1 – Données en tables

1 – Un peu d'histoire

- 1956: Invention du disque dur, qui permet d'enregistrer de grnades quantités d'information.
- 1960: Le système commercial de base de données Sabre permet de gérer des réservations aériennes.
- 1964: Première utilisation du terme *database*. Le terme databes (base de données en anglais) est utilisé pour la première fois par l'armée américaine qui rencontre des difficultés pour partager efficacement un même contenu entre plusieurs utilisateurs.
- 1970: Les langages de programmation permettent d'organiser l'information dans des tableaux à 2 dimensions, reliés entre eux par des clés.
- 1970: Invention des SGBDR. Edgar Franck Codd, informaticien britannique, invente le modèle relationnel des systèmes de gestion de base de données relationnelles. Dans une BDR, l'information est organisée dans des tableaux à 2 dimensions appelés **relation** ou **table**. Un langage comme SQL (Structured Query Language) permet de communiquer avec une BDR.
- 1972: Interopérabilité entre logiciels grâce au format CSV. Il permet d'enregistrer des données tabulées dans un fichier. Chaque ligne du fichier correspond à un enregistrement et possède un ou plusieurs champs, séparés par des virgules. Le format CSV a été utilisé dès 1972 pour échanger des données entre 2 systèmes distincts, c'est l'**interopérabilité**.
- 1979: Le logiciel *Visicalc* est le premier logiciel tableur destiné à un ordinateur individuel (l'Apple II).
- 1979 à nos jours: Les tableurs, des outils simplificateurs. De Visicalc à la toute dernière version de tableurs comme Excel ou Calc, les tableurs ont apporté au grand public une méthode simplifiée pour représenter l'information sous la forme de tableaux appelés tables, munies de **descripteurs**.

2 – Introduction

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 données pour des dizaines de milliers (voire plus) d'articles en vente, de clients, de commandes, un hopital doit pouvoir accéder efficacement à tous les détails de traitements de ses patients, etc.

Il est très pertinent de croiser les informations contenues dans différentes tables, selon une opération appelée **fusion** ou encore **jointure**. Le langage SQL, qui sera abordé en classe de terminale, permet ainsi de formuler une requête afin d'interroger une BDR.

Mais si les logiciels de traitement de base de données sont des programmes hautement spécialisés pour effectuer ce genre de tâches le plus efficacement possible, il est facile de mettre en œuvre les opérations de base dans un langage de programmation comme Python. Nous allons en illustrer quelque unes.

Le saviez-vous ?

1) Un fichier au format CSV (Comma Separated Values -> valeurs séparées par des virgules) n'est rien d'autre qu'une suite de lignes où chaque ligne est séparée en différents champs, par des virgules.

Par exemple, voici une ligne d'un fichier station.csv:
4, 43.02750, 2.981667, reveo, Place de l'Octroie 11130 Sigean, 2, 1, Non, #0000FF, 47.2

2) Le système de réservation Sabre (Semi-Automatic Business Research Environment) développé par la société IBM est la première base de données commerciale du monde. La société American Airlines pouvait ainsi effectuer près de 42 000 réservations chaque seconde dès 1960.

Le format CSV

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 réparés par une virgule (d'où le nom).

En pratique, on peut spécifier le caractère utilisé pour séparer les différents champs et on utilise fréquemment un point-virgule, une tabulation ou deux points pour cela.

Notons enfin que la première ligne d'un tel fichier est souvent utilisée pour indiquer le nom des différents champs. Dans ce cas, le premier enregistrement apparaîssant en deuxième ligne du fichier.

La première ligne est composée de descripteurs. Sur les lignes suivantes, les données correspondant aux descripteur sont rangées en lignes. Chaque descripteur définit un ensemble de champs.

Dans la suite, nous allons utiliser un fichier nommé donnees-carte-synthese-tricolore.csv qui contient quelques données sous-jacentes au tableau de bord COVID-19 publié sur gouvernement.fr. Il décrit la carte de synthèse qui sert de référence pour les mesures différenciées qui sont appliquées depuis le mardi 2 juin, selon les départements.

En voici les 6 premières lignes:

```
date,code_departement,nom_departement,nom_region,indicateur_synthese
2020-04-30,1,Ain,Auvergne-Rhône-Alpes,orange
2020-04-30,2,Aisne,Hauts-de-France,rouge
2020-04-30,3,Allier,Auvergne-Rhône-Alpes,orange
2020-04-30,4,Alpes-de-Haute-Provence,Provence-Alpes-Côte d'Azur,orange
2020-04-30,5,Hautes-Alpes,Provence-Alpes-Côte d'Azur,orange
```

Les champs sont clairement séparés par des virgules.

