STRUCTURES DE DONNÉES : LISTES

La liste est la structure de données la plus utilisée en Python. Pour programmer efficacement dans ce langage, il est donc important de savoir bien l'utiliser.

I . Notions de base

■ Définition d'une liste en extension :

Une liste est une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.

Exemple :

```
>>> mois = ['Janvier', 31, 'Fevrier', 28, 'Mars', 31, 'Avril', 30]
>>> print(mois)
['Janvier', 31, 'Fevrier', 28, 'Mars', 31, 'Avril', 30]
>>> print(type(mois))
<class 'list'>
```

Comme on peut le constater dans l'exemple ci-dessus, les éléments qui constituent une liste peuvent être de types différents. Un élément d'une liste peut lui-même être une liste.

Exemple :

```
>>> mois = [['Janvier', 31], ['Fevrier', 28], ['Mars', 31], ['Avril', 30]]
>>> print(mois)
[['Janvier', 31], ['Fevrier', 28], ['Mars', 31], ['Avril', 30]]
```

Comme les chaînes de caractères, les listes sont des *séquences*, c'est-à-dire des collections ordonnées d'objets. Les divers éléments qui constituent une liste sont en effet disposés dans un certain ordre, et l'on peut donc accéder à chacun d'entre eux individuellement si l'on connaît son index dans la liste. Comme c'est également le cas pour les caractères dans une chaîne, il faut retenir que la numérotation de ces index commence à partir de zéro, et non à partir de un.

Exemple :

```
>>> mois = ['Janvier', 31, 'Fevrier', 28, 'Mars', 31, 'Avril', 30]
>>> print(mois[0], mois[2])
Janvier Fevrier
>>> mois2 = [['Janvier', 31], ['Fevrier', 28], ['Mars', 31], ['Avril', 30]]
>>> print(mois2[3], mois2[1][1])
['Avril', 30] 28
```

■ Définition d'une liste à l'aide d'un itérateur :

Python utilise fréquemment des objets dits *itérables*. Parmi ces objets, on trouve les chaînes, les listes, les tuples, les dictionnaires, mais aussi des objets de type `iterator` qui ne sont pas accessibles dans leur globalité, mais seulement par itération (ce sont ces objets qui servent, par exemple, lors de l'éxécution d'une boucle `for`).

A partir d'un objet itérable, on peut construire une liste à l'aide de la fonction `list`.

Exemples :

```
>>> range(10)
range(0, 10)
>>> # cette instruction ne renvoie pas directement une liste, contrairement à Python 2.7
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(3,10))
[3, 4, 5, 6, 7, 8, 9]
>>> list('Bonjour')
['B', 'o', 'n', 'j', 'o', 'u', 'r']
```

■ Définition d'une liste en compréhension :

Dans les exemples ci-dessus, les listes sont définies en *extension*, c'est-à-dire que l'on énumère tous les éléments constituant la liste. Cela n'est pas très pratique pour des listes qui comportent un grand nombre d'éléments.

On peut définir une liste en précisant les propriétés que doivent vérifier ses éléments. Cela se fait par l'instruction :

```
[expression for element in liste if predicat]
```

Dans cette instruction, `expression` est une expression Python (ce peut-être une somme, un produit, une chaîne, une liste, etc...) qui dépend de la variable `element`. Elle construit une liste formée de toutes les expressions obtenues lorsque `element` parcourt la liste donnée et à condition que le prédicat indiqué soit vérifié (ce prédicat est d'ailleurs optionnel).

Exemples :

```
>>> liste = list(range(10))
>>> [element + 1 for element in liste]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [i**2 for i in liste if i % 2 == 0]
[0, 4, 16, 36, 64]
>>> [(x, 2**x) for x in range(10)]
[(0, 1), (1, 2), (2, 4), (3, 8), (4, 16), (5, 32), (6, 64), (7, 128), (8, 256), (9, 512)]
>>> liste = [[2, [3, 4]], [5, 6]]
>>> [i for li in liste for i in li]
[2, [3, 4], 5, 6]
```

■ Comparaison :

Le script Python ci-dessous compare les temps d'éxécution de différentes méthodes de construction d'une liste formée des carrés des 10.000.000 premiers entiers (100000 seulement pour la première méthode, trop lente).

