Chapitre 5
Listes et séquences

Une *structure de données* est une façon de ranger et d'ordonner des objets. Il en existe de plusieurs types, qui se distinguent par la façon d'accéder aux éléments de la structure de données et par la façon de modifier cette dernière (en ajoutant ou en ôtant des éléments).
La principale structure de données en Python est assurément la classe *list*. Il s'agit d'une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les *tableaux* et les *listes chaînées*, qui n'existent pas en tant que telles en Python. Nous allons commencer par décrire les caractéristiques principales de ces deux structures, avant d'observer en quoi la classe *list* s'en s'inspire.

1. Structures de données linéaires

Dans son acceptation la plus générale, une *structure de données* spécifie la façon de représenter en mémoire machine les données d'un problème à résoudre en décrivant :

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

Dans certains cas, la quantité de mémoire allouée à la structure de donnée est fixée au moment de la création de celle-ci et ne peut plus être modifiée ensuite ; on parle alors de structure de données *statique*. Dans d'autres cas l'attribution de la mémoire nécessaire est effectuée pendant le déroulement de l'algorithme et peut donc varier au cours de celui-ci ; il s'agit alors de structure de données *dynamique*. Enfin, lorsque le contenu d'une structure de donnée est modifiable, on parle de structure de donnée *mutable*.
Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- les *structures linéaires* : il s'agit essentiellement des structures représentables par des suites finies ordonnées ; on y trouve les tableaux et les listes chaînées, qui vont nous intéresser par la suite ;
- les *matrices* ou *tableaux multidimensionnels* ;
- les *structures arborescentes* (en particulier les arbres binaires) ;
- les *structures relationnelles* (bases de données ou graphes pour les relations binaires).

Nous nous intéresserons avant tout aux deux premières de ces familles.

1.1 Tableaux

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

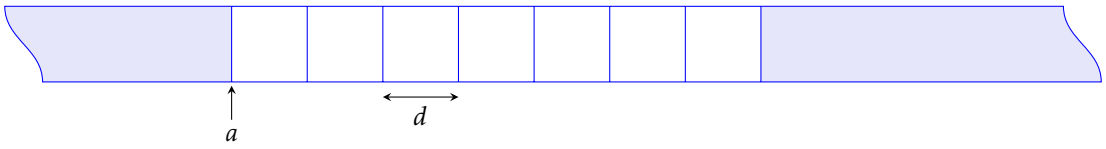


Figure 1 – Une représentation d'un tableau en mémoire.

Puisque tous les emplacements sont de même type, ils occupent tous le même nombre d de cases mémoire ; connaissant l'adresse a de la première case du tableau, on accède *en coût constant* à l'adresse de la case d'indice k en calculant $a + kd$. En revanche, ce type de structure est statique : une fois un tableau créé, la taille de ce dernier ne peut plus être modifiée faute de pouvoir garantir qu'il y a encore un espace mémoire disponible au delà de la dernière case. En résumé :

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

Remarque. Le module Numpy propose une implémentation des tableaux que nous utiliserons au second semestre : la classe *array*.

1.2 Listes chaînées

Les *listes chaînées* associent à chaque donnée (de même type) un pointeur indiquant la localisation dans la mémoire de la donnée suivante (à l'exception de la dernière, qui pointe vers une valeur particulière indiquant la fin de la liste).

FIGURE 2 – Une représentation d'une liste en mémoire.

Dans une liste chaînée, il est impossible de connaître à l'avance l'adresse d'une case en particulier, à l'exception de la première. Pour accéder à la n^e case il faut donc parcourir les $n-1$ précédentes : le coût de l'accès à une case est proportionnelle à la distance qui la sépare de la tête de la liste. En contrepartie, ce type de structure est dynamique : une fois la liste crée, il est toujours possible de modifier un pointeur pour insérer une case supplémentaire. En résumé :

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

1.3 La classe *list* en PYTHON

Contrairement à ce que pourrait laisser croire son nom, la classe *list* n'est pas une liste chaînée au sens qu'on vient de lui donner, mais une structure de donnée plus complexe qui cherche à concilier les avantages des tableaux et des listes chaînées, à savoir :

être une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.

