Listes

Ce chapitre est très similaire au précédent : les chaines de caractères et les listes étant des types conteneurs, la plupart des syntaxes sont les mêmes pour ces deux types de données. Il y a cependant des différences liées principalement à une propriété des listes que n'ont pas les chaines de caractères : **la mutabilité**.

I. Définition et création de listes

1) Définition

Définition 1 : *Listes*

Une liste est une suite finie de données. Une liste peut contenir des données de n'importe quel type et ces données peuvent être de types différents.

La liste vide se note `[]`.

Le type liste se nomme `list` en Python.

Remarque :

- ⋆ L'intérêt principal d'une liste est de réunir en une seule structure (une seule variable) un grand nombre de données (sans avoir à créer une variable pour chaque donnée). Par exemple, lorsqu'on fait des statistiques, on dispose souvent d'un échantillon de taille importante dont on souhaite calculer la moyenne, l'écart-type, la médiane, etc. Utiliser une liste pour stocker cet échantillon est bien plus pratique que de créer une variable pour chacune des valeurs de l'échantillon.
- ⋆ Bien qu'il soit possible d'avoir des données de types différents dans une même liste, c'est en général une mauvaise pratique : la plupart du temps, on manipulera des listes de nombres, des listes de booléens, des listes de chaines de caractères et même des listes de listes.

2) Création de listes

Pour créer une liste, il existe deux types de syntaxe : la définition en extension et la définition en compréhension. Ces deux syntaxes correspondent exactement aux deux manières de noter un ensemble en mathématiques :

$$E = \{3, 6, 9, 12, 15\} = \{3k \ ; \ k \in \{1, 2, 3, 4, 5\}\}.$$

⋆ **Définition d'une liste en extension**

Pour définir une liste en extension, on écrit simplement les données qui la compose entre deux **crochets** en séparant les éléments par des **virgules** (**l'ordre compte**) :

```
liste = [x_0, x_1, ..., x_n]
```

où

- `liste` est le nom de la liste (c'est une variable de type `list`).
- `x_0`, `x_1`, ..., `x_n` sont des données de n'importe quel type.

Exemple :

```
>>> liste1 = [12, 18, 24]                          # Liste de nombres
>>> liste2 = [3, 4.5, 'bla']                       # Liste mixte
>>> liste3 = [[1, 2], [3, 4], [5, 6], [7, 8]]     # Liste de listes
>>> voyelles = ['a', 'e', 'i', 'o', 'u', 'y']     # Liste de caractères
```

⋆ **Définition d'une liste en compréhension**

Pour définir une liste en compréhension, on utilise une syntaxe similaire aux boucles pour créer les éléments de la liste à l'aide d'une « formule » à un paramètre. On utilise toujours les **crochets** pour délimiteurs :

```
liste = [f(k) for k in range(a, b)]
```

ou

```
liste = [f(k) for k in autre_liste]
```

où

- `k` est une variable muette : on peut la nommer autrement et sa portée est limitée aux crochets (elle n'existe plus en dehors).
- `a` et `b` sont des entiers délimitant les valeurs possibles de `k` (`k` varie de `a` **inclu** à `b` **exclu**).
- `autre_liste` est une liste précédemment définie.

Exemple :

```
>>> liste1 = [k for k in range(10)]
>>> liste1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> liste2 = [3*k for k in range(1, 6)]
>>> liste2
[3, 6, 9, 12, 15]
>>> liste3 = [e**2 - e for e in liste2]
>>> liste3
[6, 30, 72, 132, 210]
>>> diziemes = [k / 10 for k in range(10)]
>>> diziemes
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
>>> carres = [i**2 for i in range(10)]
>>> carres
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Remarque : Il existe des variantes de syntaxes permettant d'ajouter des conditions supplémentaires sur le paramètre dans la définition d'une liste en compréhension. Ces variantes ne sont pas indispensables mais en voici un exemple :

