Fichiers CSV

Bases de Python

Introduction

Beaucoup de données se présentent sous forme de **table** : une ligne par objet décrit, une colonne par caractéristique. Un relevé météo, la liste des élèves d'une classe ou les horaires d'une ligne de bus en sont des exemples.

Pour échanger ces tables entre logiciels, on utilise très souvent le format **CSV** (*Comma Separated Values*, « valeurs séparées par des virgules »). Un fichier CSV est un simple fichier texte :

— chaque ligne du fichier est une ligne de la table, appelée **enregistrement** ;

— sur une ligne, les valeurs sont séparées par un caractère choisi, le **délimiteur** : une virgule, ou un point-virgule.

Chaque colonne porte un nom, appelé **descripteur**. Un enregistrement donne une **valeur** à chaque descripteur : c'est un **p-uplet nommé**. Une table est donc une liste d'enregistrements qui partagent les mêmes descripteurs.

Un tableur (LibreOffice Calc, Excel) ouvre un fichier CSV et sait en enregistrer un ; en Python, le module `csv` permet de le lire et de l'écrire.

Structure d'un fichier CSV

Voici le fichier `meteo.csv`, qui donne pour quelques villes la température maximale de la journée, en degrés, et la hauteur de pluie tombée, en millimètres :

```
ville,région,température,pluie
Lyon,Auvergne-Rhône-Alpes,24,0
Brest,Bretagne,17,6
Nice,Provence-Alpes-Côte d'Azur,26,0
Lille,Hauts-de-France,15,12
```

La première ligne est l'**en-tête** : elle donne les descripteurs, c'est-à-dire le nom de chaque colonne. Chacune des lignes suivantes est un enregistrement, qui décrit une ville : pour Brest, le descripteur `température` prend la valeur `17`. Le même fichier, présenté en tableau :

ville	région	température	pluie
Lyon	Auvergne-Rhône-Alpes	24	0
Brest	Bretagne	17	6
Nice	Provence-Alpes-Côte d'Azur	26	0
Lille	Hauts-de-France	15	12

Deux précautions :

— en France, les tableurs utilisent souvent le **point-virgule** comme délimiteur, parce que la virgule sert à écrire les nombres décimaux (`12,5`) ;

— une valeur qui contient le délimiteur s'écrit entre guillemets, par exemple `"Paris, 5e"`.

Lecture avec csv.reader()

La fonction `csv.reader()` lit le fichier ligne par ligne et donne chaque ligne sous forme de **liste** de valeurs :

```
import csv

with open('meteo.csv', encoding='utf-8') as fichier:
    lecteur = csv.reader(fichier, delimiter=',')
    for ligne in lecteur:
        print(ligne)
```

```
['ville', 'région', 'température', 'pluie']
['Lyon', 'Auvergne-Rhône-Alpes', '24', '0']
['Brest', 'Bretagne', '17', '6']
['Nice', "Provence-Alpes-Côte d'Azur", '26', '0']
['Lille', 'Hauts-de-France', '15', '12']
```

La première liste est l'en-tête. Pour la passer, on appelle une fois `next(lecteur)` avant la boucle : la boucle commence alors à la deuxième ligne. Dans chaque liste, `ligne[0]` est la ville, `ligne[2]` la température :

```
import csv

with open('meteo.csv', encoding='utf-8') as fichier:
    lecteur = csv.reader(fichier, delimiter=',')
    next(lecteur)  # passe la ligne d'en-tête
    for ligne in lecteur:
        print(f'À {ligne[0]}, il a fait {ligne[2]} degrés')
```

```
À Lyon, il a fait 24 degrés
À Brest, il a fait 17 degrés
À Nice, il a fait 26 degrés
À Lille, il a fait 15 degrés
```

Toutes les valeurs sont des chaînes

Dans les listes ci-dessus, la température est écrite `'24'`, entre apostrophes : le module `csv` donne **toujours des chaînes de caractères**, même quand la valeur ressemble à un nombre. Pour calculer, il faut d'abord convertir avec `int()` ou `float()`.

```
import csv

```

```
with open('meteo.csv', encoding='utf-8') as fichier:
    lecteur = csv.reader(fichier, delimiter=',')
    next(lecteur)
    total = 0
    nombre = 0
    for ligne in lecteur:
        total = total + int(ligne[2])
        nombre = nombre + 1
print(f'Température moyenne : {total / nombre} degrés')
```

```
Température moyenne : 20.5 degrés
```

Sans `int()`, l'instruction `total + ligne[2]` provoquerait une erreur, car on ne peut pas ajouter une chaîne à un entier.

