{N}^*)^3$ tels que $x^2+y^2=z^2$.

```
In [20]: [(x, y, z) for x in range(1, 20) for y in range(x, 20) for z in
range(y, 20) if x * x + y * y == z * z]
Out[20]: [(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
```

Attention, l'énumération peut être longue !

Opérations sur les listes

L'opération de concaténation se note + :

```
In [1]: [2, 4, 6, 8] + [1, 3, 5, 7]
Out[1]: [2, 4, 6, 8, 1, 3, 5, 7]
```

Opérations sur les listes

L'opération de concaténation se note + :

```
In [1]: [2, 4, 6, 8] + [1, 3, 5, 7]
Out[1]: [2, 4, 6, 8, 1, 3, 5, 7]
```

L'opérateur de duplication se note `*` ; si `lst` est une liste et `n` un entier alors `lst * n` est équivalent à `lst + lst + ... + lst` (*n* fois).

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

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.

- Modification d'un élément :

```
In [1]: l = list(range(11))

In [2]: l[3] = 'a'

In [3]: l
Out[3]: [0, 1, 2, 'a', 4, 5, 6, 7, 8, 9, 10]
```

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.

- Modification d'un élément :

```
In [1]: l = list(range(11))

In [2]: l[3] = 'a'

In [3]: l
Out[3]: [0, 1, 2, 'a', 4, 5, 6, 7, 8, 9, 10]
```

- Modification de plusieurs éléments :

```
In [4]: l[3:7] = 'abc'

In [5]: l
Out[5]: [0, 1, 2, 'a', 'b', 'c', 7, 8, 9, 10]

In [6]: l = list(range(10))

In [7]: l[1::2] = l[0::2]

In [8]: l
Out[8]: [0, 0, 2, 2, 4, 4, 6, 6, 8, 8]
```

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.

- Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))

In [10]: del l[3:6]

In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.

- Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))

In [10]: del l[3:6]

In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

- Insertion d'un élément :

```
In [12]: l = ['a', 'b', 'c', 'd']

In [13]: l.append('e')

In [14]: l
Out[14]: ['a', 'b', 'c', 'd', 'e']

In [15]: l.insert(2, 'x')

In [16]: l
Out[16]: ['a', 'b', 'x', 'c', 'd', 'e']
```

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

- la méthode `remove(x)` supprime la première occurence de `x` dans la liste ;

```
In [17]: l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

In [18]: l.remove(4)

In [19]: l
Out[19]: [1, 2, 3, 5, 1, 2, 3, 4, 5]
```

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

- la méthode `remove(x)` supprime la première occurence de `x` dans la liste ;
- la méthode `pop(i)` supprime l'élément d'indice *i* et le retourne ;

```
In [17]: l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

In [18]: l.remove(4)

In [19]: l
Out[19]: [1, 2, 3, 5, 1, 2, 3, 4, 5]

In [20]: l.pop(-1)
Out[20]: 5

In [21]: l
Out[21]: [1, 2, 3, 5, 1, 2, 3, 4]
```

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

- la méthode `remove(x)` supprime la première occurence de `x` dans la liste ;
- la méthode `pop(i)` supprime l'élément d'indice *i* et le retourne ;
- la méthode `reverse()` inverse l'ordre des éléments d'une liste ;

```
In [22]: l = [1, 2, 3, 4, 1, 2, 3, 4]

In [23]: l.reverse()

In [24]: l
Out[24]: [4, 3, 2, 1, 4, 3, 2, 1]
```

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

- la méthode `remove(x)` supprime la première occurence de `x` dans la liste ;
- la méthode `pop(i)` supprime l'élément d'indice *i* et le retourne ;
- la méthode `reverse()` inverse l'ordre des éléments d'une liste ;
- la méthode `sort()` trie la liste (par ordre croissant).

```
In [22]: l = [1, 2, 3, 4, 1, 2, 3, 4]

In [23]: l.reverse()

In [24]: l
Out[24]: [4, 3, 2, 1, 4, 3, 2, 1]

In [25]: l.sort()

In [26]: l
Out[26]: [1, 1, 2, 2, 3, 3, 4, 4]
```

Méthodes associées aux listes

append et `insert` sont des méthodes qui mutent les listes ; il en existe de nombreuses autres.