```
>>> liste4 = [e**3 for e in range(100) if e in carres and e not in liste2]
>>> liste4
[0, 1, 64, 4096, 15625, 46656, 117649, 262144, 531441]
```

II. Opérations et tests

1) Opérations sur les listes

Il existe deux opérations sur les listes :

- la concaténation : `+`

```
>>> liste1 + liste2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 6, 9, 12, 15]
>>> liste2 + voyelles + liste3
[3, 6, 9, 12, 15, 'a', 'e', 'i', 'o', 'u', 'y', 6, 30, 72, 132, 210]
```

- la multiplication par un entier : `*`

```
>>> 3 * [3, 2, 1]
[3, 2, 1, 3, 2, 1, 3, 2, 1]
>>> 10 * [0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> 0 * [3, 2, 1]
[]
>>> -3 * [3, 2, 1]
[]
```

Ainsi l'opération `3 * liste` est équivalente à l'opération `liste + liste + liste`. On remarque également que multiplier par un entier négatif ou nul donne la liste vide.

Remarque : Pour créer une liste contenant plusieurs fois la même valeur, la syntaxe

```
liste = nombre * [valeur]
```

est très pratique.
Attention cependant à ne pas écrire `liste = [nombre * valeur]` qui n'a pas le même effet !

2) Tests et comparaisons

Il est également possible de comparer deux listes avec les opérateurs de comparaison `==`, `<`, `<=`, `>`, `>=` : dans le cas des inégalités, il faut que les objets soient comparables et c'est l'ordre lexicographique qui est utilisé.

Exemple :

```
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] == [3, 2, 1]
False
>>> [1, 2, 3] < [1, 2, 2]
False
>>> [1, 2, 3] < [1, "a", 2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Enfin, il est possible de tester l'appartenance d'un élément à une liste avec l'opérateur `in` :

Exemple :

```
>>> liste = [2, 3, 5, 7, 11]
>>> 5 in liste
True
>>> 6 in liste
False
>>> 6 not in liste
True
```

III. Fonctions et méthodes

1) Fonctions sur les listes

Fonction	Résultat	Type de retour
`len(liste)`	Longueur (nombre d'éléments) de la liste	`int`
`sum(liste)`	Somme des éléments de la liste numérique	`int` ou `float`
`max(liste)`	Maximum des éléments de la liste	mixte
`min(liste)`	Minimum des éléments de la liste	mixte

Exemple :

```
>>> liste = [-3, 6, 2, 9, 1]
>>> len(liste)
5
>>> sum(liste)
15
>>> max(liste)
9
>>> min(liste)
-3
>>> mots = ["Bonjour", "Salut", "Au revoir"]
>>> max(mots)
'Salut'
>>> min(mots)
'Au revoir'
```

2) Méthodes sur les listes

Contrairement aux chaines de caractères, les méthodes suivantes sont à connaitre par cœur : nous allons les utiliser tout le temps !

Méthode	Résultat	Type de retour	Effet de bord
`liste.append(e)`	Ajoute l'élément `e` à la fin de `liste`	`None`	Oui
`liste.insert(i, e)`	Ajoute l'élément `e` à l'indice `i` dans `liste`	`None`	Oui
`liste.pop(i=-1)`	Supprime et renvoie l'élément d'indice `i` (qui vaut `-1` par défaut) dans `liste`	mixte	Oui
`liste.remove(e)`	Supprime le premier élément de `liste` ayant pour valeur `e`	`None`	Oui
`liste.index(e)`	Renvoie l'indice du premier élément de `liste` dont la valeur est `e`	`int`	Non
`liste.count(e)`	Renvoie le nombre d'occurrence de `e` dans `liste`	`int`	Non
`liste.reverse()`	Inverse l'ordre des éléments de `liste`	`None`	Oui
`liste.sort(reverse=False)`	Trie `liste` dans l'ordre croissant si `reverse = False` (valeur par défaut) et décroissant si `reverse = True`	`None`	Oui

Remarque :

