Chapitre 3

Les listes

3.1 Les listes en python : introduction

3.1.1 Motivations

• Dès que l'on commence à manipuler en python un grand nombre de données l'usage de variable numérique devient insuffisant.

Exemple : imaginons que l'on veuille stocker les notes des élèves d'une classe pour calculer leur moyenne et leur écart-type. On peut imaginer d'utiliser N variables de type `float` (N étant le nombre d'élèves de la classe), puis d'écrire des fonctions `moyenne(.)` et `ecarttype(.)` prenant N paramètres ; fastidieux lorsque N=47, et il faudra la réécrire pour chaque nouvel effectif...

```
def moyenne47(n1, n2, ..., n47):
    return (n1 + n2 + ... + n47) / 47
```

• En `python` la structure de donnée qui va nous aider est la *liste*. c'est un **objet** de type `list` qui permet de collecter des éléments : données de type quelconque : `int`, `float`, `str`, `bool`, ou d'autres listes, etc...

3.1.2 Listes : création et accès aux éléments

Exemple :

```
>>> liste = [1, 2, 3, 'toto']
>>> print(liste)
[1, 2, 3, 'toto']
>>> type(liste)
list
>>> len(liste)
4
```

Une liste est créée à l'aide d'une affectation. Ses éléments sont entre crochets `[.]`. La fonction `len(.)` prend en argument une liste et retourne son nombre d'éléments. Les éléments d'une liste s'obtiennent grâce à leur **indice** entre crochets. Attention le premier élément a pour indice 0 !

```
>>> liste[0], liste[1], liste[len(liste) - 1]
(1, 2, 'toto')
```

3.1.3 Exemple : calcul de moyenne

Exemple : reprenons notre problème de calcul de la moyenne de notes.

Saisissons la liste des notes.

```
>>> l = [10, 12, 14, 6, 8, 15, 3, 17]    # la liste de notes
```

Mettons à profit ce que nous avons vu sur les listes ainsi que la fonction `sum(.)` qui prend en argument une liste et retourne, lorsque c'est possible, le résultat de l'opération `'+'` sur ses éléments.

Définition de la fonction `moyenne(.)` qui s'applique à toute liste non vide de nombres :

```
>>> def moyenne(liste):                # fonction moyenne(.)
...     return sum(liste) / len(liste)
...
>>> moyenne(l)
10.625
```

On obtient le résultat attendu, la moyenne est de 10.625.

3.1.4 Les listes sont modifiables

Les listes sont des objets **modifiables** (on dit aussi **mutables**) : on peut modifier leurs éléments.

```
>>> liste = [1, 2, 3, 'toto']    # une liste
>>> liste[0] = 'le début' ; liste[2] = [-1, -2, -3]
>>> print(liste)
['le début', 2, [-1,- 2, -3], 'toto']
>>> print(liste[-1], liste[-2])
'toto' [-1, -2, -3]
>>> print(liste[-5], liste[4])
... IndexError: list index out of range
```

On modifie un élément de la liste en lui affectant une nouvelle valeur (de n'importe quel type, simple ou complexe). L'indice -1 permet d'obtenir le dernier élément. C'est plus simple que `liste[len(liste) - 1]`. L'indice -2 l'avant-dernier, etc... Un indice qui n'est pas compris entre `-len(liste)` et `len(liste)-1` produit une erreur `'IndexError'`.

3.1.5 Liste de listes

Reprenons la liste précédente :

```
>>> print(liste)
['le début', 2, [-1,- 2, -3], 'toto']
>>> type(liste[0]), type(liste[1]), type(liste[2])
(str, int, list)
>>> print(liste[2][0])            # liste[2] est une liste
-1
```

```
 >>> l=[['a', 'b'], ['c', 'd']]
>>> print(l[0][0], l[0][1], '\n', l[1][0], l[1][1])
a b
c d
```

Les éléments d'une liste sont des valeurs de différents types, celles qu'on lui a affectées.

Une liste de listes permet de constituer un tableau bi-dimensionnel, etc...