Dans la description de cette classe qui va suivre, on pourra distinguer les opérations qui relèvent de la structure de tableau (accès direct aux éléments) de ceux qui relèvent de la structure de liste chaînée (ajout/suppression d'éléments).

2. Construction, modification et accès à une liste

2.1 Définition et accès aux éléments

Une liste est une structure de données linéaire, les objets étant enclos par des crochets et séparés par des virgules. Par exemple,

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

est une liste qui comporte 6 éléments. Comme on peut le constater, une liste peut contenir une collection hétérogène d'objets (des entiers, des flottants, des chaînes de caractères...).
Notons déjà qu'une liste étant elle-même un objet PYTHON, il est tout à fait possible qu'une liste contienne des listes parmi ses éléments :

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

On accède à chaque élément individuel de la liste par l'intermédiaire de son index. Attention, le premier élément de la liste possède l'index 0 (tout comme les chaînes de caractères). Pour connaître l'index du dernier élément, on peut calculer la longueur d'une liste à l'aide de la fonction `len` ; l'index du dernier élément d'une liste nommée `l` est donc `len(l) - 1`.

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

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

In [3]: len(b)
Out[3]: 4
```

Lorsque l'index est négatif, le décompte est pris en partant de la fin ; ainsi le dernier élément porte l'index –1, l'avant-dernier l'index –2, etc [1].

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

En d'autres termes, si k est un entier non nul, l'élément d'indice $-k$ coïncide avec l'élément d'indice $\ell - k$, où ℓ désigne la longueur de la liste.

2.2 Slicing

À l'instar des chaînes de caractères, PYTHON permet d'effectuer des coupes (le *slicing*) : si `l` est une liste, alors `l[i:j]` crée 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 l'index 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]
```

On peut observer sur les exemples ci-dessus que `l[:n]` permet d'obtenir les n premiers éléments de la liste et `l[-n:]` les n derniers.

Sélection partielle
Si nécessaire, la syntaxe `l[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection. La commande `l[debut:fin:pas]` donne tous les éléments de la liste dont les index sont compris entre `debut` (au sens large) et `fin` (au sens strict) et espacés d'un pas égal à `pas`. Par exemple :

```
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[::2]
Out[12]: [0, 2, 4, 6, 8]
```

Il est même possible de choisir un pas négatif, auquel cas la liste est parcourue à rebours. En particulier, `l[::-1]` retourne une nouvelle liste dont l'ordre des éléments est inversé [2] :

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

2.3 Création d'une liste par compréhension

Les listes sont de type *list* et à l'instar des autres types déjà rencontrés, il existe une fonction `list` qui 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` [3]. Par exemple :

1. Les conventions d'indexation sont identiques à celles des chaînes de caractères (cf. chapitre 2).
2. On observera que pour un pas négatif les valeurs par défaut de début et de fin sont inversées.
3. Rappelons que `range(i, j, p)` énumère les entiers compris entre i (au sens large) et j (au sens strict) espacés d'un pas égal à p, les paramètres i et p étant par défaut pris égaux respectivement à 0 et 1.

```
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 conversion est aussi possible entre une chaîne de caractères et une liste :

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

Cependant, la conversion de type est toujours un peu dangereuse, faute de connaître précisément la manière dont la conversion s'opère. Aussi, il est souvent préférable de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique. Par exemple, à la définition mathématique $\left\{x \in [\![0,10]\!] \;\middle|\; x^2 \leqslant 50\right\}$ correspond la définition *par compréhension* suivante :

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

Ceci nous donne un moyen simple de calculer le 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)]
```

