Chapitre 6
Tableaux

Ce chapitre introduit les **tableaux** : des structures de données à deux dimensions, organisées en **lignes** et en **colonnes**. On apprend à les représenter avec des listes de listes, à les afficher, à accéder à leurs éléments, puis à les parcourir pour calculer des sommes, des moyennes, des extremums ou rechercher des valeurs. Le fil conducteur du chapitre est un projet : fabriquer une **grille de Sudoku**.

I. Représenter un tableau avec des lignes et des colonnes

Un tableau possède des **lignes** (sens horizontal) et des **colonnes** (sens vertical). En Python, on représente un tableau par une **liste de listes** : chaque ligne est une liste, et toutes les lignes sont regroupées dans une liste.

Définition 6.1.
Un **tableau** (ou matrice) est une liste dont chaque élément est lui-même une liste : chaque sous-liste représente une **ligne** du tableau. Les éléments d'une même ligne sont rangés dans l'ordre des **colonnes**.

■ **Exemple 6.1.**
On veut représenter la matrice suivante, qui possède 3 lignes et 3 colonnes :

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

On entre la première ligne comme une liste, puis la seconde ligne comme une liste, puis la troisième comme une liste. Toutes ces listes sont séparées par des virgules et regroupées dans une liste. ■

```
M = [[1,2,3],[4,5,6],[7,8,9]]
print(M)
```

1	2	3
4	5	6
7	8	9

R En Python, les **listes** et les **tableaux** sont la même chose : un tableau est une liste dont chaque élément est lui-même une liste. On parle de tableau à **deux dimensions** (2D) : la première dimension donne la ligne, la seconde donne la colonne. Dans ce chapitre, tous les éléments d'un tableau sont du même type (ici, des entiers).

■ **Exercice 6.1.**
Définir une variable `T` représentant le tableau suivant, à l'aide d'une liste de listes.

10	20
30	40

```
T = ...
```

■

II. Afficher et accéder au tableau

Quand on affiche notre tableau avec `print(M)`, on n'obtient pas le joli tableau de la section précédente : Python affiche la liste brute. On va l'afficher de façon plus lisible.

■ **Exemple 6.2.**
On affiche chaque ligne du tableau, l'une après l'autre : on obtient une représentation en lignes et colonnes. ■

```
for ligne in M:
    print(ligne)
```

■ **Exemple 6.3.**
On veut maintenant accéder à la valeur située à la ligne d'indice i et à la colonne d'indice j. On écrit `M[i][j]` : en effet, `M[i]` est une liste, et cette liste a des éléments ordonnés par colonnes. On cherche la valeur de la deuxième ligne (indice 1) et de la première colonne (indice 0). ■

```
print(M[1][0])
```

Exercice 6.2.
Expliquer pourquoi `M[1][0]` vaut 4 et non 1. ■

Exemple 6.4.
On affiche toutes les valeurs du tableau, une par une, avec une **double boucle** : la boucle externe parcourt les lignes, la boucle interne parcourt les colonnes de la ligne courante. ■

```
for i in range(len(M)):
    for j in range(len(M[i])):
        print(M[i][j])
```

R `len(M)` donne le nombre de lignes du tableau ; `len(M[i])` donne le nombre de colonnes de la ligne i (ici 3). On peut remplacer `len(.)` par les tailles du tableau, mais il vaut mieux les calculer : le code reste juste si le tableau change de taille.

Exemple 6.5.
On construit par **compréhension** un tableau de 3 lignes et 4 colonnes rempli de zéros : `[0] * 4` est la liste des 4 colonnes, et la compréhension répète cette liste 3 fois. ■

```
grille = [[0] * 4 for i in range(3)]
print(grille)
```

R La compréhension de liste, déjà rencontrée sur les listes, s'applique aussi aux tableaux : elle permet de construire une grille en une seule ligne, sans boucle d'initialisation.

III. Parcourir un tableau : algorithmes usuels

Les algorithmes vus sur les listes (somme, moyenne, maximum, minimum, recherche) s'appliquent aux tableaux. On peut parcourir un tableau de deux façons :

- par **valeurs** : `for element in tableau`, quand seule la valeur compte ;
- par **indices** : `for i in range(len(tableau))`, quand on a besoin de la position ou que l'on modifie le tableau.

