Listes en Python

Structures de données

Résumé

En informatique, on a souvent besoin de regrouper et de manipuler des collections de données : une liste de notes, de prénoms, de températures... Python propose pour cela le type **list**, une structure ordonnée et modifiable. Ce cours présente la création, la lecture et la modification des listes, leurs deux parcours, leur construction pas à pas ou en compréhension, ainsi que les pièges de la copie.

① Pourquoi des listes ?

Pour garder les cinq notes d'un élève, on pourrait créer cinq variables :

```
note1 = 15
note2 = 12
note3 = 18
note4 = 14
note5 = 16
```

Avec trente notes, ou les notes de toute une classe, cela devient ingérable : il faudrait autant de variables, et un simple calcul de moyenne devrait les nommer une par une. Une **liste** regroupe toutes ces valeurs dans **une seule variable** :

```
notes = [15, 12, 18, 14, 16]
print(notes)
```

```
[15, 12, 18, 14, 16]
```

② Définition

Une **liste** est une structure de données qui regroupe plusieurs valeurs dans une seule variable, les unes à la suite des autres. Une liste de courses, une liste de notes ou un classement sportif suivent le même principe.

Une liste en Python possède les propriétés suivantes :

— elle peut contenir un nombre quelconque d'éléments (y compris zéro) ;

— les éléments sont **ordonnés** : chacun a une position, son **indice** ;

— les éléments peuvent être de types différents (entiers, chaînes de caractères, etc.) ;

— elle est **modifiable** : on peut ajouter, supprimer ou remplacer des éléments.

③ Créer une liste

On crée une liste en plaçant les éléments entre **crochets** `[]`, séparés par des virgules :

```
# Une liste de prénoms
prenoms = ["Alice", "Bob", "Charlie", "Diana"]

# Une liste de notes
notes = [15, 12, 18, 14, 16]

# Une liste vide
panier = []

print(prenoms)
print(notes)
print(panier)
```

```
['Alice', 'Bob', 'Charlie', 'Diana']
[15, 12, 18, 14, 16]
[]
```

On peut aussi créer une liste à partir d'une chaîne de caractères avec la fonction `list()` : chaque caractère devient un élément.

```
lettres = list("Python")
print(lettres)
```

```
['P', 'y', 't', 'h', 'o', 'n']
```

Exercice 1 *Créer des listes*

1. Créez une liste `couleurs` contenant les chaînes `"rouge"`, `"vert"`, `"bleu"` et `"jaune"`.

2. Créez une liste `temperatures` contenant les valeurs 18.5, 22.1, 15.8, 19.3 et 21.0.

3. Quel est le type renvoyé par `type(notes)` ?

④ Lire un élément avec son indice

Comme pour une chaîne de caractères, chaque élément est repéré par son **indice**, c'est-à-dire sa position dans la liste. **Le premier élément a l'indice 0**, le deuxième l'indice 1, et ainsi de suite. Les **indices négatifs** comptent à partir de la fin : `-1` pour le dernier élément, `-2` pour l'avant-dernier.

	15	12	18	14	16
indice	0	1	2	3	4
indice négatif	−5	−4	−3	−2	−1

On lit un élément en écrivant son indice entre crochets, après le nom de la liste :

```
notes = [15, 12, 18, 14, 16]

print(notes[0])    # premier élément
print(notes[2])    # troisième élément
print(notes[-1])   # dernier élément
```

```
15
18
16
```

Question.
Que vaut `notes[-2]` ?

Réponse.
`14` : l'indice `-2` désigne l'avant-dernier élément de la liste.

5 Attention au dernier indice

La fonction `len()` renvoie le nombre d'éléments d'une liste, c'est-à-dire sa **longueur** :

```
notes = [15, 12, 18, 14, 16]
print(len(notes))
```

```
5
```

La liste contient cinq éléments : ses indices vont donc de `0` à `4`. Demander l'indice `5`, c'est chercher une case qui n'existe pas, et Python s'arrête avec une erreur :

```
notes = [15, 12, 18, 14, 16]
print(notes[5])
```

```
IndexError: list index out of range
```

À retenir. Les indices valides d'une liste `L` vont de `0` à `len(L) - 1`.

