Listes

I Définitions et opérations sur les listes

Définition (Listes)

Les listes de Python correspondent à une structure de données généralement appelée tableaux. Elles consistent en un ensemble d'objets de types quelconques (éventuellement dépareillés) ordonnés et accessibles par leur numéro de position appelé indice ou index. Attention le premier élément porte le numéro 0. On dit que c'est une structure de données de type composite.

Pour définir explicitement une liste à partir de ses éléments on utilise les délimiteurs [et] et le séparateur ,. Si il n'y a pas d'éléments, on obtient la liste vide.

```
>>> L = [-3, 'MPSI', 2.018]
>>> type(L)
<class 'list'>
>>> listeVide = []
```
1

On peut définir une liste de listes[1] ou même une liste de listes de listes, etc. La fonction **len** renvoie le nombre d'éléments d'une liste.

```
>>> tables = [[0, 2, 4, 6], [0, 3, 6, 9], [0, 4, 8, 12]]
>>> len(tables)
3
```
2

Pour accéder à un élément, on utilise son indice. Attention, les indices commencent à 0.

```
>>> L[0]
-3
>>> L[2]
2.018
>>> tables[2][1]
4
```
3

Le dépassement de la plage des indices génère une erreur classique :

```
>>> len(L)
3
>>> L[3]
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    L[3]
IndexError: list index out of range
```
4

1. C'est une bonne façon de représenter un tableau à double entrée.

Les indices négatifs permettent de parcourir la liste à l'envers :

```
>>> L[-1]
2.018
>>> L[-4]
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    L[-4]
IndexError: list index out of range
>>> L[-3]
-3
```
5

Pour supprimer un élément on utilise l'instruction **del** (delete signifie effacer en anglais).

```
>>> L
[-3, 'MPSI', 2.018]
>>> del L[1]      # effet de bord
>>> L
[-3, 2.018]
```
6

On peut modifier une liste en changeant la valeur d'un de ces éléments. On dit qu'une liste est **mutable** ou **modifiable** :

```
>>> chiffresAnnee = [2, 0, 1, 8]
>>> chiffresAnnee[3] = 9
>>> chiffresAnnee
[2, 0, 1, 9]
```
7

Intermède : notion d'objets et de méthode en Python

Python est un langage orienté objet ; tout ce qu'on manipule est objet : une variable, une liste, une fonction sont des objets. Un objet est muni de « méthodes », ce sont les actions que l'on peut lui appliquer.

Pour donner une idée de ce que c'est, on pourrait dire qu'un objet de type « chien » devrait disposer des méthodes : manger, aboyer, chercher le baton (liste non exhaustive).

Pour appliquer une méthode de sa classe à un objet on utilise la syntaxe :

nom_objet.nom_methode

Par exemple la méthode .sort() permet de trier une liste :

```
>>> listeEntiers = [4, 0, 7, 2]
>>> listeEntiers
[4, 0, 7, 2]
>>> listeEntiers.sort()   # effet de bord
>>> listeEntiers
[0, 2, 4, 7]
```
8

Les principales méthodes applicables aux listes :

```
>>> L = [1, 3, 5, 7]
>>> L.append(9)     # ajouter un élément en fin de liste
>>> L
[1, 3, 5, 7, 9]
>>> L.pop(1)    # renvoie l'élément d'indice donné et le supprime
3
>>> L
[1, 5, 7, 9]
>>> L.insert(3, 5)     # insère à la position donnée l'élément
>>> L
[1, 5, 7, 5, 9]
>>> L.count(5)     # renvoie le nombre d'occurences de la valeur
2
>>> L.index(5)     # renvoie l'indice de la première occurence
1
>>> L.remove(5)     # élimine la première occurence de la valeur
>>> L
[1, 7, 5, 9]
>>> L.extend([11, 13])    # ajoute les éléments d'une autre liste
>>> L
[1, 7, 5, 9, 11, 13]
```

9

La fonction **list**() renvoie une liste correspondant à son argument :

```
>>> list('abcbde')
['a', 'b', 'c', 'b', 'd', 'e']
>>> list(range(4, 10, 2))
[4, 6, 8]
```