Les résultats parlent d'eux-mêmes.

Comparaison des temps d'éxécution pour la construction d'une liste

```
from time import clock

def test1(n):
    l = []
    for i in range(n):
        l = l + [i * i]

def test11(n):
    l = []
    for i in range(n):
        l += [i * i]

def test2(n):
    l = []
    for i in range(n):
        l.append(i * i)

def test3(n):
    l = [i * i for i in range(n)]

def test4(n):
    l = list(map(lambda x: x * x, list(range(n))))

def duree(fonction, n=10000000):
    debut = clock()
    fonction(n)
    fin = clock()
    return(fin - debut)
```

```
29
30 print(" {:.<45} {:f} secondes".format("Concatenation (100x moins de boucles!)",\
31 duree(test1,100000)))
32 print(" {:.<45} {:f} secondes".format("Concatenation bis",duree(test11)))
33 print(" {:.<45} {:f} secondes".format("append",duree(test2)))
34 print(" {:.<45} {:f} secondes".format("Comprehension",duree(test3)))
35 print(" {:.<45} {:f} secondes".format("list avec map et range",duree(test4)))
```

```
Concatenation (100x moins de boucles!)..... 20.507924 secondes
Concatenation bis.......................... 3.507471 secondes
append..................................... 2.763036 secondes
Comprehension.............................. 1.878321 secondes
list avec map et range..................... 3.286178 secondes
```

Vous remarquerez le temps prohibitif mis par la première méthode ; cela est dû au fait que, lors de chaque instruction `l = l + [i*i]`, l'interpréteur effectue une copie complète de la liste `l`, y ajoute l'élément `i*i` puis réaffecte le résultat obtenu à la variable `l`. L'instruction `l += [i*i]`, elle, ne fait que modifier la valeur l'objet `l`.

II . Premières opérations sur les listes

■ Le problème de l'affectation :

Nous allons évoquer ici un point très important du langage, à savoir la façon dont sont représentés et manipulés les objets.

Les variables en Python ne sont que des noms associés à des objets. Lorsque l'on écrit une instruction comme : `x=1`, le programme crée l'objet correspondant à la valeur entière `1`, et le nom "x" qui référence cet objet ; si l'on écrit ensuite `y=x`, on crée un autre nom "y" qui référence le même objet que "x". On peut le vérifier en utilisant la fonction `id` de Python qui retourne l'identificateur unique associé à l'objet (correspondant à son adresse en mémoire) :

```
>>> a = 1
>>> id(a)
505910864
>>> b = a
>>> id(b)
505910864
```

Le gros avantage de cette méthode (appelée *passage par référence*) est le gain de place mémoire et de temps : si un objet est volumineux (gros tableau de données par exemple), il est inutile de le dupliquer, ce qui occuperait inutilement de la place mémoire et prendrait du temps (la duplication s'appelle le *passage par valeur*).

En contrepartie, la façon de manipuler les variables est plus complexe. En effet, dans notre exemple précédent, puisque les variables de noms "x" et "y" font référence au même objet, on peut se demander ce qui se passe si l'on modifie la variable "x". La variable "y" est-elle ou non modifiée (on parle alors *d'effet de bord*) ?

C'est là qu'intervient la distinction entre deux types d'objets : les objets *mutables* et les objets *non mutables*. Parmi les objets de type prédéfini en Python, les listes, dictionnaires et ensembles (`set`) sont mutables, les entiers (`int`), flottants (`float`), les chaînes (`str`), les tuples et les booléens sont non mutables.

Un objet est *non mutable* s'il est impossible de le modifier au même emplacement mémoire ; lorsque l'on fait des opérations sur des objets non mutables pour obtenir une nouvelle valeur, on est obligé de construire un autre objet car les opérations ne permettent pas de modifier l'original.

