III. Manipulation de fichiers CSV

1) Lecture, importation et exportation de fichiers CSV

Comme vous l'avez deviné, en Python, nous allons lire les fichiers CSV grâce à des dictionnaires. De cette manière, nous allons pouvoir faire des recherches sur les contenus plutôt que sur des numéros d'indices sans aucun sens.

Toutefois, pour nous aider dans cette tâche, nous **choisissons** de répéter les en-têtes pour chaque élément. Ainsi :

Nom	Français	Science	Histoire
Erwann	16	12	15
Céline	14	16	13

Le tableau ci-contre se notera en Python :
Table = [
{'Nom': 'Erwann', 'Français': '16', 'Science': '12', 'Histoire': '15'},
{'Nom': 'Céline', 'Français': '14', 'Science': '16', 'Histoire': '13'}
]

Convention : les fichiers CSV lus en Python se présentent toujours sous la forme d'une **liste** de dictionnaires dont les clés sont les en-têtes du tableau et les valeurs sont les contenus.

Nous allons tout d'abord utiliser une bibliothèque pour importer des fichiers CSV. Cette bibliothèque s'appelle… **csv** !

❐ — À faire vous-même **1** —

- Pour ouvrir des fichiers en Python, on utilise la commande suivante :

```
donnéesDuFichier = open( nomDuFichier, modeDOuverture)
```

avec `nomDuFichier` qui est un "string"
et `modeDOuverture` qui vaut **'r'** si l'on veut lire le fichier et **'w'** si on veut écrire un nouveau fichier. *Pour plus d'informations : "Google open python3"*

- Créez un dossier appelé NSI_CSV et téléchargez le fichier exemple.csv disponible sur mon site internet : bouillotvincent.github.io .
- Dans Spyder, Geany ou VSCode, créez un nouveau fichier appelé csvReader.py et, à l'aide de l'instruction **open**, ouvrez le fichier exemple.csv.
- En ajoutant un **print** à votre code, affichez le contenu de exemple.csv.
- Que pensez-vous de votre résultat ?

On ne va pas se mentir,
`<_io.TextIOWrapper name='exemple.csv' mode='r' encoding='UTF-8'>`
n'est pas super lisible et pas vraiment ce que l'on attendait…

❐ — À faire vous-même **2** —

On va maintenant importer la bibliothèque **csv** pour pouvoir lire ce fichier que l'on vient d'ouvrir.

- Dans votre fichier csvReader.py, importez la bibliothèque csv.
- Recopiez la fonction importCSV(fichier, separateur) qui va nous permettre d'importer des fichiers CSV.

```
def importCSV(fichier : str, separateur = ";"):
    tCSV = csv.DictReader(open(fichier,'r'), delimiter = separateur)
```

- Complétez la fonction pour afficher le tableau importé à l'aide d'un **print**, puis testez cette fonction en appelant **table = importCSV('exemple.csv')**.
- Que constatez-vous ? _____

Caramba… Encore raté : `<csv.DictReader object at 0x10335ea90>`
Toujours pas très lisible…

Pourriez-vous brièvement expliquer l'origine de cette erreur ? Dis autrement : que fait DictReader ?

_____

_____

❐ — À faire vous-même **3** —

Finalement, pour lire le contenu de notre csv.DictReader object (sic!), nous allons simplement parcourir la table `"tCSV"` à l'aide d'une boucle for.

- Dans la fonction importCSV, ajoutez une boucle for sur une variable que l'on appellera ligne et qui va parcourir la table `"tCSV".` À chaque passage dans la boucle affichez à l'aide de **print** la valeur de la variable ligne.
- Est-ce que tout fonctionne comme on le souhaitait initialement (p10) ? Pourquoi ?
  _____

❐ — À faire vous-même **4** —

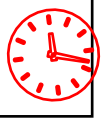
Afin de conserver un certain ordre, la bibliothèque CSV utiliser une structure Python hybride entre liste et dictionnaire qui est appellé un `OrderedDict`.

On peut transformer cette structure en dictionnaire Python simplement en appelant dict() sur celle-ci.

- ❖ Toujours dans la même fonction, afficher `dict(ligne)`.
- ❖ Est-ce que tout fonctionne comme on le souhaitait initialement (p10) ?

  _____
- ❖ Finalement, modifiez votre fonction afin remplir une **liste** que l'on va appeler tableau et qui contiendra tous les dictionnaires que vous avez créé ligne par ligne.

Si vous avez fini avant : Faites la même chose en 2 lignes : une pour importer tCSV et une pour renvoyer le résultat.



❐ — À faire vous-même **5** —

La fonction ci-dessous permet d'exporter une table au format CSV.

- ❖ Recopiez le code ci-dessous dans votre programme csvReader.py.

```
def exportCSV(tableau : list, fichier : str):
    header = tableau[0].keys()
    fichierCSV = csv.DictWriter(open(fichier, 'w'), fieldnames =
header)
    fichierCSV.writeheader()
    for ligne in tableau:
        fichierCSV.writerow(ligne)
    return None
```

- ❖ Identifiez ce que fait chaque ligne de ce programme et écrivez-le en commentaires dans votre programme.
- ❖ Pour vérifier votre code, exportez votre table dans le fichier que l'on nommera "exemple2.csv" , puis ouvrez ce fichier avec un tableur. Votre export a-t-il fonctionné ?