3.1.6 Appartenance d'un élément à une liste : `in`

• Le mot-clef `in` permet de déterminer si un élément appartient ou non à une liste.

```
>>> liste = [1, -3, 5, 17.0, 2]
>>> 1 in liste
True
>>> -1 in liste
False
>>> -3.0 in liste
True
>>> 17 in liste
True
>>> 'toto' in liste
False
```

3.1.7 Parcours d'une liste : `for Variable in Liste:`

• Avec en plus la commande `for` on peut faire parcourir à une variable les éléments d'une liste :

```
>>> for i in liste:
...     print(i)
...
1
-3
5
17.0
2
```

• Le parcours d'une liste peut se faire en sens inverse en utilisant la fonction `reversed` :

```
>>> for i in reversed(liste):
...     print(i)
...
2
17.0
5
-3
1
```

3.1.8 Exemples

• Somme des éléments d'une suite numérique (même effet que `sum`) :

```
def somme(L):
    S = 0
    for x in L:
        S += x
    return S
```

• Moyenne des éléments d'une suite numérique :

```
def moyenne(L):
    S = 0
    for x in L:
        S += x
    return S/len(L)
```

• Recherche du maximum dans une liste numérique :

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

3.1.9 La méthode append

La **méthode** list.append() appliquée aux objets de type liste permet d'ajouter un élément en queue de liste :

```
>>> liste=[ ] ; print(liste)
[]
>>> liste.append(1) ; print(liste)
[1]
>>> liste.append(2) ; print(liste)
[1, 2]
```

Exemple : créer la liste des carrés des entiers compris entre 0 et 20 :

```
>>> listeCarrés = [ ]        # initialisation
>>> for i in range(21):
...    listeCarrés.append(i ** 2)    # actualisation
...
>>> print(listeCarrés)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225, 256, 289, 324, 361, 400]
```

3.2 Les listes en python : approfondissement

3.2.1 Les méthodes de la classe list

Soit L une liste :

Méthode :	Action :
L.append(x)	Pour ajouter x à la fin de la liste L
L.extend(L2)	Pour ajouter la liste L2 à la suite de la liste L
L.insert(i,x)	Pour insérer l'élément x en position i dans L
L.pop()	Pour retirer et renvoyer le dernier élément dans L
L.pop(i)	Pour retirer et renvoyer l'élément en position i dans L
L.remove(x)	Pour retirer la première occurence de x dans L
L.index(x)	Renvoie la 1^{ere} position de x dans L. Message d'erreur si aucune.
L.count(x)	Renvoie le nombre d'occurrences de x dans L.
L.sort()	Trie la liste par ordre croissant.
L.reverse()	Renverse l'ordre des éléments de la liste.

On emploiera surtout les deux méthodes .append et .pop.

3.2.2 Saucissonnage ou slicing

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']   >>>
liste1 = liste[2:5]    # tranche (slice) de la liste
>>> print(liste1)
['c', 'd', 'e']
```

L'instruction `>>> liste1 = liste[2:5]` crée une nouvelle liste `liste1` dont les éléments sont ceux de `liste` allant de l'indice 2 (inclus) à l'indice 5 (exclu). On l'appelle une *tranche* (slice en anglais) de la liste.

Les indices entre crochets peuvent sortir de la plage d'indice de la liste :

```
>>> liste2 = liste[0:100]    # tranche des indices de 0 à 100
>>> print(liste2)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Un troisième paramètre définit un pas :

```
>>> print(liste[6:2:-1])    # de 6 à 2 par pas de -1
['g', 'f', 'e', 'd']
```

LISTE[Indice départ (inclus) : Indice arrivée (exclus) : Pas]

python crée la tranche en copiant l'élément de LISTE d'indice Indice de départ, puis tous les éléments obtenus en ajoutant successivement Pas à l'in-

dice, tant qu'on ne dépasse pas `Indice d'arrivée`. **Par défaut :** Pas = 1

— si Pas > 0 : **par défaut** départ = 0 ; arrivée = `len`(LISTE)
— si Pas < 0 : **par défaut** départ = `len(LISTE)-1` : arrivée = -1

```
>>> print(liste[::2])    # slicing par pas de 2
['a', 'c', 'e', 'g', 'i']
print(liste[::-1])    # slicing par pas de -1
['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
>>> print(liste[3::2])    # départ de 3, par pas de 2
['d', 'f', 'h']
>>> print(liste[:3:-1]) # de fin à 3 par pas de -1
['i', 'h', 'g', 'f', 'e']
```

3.2.3 Copie de liste

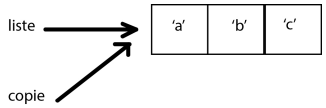
Il faut prendre garde à la façon dont `python` copie une liste... Illustrons cela sur un exemple :

```
>>> liste = ['a', 'b', 'c']    # définition d'une liste
>>> copie = liste            # puis copie de la liste
>>> print(copie)
['a', 'b', 'c']
>>> liste[0] = 'modifié'    # Modifons un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
```

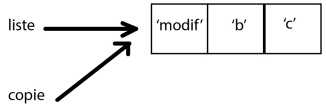
```
>>> print(copie)    # regardons ce qu'est devenue la copie
['modifié', 'b', 'c']
```

En Modifiant un élément de la liste originelle, la copie aussi a été modifiée...