Les objets *mutables* offrent des opérations permettant de modifier la valeur contenue dans l'objet, en conservant le même objet.

Pour bien comprendre, observez le comportement des deux scripts ci-dessous, l'un concernant des valeurs numériques, l'autre des listes :

```
>>> x = 1
>>> y = x
>>> id(x), id(y)
(505910864, 505910864)
>>> y += 1
>>> y
2
>>> x
1
>>> id(x), id(y)
(505910864, 505910880)
```

```
>>> li = [1, 2, 3]
>>> li2 = li
>>> id(li), id(li2)
(34178552, 34178552)
>>> li2 += [4, 5, 6]
>>> li2
[1, 2, 3, 4, 5, 6]
>>> li
[1, 2, 3, 4, 5, 6]
>>> id(li), id(li2)
(34178552, 34178552)
```

Dans le premier exemple, lors de l'affectation `y=y+1`, le programme ne pouvant pas modifier la valeur entière 1 (on ne peut pas remplacer l'entier 1 par 2 !), a créé un nouvel emplacement mémoire pour contenir le résultat et a assigné à la variable "y" ce nouvel emplacement.

Dans le second cas, les deux listes `li` et `li2` désignent le même emplacement mémoire, et le contenu de cet emplacement a été directement modifié, puisque un objet de type `list` est mutable. Par effet de bord, toute modification de l'une des variables entraîne la modification de l'autre.

Si on souhaite dupliquer la liste de façon qu'une modification sur l'une ne provoque pas une modification sur l'autre, on peut utiliser le constructeur `list`. Dans ce cas, le programme recopie l'intégralité de la liste à un autre emplacement :

```
>>> li = [1, 2, 3]
>>> li2 = list(li) # ou encore li2 = li[:]
>>> id(li), id(li2)
(34178672, 34178752)
>>> li2 += [4, 5, 6]
>>> li2
[1, 2, 3, 4, 5, 6]
>>> li
[1, 2, 3]
>>> id(li), id(li2)
(34178672, 34178752)
```

Le problème semble donc ainsi résolu. Malheureusement, non ! En fait, une liste est un ensemble de références vers des objets. Lorsque ces objets sont eux-mêmes mutables (par exemple des listes), observons ce qui se passe :

```
>>> li = [1, 2, [3, 4]]
>>> li2 = list(li)
>>> li2[1] = 5
>>> li, li2
([1, 2, [3, 4]], [1, 5, [3, 4]])
>>> li2[2][1] = 'oh!'
>>> li, li2
([1, 2, [3, 'oh!']], [1, 5, [3, 'oh!']])
>>> id(li), id(li2)
(34201648, 34178832)
>>> id(li[2]),id(li2[2])
(34178552, 34178552)
```

Comme on le voit dans les 4 dernières lignes, lors de la copie, une nouvelle liste `li2` a bien été créée. Mais ce sont seulement les références vers les objets de la liste qui ont été copiées, pas les objets eux-mêmes (les références vers les objets `li[2]` et `li2[2]` sont les mêmes). Ainsi, lorsqu'il s'agit d'un objet mutable, il y a tout naturellement effet de bord.

Le mode de copie ci-dessus s'appelle une *copie superficielle* (*shallow copy* en Anglais). La solution définitive consiste à faire une *copie profonde* (*deepcopy*), comme ci-dessous :

```
>>> from copy import *
>>> li = [1, 2, [3, 4]]
>>> li2 = deepcopy(li)
```

```
>>> li2[2][1] = 'oh!'
>>> li, li2
([1, 2, [3, 4]], [1, 2, [3, 'oh!']])
>>> id(li), id(li2)
(34180312, 34178672)
>>> id(li[2]),id(li2[2])
(34178752, 34180792)
```

■ Opérations sur les listes :

Les opérations citées ci-dessous ne sont pas spécifiques aux listes, et peuvent être aussi utilisées avec des objets de type `string` ou `tuple`.

