Chapitre 6

Traitement de données en tables

Les données informatiques sont de plus en plus nombreuses. Une fois ces données à notre disposition, il faut pouvoir les **traiter** afin d'extraire une information précise ou bien établir une interprétation de ces dernières.

Une solution classique est la gestion de données en **table**, c'est à dire sous forme de tableau. Un **tableur** comme *Calc* ou *Excel* propose une interface graphique et des fonctions pour traiter des données sous cette forme, mais il est important de comprendre comment ces fonctions sont programmées, et comment faire pour aller encore plus loin dans l'exploitation de ces données.

6.1 Le format CSV

Considérons le tableau de données (très simple) suivant :

Nom	Prénom	Date de naissance
Dupont	Alice	22/02/1981
Dupond	Bob	08/01/1992
Durand	Charlie	14/05/1987

Les informations sur la première ligne s'appellent les **champs** : Nom - Prénom - Date de naissance.
Chaque champ décrit les informations contenues dans la colonne, qui sont toutes du même type.

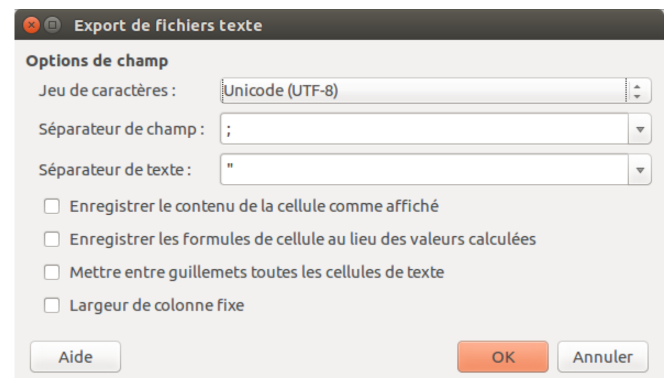
Pour enregistrer ces données dans un fichier texte, on peut utiliser le **format CSV**. Dans ce format, chaque ligne du tableau correspond à une ligne dans notre fichier. La première ligne contient les champs, et les données sont séparées par un caractère donné, comme le point-virgule. En CSV, notre tableau serait représenté de la manière suivante :

```
1 Nom;Prénom;Date de naissance
2 Dupont;Alice;22/02/1981
3 Dupond;Bob;08/01/1992
4 Durand;Charlie;14/05/1987
```

Exporter des données depuis un tableur

Depuis un tableur, il est très facile d'exporter des données au format CSV. Il suffit pour cela d'enregistrer notre fichier en précisant le format CSV (Enregistrer sous > Format CSV dans la liste déroulante).

	A	B	C
1	Nom	Prénom	Date de naissance
2	Dupont	Alice	22/02/81
3	Dupond	Bob	08/01/92
4	Durand	Charlie	14/05/87



Lors de l'export en CSV, le tableur nous demande de choisir le séparateur de champ, une virgule par défaut. On préférera le point-virgule, pour éviter les erreurs : en effet, la virgule est le séparateur décimal en français, et séparer nos colonnes avec des virgules peut générer des erreurs.

▷ ***Et si une colonne de mon tableau contient un point-virgule ?***

Dans ce cas, la donnée contenant le point-virgule sera entourée par le séparateur de texte (des guillemets par défaut).

Exercice 01 Créer le fichier CSV précédent (à la main ou en l'enregistrant depuis un tableur). On le nommera *data.csv*.

Importer des données depuis Python

En Python, vous savez déjà ouvrir un fichier et récupérer son contenu brut :

```
with open("data.csv", "r") as fichier:
    contenu = fichier.read()
```

▷ ***Le fichier* data.csv *doit se trouver dans le même dossier que votre programme Python !***

Cependant, la variable `contenu` est une chaîne de caractères ne permettant pas encore de traiter nos informations.

```
>>> contenu
'Nom;Prénom;Date de naissance\nDupont;Alice;22/02/1981\nDupond;Bob;08/01/1992\nDurand;Charlie;14/05/1987\n'
```