3 – Importation des données

Une façon de charger un fichier csv en Python est d'utiliser la bibliothèque du même nom. Voici une portion de code permettant de charger le fichier donnees-carte-synthese-tricolore.csv avec des virgules comme délimitations:

```
import csv


def depuis_csv(fichier_a_lire: str):

    departement = []

    with open(fichier_a_lire, newline='') as csvfile:
        lecteur = csv.reader(csvfile, delimiter=',')
        for row in lecteur:
            departement.append(row)

    return departement


fichier = 'donnees-carte-synthese-tricolore.csv'
liste_departement = depuis_csv(fichier)
```

Dans ce cas, les résultats sont stockés sous forme d'un tableau de tableau. On a par exemple:

```
print(liste_departement[0])
->
['date', 'code_departement', 'nom_departement', 'nom_region', 'indicateur_synthese']
```

```
print(liste_departement[1])
->
['2020-04-30', '1', 'Ain', 'Auvergne-Rhône-Alpes', 'orange']
```

On s'aperçoit que lors de la lecture du fichier, la première ligne n'a pas été utilisée comme descriptions des champs et le premier enregistrement, concernant 'Ain', apparaît dans liste_departement[1]. Plus gênant, le lien entre les valeurs du tableau liste_departement[1] et le nom des enregistrements, contenus dans liste_departement[0], n'est pas direct.

Pour y remédier, nous allons utiliser *DictReader* qui retourne un dictionnaire pour chaque enregistrement, la première ligne étant utilisée pour nommer les différents champs 1 .

```
def depuis_csv(fichier_a_lire: str):

    departement = []

    with open(fichier_a_lire, newline='') as csvfile:
        lecteur = csv.DictReader(csvfile, delimiter=',')
        for colonne in lecteur:
            departement.append(dict(colonne))

    return departement
```

Cette fois, on obtient un tableau de p-uplets représentés sous forme de dictionnaire:

```
print(liste_departement[0])
->
{'date': '2020-04-30', 'code_departement': '1', 'nom_departement': 'Ain', 'nom_region': 'Auvergne-Rhône-Alpes', 'indicateur_synthese': 'orange'}
```

4 - Exploitation des données

Nous allons donner deux types d'utilisation simples des données que l'on vient de charger : tout d'abord, l'interrogation des données pour récupérer telle ou telle information, puis le tri des données.

Interrogation

On peut traduire en Python des question simples. Par exemple, quels sont les départements en rouge ?

```
print([dep['nom_departement'] for dep in liste_departement if dep['indicateur_synthese'] == 'rouge'])
->
['Aisne', 'Ardennes', 'Aube', 'Cher', "Côte-d'Or", 'Doubs', ...]
```

ou encore:

```
for elem in liste_departement:
    if elem['indicateur_synthese'] == 'rouge':
        print(elem['nom_departement'])
->
Aisne
Ardennes
Aube
Cher
Côte-d'Or
Doubs
...
```

Demandons maintenant quelles sont les régions des départements qui sont en rouge. On peut exécuter la ligne suivante:

```
print([dep_rouge['nom_region'] for dep_rouge in liste_departement if dep_rouge['indicateur_synthese'] == 'rouge'])
```

On obtient une liste de longueur 249… mais avec beaucoup de répétitions. Pour les supprimer, on peut la transformer en un ensemble (*set* en anglais). On obtient 8 régions:

```
print(set([dep_rouge['nom_region'] for dep_rouge in liste_departement if dep_rouge['indicateur_synthese'] == 'rouge']))
```

Tri

Pour exploiter les données, il peut être intéressant de les trier. Une utilisation possible est l'obtention du classement des entrées selon tel ou tel critère. Une autre utilisation vient du fait que, comme présenté dans la partie algorithmique du programme, la recherche dichotomique dans un tableau trié est bien plus efficace que la recherche séquentielle dans un tableau quelconque.

Prennons pour exemple, un fichier nommé pays.csv qui contient quelques données sur les différents pays du monde. En voici les premières lignes :

```
iso;name;area;population;continent;currency_code;currency_name;capital
AD;Andorra;468.0;84000;EU;EUR;Euro;6
AE;United Arab Emirates;82880.0;4975593;AS;AED;Dirham;21
AF;Afghanistan;647500.0;29121286;AS;AFN;Afghani;81
AG;Antigua and Barbuda;443.0;86754;NA;XCD;Dollar;119
AI;Anguilla;102.0;13254;NA;XCD;Dollar;126
AL;Albania;28748.0;2986952;EU;ALL;Lek;157
```

La signification des différents champs est transparente, à part le dernier champ, nommé capital et dont les valeurs sont des numéros d'identifiants de villes que l'on trouvera dans un autre fichier nommé cities.csv. Nous verront comment gérer deux fichiers un peu plus tard.