Autre exemple plus élaboré, on appelle *triplet pythagoricien* tout triplet d'entiers (x,y,z) tels que $x^2+y^2=z^2$. La définition par compréhension nous donne un moyen élégant de les calculer :

```
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)]
```

Remarque. Bien que les listes définies par compréhension ressemblent beaucoup aux ensembles mathématiques, elles n'en restent pas moins différentes par le fait qu'elles sont ordonnées. Ce n'est pas non plus forcément la manière la plus efficace de procéder. Dans l'exemple précédent tous les triplets d'entiers vérifiant les relations $1 \leqslant x \leqslant y \leqslant z < 20$ sont énumérés puis testés un par un. Cette démarche peut s'avérer très longue si on remplace 20 par un entier n plus grand : le nombre de triplets énumérés est égal à $\dfrac{(n+1)n(n-1)}{6}$ donc croît proportionnellement (ou presque) avec le cube de n. Pour $n = 1000$ le nombre de triplets à envisager est de l'ordre de 166 millions !

2.4 Opérations sur les listes

À l'instar des chaînes de caractères, deux opérations sont possibles sur les listes : la concaténation et la duplication.
La concaténation de deux listes consiste simplement à les mettre bout à bout dans une nouvelle liste. 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]
```

(attention, ce n'est pas un opérateur commutatif).
L'opérateur de duplication se note `*` ; si `l` est une liste et `n` un entier alors `l * n` est équivalent à `l + l +...+ l` (n fois).

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

2.5 Mutation d'une liste

En PYTHON, une liste est un objet *mutable*, dans le sens où il est possible de *modifier* le contenu d'une liste. Il est possible de modifier un ou plusieurs éléments de la liste, d'en supprimer ou d'en ajouter.

Modification d'un élément
Si `l` est une liste, l'instruction `l[i] = x` remplace l'élément d'indice `i` par la valeur `x`.

```
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
Le slicing permet de remplacer une partie de la liste : l'instruction `l[i:j] = t` remplace les éléments d'indices $i, i+1, \ldots, j-1$ par les éléments de la séquence [4] `t`.

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

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

Il est même possible d'employer la syntaxe complète du slicing en introduisant un pas (mais attention, dans ce cas les nouveaux éléments doivent être en même nombre que ceux qu'ils remplacent). À réserver aux utilisateurs avertis !

```
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]
```

Suppression d'un ou plusieurs éléments
L'instruction **`del`** permet de supprimer un ou plusieurs éléments (définis par le slicing) :

```
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 dans une liste
L'insertion d'un élément dans une liste fonctionne suivant une technique différente : lorsqu'on crée une liste, on crée un *objet* à qui est associé une structure de données et un certain nombre de *méthodes*, qui sont en quelque sorte des fonctions particulières associées à ce type d'objet [5]. Lorsque `obj` est un représentant de cette catégorie d'objet et `meth` une méthode associée, on applique cette dernière à l'objet en suivant la syntaxe : `obj.meth()`

Il existe principalement deux méthodes associées aux listes qui permettent l'insertion de nouveaux éléments : la méthode `append(x)` ajoute l'élément `x` en fin de liste, et la méthode `insert(i, x)` insère l'élément `x` à l'index `i`.

4. Une séquence est le plus souvent une liste ou une chaîne de caractères. Voir la section suivante.
5. La programmation objet ne sera détaillée qu'en seconde année.

```
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']
```

Autres méthodes associées aux listes
Il existe d'autres méthodes associées aux listes ; on retiendra en particulier :

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

```
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]
```

On notera que les deux méthodes `remove` et `pop` modifient la liste `l`, mais `pop` retourne la valeur supprimée, alors que `remove` retourne la valeur `None`.

```
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]
```

Remarque. On peut observer que les méthodes définies ci-dessus *modifient* l'objet à qui elles s'appliquent, à la différence d'une fonction qui en général calcule un nouvel objet. Il existe par exemple une fonction `sorted` qui calcule une *nouvelle* liste comportant les mêmes éléments mais cette fois-ci triés.