Conclusion :

- ❖ Sur un plan connaissances pures, on sait maintenant importer et exporter des fichiers CSV grâce à la libraire csv, ce qui va nous permettre de réaliser des opérations sur ces fichiers.

- Sur un second plan, vous remarquerez que l'on a beaucoup travaillé pour faire fonctionner la librairie comme on le souhaitait. On peut se demander si on n'aurait pas été plus rapide en créant directement notre propre lecteur et écriveur de fichiers CSV (voir exercice 4!)

2) Opérations sur les tables

❐ — À faire vous-même **6** —

On se propose de faire quelques tests dans le programme principal (PAS les fonctions) de csvReader.py .

Si vous voulez comprendre ce qu'il se passe dans la suite, je vous conseille très fortement **d'écrire** les résultats des questions suivantes sur le cours ou une feuille que vous garderez en face de vous :

- Qu'affiche **print(table[0])** ? _____
- À quoi vous attendez-vous si vous tapez **print(table[1])** ?

_____

- Qu'affiche **print(table[0]['Nom'])** ? _____
- Comment feriez-vous pour afficher la note de Céline en Science ?

_____

- Quel résultat renvoie **print(table[0:1]['Nom'])** ? _____
  Que pouvez déduire de cela ?

_____

Nous allons maintenant étudier comment filtrer des lignes et des colonnes puis nous verrons comment trier une table en fonction d'une colonne.

❐ — À faire vous-même **7** —

Proposez une fonction filtrerLigne qui prend en argument un tableau, un critère (Nom, Science …) ainsi qu'une valeur et renvoie une liste de dictionnaire filtrée. Seules les lignes correspondantes à votre critère d'égalité (par simplicité) doivent apparaitre.

```
def filtrerLigne(tableau : list, critere : str, valeur : str):
    … … … … … … … … … … … …
    return filtrerTableau
```

Si vous avez fini avant : Faites la même chose en 1 ligne !

Filtrer suivant une colonne s'appelle aussi une **projection**.

❐ — À faire vous-même **8** —

On suppose pour l'instant que l'on cherche à faire notre projection sur un seul critère (le 'Nom' par exemple).

- Compléter la fonction filtrerColonne ci-dessous. Celle-ci prend en argument un tableau et un critère (Nom, Science …) et renvoie un tableau filtré. Seules les colonnes correspondantes à votre critère doivent apparaitre.

```
def filtrerColonne(tableau : list, listeCriteres : str):
    filtrerTableau = []
    for dico in tableau:
        dicoFiltrer = {}
        for …… in …… :
            …… # ici
        filtrerTableau.append(dicoFiltrer)
    return filtrerTableau
```

- Testez votre code grâce à un print : print(filtrerColonne(table, 'Nom')) .

Conservez une seule colonne est assez inutile… On suppose à présent que l'utilisateur peut donner une **liste** de critères. `listeCriteres` sera donc de type list.

- Modifier le programme précédent afin de prendre en compte une sélection sur une liste de critères.
  Les modifications ne concerneront que la boucle interne indiquée par # ici

Si vous avez fini avant : Faites la même chose en 1 ligne !

Finalement, le tri d'une table suivant une colonne peut se faire grâce aux algorithmes de tri.

Dans le cadre de ce chapitre, nous allons utiliser l'instruction **sorted** propre à Python. **sorted** possède un argument très pratique appelé **key** qui permet de préciser selon quel critère une liste doit être triée (cela est un objet fonction de variables à trier). Un autre argument d'intérêt est **reverse** qui est un booléen permettant d'indiquer si on souhaite que l'ordre soit croissant (False) ou décroissant (True).

❐ — À faire vous-même **9** —

❖ Recopiez le code ci-dessous :

```
def triTable(tableau: list, critere: str, decroissant = False ):
    return sorted(tableau, key=lambda k: k[critere],
reverse=decroissant)
```

❖ Testez votre code en lançant triTable sur votre table et en classant par 'Science'. Cela fonctionne-t-il ?

On a l'impression que le tableau est correctement trié : sorted doit regarder ma liste et la trier suivant le critère ['Science']. C'est bon, on a tout compris.

Non, on n'a rien compris. C'est même le drame car un nouveau mot-clé est apparu : c'est le mot clé `lambda` .

"Lambda, c'est quoi ???"

Réponse : Lambda est appelé "fonction anonyme" (ou encore fonction poubelle).
Lorsque l'on a besoin d'une fonction pour une utilisation unique, à la volée, dans un code, il est souvent fastidieux de créer une fonction complète. C'est là que la lambda entre en scène.
Par exemple, pour calculer un carré d'un nombre d'une manière ponctuelle :

Version classique

```
def carré(x: list):
    return str(x**2)
print('Carré de 4 = '+ carré(4))
```

Version lambda tranquille

```
carré = lambda x: str(x**2)
print('Carré de 4 = '+ carré(4))
```

Version lambda accrochez-vous aux déambulateurs.

```
print('le carré de 4 est ' +(lambda x: str(x**2))(4))
```

En résumé :
C'est exactement la même chose qu'une fonction, seule la syntaxe change:

❖ lambda au lieu de def ;
❖ pas de nom de fonction ;
❖ pas de paranthèses ;
❖ pas de mot clé return.