Lecture avec csv.DictReader()

La fonction `csv.DictReader()` utilise l'en-tête : elle donne chaque ligne sous forme de **dictionnaire**, dont les clés sont les descripteurs, lus dans l'en-tête. L'en-tête n'apparaît donc plus comme une ligne de données.

```
import csv

with open('meteo.csv', encoding='utf-8') as fichier:
    lecteur = csv.DictReader(fichier, delimiter=',')
    for ligne in lecteur:
        print(f'{ligne["ville"]} ({ligne["région"]}) : {ligne["pluie"]} mm de pluie')
```

```
Lyon (Auvergne-Rhône-Alpes) : 0 mm de pluie
Brest (Bretagne) : 6 mm de pluie
Nice (Provence-Alpes-Côte d'Azur) : 0 mm de pluie
Lille (Hauts-de-France) : 12 mm de pluie
```

Écrire `ligne["pluie"]` est plus clair que `ligne[3]`, et le programme reste juste si quelqu'un change l'ordre des colonnes dans le fichier.

Écriture avec csv.writer()

Pour créer un fichier CSV, on ouvre un fichier en écriture et on lui associe un `csv.writer()`. La méthode `writerow()` écrit une ligne à partir d'une liste :

```
import csv

releves = [
    ['ville', 'région', 'température', 'pluie'],
    ['Rennes', 'Bretagne', 18, 4],
    ['Toulouse', 'Occitanie', 27, 0],
]

```

```python
with open('releves.csv', 'w', encoding='utf-8', newline='') as fichier:
    ecrivain = csv.writer(fichier, delimiter=',')
    for ligne in releves:
        ecrivain.writerow(ligne)
```

Le fichier `releves.csv` contient alors :

```
ville,région,température,pluie
Rennes,Bretagne,18,4
Toulouse,Occitanie,27,0
```

Les nombres sont convertis en texte à l'écriture. Le paramètre `newline=''` évite, sous Windows, une ligne vide après chaque enregistrement.

Le mode `'w'` remplace le contenu du fichier. Pour **ajouter** une ligne à la fin d'un fichier existant, on l'ouvre en mode `'a'` (*append*, « ajouter ») :

```python
with open('releves.csv', 'a', encoding='utf-8', newline='') as fichier:
    ecrivain = csv.writer(fichier, delimiter=',')
    ecrivain.writerow(['Strasbourg', 'Grand Est', 21, 2])
```

`releves.csv` compte alors trois villes : Rennes, Toulouse et Strasbourg.

Écriture avec csv.DictWriter()

`csv.DictWriter()` écrit des **dictionnaires**. Le paramètre `fieldnames` lui donne la liste des colonnes ; la méthode `writeheader()` écrit la ligne d'en-tête, puis `writerow()` écrit un dictionnaire par ligne :

```python
import csv

releves = [
    {'ville': 'Rennes', 'température': 18},
    {'ville': 'Toulouse', 'température': 27},
]

with open('temperatures.csv', 'w', encoding='utf-8', newline='') as fichier:
    ecrivain = csv.DictWriter(fichier, fieldnames=['ville', 'température'], delimiter=','
    )
    ecrivain.writeheader()        # ligne d'en-tête
    for ligne in releves:
        ecrivain.writerow(ligne)  # un dictionnaire par ligne
```

```
ville,température
Rennes,18
Toulouse,27
```

Comparaison : reader() *vs* DictReader()

Pour la ligne de Brest, les deux fonctions donnent :

```
# csv.reader() : une liste, lue par indice
['Brest', 'Bretagne', '17', '6']
# ligne[2] vaut '17'

# csv.DictReader() : un dictionnaire, lu par nom de colonne
{'ville': 'Brest', 'région': 'Bretagne', 'température': '17', 'pluie': '6'}
# ligne['température'] vaut '17'
```

`csv.reader()` suffit pour un petit fichier dont on connaît l'ordre des colonnes. `csv.DictReader()` est préférable dès qu'il y a plusieurs colonnes : le code se lit mieux et ne dépend pas de leur ordre. Dans les deux cas, les valeurs restent des chaînes.