Opération	Résultat
`x in s`	`True` si un des éléments de s est égal à x, `False` sinon
`x not in s`	`False` si un des éléments de s est égal à x, `True` sinon
`s + t`	concaténation de s et t
`s * n`	liste où le contenu de s est copié n fois
`s[i]`	i-ème élément de s (les indexs commencent à 0)
	si i est négatif, on commence à la fin de la liste ($i \longrightarrow \text{len}(s)+i$)
`s[i:j]`	tranche entre l'élément d'index i (inclus) et j (exclu) (tranche de longueur $j-i$)
`s[i:j:k]`	tranche entre l'élément d'index i (inclus) et j (exclu) avec un pas de k
`len(s)`	nombre d'éléments de s
`min(s)`, `max(s)`	minimum, maximum des éléments de s

Exemples :

```
>>> li = [1, 2, 3]+[4, 5, 6]
>>> li
[1, 2, 3, 4, 5, 6]
>>> li[0], li[-2]
(1, 5)
>>> li2 = li*2
>>> li2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
>>> li3 = list('Bonjour')
>>> li3
['B', 'o', 'n', 'j', 'o', 'u', 'r']
>>> len(li3), min(li3), max(li3)
(7, 'B', 'u')
```

■ Quelques précisions sur le *slicing* :

Le terme *slicing* (=tranchage) fait référence à la syntaxe `liste[start:stop:step]` pour extraire une partie d'une liste (ou d'une chaîne, ou d'un tuple, ou d'une array...). Certains de ces 3 paramètres peuvent être omis. Plus précisément :

- `s[i:j]` désigne la liste formée des éléments de s dont l'index k est tel que $i \leqslant k < j$. Elle a donc pour longueur $j-i$.
  Si i ou j est plus grand que $\text{len}(s)$, il est remplacé par $\text{len}(s)$.
  Si i est omis, il est considéré égal à 0, si j est omis, il est considéré égal à $\text{len}(s)$.
- `s[i:j:k]` désigne la liste formée des éléments de s d'index i, $i+k$, $i+2k$ etc... jusqu'à ce que l'index j soit atteint (mais jamais inclus). L'entier k peut être négatif.
  Si i ou j est plus grand que $\text{len}(s)$, il est remplacé par $\text{len}(s)$.
  Si i ou j est omis, il est considéré comme étant égal à 0 ou $\text{len}(s)$ (cette valeur dépend du signe de k !).

  Exemples :

```
>>> li = [1, 0, 2, 4, 5, 6]
>>> li
[1, 0, 2, 4, 5, 6]
>>> li[1:3]
[0, 2]
>>> li[1:3] = [2,3] # les valeurs sont modifiables
>>> li
[1, 2, 3, 4, 5, 6]
>>> li[:3] # les 3 premiers éléments
[1, 2, 3]
>>> li[-3:] # les 3 derniers
[4, 5, 6]
>>> li[::2]
[1, 3, 5]
>>> li[4:1:-1]
[5, 4, 3]
```

III . Méthodes utilisées sur les listes

Les listes possèdent plusieurs méthodes très utiles à leur manipulation, et qu'il est nécessaire de connaître pour utiliser ces structures de données efficacement.

Pour obtenir une liste de toutes ces fonctions, se référer à la documentation officielle de Python ou plus simplement taper `help(list)` dans la console Python.

1. `append`

Comme son nom l'indique, cette méthode sert à ajouter un élément à la fin d'une liste.

Exemple :

```
>>> li = [1, 2, 3, 4, 5]
>>> li.append(6)
>>> print(li)
[1, 2, 3, 4, 5, 6]
```

2. `extend`

Cette méthode ajoute à la fin de la liste tous les éléments de la liste passée en paramètre.