6.2 Traitement de données au format CSV

6.2.1 Importation d'une table

En Python, il serait intéressant de manipuler nos données sous forme de listes plutôt que sous forme de chaînes de caractères. Nous allons donc créer deux listes :

- une liste `champs` contenant les champs de notre tableau

```
>>> champs
['Nom', 'Prénom', 'Date de naissance']
```

- une liste `table` contenant toutes les lignes de données de notre tableau, elles-mêmes sous forme de listes !

```
>>> table
[['Dupont', 'Alice', '22/02/1981'], ['Dupond', 'Bob', '08/01/1992'], ['Durand', 'Charlie',
 ↪ '14/05/1987']]
```

Exercice 02 Compléter la fonction `decoderCSV(contenu)` afin qu'elle retourne les listes `champs` et `table` en fonction du `contenu` passé sous forme CSV.

```
def decoderCSV(contenu):
    lignes = contenu.split("\n") # On récupère une liste de lignes en 'coupant' au niveau des \n

    champs = ...
    table = []

    for ligne in ... :
        ...

    return champs, table
```

6.2.2 Manipulation des données

Dans cette partie, nous allons travailler à partir du fichier CSV suivant, contenant des informations sur la superficie et la population de quelques communes de Corse :

`http://nsi.dellasantina.corsica/documents/corse.csv`

Le but est de rajouter un champ *Densité de population* dans nos données, en calculant les densités correspondantes pour chaque ville. Notre code sera découpé en 5 parties. Vous pouvez le récupérer à l'adresse suivante :

`http://nsi.dellasantina.corsica/documents/traitement_tables.py`

```
1  # 1. Fonctions utilisées
2
3  # 2. Importation des données CSV
4
5  # 3. Ajout d'un champ "Densité de population" dans la liste champs
6
7  # 4. Calcul des densités de population et ajout dans la liste table
8
9  # 5. Enregistrement du fichier
```

Exercice 03 Compléter la partie #3 en ajoutant l'élément `"Densité de population"` dans la liste `champs`.

Exercice 04 Compléter la partie #4 en calculant la densité de population pour chaque ville, et en ajoutant cette valeur à la fin de chaque ligne de notre table de données. On utilisera une boucle `for`.

☞ *Attention aux types de variables utilisées !*

Exercice 05 La partie 5 utilise la fonction `encoderCSV(champs, table)` qui encode les listes `champs` et `table` au format CSV : les listes sont transformées en une unique chaîne de caractères (le point-virgule est utilisé pour séparer les différentes données), et cette chaîne de caractères est retournée par la fonction.

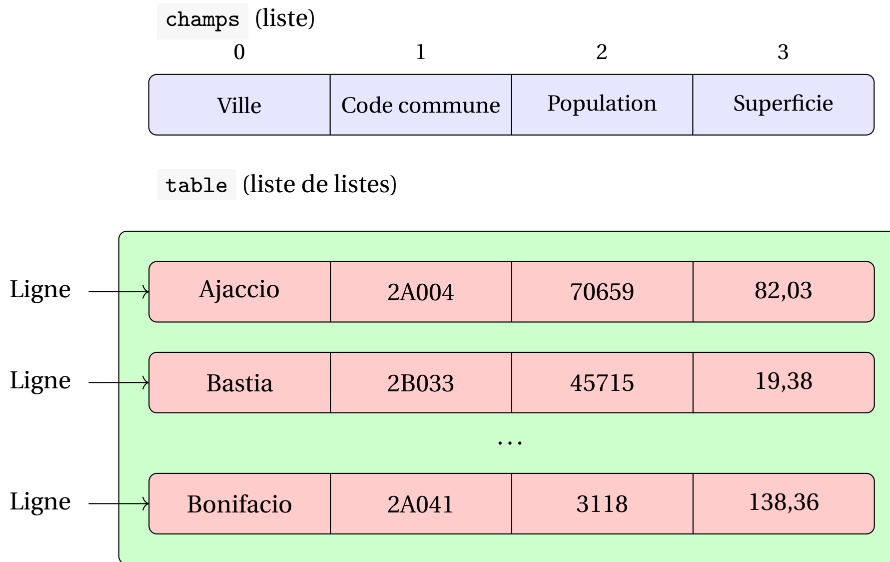
Écrire la fonction `encoderCSV`.