Charger un CSV dans une liste de dictionnaires

On charge souvent tout un fichier en mémoire, sous forme d'une **liste de dictionnaires** : chaque dictionnaire est une ligne de la table. On peut ensuite la parcourir autant de fois qu'on veut, la filtrer ou la trier.

```
import csv

def charger_csv(nom_fichier):
    """Renvoie le contenu du fichier CSV sous forme de liste de dictionnaires."""
    with open(nom_fichier, encoding='utf-8') as fichier:
        lecteur = csv.DictReader(fichier, delimiter=',')
        return list(lecteur)

table = charger_csv('meteo.csv')
print(len(table))
print(table[1])
```

```
4
{'ville': 'Brest', 'région': 'Bretagne', 'température': '17', 'pluie': '6'}
```

Pour garder, par exemple, les villes où il n'a pas plu :

```
seches = [ligne['ville'] for ligne in table if int(ligne['pluie']) == 0]
print(seches)
```

```
['Lyon', 'Nice']
```

Rechercher avec plusieurs critères

Un critère de recherche est une expression booléenne : on combine plusieurs conditions avec les opérateurs `and`, `or` et `not`.

— Avec `and`, une ligne est gardée si **toutes** les conditions sont vraies : ici, les villes à plus de 25 degrés **et** sans pluie.

— Avec `or`, une ligne est gardée si **au moins une** condition est vraie : ici, les villes à moins de 16 degrés **ou** à plus de 5 mm de pluie.

```
ensoleillees = [ligne['ville'] for ligne in table
                if int(ligne['température']) > 25
                and int(ligne['pluie']) == 0]
print(ensoleillees)

fraiches_ou_pluvieuses = [ligne['ville'] for ligne in table
                          if int(ligne['température']) < 16
                          or int(ligne['pluie']) > 5]
print(fraiches_ou_pluvieuses)
```

```
['Nice']
['Brest', 'Lille']
```

Lyon est sèche, mais à 24 degrés seulement : elle ne vérifie pas le premier critère. Brest, à 17 degrés, est gardée par le second critère grâce à ses 6 mm de pluie ; Lille vérifie les deux conditions.

Doublons et cohérence d'une table

Avant d'exploiter une table, on vérifie qu'elle ne contient pas d'erreur de saisie. Pour l'exemple, deux enregistrements fautifs sont ajoutés à la table de départ :

```
releves = table + [
    {'ville': 'Brest', 'région': 'Bretagne', 'température': '19', 'pluie': '2'},
    {'ville': 'Rennes', 'région': 'Bretagne', 'température': '18', 'pluie': '-4'},
]
```

Les doublons. Un **doublon** est un enregistrement présent deux fois. La fonction `doublons()` parcourt la table et garde dans la liste `vus` chaque valeur déjà rencontrée pour le descripteur choisi ; une valeur qui s'y trouve déjà est un doublon :

```
def doublons(table, descripteur):
    vus = []
    repetes = []
    for ligne in table:
        valeur = ligne[descripteur]
        if valeur in vus:
            repetes.append(valeur)
        else:
            vus.append(valeur)
    return repetes
```

```

print(doublons(releves, 'ville'))
```

```
['Brest']
```

La cohérence. Chaque descripteur a un **domaine de valeurs** : l'ensemble des valeurs qu'il peut prendre. Une hauteur de pluie n'est jamais négative, et une température ne dépasse pas 50 degrés. Une valeur hors de son domaine signale une erreur :

```
incoherentes = [ligne['ville'] for ligne in releves
                if int(ligne['pluie']) < 0
                or int(ligne['température']) > 50]
print(incoherentes)
```

```
['Rennes']
```

Il faut corriger ces enregistrements avant de calculer une moyenne ou de trier la table.

Trier une table suivant une colonne

La fonction `sorted()` trie une liste. Pour trier une table, c'est-à-dire une liste de dictionnaires, suivant une colonne, on lui donne avec le paramètre `key` une fonction qui, appliquée à une ligne, renvoie la valeur à comparer :

```
eleves = [
    {"nom": "Alice", "note": 15},
    {"nom": "Bob", "note": 12},
    {"nom": "Chloé", "note": 17},
]

def note(ligne):
    return ligne["note"]

par_note = sorted(eleves, key=note)
print([e["nom"] for e in par_note])      # ['Bob', 'Alice', 'Chloé']

decroissant = sorted(eleves, key=note, reverse=True)
print([e["nom"] for e in decroissant])   # ['Chloé', 'Alice', 'Bob']
```

On écrit le nom de la fonction sans parenthèses : c'est `sorted()` qui l'appelle sur chaque ligne. Le paramètre `reverse=True` inverse l'ordre. Le même paramètre `key` existe pour `max()` et `min()` : `max(eleves, key=note)` renvoie la ligne de la meilleure note.