Exemple :

```
>>> li = [1, 2, 3, 4, 5]; li2 = ['x']
>>> li.extend([6,7,8])
>>> li.extend(li2)
>>> print(li)
[1, 2, 3, 4, 5, 6, 7, 8, 'x']
```

3. `count`

Cette méthode renvoie le nombre d'occurences d'un objet dans une liste.

Exemple :

```
>>> li = [1, 2, 3, 4, 5, 1]
>>> print(li.count(1), li.count(6))
2 0
```

4. `index`

Cette méthode renvoie la position d'un objet dans une liste. Plus précisément, l'appel à `s.index(x[, i[, j]])` renvoie le plus petit entier k tel que $s[k] == x$ et $i \leqslant k \leqslant j$. Les paramètres i et j sont optionnels.

Exemple :

```
>>> li = [1, 2, 1, 5, 3, 4, 5, 1]
>>> print(li.index(1), li.index(1,1), li.index(1,3))
0 2 7
```

```
>>> li = [1, 2, 1, 3, 4, 5, 1]
>>> print(li.index(1,3,4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 1 is not in list
```

5. `insert`

Cette méthode permet d'insérer un élmément dans une liste. Plus précisément, l'appel à `s.insert(i,x)` insère l'objet x avant celui à la position i (il deviendra donc celui de position i dans la nouvelle liste).

Cas particuliers :

- Si l'indice i est plus grand que la longueur `len(s)` de la liste, on considère que $i = \text{len}(s)+1$, donc l'objet est placé à la fin.
- Si l'indice i est <0, il est remplacé par $\text{len}(s)-i$ et si le résultat est encore strictement négatif, il est remplacé par 0.

Exemple :

```
li1 = [1, 2, 3, 4, 5]
li2 = li1[:] ; li3 = li1[:]
li4 = list(li1) ; li5 = list(li1)
# pour faire des copies de la liste initiale
# en effet, l'affectation li2=li1 aurait seulement pour effet
# de désigner le même objet sous deux noms différents
# on peut aussi utiliser la fonction copy à condition de l'importer
li1.insert(2,0)
print(li1)
li2.insert(0,6)
print(li2)
li3.insert(9,7)
print(li3)
li4.insert(-2,0)
print(li4)
li5.insert(-10,0)
print(li5)
```

```
[1, 2, 0, 3, 4, 5]
[6, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 7]
[1, 2, 3, 0, 4, 5]
[0, 1, 2, 3, 4, 5]
```

6. `pop`

Cette méthode retire de la liste un élément d'indice donné. Plus précisément, l'appel à `s.pop(i)` renvoie l'élément d'indice i et le supprime de la liste.

Cas particuliers :

- Si i est strictement supérieur à la longueur de la liste, une erreur survient.
- Si l'indice i est <0, il est remplacé par $\text{len}(s)-i$ mais si le résultat est encore strictement négatif, une erreur survient.
- Si le paramètre i est omis, il est considéré égal à -1, donc le dernier élément de la liste est alors retiré.

Exemple :

```
li1 = [1, 2, 3, 4, 5]
li2 = li1[:] ; li3 = li1[:] ; li4 = li1[:]
x = li1.pop(2)
print(x, li1)
x = li2.pop(0)
```

```
print(x, li2)
x = li3.pop()
print(x, li3)
x = li4.pop(-2)
print(x,li4)
```

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

```
>>> li = [1, 2, 3, 4, 5]
>>> li.pop(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

On pourra utiliser aussi la *fonction* `del` pour supprimer des éléments d'une liste. Cette fonction agit de la même manière que `pop`, mais ne renvoie pas d'élément. Elle permet également de supprimer plusieurs éléments consécutifs.