Explication : En fait une liste en `python` ne contient que l'adresse mémoire où sont stockés ses éléments ; c'est ce que l'on appelle un pointeur.



Quand on copie `liste` dans `copie` c'est cette adresse mémoire qui est copiée. C'est un alias qui est créé.



Quand on modifie un élément d'une liste, il est alors aussi modifié dans la copie. L'avantage étant qu'on encombre moins la mémoire centrale puisque les éléments de la liste ne figurent qu'en un seul emplacement mémoire.

On peut contourner ce problème grâce à : `copie = liste[:]` qui fait une 'copie superficielle' :

```
>>> liste = ['a', 'b', 'c']    # définition d'une liste
>>> copie = liste[:]            # puis copie superficielle de
la liste
>>> print(copie)
['a', 'b', 'c']
>>> liste[0] = 'modifié'    # Modifons un élément dans liste
>>> print(liste)
['modifié', 'b', 'c']
>>> print(copie)    # regardons ce qu'est devenue la copie
['a', 'b', 'c']
```

`python` fait une copie des éléments de la liste.. mais lorsqu'un élément est une liste, c'est l'adresse mémoire des objets de la liste qui est copiée. Aussi si l'un des éléments est une liste on retombe sur le même problème :

```
>>> liste = ['a', 'b', [0, 1]]    # définition d'une liste
>>> copie = liste[:]        # puis copie superficielle
>>> liste[0] = 'modifié' # Modifons un élément dans liste
>>> liste[2][0] = 'MODIF' # et un élément dans liste[2]
>>> print(liste); print(copie)    # regardons...
['modifié', 'b', ['MODIF',1]]
['a', 'b', ['MODIF',1]]
```

Pour contourner totalement le problème utiliser la méthode `liste.deepcopy()` qui effectue une 'copie profonde'. Il faut avoir auparavant importé la bibliothèque `copy` :

```
>>> from copy import deepcopy; copie = deepcopy(liste).
```

3.2.4 Liste définies par compréhension

On peut définir une liste à l'aide des mots-clés `for`, `in` et `if` comme on définit un ensemble en mathématiques 'par compréhension' :

`[f(x) for x in liste]` correspond à $\{f(x) | x \in liste\}$

Exemple :

```
>>> liste = range(10)            # définition de la liste des
entiers de 0 à 9
>>> LISTE = [x**2 for x in liste]    # LISTE des carrés des
éléments de liste
>>> print(LISTE)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`[f(x) for x in liste if Condition(x)]`
correspond à $\{f(x) | x \in liste\ tel\ que\ Condition(x)\}$
Exemple :

```
>>> Liste = [x**2 for x in liste if (x % 2 == 0)]
>>> print(Liste)
[0, 4, 16, 36, 64]
```

Exemple : Liste des multiplies de 12 entre 0 et 100 :

```
>>> list = [x for x in range(101) if x % 12 == 0]
>>> print(list)
[0, 12, 24, 36, 48, 60, 72, 84, 96]
```

Exemple : Liste des diviseurs d'un entier naturel N :

```
>>> N=120
diviseurs = [x for x in range(1,N+1) if (N % x == 0)]
>>> print(diviseurs)
[1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120]
```

3.2.5 Exercice : le crible d'Erathostène

• Ecrire une fonction qui prend en paramètre un entier positif `n` est retourne la liste des nombres premiers inférieurs ou égaux à `n` en appliquant le crible d'Erathostène.

• Le crible d'Erathostène commence à barrer dans la liste des entiers de 2 à `n` tous les multiples stricts des nombres non barrés qui sont successivement lus. A la fin tous les nombres non barrés sont premiers.

```
def erathostene(n):
    L = [x for x in range(n+1)]
    L[1] = 0
    for x in L:
        if x!= 0:
            m = 2
            while x*m <= n:
                L[x*m] = 0
                m+= 1
    return [p for p in L if p!=0]
```

• Exemple :

```
>>> erathostene(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
59, 61, 67, 71, 73, 79, 83, 89, 97]
```

• On peut l'améliorer en ne parcourant la liste que jusqu'à $\lfloor \sqrt{n} \rfloor$:

```
def erathostene(n):
    L = [x for x in range(n+1)]
    L[1] = 0
    for k in range(2,int(n**0.5)+1):
        x = L[k]
        if x!= 0:
            m = 2
            while k*m <= n:
                L[k*m] = 0
                m+= 1
    return [p for p in L if p!=0]
```