```
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]
```

Comme on peut le constater avec l'exemple ci-dessus, la fonction `sorted` retourne un résultat (contrairement à la méthode `sort`) mais ne modifie pas la liste `l`.

3. Un mot sur les séquences

Vous avez dû observer des similitudes entre les chaînes de caractères et les listes : elles sont composées d'un nombre fini d'éléments auxquels on peut accéder par un indice, et elles se prêtent aux mêmes techniques de slicing. De tels objets sont appelés des *séquences* ; il en existe d'autres, par exemple les *tuples*, qui servent à regrouper des éléments par couples, triplets, quadruplets, (et plus généralement les n-uplets).
Toutes ces séquences ont des propriétés en commun. 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 `*` ;

Nous utiliserons principalement :

- 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 de convertir un type de séquence en un autre ; par exemple :

```
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)
```

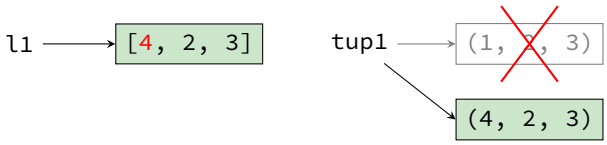
Mutabilité

Contrairement aux listes, les chaînes de caractères et les tuples *ne sont pas mutables* : on ne peut modifier ou supprimer les éléments d'une chaîne de caractères ou d'un tuple. C'est là une différence importante, qui va souvent conditionner le choix d'une structure de données (liste ou tuple) plutôt qu'une autre en fonction de ce qu'on souhaite faire.
Ainsi, pour modifier un tuple, on a pas d'autre possibilité que de le recréer entièrement.
Pour s'en convaincre, nous allons utiliser la fonction `id` qui, rappelons-le, renvoie l'adresse mémoire où est stocké l'objet à qui on l'applique. Nous savons que lorsqu'on crée une variable, c'est-à-dire lorsqu'on lie un identificateur à un objet par l'instruction `nom = obj`, on associe en fait à cet identificateur l'*adresse mémoire* où se trouve stocké cet objet. Ainsi, les instructions `l1 = [1, 2, 3]` et `tup1 = (1, 2, 3)` créent des liens symboliques (des *pointeurs*) entre les noms choisis et des emplacement en mémoire contenant les valeurs données :

```
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
```

Une liste étant mutable, on peut la modifier sans pour autant créer une nouvelle référence vers l'objet : l'adresse mémoire reste la même. L'instruction `l1[0] = 4` modifie le contenu de l'emplacement mémoire référencé par le nom `l1` mais pas l'adresse elle-même. En revanche, l'instruction `tup1 = (4, 2, 3)` recrée un nouveau référencement à une adresse différente.

```
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
```



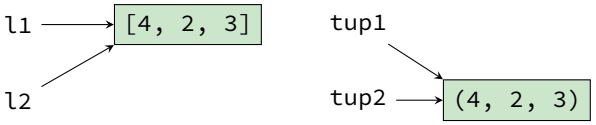
Il est important de bien comprendre le mécanisme mis en œuvre car s'agissant de structures linéaires la recopie d'une liste ou d'un tuple à un autre emplacement de la mémoire risque fort de nécessiter un temps d'exécution proportionnel à la longueur de la liste ou du tuple [6].
On peut maintenant préciser la définition d'un objet mutable :

Définition. — *Un objet* Python *est dit* mutable *lorsqu'on peut le modifier sans changer son adresse en mémoire.*

Pour l'instant, les seules structures mutables que l'on connaisse sont les listes.

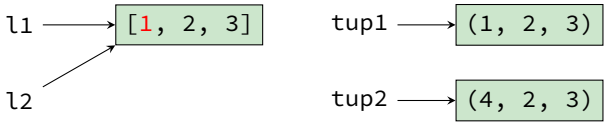
Les pièges de la mutabilité
Une méconnaissance du mécanisme de la mutabilité peut avoir une conséquence facheuse lorsqu'on cherche à créer une copie d'un objet mutable. En effet, lorsque `var1` est une variable, l'instruction `var2 = var1` se contente de créer *un nouveau référencement vers le même emplacement en mémoire*. Par exemple, les instructions `l2 = l1` et `tup2 = tup1` créent la situation suivante :

