Chapitre 10
Traitement de données en tables

Une des utilisations principales de l'informatique de nos jours est le traitement de quantités importantes de données dans des domaines très variés : un site de commerce en ligne peut avoir à gérer des bases de données pour des dizaines de milliers d'articles en vente, de clients, de commandes ; un hôpital doit pouvoir accéder efficacement à tous les détails de traitements de ses patients. Les logiciels de gestion de base de données (SGBD) sont des programmes hautement spécialisés pour effectuer ce genre de tâches le plus efficacement et sûrement possible. Cependant, il est facile de mettre en œuvre les opérations de base dans un langage de programmation comme Python, à condition d'organiser les données sous forme de **tables**.

I. Les données

Définition 10.1.
Une **donnée** (data en anglais) est une valeur attribuée à une entité pour la décrire. Cette entité peut être un objet, une personne, un événement, etc.

Une donnée peut être **élémentaire** ou **complexe**.

— Une donnée **élémentaire** représente une caractéristique de base (un nom, un numéro, etc.). Cette donnée est caractérisée par un **descripteur** qui permet de donner le format dans lequel cette donnée est représentée.

— Une donnée **complexe** est constituée de plusieurs données élémentaires.

■ **Exemple 10.1.**
Pour décrire un élève, on peut utiliser plusieurs données élémentaires : son prénom (une chaîne de caractères), son âge (un entier), sa taille en mètres (un flottant). Chaque donnée possède un descripteur — `prenom`, `age`, `taille` — qui indique ce qu'elle représente. ■

■ **Exemple 10.2.**
La position d'un objet à la surface de la Terre est une donnée complexe : elle est constituée de deux données élémentaires, la latitude et la longitude. On regroupe ces deux valeurs dans une seule entité, par exemple un p-uplet `(48.85, 2.35)`. ■

II. Les données en tables

Organisées en table, les données se présentent comme un tableau : chaque **ligne** correspond à une entité (un **enregistrement**), chaque **colonne** correspond à un descripteur (un **champ**) partagé par toutes les lignes.

Définition 10.2.
Une **table** est une liste de p-uplets nommés qui partagent les mêmes descripteurs. En Python, on la représente le plus souvent par un **tableau doublement indexé** (une liste de listes) ou par une **liste de p-uplets**. Toutes les lignes ont la même structure : la même colonne contient le même type de valeur.

■ **Exemple 10.3.**
Voici une table de trois élèves. Chaque ligne est un enregistrement, chaque colonne est un descripteur. ■

Prenom	Age	Ville
Lina	15	Antony
Sami	16	Clichy
Zoe	15	Bagneux

■ **Exemple 10.4.**
Dans la table précédente, la colonne `Prenom` est le descripteur des noms, la ligne `Sami, 16, Clichy` est un enregistrement complet : elle décrit une seule entité avec toutes ses caractéristiques. ■

Vocabulaire 10.1. **Descripteur** : nom d'une colonne (aussi appelé **champ** ou attribut).
Enregistrement : une ligne de la table, qui décrit une entité.
Table : ensemble des enregistrements partageant les mêmes descripteurs.

III. Les fichiers CSV

Définition 10.3.
Le format **CSV** (pour *comma separated values*, soit en français « valeurs séparées par des virgules ») est un format très pratique pour représenter des données structurées. Dans ce format, chaque ligne représente un enregistrement et, sur une même ligne, les différents champs de l'enregistrement sont séparés par un **séparateur**.

Le séparateur le plus courant est la virgule. Cependant, en France, la virgule est

déjà utilisée pour les nombres décimaux : un fichier CSV peut donc fonctionner avec un autre séparateur — le point-virgule, la tabulation, les deux-points.

Exemple 10.5.
Voici un extrait du fichier `donnees.csv` : les impôts des foyers de plusieurs communes des Hauts-de-Seine en 2020. Le séparateur est le point-virgule. La première ligne contient les descripteurs (l'**en-tête**). ■

```
Commune;Dep.;Revenu;Foyers;Impot_net
Antony;922;18765.611;5007;-118.186
Asnieres-sur-Seine;921;36634.296;10264;-178.069
Bagneux;922;20181.849;5342;-93.58
Boulogne-Billancourt;922;40002.583;10768;-33.458
Clichy;921;32166.164;8896;-127.296
Colombes;921;38127.728;10590;-194.874
```

R Dans le fichier réel du chapitre, les noms des colonnes sont plus longs et contiennent des accents : « Dép. », « Revenu fiscal de référence des foyers fiscaux », « Impôt net (total)* ». Pour la lisibilité des programmes, on utilise ici des noms simplifiés : `Commune`, `Dep.`, `Revenu`, `Foyers`, `Impot_net`.