Tri selon un unique critère

On ne peut pas directement trier le tableau pays… car cela ne veut rien dire. Il faut indiquer selon quels critères on veut effectuer ce tri. Pour cela, on appelle la fonction sorted ou la méthode sort avec l'argument supplémentaire key qui est une fonction renvoyant la valeur utilisée pour le tri[2]

[2]. Rappel : la méthode sort trie la liste en place, alors que la fonction sorted renvoie une nouvelle liste correspondant la liste trié, la liste initiale étant laissée intacte.

Par exemple, si l'on veut trier les pays par leur superficie, on doit spécifier la clé 'Area'. Pour cela, on définit une fonction appropriée:

```
def clé_superficie(p):
    return p['area']
```

Ainsi, pour classer les pays par superficie décroissante, on effectue:

```
pays.sort(key=clé_superficie, reverse=True)
```

Mais un petit problème demeure. Si on récupère les noms des 5 premiers pays ainsi classés, le résultat est étonnant:

```
[(p['name'], p['area']) for p in pays[:5]]

[('Canada', '9984670'),
('South Korea', '98480'),
('United States', '9629091'),
('Netherlands Antilles', '960'),
('China', '9596960')]
```

On ne s'attend pas à trouver la Corée du Sud parmi eux. La raison est que lors de l'import, tous les champs sont considérés comme des chaînes de caractères, et le tri utilise l'ordre du dictionnaire. Ainsi, de même que « aa » arrive avant « b », « 10 » arrive avant « 2 ». Cela apparaît ici en regardant les superficies qui commencent par 998, puis par 984, par 962, etc. Pour remédier à cela, on modifie la fonction de clé:

```
def clé_superficie(p):
        return float(p['area'])
```

On a alors le résultat espéré:

```
[(p['Country'], p['Area']) for p in sorted(pays, key=clé_superficie, reverse=True)[:5]]

[('Russia', '17100000.0'),
('Canada', '9984670.0'),
('United States', '9629091.0'),
('China', '9596960.0'),
('Brazil', '8511965.0')]
```

Tri selon plusieurs critères

Supposons maintenant que l'on veut trier les pays selon deux critères : tout d'abord le continent, puis le nom du pays. On peut faire cela en définissant une fonction de clé qui renvoie une paire (continent, pays):

```
def clé_combinée(p):
        return (p['continent'], p['name'])
```

Ainsi,

```
[(p['continent'], p['name']) for p in sorted(pays, key=clé_combinée)]

[('AF', 'Algeria'),
('AF', 'Angola'),
...,
('AF', 'Zambia'),
('AF', 'Zimbabwe'),
('AS', 'Afghanistan'),
('AS', 'Armenia'),
...,
('SA', 'Uruguay'),
('SA', 'Venezuela')]
```

Cependant, dans ce tri, les deux critères ont été utilisés pour un ordre croissant. Supposons maintenant que l'on veuille trier les pays par continent et, pour chaque continent, avoir les pays par population décroissante. La méthode précédente n'est pas applicable, car on a utilisé une unique clé (composée de deux éléments) pour un tri croissant. À la place, nous allons procéder en deux étapes:

- 1. trier tous les pays par populations décroissantes ;
- 2. trier ensuite le tableau obtenu par continents croissants.

Ainsi:

```
def clé_population(p):
        return int(p['population'])

pays.sort(key=clé_population, reverse=True)
pays.sort(key=clé_continent)

[(p['name'], p['continent'], p['population']) for p in pays]

[('Nigeria', 'AF', '154000000'),
('Ethiopia', 'AF', '88013491'),
...,
('Seychelles', 'AF', '88340'),
('Saint Helena', 'AF', '7460'),
('China', 'AS', '1330044000'),
('India', 'AS', '1173108018'),
...,
('French Guiana', 'SA', '195506'),
('Falkland Islands', 'SA', '2638')]
```

Pour que cela soit possible, la fonction de tri de Python vérifie une propriété très importante : la stabilité. Cela signifie que lors d'un tri, si plusieurs enregistrement ont la même clé, l'ordre initial des enregistrements est conservé. Ainsi, si on a trié les pays par ordre décroissant de population puis par continent, les pays d'un même continent seront regroupés en conservant l'ordre précédent, ici la population décroissante.

Conclusion

Nous l'avons vu, il est assez facile d'écrire en Python des commandes simples pour exploiter un ensemble de données. Cependant, une utilisation plus poussée va vite donner lieu à des programmes fastidieux. Dans la suite, nous présenterons la bibliothèque pandas qui permet une gestion plus efficace de ce genre de traîtement.