```
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
```



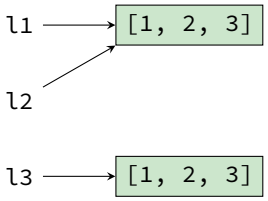
Ce n'est pas un problème lorsque la variable n'est pas mutable. Si on veut revenir à l'état initial de `tup1` il faut de toute façon le recréer entièrement en écrivant : `tup1 = (1, 2, 3)`. En revanche, l'instruction `l1[0] = 1` *va aussi modifier* la variable `l2` puisque ces deux noms référencent la *même adresse*.

```
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)
```



Copie d'un objet mutable
Comment alors copier un élément mutable ? Nous venons de constater qu'un simple référencement n'est pas suffisant. Il est nécessaire de *recréer* une liste qu'on veut copier. Cette copie peut se faire par exemple à l'aide du slicing `[:]`, puisque ceci recrée une nouvelle liste.

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

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



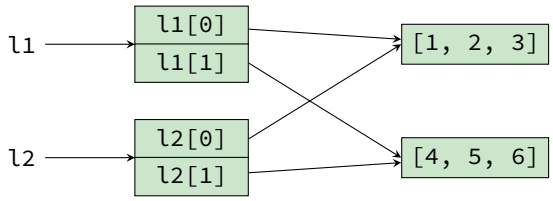
6. C'est effectivement le cas.

Retenons donc que si `l1` est une liste, l'instruction `l2 = l1` référence sous un autre nom le même objet tandis que l'instruction `l3 = l1[:]` crée une nouvelle copie de la liste.

Remarque. De manière équivalente à `l3 = l1[:]` on peut écrire `l3 = l1.copy()` dont la syntaxe est peut-être plus explicite.

Copie profonde
Considérons maintenant le cas d'une liste *dont les éléments sont eux-mêmes des listes* (ou plus généralement des objets mutables) et créons-en une copie.

```
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
```



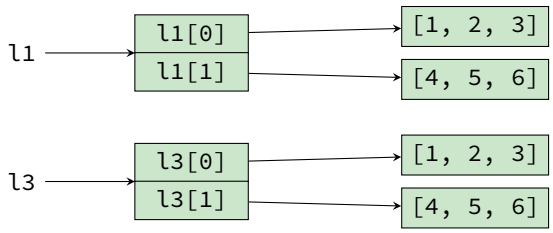
Nous constatons sur cet exemple que nous n'avons réalisé qu'une copie de premier niveau : `l1` et `l2` référencent effectivement deux emplacements mémoires distincts, mais `l1[i]` et `l2[i]` référencent toujours le même objet mutable et toute modification de l'un entrainera automatiquement une modification identique de l'autre.

Il est possible de s'en sortir avec une définition par compréhension :

```
l3 = [l[:] for l in l1]
```

mais outre le fait que la syntaxe utilisée se fait de moins en moins explicite, on ne résout pas le problème s'il y a un troisième niveau d'objets mutables, voire plus...
Dans ces situations (et même s'il faut bien reconnaître qu'il est très rare de dépasser trois niveaux d'objets mutables) on utilise un module spécialisé dans la copie : le module `copy` et plus précisément la fonction `deepcopy` de ce module qui réalise une copie profonde d'un objet mutable :