Exercice 2 *Accès par indice* Soit la liste :

```
villes = ["Paris", "Lyon", "Marseille", "Toulouse", "Nantes"]
```

Sans exécuter le code, donnez la valeur de chaque expression :

1. `villes[0]`
2. `villes[3]`
3. `villes[-1]`
4. `villes[-2]`
5. `len(villes)`
6. `villes[5]` — Que se passe-t-il ?

6 Modifier un élément

Contrairement à une chaîne de caractères, une liste est **modifiable** (on dit aussi **mutable**) : on affecte une nouvelle valeur à un indice, avec le signe `=`. Les autres éléments ne bougent pas.

```
notes = [15, 12, 18, 14, 16]
print(notes)

notes[1] = 14  # on corrige la deuxième note
print(notes)
```

```
[15, 12, 18, 14, 16]
[15, 14, 18, 14, 16]
```

Question.
Que contient la liste après ces deux affectations ?

```
notes = [15, 12, 18, 14, 16]
notes[0] = 10
notes[-1] = notes[0] + 1
print(notes)
```

Réponse.

```
[10, 12, 18, 14, 11]
```

La première affectation remplace `15` par `10`. La seconde est calculée avec la nouvelle valeur de `notes[0]`, donc `10 + 1`, et remplace le dernier élément, `16`.

Exercice 3 *Mutation* Soit la liste `animaux = ["chat", "chien", "poisson", "hamster"]`.

1. Remplacez `"poisson"` par `"perroquet"` sans recréer la liste.

2. Que vaut `animaux` après cette modification ?

3. Peut-on écrire `"bonjour"[0] = "B"` ? Pourquoi ?

7 Ajouter et supprimer des éléments

Les **méthodes** des listes s'écrivent après le nom de la liste et un point. Elles modifient la liste elle-même.

Ajouter des éléments

La méthode `append()` ajoute un élément à la fin de la liste ; la méthode `insert()` l'ajoute à l'indice choisi, et les éléments suivants se décalent.

```
couleurs = ["rouge", "vert"]

couleurs.append("bleu")       # ajoute à la fin
print(couleurs)

couleurs.insert(1, "jaune")   # ajoute à l'indice 1
print(couleurs)
```

```
['rouge', 'vert', 'bleu']
['rouge', 'jaune', 'vert', 'bleu']
```

Supprimer des éléments

La méthode `remove()` retire un élément d'après sa **valeur** ; la méthode `pop()` retire le dernier élément et le renvoie.

```python
couleurs = ["rouge", "jaune", "vert", "bleu"]

couleurs.remove("jaune")  # supprime par valeur
print(couleurs)

dernier = couleurs.pop()   # retire et renvoie le dernier
print(dernier)
print(couleurs)
```

```
['rouge', 'vert', 'bleu']
bleu
['rouge', 'vert']
```

Avec un indice en argument, `pop(i)` retire et renvoie l'élément d'indice `i` : `couleurs.pop(0)` retire donc le premier élément de la liste.

Autres méthodes utiles

Méthode	Description	Exemple
`L.append(x)`	Ajoute `x` à la fin	`notes.append(17)`
`L.insert(i, x)`	Insère `x` à l'indice `i`	`notes.insert(0, 20)`
`L.pop()`	Retire et renvoie le dernier élément	`notes.pop()`
`L.pop(i)`	Retire et renvoie l'élément d'indice `i`	`notes.pop(0)`
`L.remove(x)`	Retire la première occurrence de `x`	`notes.remove(12)`
`L.sort()`	Trie la liste en place	`notes.sort()`
`L.reverse()`	Inverse l'ordre des éléments	`notes.reverse()`
`L.index(x)`	Renvoie l'indice de la première occurrence de `x`	`notes.index(18)`
`L.count(x)`	Compte le nombre d'occurrences de `x`	`notes.count(14)`

Important. `append()`, `sort()`, `reverse()` et `remove()` modifient la liste **en place** et ne renvoient **rien** (`None`). Il ne faut donc pas écrire `L = L.sort()`.