- la méthode `remove(x)` supprime la première occurence de `x` dans la liste ;
- la méthode `pop(i)` supprime l'élément d'indice *i* et le retourne ;
- la méthode `reverse()` inverse l'ordre des éléments d'une liste ;
- la méthode `sort()` trie la liste (par ordre croissant).

En général, les méthodes qui s'appliquent aux listes mutent celles-ci (par exemple `sort`), contrairement aux fonctions qui calculent de nouvelles listes (par exemple **`sorted`**).

```
In [27]: l = [1, 2, 3, 4, 1, 2, 3, 4]

In [28]: sorted(l)
Out[28]: [1, 1, 2, 2, 3, 3, 4, 4]

In [29]: l
Out[29]: [1, 2, 3, 4, 1, 2, 3, 4]
```

Séquences

Chaînes de caractères et listes sont des séquences : composées d'un nombre fini d'éléments indexés.
Si seq est une séquence, alors :

- `seq[k]` désigne l'élément d'indice k de cette séquence (la numérotation commençant à 0) ;
- on peut effectuer des coupes à l'aide de la technique du slicing ;
- la fonction `len` donne la longueur de la séquence ;
- la concaténation et la duplication se notent respectivement `+` et `*`.

Séquences

Chaînes de caractères et listes sont des séquences : composées d'un nombre fini d'éléments indexés.
Nous utiliserons les séquences suivantes :

- les chaînes de caractères, qui sont des successions de caractères délimités par des guillemets (simples ou doubles) ;
- les listes, qui sont des successions d'objets séparés par des virgules et délimités par des crochets ;
- les tuples, qui sont des successions d'objets séparés par des virgules et délimités par des parenthèses.

Séquences

Chaînes de caractères et listes sont des séquences : composées d'un nombre fini d'éléments indexés.
Nous utiliserons les séquences suivantes :

- les chaînes de caractères, qui sont des successions de caractères délimités par des guillemets (simples ou doubles) ;
- les listes, qui sont des successions d'objets séparés par des virgules et délimités par des crochets ;
- les tuples, qui sont des successions d'objets séparés par des virgules et délimités par des parenthèses.

Les fonctions `str`, `list`, `tuple` permettent les conversions :

```
In [1]: list('abcde')
Out[1]: ['a', 'b', 'c', 'd', 'e']

In [2]: tuple([1, 2, 3, 4, 5])
Out[2]: (1, 2, 3, 4, 5)
```

Séquences

Mutabilité

Séquence mutable : on peut modifier ou supprimer les éléments de cette séquence (listes).

```
In [1]: lst = [1, 2, 3]

In [2]: lst[0] = 4

In [3]: lst
Out[3]: [4, 2, 3]
```

Séquences

Mutabilité

Séquence mutable : on peut modifier ou supprimer les éléments de cette séquence (listes).

```
In [1]: lst = [1, 2, 3]

In [2]: lst[0] = 4

In [3]: lst
Out[3]: [4, 2, 3]
```

Séquence non mutable : on peut ne modifier cette séquence (chaînes de caractères et tuples).

```
In [4]: tup = (1, 2, 3)

In [5]: tup[0] = 4
TypeError: 'tuple' object does not support item assignment

In [6]: chr = 'abc'

In [7]: chr[0] = 'd'
TypeError: 'str' object does not support item assignment
```

Séquences

Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [3]: l1 = [1, 2, 3]

In [4]: id(l1)
Out[4]: 4448440520

In [5]: tup1 = (1, 2, 3)