```
>>> li= [1, 2, 3, 4, 5, 6, 7, 8]
>>> del(li[0])
>>> li
[2, 3, 4, 5, 6, 7, 8]
>>> del(li[3:5])
>>> li
[2, 3, 4, 7, 8]
>>> del(li[:])
>>> li
[]
>>> del(li) # attention, cette instruction supprime la variable li
>>> li
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'li' is not defined
```

7. `remove`

Cette méthode retire de la liste la première valeur trouvée qui est égale au paramètre. Bien sûr, si cette valeur n'est pas dans la liste, une erreur survient.

Exemple :

```
>>> li = [1, 2, 3, 4, 5, 1]
>>> li.remove(1)
>>> print(li)
[2, 3, 4, 5, 1]
```

```
>>> li = [1, 2, 3, 4, 5, 1]
>>> li.remove(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

8. `reverse`

Cette méthode permet de renverser l'ordre des éléments d'une liste ; le renversement se fait directement dans la liste sur laquelle on a utilisé la méthode. Si on veut récupérer la valeur sans modifier la liste initiale, on peut utiliser la *fonction* `reversed` (qui n'est pas une *méthode* de la classe `list`).

Exemple :

```
>>> li = [1, 2, 3, 4, 5]
>>> li.reverse()
>>> print(li)
[5, 4, 3, 2, 1]
>>> print(reversed(li))
<list_reverseiterator object at 0x0209A2D0>
>>> li2 = list(reversed(li))
>>> print(li2)
[1, 2, 3, 4, 5]
```

On notera que la fonction `reversed` ne renvoie pas un objet de type `list` mais de type `iterator`, d'où l'utilisation de la fonction `list`.

9. `sort`

Cette méthode permet de trier une liste dans l'ordre croissant ou dans l'ordre alphabétique, selon le type des données de la liste.

De la même façon qu'il existe la fonction `reversed` pour `list.reverse()`, on a la fonction `sorted` pour `list.sort()` ; à la différence de la fonction `reversed` qui renvoie un itérateur, cette fonction renvoie une liste. De plus, la fonction `sorted` peut être appliquée à d'autres objets que les listes.

Exemple :

```
>>> li = [5, 1, 4, 6, 3, 2]
>>> li.sort()
>>> print(li)
[1, 2, 3, 4, 5, 6]
>>> str = ['zz', 'AA', 'Cc', 'BB', 'aa']
>>> print(sorted(str))
['AA', 'BB', 'Cc', 'aa', 'zz']
>>> # attention à la casse
>>> ens = {10, 1, 7, 3}.union({4, 2})
>>> print(sorted(ens))
[1, 2, 3, 4, 7, 10]
```

Des paramètres additionnels peuvent être donnés à la méthode `sort` ou à la fonction `sorted` pour changer le mode de tri et l'ordre dans lequel le tri s'effectue :

- Le paramètre `reverse=True` permet de faire le tri dans l'ordre inverse.
  Exemple :

  ```
  >>> li = [5, 1, 4, 6, 3, 2]
  >>> li.sort(reverse=True)
  >>> print(li)
  [6, 5, 4, 3, 2, 1]
  >>> str=['zz', 'AA', 'Cc', 'BB', 'aa']
  >>> print(sorted(str, reverse=True))
  ['zz', 'aa', 'Cc', 'BB', 'AA']
  ```

- Le paramètre `key` est plus intéressant. Il permet de spécifier une clé ; cette clé sera calculée pour chaque élément de la liste, et le tri sera effectué dans l'ordre des résultats.
  La méthode la plus générale pour définir la clé consiste à utiliser une fonction qui à chaque élément de la liste associera une valeur.
  Exemples :
  - Pour trier une liste de mots dans l'ordre alphabétique sans faire attention à la casse, on utilisera comme clé une fonction qui transforme une chaîne en miniscules.

