Chapitre 5
Listes

1 Listes

Une **liste** est une suite finie d'éléments de types quelconques. Les listes permettent de stocker dans une seule structure un grand nombre de données (sans créer une variable pour chacune des données). Très souvent, tous les éléments d'une même liste seront du même type.

En Python, les listes, de type **`list`**, sont notées entre crochets et les éléments d'une liste sont séparés par des virgules.

```
type([0, 1, 2])

type([-7, "Bonjour", 12.3, True])

type([]) # La liste [] est vide.
```

1.1 Opérations sur les listes

Les opérations sur les listes sont les suivantes.

Opérations sur les listes	
==	Égalité
len	**Longueur** : nombre d'éléments
+	**Concaténation**
$n*$	Concaténation n fois avec elle-même

```
[0, 1, 2] == [1, 0, 2]
# Dans une liste, l'ordre des éléments est important.

len([0, 1, 2, 3, 4, "Python"])

[1, 3, 6] + [8, "coucou"]

li = 3 * [1, "orange"] + [2.3, 4]
```

1.2 Accès aux éléments

Dans une liste de longueur n, les éléments sont **indexés** de 0 à $n-1$.

Pour tout $k \in [\![0; n-1]\!]$, on accède à l'**élément d'indice** k d'une telle liste `liste` ainsi :

```
liste[k]
```

Remarquons que comme la numérotation des éléments démarre à zéro, le premier élément de la liste est celui d'indice 0 et le dernier est celui d'indice `len(liste)` -1.

```
L = [2, 7, -1, [4, -5, 3], "Python"]

len(L)

L[2]

L[3][1]

type(L[3])

L[4]

L[len(L)]

L[len(L) - 1]
```

1.3 Slicing

Pour former des sous-listes d'une liste, on peut utiliser la technique du **slicing**, en spécifiant une **tranche** d'indices, comme pour les chaines de caractères.
Dans la liste `liste`, le slicing s'écrit de la manière suivante :

- ⋆ si i et j sont deux entiers tels que $i \leqslant j$, alors

  ```
  liste[i : j]
  ```

  désigne la liste constituée des éléments de `liste` dont l'indice est compris entre i inclus et j exclu ;

- ⋆ si i est un entier, alors

  ```
  liste[i : ]
  ```

  désigne la liste constituée des éléments de `liste` dont l'indice est supérieur ou égal à i : c'est la même sous-liste que `liste[i : len(liste)]` ;

- ⋆ de même, si j est un entier, alors

  ```
  liste[ : j]
  ```

  désigne la liste constituée des éléments de `liste` dont l'indice est strictement inférieur à j : c'est la même sous-liste que `liste[0 : j]` ;

- ⋆ si i, j et r sont trois entiers, alors

  ```
  liste[i : j : r]
  ```

  est la chaine constituée des éléments de `liste` dont les indices vont de i inclus à j exclu, en faisant des pas de r.

```
L = list(range(15))

L

L[2 : 4]

L[1 : 13 : 2]

L[13 : 1 : -2]

L[ : 2] + L[3 : ]

L[ : ] # C'est la liste tout entière !
       # On pourra utiliser cette technique
       # pour créer une copie de la liste L.
```

2 Modification d'une liste

Les listes, contrairement aux chaines de caractères, sont des objets **mutables** : on peut les modifier (leur ajouter un élément, leur supprimer un élément, modifier un de leurs éléments) directement, sans changer leur emplacement en mémoire, alors que pour modifier une chaine de caractères, on doit en fait la recréer entièrement.

La mutabilité est une caractéristique très pratique et très importante des listes. Nous allons voir comment elle s'opère et, plus tard, quelles conséquences impromptues elle peut avoir.