Exemple 10.6.
Un fichier CSV n'est rien d'autre qu'un **fichier texte** : on peut l'ouvrir dans n'importe quel éditeur de texte, et le lire en Python avec les fonctions habituelles de lecture de fichiers. ■

IV. Importer une table en Python

1. Un tableau de listes

La représentation la plus naturelle d'une table en Python est une **liste de listes** : chaque sous-liste est une ligne de la table. C'est un **tableau doublement indexé** : pour accéder à la valeur de la ligne i et de la colonne j, on écrit `t[i][j]`.

Exemple 10.7.
On construit la table des notes de la classe sous forme d'une liste de listes. Chaque ligne contient le prénom de l'élève et sa note. ■

```
notes = [["Lina", 15], ["Sami", 8], ["Zoe", 12]]
```

Exemple 10.8.
Le premier indice sélectionne la ligne, le second l'élément dans cette ligne : `notes[0]` est la première ligne `["Lina", 15]`, et `notes[1][1]` est la note de Sami, c'est-à-dire 8. ■

```
print(notes[0]) # ["Lina", 15]
print(notes[1][1]) # 8
```

■ **Exemple 10.9.**
Le nombre de lignes de la table est `len(notes)` ; le nombre de colonnes est `len(notes[0])` (la longueur d'une ligne). ■

```
print(len(notes)) # 3 lignes
print(len(notes[0])) # 2 colonnes
```

■ **Exemple 10.10.**
On peut lire un fichier CSV avec les outils de base de Python : on ouvre le fichier, on découpe chaque ligne avec `split` sur le séparateur, et on construit la table ligne par ligne. Les valeurs numériques sont converties avec `float` ou `int`. ■

```
table = []
with open("donnees.csv", encoding="utf-8") as fichier:
    for ligne in fichier:
        champs = ligne.strip().split(";")
        table.append(champs)
print(table[1]) # premiere ligne de donnees
```

2. La bibliothèque pandas

Pour traiter des données de façon efficace, on utilise souvent la bibliothèque spécialisée **pandas**. Elle fournit la structure `DataFrame` : un tableau avec toutes les données, muni de nombreuses méthodes de manipulation.

■ **Exemple 10.11.**
On importe la bibliothèque et on charge les données du fichier CSV dans un `DataFrame`. Le paramètre `sep = ";"` indique que le séparateur est le point-virgule. ■

```
import pandas as pd

dataframe = pd.read_csv("donnees.csv", sep = ";")
```

> **R** Que signifie `sep = ";"` ? Et pourquoi le met-on ? Le fichier CSV utilise le point-virgule comme séparateur, car la virgule sert déjà aux nombres décimaux en France. Sans ce paramètre, pandas supposerait une virgule et découperait les nombres décimaux en deux champs, ce qui détruirait la structure de la table.

■ **Exemple 10.12.**
Le `DataFrame` s'affiche comme un tableau propre : les colonnes portent les noms de l'en-tête du fichier, et les lignes sont numérotées automatiquement. ■

```
dataframe
```

V. Parcourir une table

Parcourir une table, c'est visiter ses lignes une à une. Avec une liste de listes, une simple boucle `for` suffit.

Exemple 10.13.
On affiche toutes les lignes de la table des notes, puis uniquement la première colonne (les prénoms). ■

```
for ligne in notes:
    print(ligne) # toute la ligne

for ligne in notes:
    print(ligne[0]) # premiere colonne : les prenoms
```

Exemple 10.14.
Pour parcourir une colonne, on garde le même indice de colonne pour toutes les lignes. On peut ainsi calculer la somme ou la moyenne d'une colonne numérique. ■

```
total = 0
for ligne in notes:
    total = total + ligne[1]
print(total / len(notes)) # moyenne des notes
```

Exemple 10.15.
Avec pandas, on peut parcourir les lignes d'un `DataFrame` avec la méthode `iterrows`. Chaque itération fournit un couple : l'indice de la ligne et la ligne elle-même (une `Series` pandas). ■

```
for indice, ligne in dataframe.iterrows():
    print(ligne["Commune"])
```

Exemple 10.16.
Quel est le format d'une ligne produite par `iterrows` ? C'est un p-uplet de deux éléments : l'indice de la ligne, puis une `Series` dont on peut extraire une valeur par son descripteur. Pour accéder à la donnée `Commune` d'une ligne, on écrit `ligne["Commune"]`. ■

```
for indice, ligne in dataframe.iterrows():
    print(indice, ligne["Commune"], ligne["Foyers"])
```

VI. Recherche dans une table

Définition 10.4.
La **recherche dans une table** consiste à obtenir les valeurs de certains descripteurs (champs), avec éventuellement des critères les concernant.