☞ *On pourra utiliser la méthode `join()` qui permet de transformer une liste en chaîne de caractères en* recollant *chaque élément avec un caractère donné. C'est l'inverse de la méthode `split`.Quelques exemples ci-dessous :*

```
>>> "#".join(['hello', 'world']
'hello#world'
>>> ".".join(['01', '02', '03', '04', '05']
'01.02.03.04.05'
>>> ",".join(['data1', 'data2', 'data3']
'data1,data2,data3'
```

6.2.3 Recherche dans une table

Notre fonction `decoderCSV` retourne deux variables `champs` et `table`, que l'on peut schématiser - dans le cas où l'on décode le fichier `corse.csv` - de la manière suivante :



Rechercher dans notre table, c'est **sélectionner** les lignes qui vérifient un certain **critère**. Ce critère dépend des données dont on dispose, et du type d'information qui nous intéresse. Supposons par exemple que l'on veuille récupérer seulement les lignes correspondantes à des villes de plus de 10 000 habitants.
C'est manifestement la colonne *Population* qui nous intéresse, dont l'indice est 2 dans notre exemple précédent.
Pour récupérer les lignes recherchées, on parcourt donc la variable `table` avec une boucle `for` et on récupère les lignes dont la valeur correspondant à l'indice 2 est supérieure à 10 000. Ces lignes sont stockées dans une liste `rep` (pour *réponse*).

Exercice 06 Compléter la fonction `recherche1(table)` qui retourne la liste `rep` contenant les lignes correspondantes aux villes de plus de 10 000 habitants.

```
1  def recherche1(table):
2      rep = []
3      for ligne in table:
4          if ....... :
5              rep.append(......)
6
7      return rep
```

Exercice 07 Écrire une fonction `recherche2(table)` qui retourne la liste des lignes correspondantes aux villes dont la superficie est supérieure à $100 km^2$.

Exercice 08 Écrire une fonction `recherche3(table)` qui retourne la liste des lignes correspondantes aux villes dont la densité de population est supérieure à $100 hab/km^2$.

Exercice 09 Écrire une fonction `recherche4(table)` qui retourne la liste des lignes correspondantes aux villes de Corse-du-Sud.

6.2.4 Tri d'une table

L'objectif du tri est d'obtenir les lignes ordonnées suivant les valeurs d'un champ. Ce champ peut-être un champ existant (population, superficie...) ou calculé (densité de population...), et l'ordre peut être croissant ou décroissant.
Pour trier notre table, nous n'allons pas programmer l'algoritme de tri « à la main » mais utiliser une fonction déjà définie, et déjà vue dans un précédent cours : `sorted`. Pour rappel, la fonction `sorted` prend en argument une liste et retourne une copie de cette liste dont les valeurs sont rangées dans l'ordre croissant. Ceci suppose néanmoins que l'ensemble des valeurs soit **ordonné**, c'est à dire qu'il existe un ordre permettant de trier ces valeurs.

```
>>> sorted([1, -2, 4, 2.9]) # Tri de valeurs numériques : OK
[-2, 1, 2.9, 4]
```

```
>>> sorted(["bonsoir", "salut", "bonjour", "euh"]) # Tri de chaînes de caractères : OK
['bonjour', 'bonsoir', 'euh', 'salut']
```

☞ *Dans le cas de chaînes de caractères, l'ordre utilisé est appelé ordre* ***lexicographique*** *(ordre du dictionnaire)*