- Les méthodes à effet de bord sont des méthodes qui **modifient la liste** sur laquelle elles s'appliquent. Pour la plupart, c'est d'ailleurs leur seul rôle puisqu'elles ne renvoient pas de valeur (à l'exception de `pop`). Ce genre de méthodes n'existe que pour les types de données mutables comme les listes et les dictionnaires mais pas pour les types immuables comme les chaines de caractères.
- Les méthodes dont le type de retour est `None` ne renvoient rien. Il est donc très fortement déconseillé d'attribuer le résultat de la méthode à une variable. Ainsi pour ajouter un élément à une liste :

```
L = [1, 3, 5]
L.append(7)          # Ajout de 7 à la fin
L = L.append(9)      # Ne produit pas d'erreur mais efface la liste !
```

- L'argument de la méthode `pop` est optionnel : si on ne le met pas, il vaut par défaut `-1` c'est-à-dire que le dernier élément de la liste et renvoyé et supprimé de la liste. De même, l'argument de la méthode `sort` est optionnel et vaut `False` :

```
L.sort()                # Trie L dans l'ordre croissant
L.sort(reverse=True)    # Trie L dans l'ordre décroissant
```

Exemple :

```
>>> liste = [k**2-k for k in range(-5, 5)]
>>> liste
[30, 20, 12, 6, 2, 0, 0, 2, 6, 12]
>>> liste.append(6)        # liste = [30, 20, 12, 6, 2, 0, 0, 2, 6, 12, 6]
>>> liste.insert(3, 0)     # liste = [30, 20, 12, 0, 6, 2, 0, 0, 2, 6, 12, 6]
>>> liste.pop(1)           # liste = [30, 12, 0, 6, 2, 0, 0, 2, 6, 12, 6]
20
>>> x = liste.pop()        # liste = [30, 12, 0, 6, 2, 0, 0, 2, 6, 12]
>>> x
6
>>> liste.remove(2)        # liste = [30, 12, 0, 6, 0, 0, 2, 6, 12]
```

```
>>> liste.reverse()                  # liste = [12, 6, 2, 0, 0, 6, 0, 12, 30]
>>> liste.sort()                     # liste = [0, 0, 0, 2, 6, 6, 12, 12, 30]
>>> liste.sort(reverse=True)         # liste = [30, 12, 12, 6, 6, 2, 0, 0, 0]
>>> liste.index(0)
6
>>> liste.count(0)
3
```

IV. Accès aux éléments d'une liste

1) Indices et éléments

Définition 2 : *Indices et éléments*

- ⋆ Dans une liste, les éléments sont numérotés par leurs indices. Les indices sont attribués de la façon suivante : le premier élément de la liste a **toujours** pour indice 0, le deuxième a pour indice 1 et ainsi de suite jusqu'à la fin de la liste.
- ⋆ Les éléments d'une liste possèdent également un indice négatif attribué de la façon suivante : le dernier élément de la liste a pour indice négatif -1, l'avant-dernier a pour indice négatif -2 et ainsi de suite jusqu'au début de la liste.
- ⋆ Pour accéder à un élément particulier d'une liste, on utilise la syntaxe suivante :

  ```
  liste[indice]
  ```

  où `liste` est le nom de la variable de type `list` et `indice` est l'indice (positif ou négatif) de l'élément auquel on veut accéder.

Par exemple, en considérant la liste `premiers = [2, 3, 5, 7, 11, 13, 17, 19]` qui a 8 éléments :

Éléments	2	3	5	7	11	13	17	19
Indice	0	1	2	3	4	5	6	7
Indice négatif	-8	-7	-6	-5	-4	-3	-2	-1

Exemple :

```
>>> premiers = [2, 3, 5, 7, 11, 13, 17, 19]
>>> premiers[0]
2
>>> premiers[1]
3
>>> premiers[5]
13
>>> premiers[-1]
19
>>> premiers[-5]
7
>>> premiers[8]      # Indice trop grand ou trop petit = IndexError
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
```

Remarque :

- ⋆ Dans une liste `L`, les indices des éléments peuvent varier de `-len(L)` à `len(L)-1`.
- ⋆ Attention à ne pas confondre l'indice et l'élément : c'est une erreur très fréquente dans le cas d'une liste de nombres.