1. Filtrage par une boucle

La méthode la plus simple consiste à parcourir les lignes avec une boucle et à tester un critère avec un `if`. On parle de **filtrage** : on ne garde que les lignes qui vérifient la condition.

■ **Exemple 10.17.**
On affiche toutes les communes dont le département (`Dep.`) est 922. Le critère est une condition de comparaison `ligne[1] == 922` dans la boucle. ■

```
for ligne in table:
    if ligne[1] == "922":
        print(ligne[0])
```

■ **Exemple 10.18.**
On affiche tous les revenus de référence des foyers de Clichy. Le critère porte sur le nom de la commune, une chaîne de caractères. ■

```
for ligne in table:
    if ligne[0] == "Clichy":
        print(ligne[2])
```

■ **Exemple 10.19.**
Les critères peuvent se combiner avec les opérateurs de la **logique propositionnelle** : `and` (et), `or` (ou), `not` (non). On affiche ici les communes du 921 ayant plus de 8000 foyers fiscaux. ■

```
for ligne in table:
    if ligne[1] == "921" and int(ligne[3]) > 8000:
        print(ligne[0])
```

2. Le filtrage avec pandas

Pandas est optimisé pour ne pas faire de boucle. La façon de faire est la suivante : on place la **condition** entre crochets, directement après le `DataFrame`. La condition porte sur une colonne entière, et pandas sélectionne toutes les lignes pour lesquelles elle est vraie.

■ **Exemple 10.20.**
Qu'affiche `dataframe[dataframe["Dep."] == 922]` ? Un nouveau `DataFrame` contenant uniquement les lignes dont la colonne `Dep.` vaut 922 : toutes les communes du département 922. ■

```
dataframe[dataframe["Dep."] == 922]
```

■ **Exemple 10.21.**
Si l'on ne veut avoir que les communes, on ajoute un deuxième crochet avec le nom de la colonne : `dataframe[condition]["Commune"]`. ■

```
dataframe[dataframe["Dep."] == 922]["Commune"]
```

■ **Exemple 10.22.**
La commande pour obtenir l'impôt net de Clichy combine le filtrage sur la commune et la sélection de la colonne `Impot_net`. ■

```
dataframe[dataframe["Commune"] == "Clichy"]["Impot_net"]
```

3. La recherche de doublons

Une table peut contenir des **doublons** : plusieurs lignes qui présentent la même valeur dans une colonne (par exemple la même commune, si le fichier contient plusieurs tranches de revenus). La recherche de doublons consiste à détecter ces répétitions.

■ **Exemple 10.23.**
Pour connaître les villes présentes dans le fichier, on collecte la colonne `Commune` dans une liste. Certaines communes apparaissent plusieurs fois : ce sont des doublons. On peut les repérer en construisant un ensemble (`set`), qui ne garde qu'une occurrence de chaque valeur. ■

```
villes = [ligne[0] for ligne in table]
print(villes)
print(len(villes), len(set(villes)))
```

R Un **test de cohérence** consiste à vérifier que les données de la table sont plausibles. Par exemple, dans le fichier réel des impôts, certaines valeurs sont marquées `n.c.` (non communiqué) : il faut les détecter et décider comment les traiter (les ignorer, les signaler) avant de faire des calculs. Une moyenne calculée sur des valeurs manquantes serait fausse sans ce test.

VII. Tri d'une table

Trier une table, c'est réordonner ses lignes suivant une colonne. Avec une liste de listes, on utilise la fonction `sorted` avec le paramètre `key`, qui indique sur quelle partie de chaque ligne porte le tri.

Exemple 10.24.
On trie la table des notes par note croissante : la clé de tri est `ligne[1]`, c'est-à-dire la deuxième colonne. ■

```
trie = sorted(notes, key=lambda ligne: ligne[1])
print(trie)
```

R La fonction `sorted` ne modifie pas la table d'origine : elle renvoie une **nouvelle liste** triée. Pour trier en place, on utiliserait `liste.sort()`.