In [6]: id(tup1)
Out[6]: 4448266352
```

Séquences

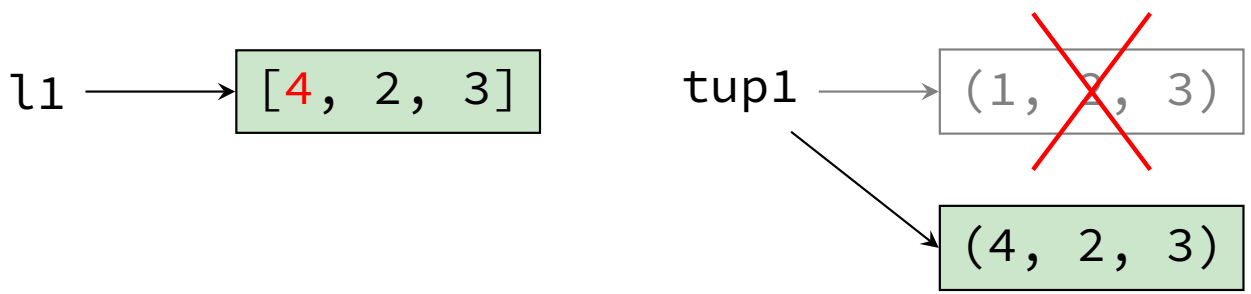
Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [7]: l1[0] = 4

In [8]: id(l1)
Out[8]: 4448440520

In [9]: tup1 = (4, 2, 3)

In [10]: id(tup1)
Out[10]: 4448266640
```



Une liste est mutable, on peut la modifier sans pour autant créer une nouvelle référence vers l'objet, contrairement à un tuple.

Séquences

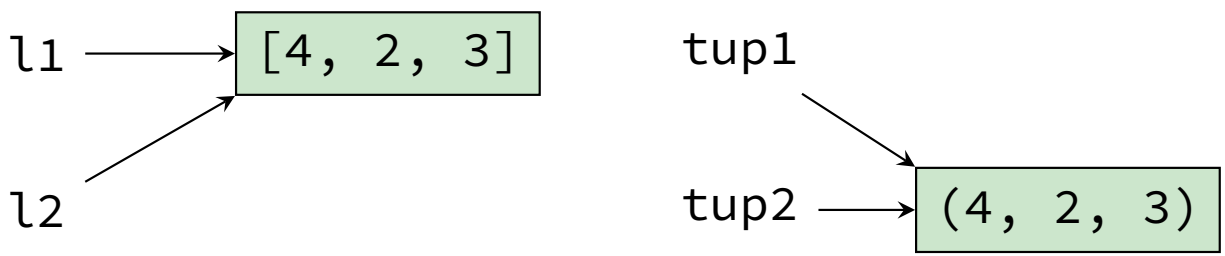
Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [11]: l2 = l1

In [12]: id(l1) == id(l2)
Out[12]: True

In [13]: tup2 = tup1

In [14]: id(tup1) == id(tup2)
Out[14]: True
```



lorsque `var1` est une variable, l'instruction `var2 = var1` crée un nouveau référencement vers le même emplacement en mémoire.

Séquences

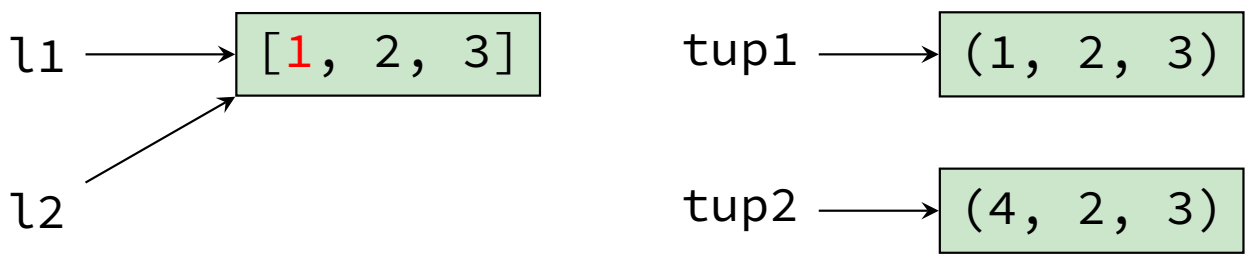
Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [15]: l1[0] = 1

In [16]: l2
Out[16]: [1, 2, 3]

In [17]: tup1 = (1, 2, 3)

In [18]: tup2
Out[18]: (4, 2, 3)
```



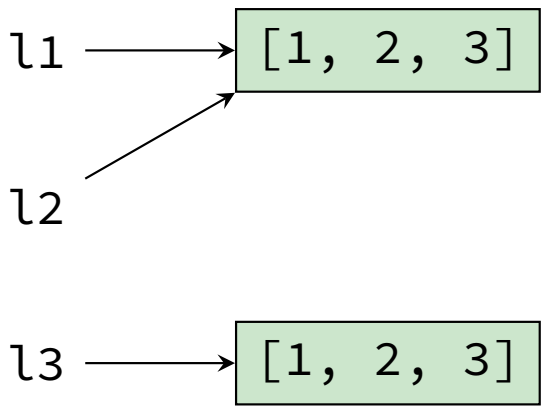
L'instruction `l1[0] = 1` modifie aussi la variable `l2` puisque ces deux noms référencent la même adresse.