Modifier un élément : mutabilité

C'est la principale différence entre les listes et les chaînes de caractères (en plus du fait que les chaînes de caractères ne peuvent contenir que des caractères) : les listes sont dites mutables.
Une conséquence de la mutabilité des listes (il y en a d'autres dont les effets de bords vus plus haut) est qu'il est possible de modifier un élément en particulier d'une variable de type `list` de la façon suivante :

```
liste[indice] = valeur
```

Par exemple :

```
>>> L = [3, 6, 19, 5]
>>> L[2] = 0
>>> L
[3, 6, 0, 5]
```

2) Tranches et sous-listes

Définition 3 : *Tranches (slicing)*

Pour extraire une sous-liste d'une liste, on peut utiliser l'une des syntaxes suivantes :

```
liste[i:j]          # Liste composée des éléments d'indices i à j-1
liste[:j]           # Liste composée des éléments d'indices 0 (premier) à j-1
liste[i:]           # Liste composée des éléments d'indices i à -1 (dernier)
liste[:]            # Copie complète de la liste
```

On parle alors de tranches (slicing en anglais). Les indices positifs ou négatifs peuvent être utilisés.

Exemple :

```
>>> premiers = [2, 3, 5, 7, 11, 13, 17, 19]
>>> premiers[2:5]
[5, 7, 11]
>>> premiers[2:-1]
[5, 7, 11, 13, 17]
>>> premiers[-5:6]
[7, 11, 13]
>>> premiers[5:3]          # Liste vide si le premier élément est après le
    dernier
[]
>>> premiers[:4]           # Les 4 premiers éléments
[2, 3, 5, 7]
>>> premiers[-3:]          # Les 3 derniers éléments
[13, 17, 19]
>>> premiers[:]            # Copie de la liste
[2, 3, 5, 7, 11, 13, 17, 19]
```

Remarque :

- ⋆ Dans la syntaxe `liste[i:j]`, le caractère d'indice `j` **est exclu**.
- ⋆ Il est très pratique de retenir les longueurs des listes obtenues par ces syntaxes.
  Si $0 \leqslant$ `i` $\leqslant$ `j` $\leqslant$ `len(liste)`, alors
  - `len(liste[i:j]) = j - i`
  - `len(liste[:j]) = j`
  - `len(liste[i:]) = len(liste) - i`
  - `len(liste[:]) = len(liste)`

3) Parcours d'une liste

Pour parcourir une liste, c'est-à-dire examiner les éléménts de la liste un par un à l'aide d'une boucle, on dispose de deux moyens :

- ⋆ **Parcourir les indices de la liste :**

```
for i in range(len(liste)):
    print(f"L'élément d'indice {i} de la liste est {liste[i]}.")
```

- ⋆ **Parcourir les éléments de la liste :**

```
for e in liste:
    print(f"{e} est un élément de la liste mais on n'a pas accès à son indice.")
```

Pour certains algorithmes, un des deux parcours est plus adapté que l'autre mais dans la plupart des cas, on peut choisir entre les deux parcours. On peut également retenir que le parcours des indices fonctionne toujours ce qui n'est pas le cas du parcours des caractères.

Exemple :

- ⋆ **Code :**

```
premiers = [2, 3, 5, 7, 11, 13, 17, 19]
for i in range(len(premiers)):
    print(premiers[i])
```

Résultat :

```
2
3
5
7
11
13
17
19
```

- ⋆ **Code :**

```
premiers = [2, 3, 5, 7, 11, 13, 17, 19]
for i in range(len(premiers)):
    print(i)
```

Résultat :

```
0
1
2
3
4
5
6
7
```

- ⋆ **Code :**

```
premiers = [2, 3, 5, 7, 11, 13, 17, 19]
for e in premiers:
    print(e)
```

Résultat :

```
2
3
5
7
11
13
17
19
```

V. Algorithmes sur les listes

Dans cette partie, nous allons étudier un certain nombre de problèmes classiques sur les listes. Pour la plupart de ces problèmes, une fonction ou une méthode existe déjà pour le résoudre mais il est très important d'apprendre à le faire nous-mêmes pour :

- ⋆ s'entrainer à travailler sur les listes sur des problèmes simples ;
- ⋆ pouvoir adapter un algorithme connu à un problème spécifique ;
- ⋆ estimer la complexité temporelle d'un algorithme pour la comparer à un autre.