Exemple 10.25.
Avec pandas, la méthode `sort_values` trie le `DataFrame` suivant une colonne : ici, par nombre croissant de foyers fiscaux. ■

```
dataframe.sort_values("Foyers")
```

VIII. Fusion de tables

Construire une nouvelle table en combinant les données de deux tables s'appelle une **fusion**. Les deux tables doivent partager les mêmes descripteurs : on parle de **domaine de valeurs** commun.

Exemple 10.26.
En Python, si deux listes de listes ont la même structure, l'opérateur `+` les concatène en une seule table. ■

```
table1 = [["Lina", 15], ["Sami", 8]]
table2 = [["Zoe", 12], ["Theo", 6]]
toutes = table1 + table2
print(toutes)
```

Exemple 10.27.
Avec pandas, la fonction `pd.concat` combine plusieurs `DataFrame` en un seul, à condition qu'ils aient les mêmes colonnes. ■

```
asnieres = dataframe[dataframe["Commune"] == "Asnieres-sur-Seine"]
clichy = dataframe[dataframe["Commune"] == "Clichy"]
fusion = pd.concat([asnieres, clichy])
```

IX. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre : représentation des données en tables, fichiers CSV, parcours, recherche, doublons, tri et fusion. 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 : lecture de code court, prédiction d'affichage, accès aux cases d'une table, complétion. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 10.1.

Rappels : types, calculs et booléens. Répondre aux questions suivantes sans exécuter les programmes.

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

```
4 + 0.5
```

2. Que vaut l'expression suivante ?

```
2 ** 6
```

3. Que vaut l'expression suivante ?

```
23 // 5
```

4. Que vaut l'expression suivante ?

```
(8 > 3) or (2 > 5)
```

■

■ Exercice 10.2.

Tables : accéder aux cases d'une table.

1. Qu'affiche le programme suivant ?

```
t = [[1, 3, 5], [7, 9, 11], [13, 15, 17]]
print(t[1][2])
```

2. Que renvoie `len(t)` dans le programme suivant ?

```
t = [[1, 3, 5], [7, 9, 11], [13, 15, 17]]
print(len(t))
```

3. Que renvoie `len(t[0])` dans le programme suivant ?

```
t = [[1, 3, 5], [7, 9, 11], [13, 15, 17]]
print(len(t[0]))
```

4. Qu'affiche le programme suivant ?

```
t = [[1, 3, 5], [7, 9, 11], [13, 15, 17]]
print(t[0][2] + t[2][0])
```

■

■ **Exercice 10.3.**

Tables : filtrer des lignes.

1. Quelle condition complète ce programme, qui doit afficher le nom des élèves ayant une note supérieure ou égale à 10 ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12]]
for eleve in eleves:
    if ____:
        print(eleve[0])
```

2. Qu'affiche le programme suivant ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 6]]
resultat = []
for eleve in eleves:
    if eleve[1] < 10:
        resultat.append(eleve[0])
print(resultat)
```

3. Qu'affiche le programme suivant, qui filtre avec deux conditions ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 6]]
resultat = []
for eleve in eleves:
    if eleve[1] >= 8 and eleve[1] <= 12:
        resultat.append(eleve[0])
print(resultat)
```

4. Quelle condition complète ce programme, qui doit afficher le nom des élèves en échec (note strictement inférieure à 10) ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 6]]
for eleve in eleves:
    if ____:
        print(eleve[0])
```

■

■ **Exercice 10.4.**

Tables : calculer sur une colonne.

1. Quel indice complète ce programme, qui doit additionner la colonne des notes ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12]]
total = 0
for eleve in eleves:
    total = total + eleve[____]
print(total)
```

2. Qu'affiche le programme suivant, qui calcule une moyenne ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 9]]
total = 0
for eleve in eleves:
    total = total + eleve[1]
print(total / len(eleves))
```

3. Qu'affiche le programme suivant, qui compte les lignes vérifiant un critère ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 6]]
compteur = 0
for eleve in eleves:
    if eleve[1] >= 10:
        compteur = compteur + 1
print(compteur)
```

4. Quelle expression complète correctement le calcul de la moyenne ?

```
def moyenne_notes(eleves):
    total = 0
    for eleve in eleves:
        total = total + eleve[1]
    return total / ____
```

5. Qu'affiche le programme suivant, qui cherche la meilleure note ?

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12]]
meilleur = eleves[0]
for eleve in eleves:
    if eleve[1] > meilleur[1]:
        meilleur = eleve
print(meilleur[0])
```

■

■ **Exercice 10.5.**

Rappels : listes, dictionnaires et fichiers CSV.