Exemple 6.6.
On calcule la somme des valeurs d'un tableau de notes. ■

```
notes = [12, 8, 15, 10]
somme = 0
for note in notes:
    somme = somme + note
print(somme)
```

■ **Exemple 6.7.**
On calcule la moyenne des notes : la somme divisée par le nombre de notes, donné par `len`. ■

```
moyenne = somme / len(notes)
print(moyenne)
```

■ **Exemple 6.8.**
On cherche la plus grande valeur du tableau (le maximum) : on la compare à chaque valeur du parcours, en partant du premier élément. ■

```
maxi = notes[0]
for note in notes:
    if note > maxi:
        maxi = note
print(maxi)
```

R Il ne faut pas initialiser `maxi` à 0 : si toutes les valeurs du tableau sont négatives, aucune ne dépasserait 0 et le résultat serait faux. On initialise avec `notes[0]`, ce qui fonctionne quel que soit le contenu du tableau.

■ **Exemple 6.9.**
On teste si une valeur est présente dans le tableau : c'est la **recherche séquentielle**. On parcourt le tableau ; dès qu'on trouve la valeur cherchée, on met le booléen à `True`. ■

```
trouve = False
for note in notes:
    if note == 15:
        trouve = True
print(trouve)
```

■ **Exemple 6.10.**
On recherche l'**indice** d'une valeur : la variable `indice` garde la valeur `-1` tant que la valeur n'a pas été trouvée. C'est une valeur **sentinelle** : elle signale que la valeur n'est pas dans le tableau. ■

```
indice = -1
for i in range(len(notes)):
    if notes[i] == 15:
        indice = i
print(indice)
```

Exemple 6.11.
On **modifie** un tableau en place : on double chaque valeur. Il faut parcourir par indices, car on a besoin d'écrire dans la case `notes[i]`. ■

```
for i in range(len(notes)):
    notes[i] = notes[i] * 2
print(notes)
```

> **R** Avec `for note in notes`, la variable `note` est une **copie** de la valeur courante : la modifier ne change pas le tableau. Pour modifier le tableau, on passe par l'indice : `notes[i] = ...`. C'est pour cela qu'on choisit le parcours par indices dès qu'il faut modifier ou repérer une position.

Exemple 6.12.
On construit un **nouveau** tableau par compréhension : les carrés des entiers de 0 à 4. ■

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

> **R** On peut aussi parcourir un tableau à deux dimensions, ligne par ligne : une boucle externe sur les lignes, puis une boucle interne sur les colonnes. Tous les algorithmes de cette section s'adaptent alors à chaque ligne ou à tout le tableau.

Le projet du chapitre consiste à construire une **grille de Sudoku** : un plateau de nombres que l'on mélange (en échangeant des lignes et des colonnes), puis dont on cache des valeurs pour fabriquer la grille à faire résoudre. Les fonctions nécessaires sont construites dans la section Exercices d'application.

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : calcul mental d'expressions Python, lecture de code court, prédiction d'affichage, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 6.3.

Types et valeurs : répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
15 / 4
```

2. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
10 // 3
```

3. Quel est le type de la variable `a` après l'affectation suivante ?

```
a = "tableau"
```

4. Que vaut l'expression suivante ?

```
2 + 3 * 4
```

5. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
len("bonjour")
```

■

■ Exercice 6.4.

Listes et indices : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
notes = [12, 15, 8, 17]
print(notes[2])
```

2. Qu'affiche le programme suivant ?

```
fruits = ["pomme", "banane", "kiwi"]
print(fruits[0])
```

3. Qu'affiche le programme suivant ?

```
t = [10, 20, 30]
```

```
print(t[-1])
```

4. Qu'affiche le programme suivant, avec une liste de listes ?

```
m = [[1, 2], [3, 4]]
print(m[1][0])
```

5. Que se passe-t-il à l'exécution du programme suivant ?

```
t = [10, 20, 30]
print(t[3])
```

■

■ **Exercice 6.5.**

Boucles et parcours : répondre aux questions suivantes.

1. Quelles valeurs produit `range(5)` ?
2. Quelles valeurs produit `range(2, 10, 3)` ?
3. Combien de fois s'exécute la boucle `for i in range(4, 4)` ?
4. Pour une liste `t` de longueur `n`, quel est l'indice de son dernier élément ?
5. Quelle instruction parcourt directement les valeurs d'une liste `t`, sans passer par un indice ?

■

■ **Exercice 6.6.**

Somme et moyenne : répondre aux questions suivantes.