2.1 Modification d'un élément

Pour modifier l'un des éléments d'une liste, on procède de la manière suivante : l'instruction

```
liste[k] = nouveau
```

modifie l'élément d'indice `k` de la liste `liste` et lui affecte la valeur `nouveau`.

```
L = [3, 1, -2, -7, 3, 0, 5, 4, 9, -1]

L

L[4] = "Tiens, ça a changé ici !"

L

L[2] = L[8]

L
```

Sur le même principe, on peut également modifier toute une sous-liste d'une liste donnée.

```
L = list(range(10))

L

Li = [2, 4, 6, 8]

L[3 : 7] = Li

L
```

2.2 Suppression du dernier élément

L'instruction

```
liste.pop()
```

supprime le dernier élément de la liste `liste` et le renvoie, de sorte que l'on puisse ensuite l'affecter à une variable.

```
L = [-1, 8, -4, 5, 2]

x = L.pop()

L

x
```

2.3 Ajout d'un élément

L'une des modifications les plus fréquentes opérées sur une liste est l'ajout d'un nouvel élément en fin de liste. Pour ce faire, on utilise la méthode **append** : l'instruction

```
liste.append(nouveau)
```

ajoute l'élément `nouveau` à la fin de la liste `liste`.

```
L = [1, 3, -8, 4]

L.append(7)

L

L.append("Python")

L
```

2.4 Une conséquence de la mutabilité

Une conséquence importante de la mutabilité — nous en verrons d'autres plus tard dans l'année — est qu'une fonction prenant en entrée une liste peut modifier son argument !

```
def echange(L, i, j):
    """
    Entrées : une liste L, deux indices i et j de la liste L.
    Échange les éléments d'indices i et j dans la liste L.
    Sortie : aucune, la liste L est modifiée en place.
    """
    aux = L[i]
    L[i] = L[j]
    L[j] = aux
```

```
L = [5, 12, -1, 6, 8]

echange(L, 1, 3)
```

```
L
```

3 Création de listes

3.1 Par append successifs

Il arrive très souvent que l'on construise une liste à partir d'une liste vide (ou d'une liste de petite taille) en lui ajoutant des éléments au fur et à mesure du programme.

C'est par exemple le cas lorsque l'on souhaite créer la liste des premiers termes d'une suite récurrente.

Considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, u_{n+1} = 3u_n^2 - 1. \end{cases}$$

```
def liste_premiers_termes(n):
    """
    Entrée : un entier naturel n.
    Sortie : la liste des n + 1 premiers termes de la suite
             (u_n)_{n in N}, de u_0 à u_n.
    """
    terme = 1
    Liste = [1] # Initialement, la liste contient le premier terme.
    for k in range(1, n + 1): # k va de 1 à n,
    # on calculera n nouveaux termes : les termes u_1, ..., u_n.
        terme = 3 * terme**2 - 1
        Liste.append(terme)
        # On ajoute le nouveau terme calculé (u_k) à la liste.
    return Liste

liste_premiers_termes(10)
```

3.2 Définition en compréhension

Une manière très rapide de créer une liste (dont on sait à l'avance quels vont être les éléments) est de la définir de manière paramétrique, en décrivant ses éléments à l'aide d'une variable qui parcourt une boucle `for` (sur une structure itérable quelconque : un `range`, une chaine de caractères et même, plus tard, une liste !). En informatique, on parle alors de **définition en compréhension**.

Illustrons cette méthode par les exemples suivants.

```
[k ** 2 for k in range(10)]
```

```
[k ** 2 for k in range(37, 3, -9)]

[3 * c for c in "Bonjour"]

[3 * [0] for k in range(5)] # C'est une liste de listes !
```

4 Parcours d'une liste

De même que pour les chaines de caractères, il y a deux manières de parcourir tous les éléments d'une liste.

- La première consiste à écrire une boucle `for` dont la variable dont la variable parcourt tous les entiers de 0 à sa longueur moins 1, qui se trouvent être exactement les indices des éléments dans la liste :

```
for k in range(len(liste)) :
# L'entier k va de 0 à len(liste) - 1,
# c'est-à-dire que k parcourt le indices des éléments de la liste.
    bloc d'instructions
    dépendant de l'élément liste[k]
```

- La seconde consiste à écrire une boucle `for` dont la variable parcourt directement les éléments de liste un à un :

```
for x in liste :
# La variable x prend successivement pour valeur
# chaque élément de la liste liste.
    bloc d'instructions
    dépendant de l'élément x
```

⚠ On prendra toujours garde à ne pas confondre ces deux approches et soin de distinguer un élément de son indice dans une liste.