1. Qu'affiche le programme suivant ?

```
d = {"a": 2, "b": 3}
print(d["a"] + d["b"])
```

2. Qu'affiche le programme suivant ?

```
print([x * 2 for x in range(1, 5)])
```

3. Dans un fichier CSV dont les champs sont séparés par des points-virgules, quelle ligne du programme suivant permet de récupérer la liste des champs d'une ligne ?

```
ligne = "Antony;922;5007"
champs = ____
```

4. Quelle commande permet de charger le fichier `donnees.csv` (séparé par des points-virgules) dans un `DataFrame` pandas ?

```
import pandas as pd
dataframe = ____
```

5. Qu'affiche le programme suivant ?

```
data = [["Antony", "922"], ["Clichy", "921"], ["Bagneux", "922"]]
for ligne in data:
    print(ligne[0])
```

■

Exercices d'application

Les exercices d'application manipulent la table des impôts du chapitre. On suppose que le fichier `donnees.csv` contient l'en-tête et des lignes de la forme :

```
Commune;Dep.;Revenu;Foyers;Impot_net
```

On utilisera la représentation en liste de listes (tableau doublement indexé) pour les exercices 1 à 4 et 7, et le `DataFrame` pandas `dataframe` pour les exercices 5, 6 et 8.

■ Exercice 10.6.

Afficher toutes les communes. Faire une boucle qui affiche toutes les valeurs de la colonne `Commune` de la table.

```
for ligne in table:
    # Ecrire ici le parcours
```

■

■ **Exercice 10.7.**
Afficher tous les revenus. Faire une boucle qui affiche tous les `Revenu` (revenu fiscal de référence des foyers fiscaux) de la table.

```
for ligne in table:
    # Ecrire ici le parcours
```

■

■ **Exercice 10.8.**
Afficher les communes du 922. Faire une boucle qui affiche toutes les communes dont le `Dep.` est 922. Astuce : ajouter un `if` dans la boucle.

```
for ligne in table:
    # Ecrire ici le parcours avec condition
```

■

■ **Exercice 10.9.**
Afficher les revenus de Clichy. Faire une boucle qui affiche tous les `Revenu` de Clichy.

```
for ligne in table:
    # Ecrire ici le parcours avec condition
```

■

■ **Exercice 10.10.**
L'impôt net de Clichy avec pandas. Écrire la commande pandas qui permet d'obtenir la colonne `Impot_net` de Clichy.

```
# Ecrire ici la commande pandas
```

■

■ **Exercice 10.11.**
Moyenne des impôts de Clichy. Grâce à la commande `list` et à quelques outils de Python, calculer la moyenne des `Impot_net` de Clichy. Attention : le fichier contient une ligne par tranche de revenus ; penser à multiplier chaque impôt par

le nombre de foyers fiscaux de la tranche (`Foyers`) avant de faire la moyenne.

```
# Ecrire ici le calcul de la moyenne ponderee
```

■

■ **Exercice 10.12.**

Recherche de doublons.

1. Faire une liste `villes` contenant toutes les communes de la table (une valeur par ligne).
2. Faire une seconde liste qui répertorie les communes apparaissant plusieurs fois (les doublons).

```
# Ecrire ici la collecte des villes puis des doublons
```

■

■ **Exercice 10.13.**

Tri et fusion avec pandas.

1. Trier la table par nombre croissant de foyers fiscaux.
2. Faire deux tables (`DataFrame`) : une pour Asnières-sur-Seine et une pour Clichy.
3. Faire une table regroupant ces deux tables.

```
# Ecrire ici les commandes pandas
```

■

■ **Exercice 10.14.**

Construire une table de notes. Créer une table `eleves` contenant les lignes `["Aya", 14]`, `["Leo", 7]`, `["Mia", 11]`, `["Noe", 9]`, puis afficher chaque ligne avec une boucle.

```
eleves = [["Aya", 14], ["Leo", 7], ["Mia", 11], ["Noe", 9]]
# Ecrire ici le parcours
```

■

■ **Exercice 10.15.**

Extraire une colonne. Écrire une fonction `colonne(table, j)` qui prend une table et un indice de colonne, et qui renvoie la liste des valeurs de cette colonne. Tester avec la table `eleves` et la colonne 0.

```
def colonne(table, j):
    # Ecrire ici la collecte
```