Copie d'un objet mutable

Un nouveau référencement n'est pas suffisant pour copier un objet mutable ; il est nécessaire de le recréer entièrement, par exemple à l'aide du slicing `[:]`.

```
In [19]: l3 = l1[:]

In [20]: id(l1) == id(l3)
Out[20]: False
```



De manière équivalente on peut écrire :

`l3 = l1.copy()` ou `l3 = [x for x in l1]`.

Copie d'un objet mutable

Copie profonde

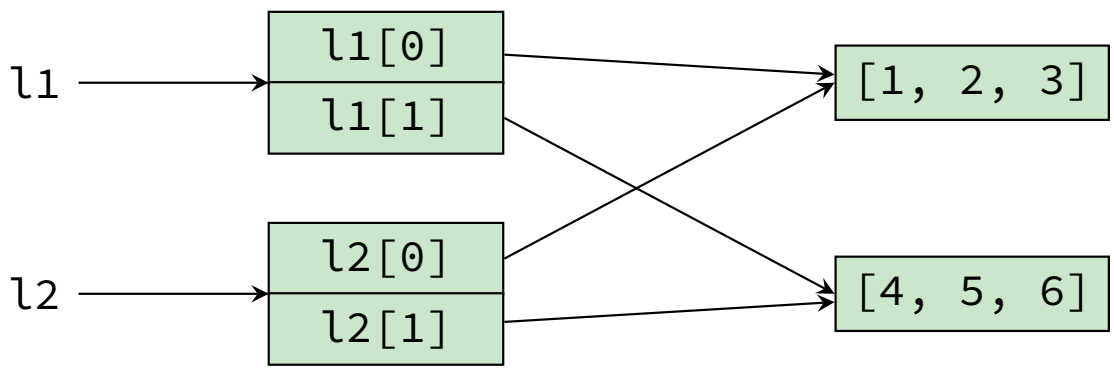
Si une liste contient des objets eux-mêmes mutables, la copie de premier niveau est insuffisante.

```
In [21]: l1 = [[1, 2, 3], [4, 5, 6]]

In [22]: l2 = l1[:]

In [23]: id(l1) == id(l2)
Out[23]: False

In [24]: id(l1[0]) == id(l2[0])
Out[24]: True
```



`l1` et `l2` référencent deux emplacements mémoires distincts, mais `l1[i]` et `l2[i]` référencent toujours le même objet mutable.

Copie d'un objet mutable

Copie profonde

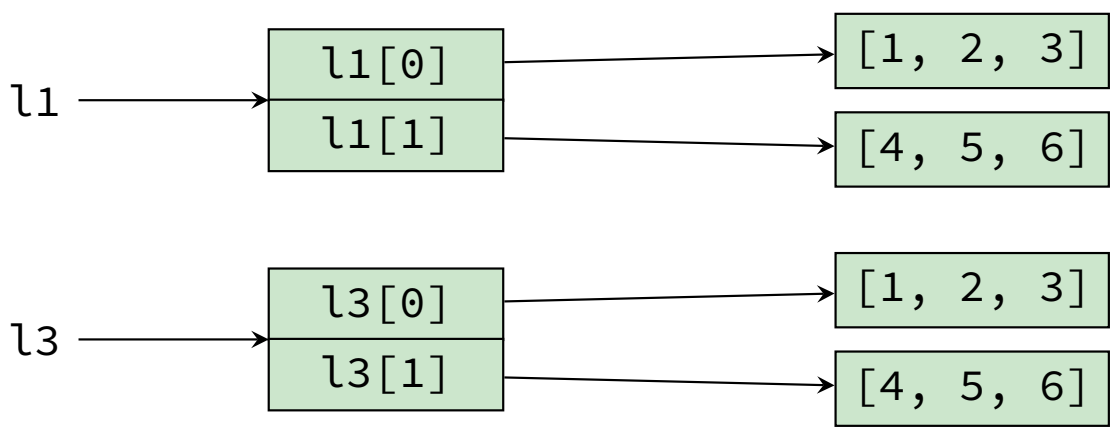
Si une liste contient des objets eux-mêmes mutables, la copie de premier niveau est insuffisante.

```
In [25]: from copy import deepcopy

In [26]: l3 = deepcopy(l1)

In [27]: id(l1) == id(l3)
Out[27]: False

In [28]: id(l1[0]) == id(l3[0])
Out[28]: False
```