```
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
```



4. Parcours d'une liste

Nous allons maintenant nous intéresser à la façon de parcourir une liste. Puisqu'il ne s'agit pas ici de modifier les éléments de la liste mais seulement d'y accéder, ce qu'on va dire pourra aussi s'appliquer aux chaînes de caractères et aux tuples.

4.1 Parcours complet par boucle énumérée

Les boucles énumérées `for i in range(...)` que nous avons utilisées jusqu'à présent ne sont qu'un cas particulier d'une syntaxe plus générale :

```
for x in seq
```

où seq est une séquence (donc une liste, une chaîne de caractères ou un tuple). Une telle boucle définit une variable (ici nommée `x`) qui prend successivement toute les valeurs de la séquence (par ordre d'indice croissant) ; cela en fait le mode privilégié de parcours complet d'une liste.

Supposons par exemple que l'on veuille calculer la somme des éléments d'une liste d'entier. On définira la fonction :

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

Quelques exemples d'application :

- **Calcul de la moyenne**

Rappelons que la moyenne d'une suite de valeurs $(x_1, x_2, \dots, x_n)$ se définit par : $\overline{x} = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n} x_k$.

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

- **Calcul de la variance**

Rappelons que la variance d'une suite finie de valeurs $(x_1, x_2, \dots, x_n)$ se définit par : $v = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n}(x_k - \overline{x})^2$ et caractérise la dispersion d'un échantillon autour de sa moyenne. Cependant cette formule exige deux parcours de la liste : le premier pour calculer $\overline{x}$ et le second pour calculer v. Il est préférable d'utiliser la formule de König-Huygens :

$$\frac{1}{n}\sum_{k=1}^{n}(x_k - \overline{x})^2 = \frac{1}{n}\sum_{k=1}^{n}x_k^2 - \frac{2\overline{x}}{n}\sum_{k=1}^{n}x_k + \overline{x}^2 = \frac{1}{n}\sum_{k=1}^{n}x_k^2 - \overline{x}^2 = \frac{1}{n}\sum_{k=1}^{n}x_k^2 - \left(\frac{1}{n}\sum_{k=1}^{n}x_k\right)^2$$

qui permet le calcul de la variance après un seul parcours de liste :

```
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)
```

- **Calcul du maximum**

Intéressons nous maintenant au problème du calcul de la valeur maximale d'une liste d'entiers :

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

Notons qu'il existe déjà une fonction `max` (ainsi qu'une fonction `min`) qui calcule le maximum d'une séquence d'entiers.

Si on souhaite obtenir l'indice de l'élément maximal de la liste, il faut cette fois parcourir non plus la liste elle-même, mais la liste de ces indices :

```
def indice_du_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
```

On peut constater que le code de la fonction ci-dessus s'est alourdi et va à l'encontre du principe des itérateurs en Python. C'est pourquoi le langage propose une fonction `enumerate` qui renvoie une liste de tuples `(indice, valeur)` lorsqu'on l'applique à une liste (ou plus généralement à tout objet énumérable). Au code précédent on préférera donc :

```
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
```

4.2 Parcours incomplet : recherche dans une liste

Les algorithmes de recherche dans une liste correspondant le plus souvent à des parcours incomplets, puisqu'on stoppe la recherche une fois l'élément trouvé. On utilise en général une boucle conditionnelle, à moins qu'on préfère utiliser une sortie prématurée de boucle énumérée.

• Recherche d'un élément

Un problème fréquent est de déterminer la présence ou non d'un élément `x` dans une liste `l`. On peut répondre à ce problème de plusieurs façons.
La première version, la plus classique, consiste à parcourir la liste à l'aide d'une boucle conditionnelle : on continue le parcours tant que l'élément n'est pas trouvé et qu'on est pas en bout de liste.

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

Le code produit est assez lourd et n'est guère dans l'esprit de la programmation en Python.
La seconde façon, beaucoup plus dans l'esprit des itérateurs Python, consiste à interrompre une boucle énumérée dès qu'on a trouvé l'élément recherché :

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

Cette dernière version est déjà particulièrement simple, mais on peut faire encore plus court en utilisant l'instruction `in`. Ce mot-clé, qu'on utilise déjà dans les boucles énumérées, a un second rôle : celui justement de déterminer la présence ou non d'un élément dans une séquence. Les deux fonctions ci-dessus sont donc équivalentes à la simple expression :