Exercice 4 *Méthodes de listes* Soit `L = [3, 1, 4, 1, 5, 9, 2, 6]`.

Donnez la valeur de `L` après chaque instruction (les instructions sont exécutées à la suite) :

1. `L.append(5)`
2. `L.sort()`
3. `L.pop()`
4. `L.remove(1)`

5. `L.insert(0, 7)`

6. Après toutes ces opérations, quelle est la valeur de `len(L)` ?

8 Parcourir une liste par ses valeurs

Pour traiter tous les éléments d'une liste, on la parcourt avec une boucle `for`. Il existe deux façons de le faire. La première, la plus simple, parcourt directement les **valeurs** : à chaque tour de boucle, la variable `note` reçoit l'élément suivant de la liste.

```
notes = [15, 12, 18]

for note in notes:
    print(note)
```

```
15
12
18
```

Pour comprendre ce qui se passe, on suit la valeur des variables tour après tour : c'est la **trace d'exécution** du programme.

Tour	Valeur de note	Affichage
1	15	15
2	12	12
3	18	18

Après le troisième tour, la liste est terminée et la boucle s'arrête. On utilise ce parcours quand on a seulement besoin des valeurs, par exemple pour les afficher ou les additionner :

```
fruits = ["pomme", "banane", "cerise", "mangue"]

for fruit in fruits:
    print(f"J'aime les {fruit}s")
```

```
J'aime les pommes
J'aime les bananes
J'aime les cerises
J'aime les mangues
```

⑨ Parcourir une liste par ses indices

La seconde façon parcourt les **positions** des éléments. La fonction `len()` donne la longueur de la liste, et la fonction `range()` fournit alors les indices `0`, `1`, `2`... jusqu'à `len(notes) - 1`. L'élément s'obtient ensuite avec `notes[i]`.

```
notes = [15, 12, 18]

for i in range(len(notes)):
    print(f"Note {i} : {notes[i]}")
```

```
Note 0 : 15
Note 1 : 12
Note 2 : 18
```

Tour	Valeur de i	notes[i]	Affichage
1	0	15	`Note 0 : 15`
2	1	12	`Note 1 : 12`
3	2	18	`Note 2 : 18`

On utilise ce parcours quand on a besoin de la **position** de chaque élément, par exemple pour l'afficher ou pour **modifier** la liste :

```
notes = [15, 12, 18]

for i in range(len(notes)):
    notes[i] = notes[i] + 1   # un point de plus pour chaque note
print(notes)
```

```
[16, 13, 19]
```

Question.
Pourquoi ne peut-on pas obtenir ce résultat avec le parcours par les valeurs, en écrivant `note = note + 1` dans la boucle `for note in notes` ?

Réponse.
Dans le parcours par les valeurs, la variable `note` reçoit la valeur de chaque élément : `note = note + 1` change la variable `note`, pas la liste. Pour modifier la liste, il faut écrire `notes[i] = ...`, donc connaître l'indice `i`.

Parcours avec `enumerate()`

La fonction `enumerate()` fournit à la fois l'indice et l'élément :

```
fruits = ["pomme", "banane", "cerise", "mangue"]

for i, fruit in enumerate(fruits):
```

```
    print(f"Fruit n°{i} : {fruit}")
```

```
Fruit n°0 : pomme
Fruit n°1 : banane
Fruit n°2 : cerise
Fruit n°3 : mangue
```

⚠ **Attention.** on ne modifie pas la longueur d'une liste pendant qu'on la parcourt (en lui ajoutant ou en lui retirant des éléments) : cela peut produire des résultats imprévisibles. Il vaut mieux construire une nouvelle liste, comme dans la section suivante.