10

Les listes sont *itérables*. On peut les parcourir avec une boucle for **sans utiliser les indices** :

```
>>> noms = ['Alice', 'Bernard', 'Cécile']
>>> for x in noms:
        print(x)

Alice
Bernard
Cécile
```

11

Mais le parcours par indice est aussi possible (il est plus général et peut être nécessaire) :

```
>>> nbPremiers = [2, 3, 5, 7, 11]
>>> for i in range(len(nbPremiers)):
        print(nbPremiers[i])
2
3
5
7
11
```

12

On peut tester l'appartenance à une liste :

```
>>> 4 in nbPremiers
False
```

13

Exercice 1

On suppose donnée une liste L et une variable x. Écrire une fonction qui teste si x apparait dans L.

Exercice 2

On suppose donnée une liste d'entiers L et un entier x. Écrire un code qui renvoie la liste (éventuellement vide) des indices des occurrences de la valeur x dans la liste L.

Exercice 3 Programmer la suite (u_n) définies par $u_0 = 1$ et $u_{n+1} = \sum_{k=0}^{n} (nk - 1)u_k$

Exercice 4 Programmer une fonction `extension(L, M)` qui prend en paramètres deux listes `L` et `M` et qui ajoute les éléments de `M` à la liste `L`. C'est bien un effet de bord recherché !

Entraînement 1 

Écrire une fonction `somme(L)` qui prend en paramètre une liste de nombres `L` et qui renvoie la somme de ses éléments (bien entendu on n'utilisera pas la fonction **sum**() intégrée à Python qui fait cela).

Entraînement 2 

Programmer la suite (u_n) définie par $u_0 = 1$ et $u_n = \sum_{k=0}^{n-1} \frac{u_k}{n-k}$.

Entraînement 3 

`L` est une liste non vide d'entiers positifs.

Écrire une fonction `indice_max(L)` qui renvoie l'indice de la première occurence d'un élément maximal de `L`.

Exemple : `indice_max([2, 5, 0, 5])` doit renvoyer 1.

Entraînement 4 

Les listes `L` et `M` sont de même longueur. Écrire une fonction `alterne(L, M)` qui renvoie une liste obtenue en intercalant les éléments de `L` et `M`. On commence par un élément de `L`.

II Concaténation et tranchage

On peut mettre bout à bout deux listes pour en former une troisième.

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, 5]
>>> L1_2 = L1 + L2
>>> L1_2
[1, 2, 3, 4, 5]
```
14

Chacun sait que la répétition d'une addition c'est une multiplication :

```
>>> L1*4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```
15

On peut découper une tranche (en anglais : *slicing*) dans une liste.

```
>>> L = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
>>> L[1:5]
[1, 4, 1, 5]
>>> L[4:]
[5, 9, 2, 6, 5, 3]
>>> L[:5]
[3, 1, 4, 1, 5]
```
16

La syntaxe est celle de la fonction **range** :

L[i:j:p] renvoi la liste des éléments L[k] pour k partant de i avec un pas de p en ne prenant que des valeurs strictement inférieures à j.

La valeur par défaut de i est 0, celle de j est la longueur de la liste et celle de p est 1.

```
>>> L[2:8:2]
[4, 5, 2]
>>> L[:5:2]
[3, 4, 5]
>>> L[2::2]
[4, 5, 2, 5]
```
17

On peut même remplacer une tranche par une autre liste de taille différente :

```
>>> L = [0, 4, 8, 12]
>>> L[1:3] = [-1, -2, -3, -4, -5]
>>> L
[0, -1, -2, -3, -4, -5, 12]
```
18

Ou la supprimer :

```
>>> del L[2:4]
>>> L
[0, -1, -4, -5, 12]
```
19

Exercice 5 Prévoir le résultat.

```
>>> M = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L = [1, 3, 5, 7, 9]
>>> M[::2] = L
>>> M
```
20

III Listes par compréhension

Exercice 6

Quelle liste produit ce code ?

```
L = []
for k in range(10):
    L.append(k**2)
```
21