Mais lorsque la liste contient des valeurs numériques et des chaînes de caractères, Python ne sait plus où donner de la tête :

```
>>> sorted([1, 2, "a", "b"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Et pour cause, il ne sait pas ordonner des variables de type `str` (chaînes de caractères) et `int` (entier) (vous non plus d'ailleurs...). C'est encore pire si on lui demande de trier des listes de listes (comme notre variable `table`) : comment peut-il affirmer qu'une certaine liste est *supérieure* à une autre? C'est impossible! À moins d'**évaluer** ces listes...

Reprenons notre table de données. Pour trier les différentes lignes la composant (ici des objets de type `list`), il faudrait spécifier par rapport à quelle colonne on souhaite effectuer le tri. Supposons que l'on veuille ordonner nos lignes par population croissante : il suffit alors d'expliquer à Python que chaque liste est équivalente à un certain nombre (ici le nombre d'habitants) que Python pourra trier. Cette association *liste / nombre* est réalisée par une **fonction d'évaluation**, dont le rôle est d'associer une valeur ordonnable à une liste.

Listes (non ordonnables)					**Valeurs ordonnables**
Ajaccio	2A004	70659	82,03	*Évaluation* →	70659
Bastia	2B033	45715	19,38	*Évaluation* →	45715
		...			
Bonifacio	2A041	3118	138,36	*Évaluation* →	3118

Grâce à notre fonction d'évaluation, les listes sont maintenant représentées par des nombres, que Python sait ordonner!

La fonction d'évaluation

D'ailleurs, quelle est cette fonction d'évaluation ? Dans notre cas, on souhaite récupérer, pour une ligne donnée, le nombre d'habitants, qui correspond à l'indice 2. Notre fonction d'évaluation s'écrit donc simplement :

```
def evaluation(liste):
    return liste[2]
```

Utilisation de `sorted` avec la fonction d'évaluation

La fonction `sorted` peut prendre plus d'un argument, notamment pour lui renseigner une fonction d'évaluation. Il est cependant nécessaire de nommer ce paramètre, `key` pour notre fonction d'évaluation.

```
def evaluation(liste):
    return liste[2]

tableOrdonnee = sorted(table, key = evaluation)
```

On récupère alors dans `tableOrdonnee` la liste de listes `table`, triée dans l'ordre croissant selon l'indice 2 (ici le nombre d'habitants) en spécifiant l'indice `key` égal à notre fonction d'évaluation.

▷ ***Et si je veux trier dans l'ordre décroissant ?***

On utilise un autre paramètre de la fonction `sorted` : `reverse`.

```
tableOrdonnee = sorted(table, key = evaluation, reverse = True)
```

Utilisation de `sorted` avec une fonction lambda

Notre fonction d'évaluation peut se définir « à la volée », directement lors de l'appel à la fonction `sorted`, grâce à une **fonction lambda**. Les fonctions lambda sont des fonctions définies en une seule ligne, à l'aide du mot clé `lambda`. Ce sont généralement des fonctions simples, comme notre fonction d'évaluation. La syntaxe est la suivante :

```
nomDeLaFonction = lambda parametres : valeurDeRetour
```

Dans notre cas, on peut alors écrire :

```
evaluation = lambda liste : liste[2]
```

On définit ainsi la fonction `evaluation` différemment, mais son utilisation est inchangée. L'avantage de définir une fonction de cette manière, c'est qu'il n'est pas nécessaire de « stocker » la fonction dans une variable, on peut la passer directement en paramètre à la fonction `sorted`. Magique ?

```
tableOrdonnee = sorted(table, key = lambda liste : liste[2])
```

L'utilisation du nom `liste` en paramètre de la fonction lambda n'est pas obligatoire, et il n'est pas rare de voir une fonction lambda utilisée avec une variable `x` comme ci-dessous :

```
tableOrdonnee = sorted(table, key = lambda x : x[2])
```

Résumons...

Voici notre programme, pour obtenir une table triée par population croissante (la fonction `decoderCSV` n'est pas écrite) :

```
1   # On récupère le contenu du fichier CSV
2
3   with open("corse.csv", "r") as fichier:
4     contenu = fichier.read()
5
6   # On récupère les listes champs et table avec la fonction decoderCSV
7
8   champs, table = decoderCSV(contenu)
9
10  # On trie notre table par population croissante
11
12  tableOrdonnee = sorted(table, key = lambda x : x[2])
```

Exercice 10 Créer une variable `tableOrdonnee2` contenant la table triée par superficie croissante.

Exercice 11 Créer une variable `tableOrdonnee3` contenant la table triée par nom de ville, dans l'ordre décroissant.

Exercice 12 Créer une variable `tableOrdonnee4` contenant la table triée par densité de population décroissante.