C'est pourquoi nous nous interdirons d'utiliser la fonction ou la méthode qui fait déjà tout le travail à notre place.

1) Sommer les éléments d'une liste numérique

Pour calculer une somme, on se souvient qu'il faut toujours initialiser une variable à 0. Ensuite il suffit de parcourir la liste et d'additionner les résultats dans la variable initialisée (qui joue un rôle d'accumulateur).

- ⋆ **Avec un parcours des indices** :

```
def somme(liste):
    """
    Entrée : une liste numérique
    Sortie : somme des éléments de liste
    """
    S = 0
    for i in range(len(liste)):
        S += liste[i]
    return S
```

- ⋆ **Avec un parcours des éléments** :

```
def somme(liste):
    """
    Entrée : une liste numérique
    Sortie : somme des éléments de liste
    """
    S = 0
    for e in liste:
        S += e
    return S
```

2) Rechercher un élément dans une liste

Plusieurs variantes du même problème peuvent être résolues :

- ⋆ rechercher la présence d'un élément
- ⋆ rechercher la position d'un élément (s'il est présent)
- ⋆ compter le nombre d'occurrence d'un élément

```
def est_present(element, liste):
    """
    Entrées : element (objet quelconque) et liste
    Sortie : True si element est dans liste, False sinon
    """
    for e in liste:
        if e == element:
```

```
            return True
    return False

def indice(element, liste):
    """
    Entrées : element (objet quelconque) et liste
    Sortie : indice de element s'il est présent dans liste et None sinon
    """
    for i in range(len(liste)):
        if liste[i] == element:
            return i
    return None

def occurrences(element, liste):
    """
    Entrées : element (objet quelconque) et liste
    Sortie : nombre d'occurrence de element dans liste
    """
    cpt = 0
    for e in liste:
        if e == element:
            cpt += 1
    return cpt
```

Remarque :

- ⋆ On remarque que les trois fonctions se ressemblent beaucoup.
- ⋆ La fonction `indice` est la seule qui nécessite un parcours des indices. Les deux autres auraient pu aussi utiliser le même type de parcours.
- ⋆ La dernière ligne `return None` de la fonction `indice` est optionnelle : si on l'enlève, ça ne changera rien car c'est la valeur par défaut lorsqu'il n'y a pas de `return`.

3) Trouver le maximum ou le minimum d'une liste numérique

Pour trouver le maximum ou le minimum d'une liste numérique, on utilise le principe du « record à battre » :

- ⋆ On commence par initialiser une variable égale au premier élément de la liste qui sera la valeur record initiale.
- ⋆ Ensuite on parcourt la liste en actualisant le record dès qu'il est battu (valeur plus grande pour un maximum ou plus petite pour un minimum).
- ⋆ À la fin du parcours, le record est égal au maximum ou au minimum de la liste.

```python
def maximum(liste):
    """
    Entrée : une liste numérique
    Sortie : maximum de liste
    """
    record = liste[0]
    for e in liste:
        if e > record:      # le record est battu
            record = e      # on enregistre le nouveau record
    return record

def minimum(liste):
    """
    Entrée : une liste numérique
    Sortie : minimum de liste
    """
    record = liste[0]
    for e in liste:
        if e < record:      # le record est battu
            record = e      # on enregistre le nouveau record
    return record
```

Remarque : D'autres variantes du même algorithme peuvent être résolues en adaptant les programmes précédents :

- ⋆ Trouver l'indice de la première occurrence du maximum/minimum d'une liste.
- ⋆ Trouver le deuxième maximum/minimum d'une liste.