```
x in l
```

qui donne un booléen correspondant à la réponse cherchée.

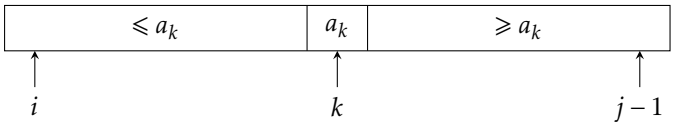
• Recherche dichotomique

Les trois fonctions précédentes ont en commun de nécessiter dans le pire des cas n comparaisons (où n est la longueur de la liste) pour déterminer la présence ou non de l'élément cherché dans la liste. On peut faire beaucoup mieux dans le cas d'une liste *triée*.
Pour chercher x dans la liste triée par ordre croissant $[a_0,\dots,a_{n-1}]$, nous allons appliquer le principe *dichotomique* : comparer x et a_p, avec $p = \left\lfloor \frac{n}{2} \right\rfloor$.

- si $x < a_p$ alors x, s'il se trouve dans la liste, ne peut qu'être dans la liste $[a_0,\dots,a_{p-1}]$;
- si $x = a_p$, la recherche est terminée ;
- si $x > a_p$ alors x, s'il se trouve dans la liste, ne peut qu'être dans la liste $[a_{p+1},\dots,a_{n-1}]$.

La mise en œuvre pratique utilise deux variables entières i et j délimitant le domaine $[a_i,\dots,a_{j-1}]$ dans lequel on cherche x. Les valeurs initiales de ces variables sont donc $i = 0$ et $j = n$.

On compare ensuite x à l'élément médian d'indice $k = \left\lfloor \frac{i+j}{2} \right\rfloor$ et si nécessaire on poursuit la recherche en remplaçant j par k ou i par $k+1$.



La recherche s'achève (par une réponse négative) lorsque $i = j$ car le domaine de recherche est vide.

```
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
```

Évaluation du nombre de comparaison
Notons c_n le nombre de comparaison que cet algorithme effectue dans le pire des cas. Le principe dichotomique sépare la liste en trois parties de respectivement $\left\lfloor \frac{n-1}{2} \right\rfloor$, 1 et $\left\lfloor \frac{n}{2} \right\rfloor$ cases, et une comparaison permet de restreindre la recherche à l'une de ces trois sous-listes.
Dans le pire des cas, on a donc : $c_n = 1 + c_{\lfloor n/2 \rfloor}$.
On résout cette relation de récurrence en utilisant la décomposition en base 2 de n : $n = (b_p \cdots b_1 b_0)_2$ car dans ce cas, $\left\lfloor \frac{n}{2} \right\rfloor = (b_p \cdots b_2 b_1)_2$. Ainsi, $c_n = p + c_1 = p + 1$.
Il reste à exprimer l'entier p en fonction de n. Or : $(1\underbrace{00\cdots00}_{p \text{ fois}})_2 \leqslant n \leqslant (1\underbrace{11\cdots11}_{p \text{ fois}})_2$ donc $2^p \leqslant n \leqslant 2^{p+1} - 1$.
En composant par le logarithme en base 2 on obtient : $p \leqslant \log_2(n) < p+1$, et donc $p = \lfloor \log_2(n) \rfloor$.
Le nombre de comparaisons dans le pire des cas de l'algorithme de recherche dichotomique est donc égal à $\lfloor \log_2(n) \rfloor + 1$, à comparer aux n comparaisons que nécessite l'algorithme de recherche linéaire, ce qui peut constituer une différence importante dans le cas de nombreuses recherches dans des listes de grande taille.