    ```
    >>> li = ['zz', 'AA', 'Cc', 'BB', 'aa']
    >>> li.sort(key=lambda s: s.lower())
    >>> print(li)
    ['AA', 'aa', 'BB', 'Cc', 'zz']
    ```

  - Pour trier une liste de mots selon leurs longueurs, on utilisera comme clé une fonction qui à une chaîne de caractères associe sa longueur.

```
>>> li = "Les sanglots longs des violons de l automne".split()
>>> print(li)
['Les', 'sanglots', 'longs', 'des', 'violons', 'de', 'l', 'automne']
>>> print(sorted(li,key=lambda s: len(s)))
['l', 'de', 'Les', 'des', 'longs', 'violons', 'automne', 'sanglots']
```

Si, dans l'exemple précédent, on veut de plus que les chaînes de même longueur soient triées par ordre alphabétique, on pourra écrire :

```
>>> li = "Les sanglots longs des violons de l automne".split()
>>> print(li)
['Les', 'sanglots', 'longs', 'des', 'violons', 'de', 'l', 'automne']
>>> print(sorted(li,key=lambda s: (len(s), s.lower())))
['l', 'de', 'des', 'Les', 'longs', 'automne', 'violons', 'sanglots']
```

– On peut aussi trier des objets plus complexes en prenant pour clé d'un objet l'une ou plusieurs de ses composantes.

```
1  Liste_Eleves = [['Jean', '6emeA', '11 ans'], ['Jacques', '6emeB', '12 ans'],
2                  ['Jeanne', '5emeA', '12 ans'], ['Jules', '5emeA', '13 ans'] ,
3                  ['Julie', '5emeA', '12 ans']]
4
5  def MyKey(Eleve):
6      return(Eleve[2]+Eleve[1]+Eleve[0])
7
8  Liste_Eleves.sort(key=MyKey)
9  # 'joli' affichage:
10 l = map(reversed,Liste_Eleves)
11 l = map(','.join,l)
12 print('\n'.join(l))
```

```
11 ans,6emeA,Jean
12 ans,5emeA,Jeanne
12 ans,5emeA,Julie
12 ans,6emeB,Jacques
13 ans,5emeA,Jules
```

IV . Modification fonctionnelle d'une liste

On peut créer de nouvelles listes à partir d'une liste existante à l'aide des fonctions `map`, `filter` et `zip`.

Le principe est d'appliquer une même fonction que vous avez créée à tous les éléments d'une liste pour en créer une nouvelle. Cela évite d'écrire une boucle `for` ou `while` explicite ; la boucle implicite contenue dans ces trois fonctions est en effet plus rapide à l'exécution.

Attention : Depuis la version 3 de Python, ces instructions ne renvoient pas directement une liste mais un itérateur.

1. `map`

La syntaxe de cette fonction est la suivante :

`map` (*func, sequence*, [*sequence*,...])

Cette fonction crée une nouvelle liste (ou plutôt un itérateur sur une liste) en appliquant la fonction *func* aux objets de la *sequence*. Si plusieurs séquences sont mentionnées, la fonction *func* doit être une fonction de plusieurs variables qui s'appliquera aux objets correspondants de chaque séquence. Ces séquences doivent alors être de même longueur.

Exemples :

```
1  def carre(x): return(x * x)
2  def pair(x): return (x % 2 == 0)
3  li1 = list(range(8))
4  print(li1)
```

```
print(map(carre,li1)) # ce n'est pas une liste mais un itérateur
print(list(map(carre,li1))) # d'où l'emploi de la fonction list
print(list(map(pair,li1)))
li2 = [0, 2, 1, 4, 3, 7, 6, 5]
print(list(map(lambda x, y: x if (x > y) else y, li1, li2)))
li3 = list(map(ord,"Bonjour.")) # la fonction ord donne le code Ascii d'un caractère
print(li3)
print(list(map(lambda x, y, z: x + y * z, li1, li2, li3)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
<map object at 0x01E67930>
[0, 1, 4, 9, 16, 25, 36, 49]
[True, False, True, False, True, False, True, False]
[0, 2, 2, 4, 4, 7, 6, 7]
[66, 111, 110, 106, 111, 117, 114, 46]
[0, 223, 112, 427, 337, 824, 690, 237]
```

2. `filter`

La syntaxe de cette fonction est la suivante :

`filter` (*func, sequence*)

Cette fonction crée un itérateur sur une liste formée des objets de la séquence pour lesquels la fonction *func* renvoie le résultat `True`.