Exercice 5 *Parcours et accumulation* Soit la liste `notes = [15, 8, 12, 18, 6, 14, 11]`.

1. Écrivez un programme qui calcule la **somme** de toutes les notes en utilisant un parcours par élément.

2. Modifiez votre programme pour calculer la **moyenne**.

3. Écrivez un programme qui **compte** le nombre de notes supérieures ou égales à 10.

4. Écrivez un programme qui affiche chaque note avec son rang : `"Note n°0 : 15"`, etc. Utilisez `enumerate()`.

Exercice 6 *Recherche dans une liste*

1. Écrivez une fonction `chercher(liste, x)` qui renvoie `True` si `x` est dans `liste`, `False` sinon, en utilisant un parcours par élément.

2. Écrivez une fonction `indice_de(liste, x)` qui renvoie l'**indice** de la première occurrence de `x` dans `liste`, ou `None` si `x` est absent.

3. Quel opérateur Python permet de faire la même chose que votre fonction `chercher` en une seule expression ?

Exercice 7 *Maximum d'une liste*

1. Complétez la fonction suivante qui renvoie le maximum d'une liste de nombres, **sans utiliser** la fonction `max()` de Python :

```
def maximum(liste):
    """Renvoie le plus grand élément de liste."""
    maxi = liste[0]
    for element in liste:
        if ...:       # compléter
            maxi = ...  # compléter
    return maxi
```

2. Testez votre fonction avec `maximum([3, 7, 2, 9, 4])` et `maximum([-5, -1, -8])`.

3. Écrivez une fonction `indice_maximum(liste)` qui renvoie l'**indice** du plus grand élément.

10 Construire une liste pas à pas

Très souvent, on ne connaît pas les éléments d'une liste à l'avance. On part alors d'une **liste vide**, écrite `[]`, et on la remplit au fur et à mesure, avec la méthode `append()` dans une boucle.

```
carres = []
for x in range(5):
    carres.append(x ** 2)
```

```
print(carres)
```

Question.
Que contient la liste `carres` à la fin de la boucle ?

Réponse.

```
[0, 1, 4, 9, 16]
```

La boucle fait cinq tours, avec `x` de `0` à `4`, et chaque tour ajoute le carré de `x` à la fin de la liste :

Tour	Valeur de x	Élément ajouté	carres après le tour
1	0	0	[0]
2	1	1	[0, 1]
3	2	4	[0, 1, 4]
4	3	9	[0, 1, 4, 9]
5	4	16	[0, 1, 4, 9, 16]

La même méthode permet de **filtrer** une liste : on n'ajoute un élément que s'il vérifie une condition.

```
notes = [15, 8, 12, 6, 18]

admis = []
for note in notes:
    if note >= 10:
        admis.append(note)
print(admis)
```

```
[15, 12, 18]
```

À retenir. Liste vide, boucle, puis `append()` à chaque tour : c'est la façon la plus courante de construire une liste.

Exercice 8 *Comptage et filtrage* Soit `prenoms = ["Zoé", "Léa", "Léo", "Mia", "Tom"]` et `ages = [17, 19, 12, 15, 21]` : la personne d'indice `i` s'appelle `prenoms[i]` et a `ages[i]` ans.

1. Écrivez une fonction `moyenne_age(ages)` qui renvoie la moyenne d'âge arrondie à l'entier inférieur.

2. Écrivez une fonction `majeurs(prenoms, ages)` qui renvoie la liste des prénoms des personnes ayant 18 ans ou plus.