Une façon plus courte d'obtenir le même résultat est d'utiliser les listes décrites par compréhension :

```
>>> listeCarres = [k**2 for k in range(10)]
>>> listeCarres
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```
22

On peut y ajouter des conditions :

```
>>> premiers = [2, 3, 5, 7, 11, 13]
>>> composes = [k for k in range(2, 14) if not(k in premiers)]
>>> composes
[4, 6, 8, 9, 10, 12]
```

23

Exercice 7 Écrire une fonction `multiples(n, p)` qui renvoie une liste de listes des multiples des nombres k de 1 à p en commençant par 0 et en finissant par kn. Par exemple

```
multiples(4, 3)
[[0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12]]
```

Exercice 8

Écrire une liste décrite en compréhension équivalente au tranchage L[i:j:p].

Bilan des méthodes pour la construction des listes

On peut construire une liste

1. Explicitement par ses éléments,

```
L = [1, 2, 3, 4]
```

2. avec la fonction **list** qui transforme un itérable en liste,

```
>>> list(range(1,15,2))
[1, 3, 5, 7, 9, 11, 13]
```

3. par une description en compréhension,

```
>>> [k for k in range(20) if k%3 != 0]
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

4. à l'aide d'une boucle en partant de la liste vide.

```
>>> L = []
>>> for k in range(10):
        L.append(2k)
>>> L
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

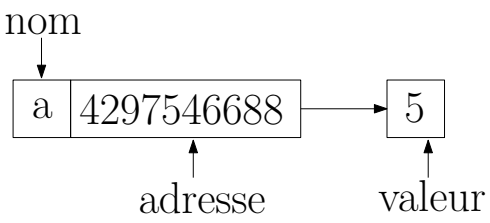
Entraînement 5

Écrire une fonction `images(f, L)` qui prend en argument une fonction `f` et une liste `L` et renvoie la liste formée des images de chaque élément de `L` par `f`. Écrire une fonction `appliquer(f, L)` qui modifie la liste `L` en remplaçant chaque élément par son image.

Remarque : Ne pas utiliser la fonction **map** de python qui réalise cette opération mais en générant un objet itérable.

IV Conséquences du caractère modifiable des listes

Avant de continuer l'étude des listes, il faut revenir aux variables et regarder d'un peu plus près comment elles sont gérées par Python. Contrairement à ce que nous avons fait mine de croire, une variable est un nom et une adresse. Il s'agit de l'adresse mémoire où se trouve la valeur de la variable.

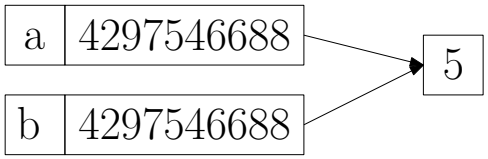


On obtient la référence d'une variable grâce à la fonction **id**() :

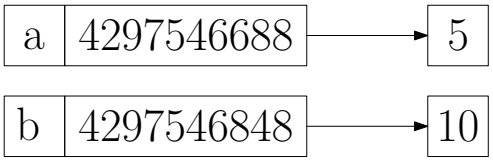
```
>>> a = 5
>>> id(a)
4297546688
```
24

Le type entier ('int') n'est **pas modifiable** ce qui signifie qu'une valeur de type entier ne peut pas être modifiée après sa création. Concrètement, voilà ce qui se passe :

```
>>> b = a
>>> id(b)
4297546688
```
25




```
>>> b = 2*b
>>> id(b)
4297546848
>>> b
10
```
26



Lors de l'exécution de l'instruction b = 2*b, il n'y a pas modification de la valeur contenue dans la mémoire [2] à l'adresse 4297546688 mais Python crée une nouvelle valeur pour le résultat. Ensuite la référence de b est modifiée (4297546848) pour pointer vers cette valeur.

NB : Les types : chaine de caractères ('str'), flottant ('float'), booléen ('bool'), tuple ('tuple') et fonction ('fonction') sont tous non modifiables et se comporteront donc comme les entiers.

Regardons maintenant ce qui se passe pour les listes :

```
>>> L = [8, 2, 9]
>>> id(L)
193491552
>>> L[0] = 10              # il y a modification de la liste
>>> L
[10, 2, 9]
>>> id(L)
193491552
```
27

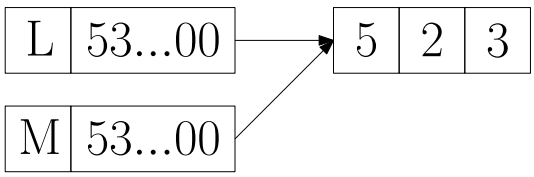
2. Ce serait catastrophique car la valeur de a ne serait plus 5 !