Exemple :

```
>>> liste = [i - 4 for i in range(10)]
>>> print(liste)
[-4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
>>> print(list(filter(lambda x: (x > 0)  and (x % 2 != 0),liste)))
[1, 3, 5]
```

On peut aussi écrire directement :

```
>>> [i - 4 for i in range(10) if (i > 4) and (i % 2 != 0)]
[1, 3, 5]
```

Remarque : Pour construire une liste, il est préférable d'utiliser la définition d'une liste en compréhension plutôt que d'utiliser les fonctions `map` et `filter`. Cela est plus lisible et souvent plus rapide, particulièrement lors de l'utilisation de fonctions lambda. Exemple :

```
from time import clock

def test1(n):
    l = [i * i for i in range(n) if (i > n/2) and (i % 2 != 0)]

def test2(n):
    l = list(map(lambda x: x * x, filter(lambda x: (x > n/2)  and (x % 2 != 0),\
        range(n))))

def test3(n):
    l = [hex(i) for i in range(n)]

def test4(n):
    l = list(map(hex, range(n)))

def duree(fonction, n=10000000):
    debut = clock()
    fonction(n)
    fin = clock()
    return(fin - debut)

```

```
print("*"*10,"Utilisation avec des fonctions lambda de l'utilisateur:")
print(" {:.<20} {:f} secondes".format("Comprehension", duree(test1)))
print(" {:.<20} {:f} secondes".format("map + filter", duree(test2)))
print("*"*10,"Utilisation avec une fonction predefinie du langage:")
print(" {:.<20} {:f} secondes".format("Comprehension", duree(test3)))
print(" {:.<20} {:f} secondes".format("map", duree(test4)))
```

```
********** Utilisation avec des fonctions lambda de l'utilisateur:
 Comprehension....... 4.014930 secondes
 map + filter........ 5.635100 secondes
********** Utilisation avec une fonction predefinie du langage:
 Comprehension....... 2.431466 secondes
 map................. 2.281591 secondes
```

3. `zip`

La syntaxe de cette fonction est la suivante :

`zip` (*sequence1*, [*sequence2*, ...])

et le résultat est un *iterateur* de tuples.

Les séquences passées en paramètres doivent avoir le même nombre d'éléments n. Cette fonction crée un iterateur formé de n tuples de longueur le nombre de paramètres, le ième tuple étant formé des éléments d'index i de chaque séquence. Si les séquences ne sont pas de même longueur, elles sont tronquées à la longueur de la plus petite d'entre elles.

Exemples :

```
>>> li1 = list('abcdef')
>>> print(li1)
['a', 'b', 'c', 'd', 'e', 'f']
>>> li2=[i + 1 for i in range(6)]
>>> print(li2)
[1, 2, 3, 4, 5, 6]
>>> print(zip(li1, li2)) # ce n'est pas une liste mais un itérateur
<zip object at 0x0209E5F8>
>>> print(list(zip(li1, li2))) # d'où l'emploi de la fonction list
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5), ('f', 6)]
>>> li3=list('ABCD')
>>> print(list(zip(li2, li1, li3)))
[(1, 'a', 'A'), (2, 'b', 'B'), (3, 'c', 'C'), (4, 'd', 'D')]
```

Les exemples ci-dessous montrent l'utilisation de la fonction `zip` pour calculer la somme puis le produit scalaire de deux vecteurs dont les coordonnées sont données sous forme d'une liste. Cela permet de façon élégante d'éviter l'emploi d'une boucle `for`.

```
>>> u = [1, 2, 3, 4]
>>> v = [5, 6, 7, 8]
>>> print([x[0] + x[1] for x in zip(u, v)])
[6, 8, 10, 12]
>>> print([x + y for (x, y) in zip(u, v)])
[6, 8, 10, 12]
>>> print(sum(x * y for x, y in zip(u, v)))
70
```

★ ★ ★ ★
★ ★ ★
★ ★
★