Les deux boucles suivantes affichent toutes les deux chaque élément de la liste $L = [1, 4, 5, 7, -2]$.

```
L = [1, 4, 5, 7, -2]

for k in range(5): # Pour k allant de 0 à 4,
                   # càd pour k parcourant les indices des éléments de L :
    print(L[k]) # on affiche l'élément d'indice k de la liste L.

for x in L: # Pour x parcourant les éléments de la liste L :
    print(x) # on affiche l'élément x.
```

5 Algorithmes importants de parcours de listes

5.1 Somme des éléments d'une liste

Pour sommer tous les éléments d'une liste de nombres, on commence par créer une variable qui contiendra la valeur de la somme de tous les éléments rencontrés, et qui est initialisée à 0 (pour la somme vide).

On parcourt ensuite un à un les éléments de la liste et on ajoute au fur et à mesure chaque élément rencontré à la somme.

Le parcours de la liste peut s'effectuer avec une boucle `for`, soit sur les éléments de la liste, soit sur leurs indices.

- Avec une boucle `for` qui parcourt directement les éléments de la liste :

```
def somme(L):
    """
    Entrée : une liste L de nombres.
    Sortie : la somme des éléments de la liste L.
    """
    S = 0 # On initialise la somme à 0.
    for x in L: # Pour x parcourant les éléments de la liste L :
        S = S + x # on ajoute l'élément x à la somme.
    return S
```

- Avec une boucle `for` qui parcourt les indices des éléments de la liste :

```
def somme_bis(L):
    """
    Entrée : une liste L de nombres.
    Sortie : la somme des éléments de la liste L.
    """
    S = 0 # On initialise la somme à 0.
    for k in range(len(L)): # Pour k allant de 0 à len(L) - 1,
                            # càd pour k parcourant les indices
                            # des éléments de la liste L :
        S = S + L[k]
        # on ajoute à la somme l'élément L[k] d'indice k.
    return S
```

5.2 Plus grand élément d'une liste

Un autre exemple important d'algorithme nécessitant un parcours de liste est la recherche du plus grand élément dans une liste de nombres.

5.2.1 Maximum

Pour trouver le plus grand élément d'une liste de nombres, on commence par créer une variable qui, au fur et à mesure du programme, contiendra la plus grande valeur rencontrée et qui prend initialement pour valeur le premier élément de la liste.

On parcourt ensuite la liste : dès que l'on rencontre un élément plus grand que la plus grande valeur rencontrée jusqu'alors, il en prend la place.

- Avec une boucle `for` qui parcourt directement les éléments de la liste :

```
def maximum(L):
    """
    Entrée : une liste L de nombres.
    Sortie : le plus grand élément de la liste L.
    """
    maxi = L[0]
    # Le maximum courant est initialisé au premier élément.
    for x in L: # x parcourt les éléments de la liste L.
        if x > maxi:
        # Si l'élément x est supérieur au maximum courant :
            maxi = x
            # l'élément x rencontré devient le maximum courant :
            # c'est la plus grande valeur qu'on ait rencontrée.
    return maxi
```

```
maximum([-5, -35, -2, -17, -2, -4])
```

- Avec une boucle `for` qui parcourt les indices des éléments de la liste :

```
def maximum_bis(L):
    """
    Entrée : une liste L de nombres.
    Sortie : le plus grand élément de la liste L.
    """
    maxi = L[0]
    for k in range(len(L)): # L'entier k va de 0 à len(L) - 1,
    # c'est-à-dire que k parcourt les indices de L.
        if L[k] > maxi:
            maxi = L[k]
        # Si l'élément d'indice k est supérieur à maxi,
        # alors l'élément d'indice k devient le maximum courant :
        # c'est la plus grande valeur qu'on ait rencontrée.
    return maxi
```

5.2.2 Indice de la première occurrence du maximum

Pour trouver la position de la première occurrence du maximum dans une liste, on peut procéder de plusieurs manières différentes.
Dans tous les cas, on devra parcourir la liste sur les indices, puisque l'on souhaite avoir accès à la position des éléments rencontrés.

- Une première manière de faire, qui n'est pas la plus efficace, consiste à trouver tout d'abord le maximum de la liste en appliquant la fonction `maximum` précédente, puis dans un second temps, à parcourir une seconde fois la liste, sur les indices cette fois-ci, jusqu'à trouver une occurrence du maximum (qui sera donc la première). On sortira de la fonction dès que l'on aura rencontré le maximum.