1. Que vaut `somme` après l'exécution du programme suivant ?

```
somme = 0
for v in [7, 2, 9, 4, 1]:
    somme = somme + v
```

2. Quelle valeur initiale faut-il donner à un accumulateur qui calcule une somme ?
3. Quelle est la moyenne du tableau de notes suivant ?

```
notes = [12, 8, 15, 10]
```

4. Que vaut `somme` après l'exécution du programme suivant, qui n'additionne que certaines valeurs ?

```
somme = 0
for v in [7, 2, 9, 4, 1]:
    if v % 2 == 0:
        somme = somme + v
```

5. Quelle expression donne le nombre de valeurs d'un tableau `notes` ?

■

■ **Exercice 6.7.**

Maximum, minimum et recherche : répondre aux questions suivantes.

1. Qu'affiche le programme suivant ?

```
tableau = [3, 7, 2, 9, 4]
maxi = tableau[0]
for element in tableau:
    if element > maxi:
        maxi = element
print(maxi)
```

2. Pourquoi initialise-t-on `maxi = tableau[0]` plutôt que `maxi = 0` ?
3. Qu'affiche le programme suivant ?

```
tableau = [5, 3, 8, 1]
cible = 10
trouve = False
for element in tableau:
    if element == cible:
        trouve = True
print(trouve)
```

4. Qu'affiche le programme suivant, lorsque la valeur cherchée n'est pas dans le tableau ?

```
tableau = [5, 3, 8, 1]
cible = 6
indice_trouve = -1
for i in range(len(tableau)):
    if tableau[i] == cible:
        indice_trouve = i
print(indice_trouve)
```

5. Pour trouver la **position** de la plus grande valeur d'un tableau, faut-il parcourir par valeurs ou par indices ?

■

■ **Exercice 6.8.**

Fonctions, chaînes et dictionnaires : répondre aux questions suivantes.

1. Que renvoie l'appel `f(5)` ?

```
def f(x):
```

```
    return x + 10

print(f(5))
```

2. Que vaut l'expression `"a" * 3` ?
3. Que renvoie l'expression `[1, 2] + [3]` ?
4. Que renvoie `d["nom"]` pour `d = {"nom": "Naruto", "annee": 2002}` ?
5. Que renvoie `len({"a": 1, "b": 2})` ?

■

Exercices d'application

Les exercices d'application construisent le projet du chapitre, la grille de Sudoku, puis ajoutent des exercices classiques sur les tableaux. Ils sont classés du plus simple au plus difficile.

■ **Exercice 6.9.**

Exercice filé — Sudoku, partie 1 : le plateau.

Le projet du chapitre est de fabriquer une grille de Sudoku : un plateau de nombres que l'on mélange, puis dont on cache des valeurs. On commence par le plateau. On dispose de la fonction suivante pour afficher un tableau ligne par ligne.

```
def afficher_tableau(tableau):
    for i in tableau:
        print(i)
```

1. Créer une variable `M` représentant le tableau suivant.

1	2	3
4	5	6
7	8	9

2. Contrôler votre tableau en l'affichant avec la fonction `afficher_tableau`.
3. Écrire une fonction `permute_lignes(M, i, j)` qui prend en argument un tableau `M` et deux indices de ligne `i` et `j`, et renvoie un nouveau tableau où les lignes d'indices `i` et `j` ont été échangées.
4. Écrire une fonction `permute_colonnes(M, i, j)` qui prend en argument un tableau `M` et deux indices de colonne `i` et `j`, et renvoie un nouveau tableau où les colonnes d'indices `i` et `j` ont été échangées.
5. Écrire une fonction `couple_aleatoire()` qui utilise le module `random` et renvoie un couple (un p-uplet à 2 éléments) de nombres entiers compris entre 0 et 2.

■

■ **Exercice 6.10.**

Exercice filé — Sudoku, partie 2 : le mélange et la grille question.

On combine maintenant les fonctions de la partie 1.

1. Écrire une fonction `melange_lignes(M)` qui choisit deux lignes au hasard, les échange à l'aide de `permute_lignes`, et renvoie le tableau mélangé.
2. Écrire une fonction `melange_colonnes(M)` qui choisit deux colonnes au hasard, les échange à l'aide de `permute_colonnes`, et renvoie le tableau mélangé.
3. Écrire une fonction `melange_complet(M)` qui échange deux lignes choisies au hasard puis deux colonnes choisies au hasard, et renvoie le tableau obtenu.
4. Écrire une fonction `melange_n_fois(M, n)` qui applique `n` fois la fonction `melange_complet` au tableau `M`.
5. Tester plusieurs fois la fonction suivante avec plusieurs valeurs de `p` comprises entre 0 et 1. Que comprenez-vous ?

```
import random as rd
def random_hide(p, k):
    if rd.random() < p:
        return ""
    else:
        return k
p = 0.7
print(random_hide(p, 3))
```

6. Écrire une fonction `affiche_valeurs(M)` qui affiche **valeur par valeur** toutes les valeurs du tableau (c'est différent de `afficher_tableau`, qui affiche les valeurs ligne par ligne).
7. Adapter cette fonction pour qu'elle **cache aléatoirement** des valeurs du tableau : la nouvelle fonction `cache_valeurs(M, p)` prendra en argument un `p` compris entre 0 et 1, et affichera chaque valeur ou une chaîne vide selon le tirage.