3. Vérifiez : `moyenne_age(ages)` doit renvoyer `16` et `majeurs(prenoms, ages)` doit renvoyer `["Léa", "Tom"]`.

11 Construire une liste en compréhension

Python propose une écriture plus courte pour construire une liste : la **liste en compréhension**. Entre crochets, on écrit d'abord l'élément à ajouter, puis la boucle qui fournit les valeurs : `[expression for variable in itérable]`. On obtient exactement la même liste qu'avec la boucle et `append()`, mais en une seule ligne.

```
carres = [x ** 2 for x in range(5)]
print(carres)
```

```
[0, 1, 4, 9, 16]
```

On peut ajouter une condition à la fin, avec `if` : seuls les éléments qui la vérifient entrent dans la liste. Le filtrage de la section précédente s'écrit ainsi :

```
notes = [15, 8, 12, 6, 18]
admis = [note for note in notes if note >= 10]
print(admis)
```

```
[15, 12, 18]
```

Question.
Que contient la liste `pairs` ?

```
pairs = [x for x in range(10) if x % 2 == 0]
print(pairs)
```

Réponse.

```
[0, 2, 4, 6, 8]
```

La condition garde les nombres dont le reste de la division par 2 vaut 0, c'est-à-dire les nombres pairs de 0 à 9.

Autre exemple : extraire les initiales d'une liste de prénoms.

```python
prenoms = ["Alice", "Bob", "Charlie", "Diana"]
initiales = [p[0] for p in prenoms]
print(initiales)
```

```
['A', 'B', 'C', 'D']
```

Exercice 9 *Compréhensions de listes* Écrivez en compréhension de liste :

1. La liste des cubes de 0 à 10.

2. La liste des multiples de 3 entre 0 et 30 (inclus).

3. À partir de `mots = ["chat", "anticonstitutionnellement", "le", "python", "a"]`, la liste des mots de plus de trois lettres.

4. À partir de `notes = [15, 8, 12, 18, 6, 14]`, la liste des notes supérieures ou égales à 10.

12 Opérations sur les listes

```python
# Concaténation avec +
a = [1, 2, 3]
b = [4, 5]
print(a + b)

# Répétition avec *
print([0] * 5)

# Test d'appartenance avec in
print(3 in a)
print(7 in a)
```

```
[1, 2, 3, 4, 5]
[0, 0, 0, 0, 0]
True
False
```

13 Fonctions utiles sur les listes

```python
notes = [15, 12, 18, 14, 16]

print(len(notes))    # nombre d'éléments
print(min(notes))    # minimum
print(max(notes))    # maximum
print(sum(notes))    # somme
print(sum(notes) / len(notes))  # moyenne
```

```
5
12
18
75
15.0
```

Exercice 10 *Opérations sur les listes*

1. Que vaut `[1, 2] + [3, 4]` ?

2. Que vaut `[0] * 5` ?

3. Soit `notes = [15, 8, 12, 18, 6]`. Écrivez une expression Python qui calcule la moyenne en utilisant `sum()` et `len()`.

14 Des chaînes aux listes : `split()` et `join()`

Deux méthodes des chaînes de caractères font le lien avec les listes : `split()` découpe une chaîne en une liste de morceaux, et `join()` assemble une liste de chaînes en une seule chaîne. Ce sont les deux méthodes les plus utilisées pour traiter des données textuelles.

`split()` : découper une chaîne en liste

```python
date = "28-03-2026"
morceaux = date.split("-")
print(morceaux)

# Accéder aux éléments
jour = morceaux[0]
mois = morceaux[1]
annee = morceaux[2]
print(f"Jour : {jour}, Mois : {mois}, Année : {annee}")
```

```
['28', '03', '2026']
Jour : 28, Mois : 03, Année : 2026
```

Sans argument, `split()` découpe sur les espaces (un ou plusieurs) :

```
phrase = "  Python   est    génial  "
mots = phrase.split()
print(mots)
```

```
['Python', 'est', 'génial']
```

`join()` : assembler une liste en chaîne

La méthode `join()` s'appelle sur le **séparateur** et reçoit la liste en argument :

```
mots = ["Python", "est", "génial"]
phrase = " ".join(mots)
print(phrase)

csv_ligne = ";".join(["Alice", "17", "1NSI"])
print(csv_ligne)
```

```
Python est génial
Alice;17;1NSI
```

Application. Inverser les mots d'une phrase :

```
phrase = "les chaînes sont utiles"
mots = phrase.split()
mots.reverse()
resultat = " ".join(mots)
print(resultat)
```

```
utiles sont chaînes les
```

15 Copier une liste

Voici un piège très fréquent.