Parcours d'une liste

Parcours complet par boucle énumérée

La syntaxe `for x in seq:` permet de parcourir tous les éléments x d'une séquence seq.

Exemple : calcul de la somme des éléments d'une liste (ou d'un tuple).

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

Parcours d'une liste

Parcours complet par boucle énumérée

La syntaxe `for x in seq:` permet de parcourir tous les éléments x d'une séquence seq.
Exemple : calcul de la somme des éléments d'une liste (ou d'un tuple).

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

- Calcul de la moyenne $\overline{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$.

```
def moyenne(l):
    s = 0
    for x in l:
        s += x
    return s / len(l)
```

Parcours d'une liste

Parcours complet par boucle énumérée

La syntaxe `for x in seq:` permet de parcourir tous les éléments x d'une séquence seq.
Exemple : calcul de la somme des éléments d'une liste (ou d'un tuple).

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

- Calcul de la variance $v = \frac{1}{n}\sum_{k=1}^{n}(x_k - \overline{x})^2 = \frac{1}{n}\sum_{k=1}^{n} x_k^2 - \overline{x}^2$.

```
def variance(l):
    n = len(l)
    s = c = 0
    for x in l:
        s += x
        c += x * x
    return c/n - (s/n) * (s/n)
```

Parcours d'une liste

Calcul du maximum

- Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

Parcours d'une liste

Calcul du maximum

- Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

- Calcul de l'indice de l'élément maximal : on parcourt la liste des indices.

```
def indice_max(l):
    i, m = 0, l[0]
    for k in range (1, len(l)):
        if l[k] > m:
            i, m = k, l[k]
    return i
```

Parcours d'une liste

Calcul du maximum

- Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

- La fonction `enumerate` renvoie une liste de tuples `(indice, valeur)` lorsqu'on l'applique à un objet énumérable.

```
def indice_du_max(l):
    i, m = 0, l[0]
    for (k, x) in enumerate(l):
        if x > m:
            i, m = k, x
    return i
```

Parcours d'une liste

Parcours incomplet (recherche d'un élément)

Un problème fréquent : déterminer la présence ou non d'un élément x dans une liste l.

- **Première méthode** : parcourir la liste par une boucle conditionnelle.

```
def cherche(x, l):
    k, rep = 0, False
    while k < len(l) and not rep:
        rep = l[k] == x
        k += 1
    return rep
```

Parcours d'une liste

Parcours incomplet (recherche d'un élément)

Un problème fréquent : déterminer la présence ou non d'un élément x dans une liste l.

- **Première méthode** : parcourir la liste par une boucle conditionnelle.

```
def cherche(x, l):
    k, rep = 0, False
    while k < len(l) and not rep:
        rep = l[k] == x
        k += 1
    return rep
```

- **Deuxième méthode** : interrompre une boucle énumérée dès l'élément trouvé.

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

Parcours d'une liste

Parcours incomplet (recherche d'un élément)

Un problème fréquent : déterminer la présence ou non d'un élément x dans une liste l.

- **Première méthode** : parcourir la liste par une boucle conditionnelle.

```
def cherche(x, l):
    k, rep = 0, False
    while k < len(l) and not rep:
        rep = l[k] == x
        k += 1
    return rep
```

- **Deuxième méthode** : interrompre une boucle énumérée dès l'élément trouvé.