```
>>> L.append(18)          # L.append(a) opère sur la liste
>>> L
[10, 2, 9, 18]
>>> id(L)
193491552
>>> L = L + [12]          # L = L + [a] crée une nouvelle liste
>>> L
[10, 2, 9, 18, 12]                                          28
>>> id(L)
184650288
>>> L += [98, 56]                  # L += M opère sur la liste
>>> L
[10, 2, 9, 18, 12, 98, 56]
>>> id(L)
184650288
```

Il est évidemment en général plus économe de travailler directement sur une liste en rajoutant un élément plutôt que de recopier toute la liste pour en créer une nouvelle (si on veut du non-modifiable les tuples sont là pour ça).

```
>>> L = [5, 2, 3]
>>> M = L
>>> id(L), id(M)
(53566400, 53566400)
>>> L[0] = 8
>>> L
[8, 2, 3]
```

```
>>> M
[8, 2, 3]
>>> M[0] = 9
>>> L
[9, 2, 3]
>>> M
[9, 2, 3]
```



On constate que pour des listes (et des objets mutables en général), l'affectation `M = L` ne s'exécute pas comme pour les types simples : il n'y a pas évaluation puis affectation mais création d'une nouvelle variable qui pointe vers la même adresse. C'est l'adresse de `L` qui est utilisée et pas sa valeur.

```
def ajoute(L, x):
    L.append(x)
    return L
```

```
>>> L = [3, 8, 1]
>>> ajoute(L, 7)
[3, 8, 1, 7]
>>> L
[3, 8, 1, 7]
```

Comme on le voit, cette fonction est à effet de bord : elle modifie la liste passée en paramètre. Il y a encore ici un phénomène important que l'on avait passé sous silence jusqu'ici. Dans le cas des objets mutables, ils ne sont pas passés par valeur mais par adresse et peuvent donc être modifiés par l'exécution d'une fonction (contrairement à ce que l'on a vu pour les arguments de type simple).

Comme nous l'avons vu, l'affectation `M = L` ne crée pas une copie de `L` mais une variable qui pointe à la même adresse que `L` ce qui peut être gênant puisque toute modification d'une liste est répercutée sur l'autre.

Pour créer une copie d'une liste, il y a plusieurs méthodes :

```
>>> L = [6, 9, 8]
>>> M = []
>>> for k in L:
        M.append(k)
>>> M
[6, 9, 8]
>>> L[0] = 19
>>> (L, M)
([19, 9, 8], [6, 9, 8])
```

à la main

```
>>> L = [6, 9, 8]
>>> M = L[:]
>>> M
[6, 9, 8]
>>> (id(L), id(M))
(72865472, 72865072)
>>> L[0] = 87
>>> (L, M)
([87, 9, 8], [6, 9, 8])
```

par tranchage

```
>>> L = [6, 9, 8]
>>> M = list(L)
>>> M
[6, 9, 8]
>>> (id(L), id(M))
(72569000, 73451600)
>>> L[1] = 98
>>> (L, M)
([6, 98, 8], [6, 9, 8])
```

avec la fonction `list`