■

■ **Exercice 6.11.**

Écrire une fonction `somme_tableau(t)` qui prend en argument un tableau de nombres et renvoie la somme de ses valeurs. On pourra utiliser un parcours par valeurs. ■

■ **Exercice 6.12.**

Écrire une fonction `moyenne_tableau(t)` qui prend en argument un tableau de nombres et renvoie sa moyenne : la somme des valeurs divisée par le nombre de valeurs. ■

■ **Exercice 6.13.**
Écrire une fonction `max_tableau(t)` qui prend en argument un tableau de nombres et renvoie sa plus grande valeur. On initialisera le maximum avec `t[0]`. ■

■ **Exercice 6.14.**
Écrire une fonction `indice_max_tableau(t)` qui prend en argument un tableau de nombres et renvoie l'**indice** de sa plus grande valeur. En cas d'égalité, on renverra l'indice de la première occurrence. ■

■ **Exercice 6.15.**
Écrire une fonction `compte_occurrences(t, v)` qui prend en argument un tableau et une valeur, et renvoie le nombre de fois où la valeur `v` apparaît dans le tableau. ■

■ **Exercice 6.16.**
Écrire une fonction `est_present(t, v)` qui prend en argument un tableau et une valeur, et renvoie le booléen `True` si la valeur `v` est dans le tableau, `False` sinon. On pourra utiliser une valeur sentinelle. ■

■ **Exercice 6.17.**
Écrire une fonction `double_valeurs(t)` qui prend en argument un tableau et le **modifie en place** : chaque valeur est remplacée par son double. La fonction ne renvoie rien : c'est le tableau passé en argument qui est modifié. ■

■ **Exercice 6.18.**
Écrire une fonction `grille_zeros(n)` qui renvoie un tableau à `n` lignes et `n` colonnes rempli de zéros, construit par compréhension de liste. Tester la fonction avec `n = 3`, puis afficher le tableau obtenu avec une double boucle. ■

Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : déroulement de programmes, repérage d'erreurs, questions ouvertes sur les tableaux. Ils sont classés du plus abordable au plus exigeant.

Exercice 6.19.
Le programme suivant est censé doubler chaque valeur du tableau. Il ne fait rien.

```
tableau = [3, 5, 2]
for element in tableau:
    element = element * 2
print(tableau)
```

1. Sans l'exécuter, prédire ce qui est affiché.
2. Expliquer pourquoi le tableau n'est pas modifié.
3. Proposer une correction.

■

Exercice 6.20.
Le programme suivant est censé afficher toutes les valeurs du tableau. Il provoque une erreur.

```
tableau = [1, 2, 3]
for i in range(len(tableau) + 1):
    print(tableau[i])
```

1. Quelle erreur est levée, et pourquoi ?
2. Proposer une correction.

■

Exercice 6.21.
Écrire une fonction `est_trie(t)` qui prend en argument un tableau de nombres et renvoie le booléen `True` si le tableau est trié dans l'ordre croissant (chaque valeur est inférieure ou égale à la suivante), `False` sinon. On comparera chaque élément à son voisin suivant, en faisant attention aux bornes de la boucle. ■

Exercice 6.22.
Écrire une fonction `compte_au_dessus(t, seuil)` qui prend en argument un tableau de nombres et un entier `seuil`, et renvoie le nombre de valeurs strictement

supérieures à `seuil`. ■

■ **Exercice 6.23.**
Écrire une fonction `tableaux_identiques(a, b)` qui prend en argument deux tableaux de nombres de même taille et renvoie le booléen `True` s'ils sont identiques terme à terme, `False` sinon. Que se passerait-il si les deux tableaux n'avaient pas la même taille ? ■

■ **Exercice 6.24.**
Dans la recherche de l'indice d'une valeur, on utilise `-1` comme valeur initiale de la variable `indice`.

1. Pourquoi `-1` ne peut-il pas être un indice valide d'un tableau ?
2. Quelle est la différence entre « trouver la valeur » et « trouver l'indice » d'un élément dans un tableau ?
3. Proposer une autre façon de signaler qu'une valeur n'a pas été trouvée.

■