■

■ **Exercice 10.16.**
Filtrer des lignes. Écrire une fonction `admis(eleves)` qui renvoie une nouvelle table contenant uniquement les lignes dont la note est supérieure ou égale à 10. Tester avec la table `eleves`.

```
def admis(eleves):
    # Ecrire ici le filtrage
```

■

■ **Exercice 10.17.**
Somme et moyenne d'une colonne. Écrire un script qui calcule et affiche la somme puis la moyenne des notes de la table `eleves`.

```
eleves = [["Aya", 14], ["Leo", 7], ["Mia", 11], ["Noe", 9]]
# Ecrire ici le calcul
```

■

■ **Exercice 10.18.**
Maximum d'une colonne. Écrire un script qui affiche la meilleure note de la table `eleves`, puis le prénom de l'élève qui l'a obtenue.

```
eleves = [["Aya", 14], ["Leo", 7], ["Mia", 11], ["Noe", 9]]
# Ecrire ici la recherche du maximum
```

■

■ **Exercice 10.19.**
Compter des lignes. Écrire un script qui compte et affiche le nombre de communes

de la table des impôts dont le `Dep.` est 921.

```
# Ecrire ici le comptage avec condition
```

■

■ **Exercice 10.20.**
Fusionner et trier. Construire deux petites tables de notes (`classe1` et `classe2`), les fusionner en une seule table, puis l'afficher triée par note croissante.

```
classe1 = [["Aya", 14], ["Leo", 7]]
classe2 = [["Mia", 11], ["Noe", 9]]
# Ecrire ici la fusion puis le tri
```

■

Exercices de réflexion

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

■ **Exercice 10.21.**
Table de tables. Construire par compréhension une table `m` de 3 lignes et 3 colonnes contenant les entiers de 1 à 9, puis répondre aux questions suivantes.

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

1. Que vaut `m[1][2]` ? Que vaut `m[2][0]` ?
2. Écrire une boucle qui affiche toutes les valeurs de la première colonne.
3. Écrire une boucle qui affiche toutes les valeurs de la deuxième ligne.
4. Que se passe-t-il si l'on écrit `m[3][0]` ? Expliquer.

■

■ **Exercice 10.22.**
Erreur à corriger. Le programme suivant devrait afficher la somme des notes de la table `eleves`, mais il affiche 4. Expliquer l'erreur et corriger le programme.

```
eleves = [["Lina", 15], ["Sami", 8], ["Zoe", 12], ["Theo", 6]]
total = 0
for eleve in eleves:
    total = total + eleve[0]
print(total)
```

1. Pourquoi le programme affiche-t-il 4 ?
2. Corriger le programme pour qu'il affiche la somme des notes.

■

■ **Exercice 10.23.**
Agrégation : la moyenne pondérée. Le fichier des impôts contient une ligne par tranche de revenus. Pour calculer le revenu moyen d'une commune, il ne faut pas faire la moyenne des colonnes `Revenu` : il faut **pondérer** chaque revenu par le nombre de foyers fiscaux de la tranche.

1. Expliquer pourquoi la simple moyenne des `Revenu` serait fausse.
2. Écrire une fonction `revenu_moyen(table, commune)` qui renvoie la moyenne pondérée des revenus d'une commune : somme des produits `Revenu * Foyers`, divisée par la somme des `Foyers`.

```
def revenu_moyen(table, commune):
    # Ecrire ici le calcul de la moyenne ponderee
```

■

■ **Exercice 10.24.**
Test de cohérence. Dans le fichier réel des impôts, certaines valeurs sont marquées `n.c.` (non communiqué) : ce ne sont pas des nombres.

1. Que se passe-t-il si l'on essaie de convertir `"n.c."` avec `float` ?
2. Proposer un test de cohérence qui détecte les valeurs `n.c.` dans une colonne avant de faire un calcul.
3. Écrire une boucle qui compte les valeurs `n.c.` de la colonne `Impot_net` et les signale.

■

■ **Exercice 10.25.**
Doublons sans `set`. On veut construire la liste des communes sans doublon, sans utiliser `set`.

1. Écrire un programme qui construit une liste `uniques` : pour chaque commune de la table, on l'ajoute seulement si elle n'y est pas déjà (test avec `in`).
2. Quelle est la taille de `uniques` par rapport au nombre de lignes de la table ? Expliquer.
3. Proposer une méthode pour compter le nombre d'occurrences de chaque commune.