```
>>> L = [6, 9, 8]
>>> from copy import *
>>> M = copy(L)
>>> (id(L), id(M))
(72868008, 73752448)
>>> M
[6, 9, 8]
```

avec la fonction `copy` (du module `copy`)

L'utilisation de la fonction `list` est simple et facilement compréhensible. On pourra donc retenir cette technique (mais attention, elle ne fait pas de copie en profondeur, voir plus loin).

Exercice 9

Écrire une fonction permettant de permuter deux éléments d'une liste. On écrira une version avec effet de bord : `permute(L, a, b)` et une version sans : `permutation(L, a, b)`.

Entraînement 6

Écrire une fonction prenant en paramètre une liste L (que l'on supposera constituée d'entiers) et renvoyant `True` ou `False` suivant que L est dans l'ordre croissant ou pas. On écrira un code avec une boucle `while` puis avec une boucle `for`.

Entraînement 7 

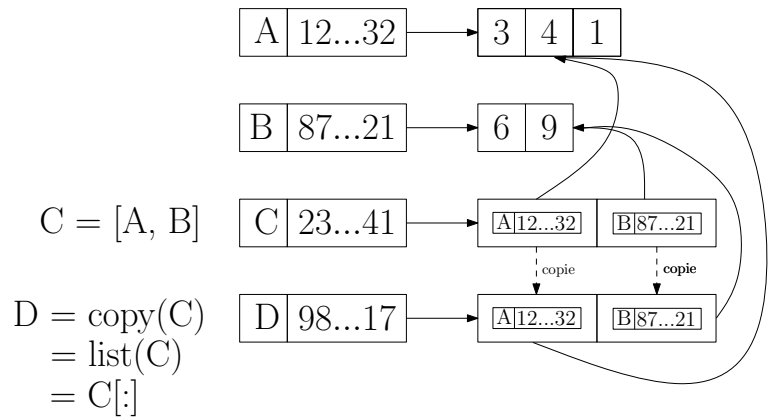
Écrire une fonction prenant en paramètres une liste L et un objet x et supprimant de L toutes les occurrences de x (on écrira une version modifiant L et une version renvoyant une nouvelle liste).

Voici enfin une autre difficulté dans la gestion des listes par adresse : les listes de listes (problème que l'on rencontre lorsque l'on travaille avec des matrices par exemple).

```
>>> L = [1, 7, 8]
>>> M = [9, 1, 3]
>>> N = [L, M]
>>> N
[[1, 7, 8], [9, 1, 3]]
>>> L[1] = 14
>>> N
[[1, 14, 8], [9, 1, 3]]
```

```
>>> A = [3, 4, 1]
>>> B = [6, 9]
>>> C = [A, B]
>>> D = list(C)
>>> (C, D)
([[3, 4, 1], [6, 9]], [[3, 4, 1], [6, 9]])
>>> A[1] = 69
>>> (C, D)
([[3, 69, 1], [6, 9]], [[3, 69, 1], [6, 9]])

>>> from copy import *
>>> E = deepcopy(C)
>>> (C, E)
([[3, 69, 1], [6, 9]], [[3, 69, 1], [6, 9]])
>>> A[0] = 18
>>> (C, E)
([[18, 69, 1], [6, 9]], [[3, 69, 1], [6, 9]])
```



La fonction `deepcopy` (du module `copy`) permet de faire une copie en profondeur d'une liste de listes (récursivement, tant qu'il y a une liste). On peut aussi se débrouiller à la main ...

Exercice 10 Expliquer le phénomène suivant :

```
>>> L = [[0]*2]*2
>>> L
[[0, 0], [0, 0]]
>>> L[0][0] = 1
>>> L
[[1, 0], [1, 0]]
```

```
>>> L = [[0]*2 for k in range(2)]
>>> L
[[0, 0], [0, 0]]
>>> L[0][0] = 1
>>> L
[[1, 0], [0, 0]]
```

Exercice 11 Expliquer le phénomène suivant :

```
>>> L = [[0,1], [2,3], [4,5]]
>>> M = L[1:3]
>>> M
[[2, 3], [4, 5]]
>>> M[0][0] = 12
>>> M
[[12, 3], [4, 5]]
>>> L
[[0, 1], [12, 3], [4, 5]]
```

```
>>> L = [1, 2, 3, 4]
>>> M = L[1:4]
>>> M
[2, 3, 4]
>>> M[0] = 12
>>> M
[12, 3, 4]
>>> L
[1, 2, 3, 4]
```