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a)
```

Question.
Que contient la liste `a` après ces lignes ?

Réponse.

```
[99, 2, 3]
```

La liste `a` est modifiée, elle aussi ! Écrire `b = a` ne crée pas de copie : `a` et `b` sont deux noms pour **la même liste en mémoire**. On dit que `a` et `b` sont des **alias**. Toute modification faite avec `b` se voit avec `a`, et inversement.

Pour obtenir une vraie copie, indépendante, on utilise la méthode `copy()` ou la fonction `list()`. Dans les deux cas, on obtient une **nouvelle liste**, et la liste de départ reste intacte :

```
a = [1, 2, 3]

b = a.copy()      # méthode 1 : la méthode copy()
c = list(a)       # méthode 2 : la fonction list()

b[0] = 99
print(a)   # a n'est pas modifiée
print(b)   # seule b est modifiée
```

```
[1, 2, 3]
[99, 2, 3]
```

À retenir. `=` crée un alias, pas une copie. Pour copier une liste, il faut utiliser `copy()` ou `list()`.

Exercice 11 *Copie et référence*

1. Qu'affiche le programme suivant ?

```
x = [10, 20, 30]
y = x
y.append(40)
print(x)
print(len(x))
```

2. Réécrivez le programme pour que la modification de `y` ne change pas `x`.

16 Effets de bord : listes et fonctions

Quand on passe une liste à une fonction, la fonction reçoit un **alias** de la liste, pas une copie. Si elle modifie la liste (ajout, suppression, affectation `L[i] = ...`), la modification est visible par l'appelant après l'appel : on dit que la fonction a un **effet de bord**.

```
def ajouter_zero(L):
    L.append(0)

notes = [12, 15]
ajouter_zero(notes)
print(notes)
```

```
[12, 15, 0]
```

Au contraire, réaffecter le paramètre ne touche pas la liste de l'appelant : `L = []` fait désigner au nom local `L` une nouvelle liste, et `notes` continue de désigner l'ancienne.

```
def vider(L):
    L = []

notes = [12, 15]
vider(notes)
print(notes)
```

```
[12, 15]
```

Deux styles de fonctions existent, et il faut choisir l'un ou l'autre et l'indiquer dans la docstring :

— une fonction qui **modifie en place** la liste reçue et ne renvoie rien, comme `L.sort()` ou `L.append()` ;

— une fonction **sans effet de bord** qui renvoie une nouvelle liste et laisse son argument intact, comme `sorted(L)`.

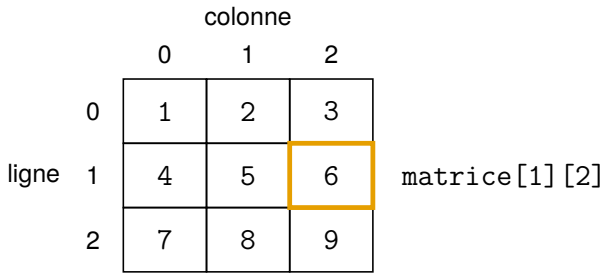
```
def doubler_en_place(L):
    """Double chaque élément de L. Modifie L, ne renvoie rien."""
    for i in range(len(L)):
        L[i] = 2 * L[i]

def doubler(L):
    """Renvoie une nouvelle liste dont les éléments sont les doubles de ceux de L. L n'
    est pas modifiée."""
    resultat = []
    for x in L:
        resultat.append(2 * x)
    return resultat
```

Les effets de bord sont une source classique de bugs : une liste modifiée « à distance » par une fonction appelée ailleurs dans le programme. Quand une fonction doit travailler sur une copie, on la crée explicitement avec `L.copy()` ou `list(L)`.

17 Listes de listes

Les éléments d'une liste peuvent eux-mêmes être des listes. On représente ainsi un tableau à deux dimensions, par exemple une **matrice** : chaque élément de la liste est une ligne.