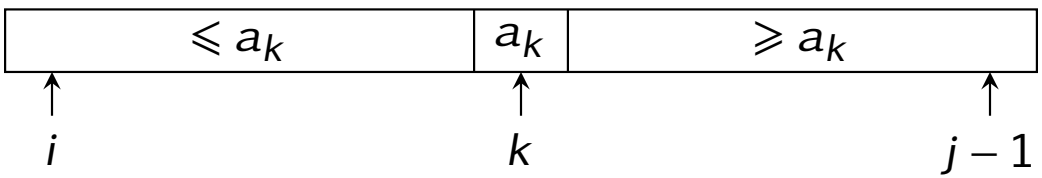
```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

- **Troisième méthode** : `(x in l)` renvoie le booléen recherché !

Recherche dichotomique

Pour chercher x dans la liste triée par ordre croissant $[a_i, \ldots, a_{j-1}]$, on applique le principe dichotomique : comparer x et a_k, avec $k = \left\lfloor \frac{i+j}{2} \right\rfloor$.

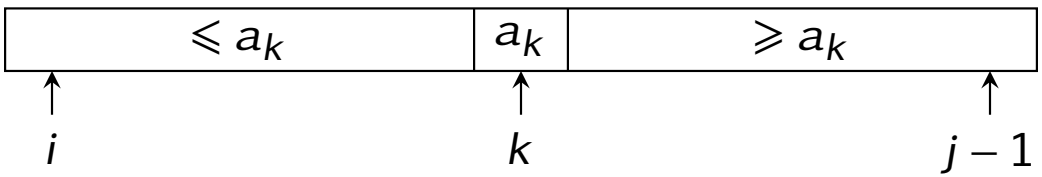
- si $x < a_k$ le recherche se poursuit dans $[a_i, \ldots, a_{k-1}]$;
- si $x = a_k$ la recherche est terminée ;
- si $x > a_k$ la recherche se poursuit dans $[a_{k+1}, \ldots, a_{j-1}]$.



Recherche dichotomique

Pour chercher x dans la liste triée par ordre croissant $[a_i, \ldots, a_{j-1}]$, on applique le principe dichotomique : comparer x et a_k, avec $k = \left\lfloor \frac{i+j}{2} \right\rfloor$.

- si $x < a_k$ le recherche se poursuit dans $[a_i, \ldots, a_{k-1}]$;
- si $x = a_k$ la recherche est terminée ;
- si $x > a_k$ la recherche se poursuit dans $[a_{k+1}, \ldots, a_{j-1}]$.

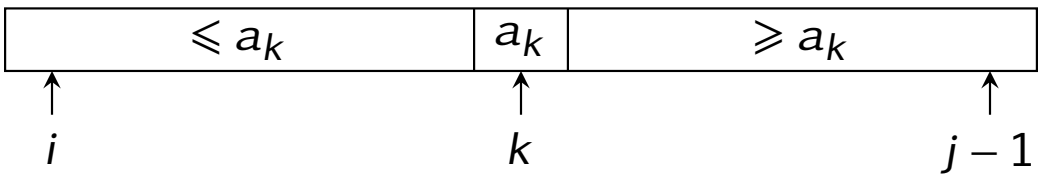



```
def cherche_dicho(x, l):
    i, j = 0, len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

Recherche dichotomique

Pour chercher x dans la liste triée par ordre croissant $[a_i, \ldots, a_{j-1}]$, on applique le principe dichotomique : comparer x et a_k, avec $k = \left\lfloor \frac{i+j}{2} \right\rfloor$.

- si $x < a_k$ le recherche se poursuit dans $[a_i, \ldots, a_{k-1}]$;
- si $x = a_k$ la recherche est terminée ;
- si $x > a_k$ la recherche se poursuit dans $[a_{k+1}, \ldots, a_{j-1}]$.



c_n : nombre de comparaison effectuées dans le pire des cas. Alors :

$$c_n = 1 + c_{\lfloor n/2 \rfloor}.$$

Si $n = (b_p \cdots b_1 b_0)_2$ alors $\left\lfloor \frac{n}{2} \right\rfloor = (b_p \cdots b_2 b_1)_2$ donc : $c_n = p + c_1 = p + 1$.
On a : $(100\cdots00)_2 \leqslant n \leqslant (111\cdots11)_2$ donc $2^p \leqslant n < 2^{p+1}$, d'où :

$$p = \lfloor \log_2(n) \rfloor.$$