  Le parcours peut s'effectuer

  ⋆ soit à l'aide d'une boucle `for` :

```
def indice_maximum_for(L):
    """
    Entrée : une liste L de nombres.
    Sortie : l'indice de la première occurrence
             du maximum de la liste L.
    """
    maxi = maximum(L) # Le plus grand élément de la liste L.
    for k in range(len(L)):
    # Pour k allant de 0 à len(L) - 1,
    # c'est-à-dire pour k parcourant les indices des éléments de L :
        if L[k] == maxi:
            return k
        # si l'élément d'indice k de la liste L est le maximum,
        # alors on renvoie k (et on arrête la fonction).
```

  ⋆ soit à l'aide d'une boucle `while` :

```
def indice_maximum_while(L):
    """
    Entrée : une liste L de nombres.
    Sortie : l'indice de la première occurrence
             du maximum de la liste L.
    """
    maxi = maximum(L)
    i = 0 # Le premier indice considéré est 0.
    while L[i] != maxi:
    # Tant que l'élément d'indice i dans la liste L
    # n'est pas le maximum,
    # on passe à l'indice suivant.
        i += 1
    return i
```

- Une méthode plus efficace consiste à trouver du même coup le maximum et l'indice de sa première occurrence dans la liste.

Pour cela, on procède comme dans la fonction `maximum_bis` : on initialise le maximum courant au premier élément de la liste, puis dès que l'on rencontre un élément strictement plus grand que le maximum courant, on garde à la fois en mémoire sa valeur et sa position.

```
def indice_maximum(L):
    """
    Entrée : une liste L de nombres.
    Sortie : l'indice de la première occurrence
             du maximum de la liste L.
    """
    maxi = L[0]
    indice_maxi = 0 # Indice de la première occurrence
                    # du maximum courant.
    for k in range(len(L)):
    # Pour k allant de 0 à len(L) - 1,
    # càd pour k parcourant les indices des éléments de L.
        if L[k] > maxi:
            maxi = L[k]
            indice_maxi = k
        # Si l'élément d'indice k est strictement supérieur
        # au maximum courant,
        # alors il devient le maximum courant
        # et k devient l'indice du maximum courant.
    return indice_maxi
```

```
indice_maximum([-5, -35, -2, -17, -2, -4])
```

5.3 Recherche des deux valeurs les plus proches

Présentons maintenant un algorithme qui permet de trouver les deux valeurs les plus proches dans une liste de nombres et qui utilisera deux boucles imbriquées l'une dans l'autre.

Pour trouver les deux valeurs les plus proches dans une liste, on commence par créer une liste `Proches` qui contiendra les deux valeurs les plus proches et qui contient initialement la valeurs des deux premiers éléments de la liste. On crée également une variable `distance_min` qui contiendra la valeur de la plus petite distance entre deux éléments de la liste, initialisée à la distance entre les deux premiers éléments.

Pour décrire toutes les paires d'éléments de la liste, on utilise deux indices : l'indice `i` qui parcourt tous les indices de la liste et l'indice `j` qui parcourt les indices des éléments situés strictement après l'indice `i` dans la liste.
Pour chaque paire d'éléments de la liste, si la distance entre les deux éléments de la paire est strictement inférieure à `distance_min`, elle en prend la place et `Proches` devient la liste des deux éléments de la paire.

À la fin du parcours, on renvoie la liste `Proches`, qui contient les deux éléments de la liste dont les valeurs sont les plus proches.

```
import numpy as np # Pour la fonction valeur absolue.

def plus_proches(L):
    """
    Entrée : une liste L de nombres.
    Sortie : une liste contenant les deux éléments de L
             dont les valeurs sont les plus proches.
    """
    Proches = [L[0], L[1]]
    distance_min = np.abs(L[0] - L[1])
    # Au début, distance_min est la distance
    # entre les deux premiers éléments de la liste.
    for i in range(len(L)): # i va de 0 à len(L) - 1.
        for j in range(i + 1, len(L)): # j va de i + 1 à len(L) - 1.
            dist = np.abs(L[i] - L[j])
            if dist < distance_min :
                Proches = [L[i], L[j]]
                distance_min = dist
    return Proches
```

On peut aussi modifier ce programme pour qu'il renvoie plutôt les indices de deux éléments les plus proches de la liste.