```
matrice = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]

print(matrice[1])      # la ligne d'indice 1
print(matrice[1][2])   # dans cette ligne, l'élément d'indice 2
```

```
[4, 5, 6]
6
```

La notation `matrice[i][j]` désigne l'élément de la ligne `i` et de la colonne `j` : on écrit d'abord l'indice de la ligne, puis celui de la colonne. Pour parcourir tous les éléments, on imbrique deux boucles : la première parcourt les lignes, la seconde les éléments de chaque ligne.

```
for ligne in matrice:
    for element in ligne:
        print(element, end=" ")
    print()
```

```
1 2 3
4 5 6
7 8 9
```

Copier une liste de listes

Avec une liste de listes, la copie cache un second piège.

```
a = [[1, 2], [3, 4]]
b = a.copy()

b[0][0] = 99
print(a)
```

Question.
La liste `a` est-elle modifiée ?

Réponse.

```
[[99, 2], [3, 4]]
```

Elle l'est. La méthode `copy()` et la fonction `list()` ne copient que la liste principale : la copie est une nouvelle liste, mais ses éléments sont **les mêmes listes intérieures**, que `a` et `b` partagent. Modifier une ligne de la copie modifie donc aussi l'original.

Pour copier aussi les listes intérieures, on utilise la fonction `deepcopy()` du module `copy`, qui fait une **copie profonde** :

```
from copy import deepcopy

a = [[1, 2], [3, 4]]
b = deepcopy(a)

b[0][0] = 99
print(a)
print(b)
```

```
[[1, 2], [3, 4]]
[[99, 2], [3, 4]]
```

18 À RETENIR

1. Une liste regroupe plusieurs valeurs dans une seule variable, entre crochets : `notes = [15, 12, 18]`.
2. Le premier élément a l'indice `0`, le dernier l'indice `len(L) - 1`, qu'on écrit aussi `-1`.
3. Une liste se modifie : `L[i] = x`, `L.append(x)`, `L.insert(i, x)`, `L.pop()` et `L.remove(x)`.
4. On parcourt une liste par ses valeurs, `for x in L`, ou par ses indices, `for i in range(len(L))`, quand on a besoin des positions.
5. Pour construire une liste, on part souvent d'une liste vide et on ajoute un élément à chaque tour avec `append()` ; la liste en compréhension fait la même chose en une ligne.
6. `b = a` ne copie pas la liste : `a.copy()` ou `list(a)` en font une vraie copie, et `deepcopy(a)` copie aussi les listes intérieures.
7. Une fonction qui reçoit une liste peut la modifier : c'est un effet de bord.
8. Dans une liste de listes, `M[i][j]` désigne l'élément de la ligne `i` et de la colonne `j`.

19 Résumé

Concept	Syntaxe	Exemple
Création	`[e1, e2, ...]`	`[15, 12, 18]`
Accès	`L[i]`	`notes[0]` → `15`
Longueur	`len(L)`	`len(notes)` → `3`
Ajout	`L.append(x)`	`notes.append(17)`
Parcours par les valeurs	`for x in L`	`for n in notes`
Parcours par les indices	`for i in range(len(L))`	`notes[i]`
Construction	`L = []` puis `L.append(x)`	`carres.append(x ** 2)`
Compréhension	`[expr for x in L]`	`[x**2 for x in range(5)]`
Copie	`L.copy()` ou `list(L)`	`b = a.copy()`
Copie d'une liste de listes	`deepcopy(L)`, module `copy`	`b = deepcopy(matrice)`
Appartenance	`x in L`	`15 in notes` → `True`
Découper une chaîne	`s.split(sep)`	`"a-b".split("-")` → `["a", "b"]`
Assembler	`sep.join(L)`	`"-".join(["a", "b"])` → `"a-b"`
Matrice	`M[i][j]`	`matrice[1][2]` → `6`