Fusionner deux tables

Fusionner deux tables, c'est construire une nouvelle table qui combine leurs données. Les deux tables doivent avoir un descripteur commun, qui sert de **clé** de fusion : deux lignes sont réunies quand elles ont

la même valeur pour ce descripteur. Ce descripteur doit avoir le même domaine de valeurs dans les deux tables : les mêmes noms de villes, écrits de la même façon (`'LYON'` ne correspond pas à `'Lyon'`).

Voici le fichier `departements.csv`, qui donne le département de quelques villes :

```
ville,département
Lyon,Rhône
Brest,Finistère
Lille,Nord
Marseille,Bouches-du-Rhône
```

Pour chaque ligne de la première table, la fonction `fusionner()` parcourt toute la seconde table. Quand les deux lignes ont la même valeur pour la clé, elle crée une nouvelle ligne, qui reçoit les colonnes de la première ligne, puis celles de la seconde :

```
def fusionner(table1, table2, cle):
    resultat = []
    for ligne1 in table1:
        for ligne2 in table2:
            if ligne1[cle] == ligne2[cle]:
                nouvelle_ligne = {}
                for k in ligne1:
                    nouvelle_ligne[k] = ligne1[k]
                for k in ligne2:
                    nouvelle_ligne[k] = ligne2[k]
                resultat.append(nouvelle_ligne)
    return resultat

meteo = charger_csv('meteo.csv')
departements = charger_csv('departements.csv')
for ligne in fusionner(meteo, departements, 'ville'):
    print(f'{ligne["ville"]} ({ligne["département"]}) : {ligne["température"]} degrés')
```

```
Lyon (Rhône) : 24 degrés
Brest (Finistère) : 17 degrés
Lille (Nord) : 15 degrés
```

La table fusionnée ne contient que les villes présentes dans les **deux** tables : Nice n'a pas de ligne dans `departements.csv`, et Marseille n'en a pas dans `meteo.csv`.

Résumé des fonctions du module csv

Fonction	Rôle	Type renvoyé par ligne
`csv.reader()`	Lecture ligne par ligne	liste de chaînes
`csv.DictReader()`	Lecture avec en-tête	dictionnaire
`csv.writer()`	Écriture ligne par ligne	—
`csv.DictWriter()`	Écriture avec en-tête	—

Exercices

Exercice 1 On dispose d'un fichier `notes.csv` contenant :

```
nom,maths,physique,info
Alice,15,12,18
Bob,10,14,16
Charlie,8,11,13
```

Écrire un programme qui lit ce fichier et affiche pour chaque élève son nom et sa moyenne (arrondie à un chiffre après la virgule).

Exercice 2 À partir du même fichier `notes.csv`, écrire une fonction `meilleurs_en(matiere, seuil)` qui renvoie la liste des noms des élèves ayant une note supérieure ou égale à `seuil` dans la matière donnée.

Exemple d'appel attendu :

```
print(meilleurs_en('info', 16))  # doit afficher ['Alice', 'Bob']
```

Exercice 3 Écrire un programme qui crée un fichier `carres.csv` contenant deux colonnes : `nombre` et `carre`, pour les entiers de 1 à 20.

Exercice 4 On dispose d'un fichier `temperatures.csv` contenant :

```
ville,janvier,fevrier,mars,avril,mai,juin
Paris,5,6,10,13,17,21
Marseille,7,8,11,14,18,23
Lyon,3,5,10,13,17,21
Lille,3,4,8,11,15,18
```

Écrire un programme qui :

1. affiche la ville la plus chaude en juin ;
2. calcule la température moyenne de chaque ville sur les six mois ;
3. écrit les résultats dans un nouveau fichier `moyennes.csv` avec les colonnes `ville` et `moyenne`.

Exercice 5 On dispose d'un fichier `livres.csv` contenant :

```
titre,auteur,annee,pages
Les Misérables,Victor Hugo,1862,1900
Le Petit Prince,Antoine de Saint-Exupéry,1943,96
Germinal,Émile Zola,1885,592
L'Étranger,Albert Camus,1942,185
```

Écrire une fonction `livres_par_pages(nom_fichier)` qui charge le fichier, trie les livres par nombre de pages décroissant et affiche le résultat.

Indication : la fonction `sorted()` accepte un paramètre `key` qui permet de spécifier le critère de tri. Les valeurs lues dans un CSV sont toujours des chaînes : il faut les convertir en entier avec `int()`.