éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

MANIPULATION DE TABLES AVEC LA BIBLIOTHÈQUE PANDAS

1. Introduction

Nous avons vu dans le document « Manipulations de tables » comment faire en Python du traîtement de données, à partir notamment de fichiers `csv`. Mais, bien que les différents programmes présentés soient tous relativement simples, l'enchaînement des étapes de traîtements vont vite conduire à des constructions fastidieux et peu lisibles d'instructions.
Pour remédier à ce problème, nous allons utiliser la bibliothèque `pandas` qui permet d'exprimer de façon simple, lisible et concise de genre de transformations de données. Il faut cependant noter qu'il n'y a rien de *magique* dans cette bibliothèque, et que les commandes présentées dans le document précédent continuent d'illustrer les traitements de départ. Seulement, dans la bibliothèque `pandas`, une représentation adaptée des données permet de rendre le tout plus efficace.

2. Prise en main

2.1. Lecture de fichiers

De façon classique en Python, nous allons commencer par charger le module.

```
1 import pandas
```

La lecture d'un fichier `csv` se fait alors aisément grâce à la commande [1] :

```
1 pays = pandas.read_csv("countries.csv", delimiter=";", keep_default_na=False)
```

où, en particulier, on spécifie explicitement le caractère utilisé pour délimiter les champs du fichier, ici un point-virgule. Chargeons aussi une liste des villes du monde de plus de 5000 mondes (et des capitales) :

```
1 villes = pandas.read_csv("cities.csv", delimiter=";")
```

Le fichier `countries.csv` est le même que dans le document précédent, explorons un peu l'autre. Pour cela, la bibliothèque `pandas` propose quelques commandes utiles :

1. L'option `keep_default_na=False` est nécessaire à cause de la gestion des données manquantes. Une absence est parfois précisée spécifiquement en écrivant NA plutôt que de ne rien mettre. Ainsi, à la base, la lecture de NA est interprété comme une donnée manquante. On est obligé de désactiver ce traîtement de base pour pouvoir utiliser NA comme tel, comme code de l'Amérique du Nord.

- `villes.head()` affiche les premières entrées de la table ;
- `villes.sample(7)` affiche 7 enregistrements de la table pris au hasard ;
- `villes.columns` retourne la liste des champs ;
- `villes.dtypes` affiche la liste des champs avec, à chaque fois, le type de données correspondant.

Ainsi, on a :

```
>>> villes.dtypes
id             object
name           object
latitude      float64
longitude     float64
country        object
population    float64
dtype: object
```

On remarque en particulier que `pandas` a reconnu que les champs latitude, longitude et population correspondent à des données numériques, et le traîtent comme tels.
On peut aussi avoir des données statistiques (bien sûr, seules celles concernant la population soient pertinentes) :

```
>>> villes.describe()
                 id      latitude     longitude    population
count  49699.000000  49699.000000  49699.000000  4.969900e+04
mean   24849.000000     29.863320      7.474766  5.854476e+04
std    14347.009851     23.819937     70.413201  3.284362e+05
min        0.000000    -54.810840   -178.158330  0.000000e+00
25%    12424.500000     18.007330    -56.501805  7.813000e+03
50%    24849.000000     38.492900      9.985580  1.445100e+04
75%    37273.500000     47.407980     45.341595  3.442050e+04
max    49698.000000     78.223340    179.364510  2.231547e+07
```

Enfin, on peut facilement ne conserver que les champs qui nous intéressent. Par exemple, si l'on ne veut que les noms des villes et leurs coordonnées, on utilise :

```
villes[['name', 'latitude', 'longitude']]
```

Dataframes et series

Les tables lues dans les fichiers `csv` sont stockés par `pandas` sous forme de *dataframes*. On peut les voir comme un tableau de p-uplets nommés. Par exemple, l'enregistrement numéro 10 (obtenu grâce à la méthode `loc`) s'obtient en exécutant :

```
>>> villes.loc[10]
id                    10
name          Zayed City
latitude         23.6542
longitude        53.7052
country               AE
population         63482
Name: 10, dtype: object
```

et son nom s'obtient comme pour un dictionnaire :

```
>>> villes.loc[10]['name']
'Zayed City'
```

Une série est ce que l'on obtient à partir d'un dataframe en ne sélectionnant qu'un seul champ.

```
>>> villes['name']
0                              Sant Julià de Lòria
1                                           Ordino
2                                       les Escaldes
                        ...
49697                                       Epworth
49698                                    Chitungwiza
Name: name, Length: 49699, dtype: object
>>> type(villes['name'])
pandas.core.series.Series
```

Lors de la sélection d'un unique champ, `pandas` permet d'utiliser une syntaxe légère en n'écrivant que `villes.name` plutôt que `villes['name']`.

Il convient, pour finir, de différentier :

- ▷ la *serie* `villes['name']` (ou `ville.name`, donc) et
- ▷ le *dataframe* à un seul champ `villes[['name']]`.

2.2. Interrogations simples

Reprenons les interrogations présentées dans le document précédent, et exprimons-les à l'aide de `pandas`.

Noms des pays où l'on paye en euros

On sélectionne la bonne valeur de `currency_code` ainsi :
`pays[pays.currency_code == 'EUR']`.
Ensuite, on ne garde que les noms des pays ainsi obtenus, pour obtenir :

```
pays[pays.currency_code == 'EUR'].name
```

Codes des monnaies appelées *Dollar*

On peut écrire :

```
pays[pays.currency_name == 'Dollar'].currency_code.unique()
```

La méthode `unique` s'applique à une série et non un dataframe.

2.3. Tris

Les méthodes `nlargerst` et `nsmallest` permettent de déterminer les plus grands et plus petits éléments selon un critère donné. Ainsi, pour obtenir les pays les plus grands en superficie et ceux les moins peuplés, on peut écrire :

```
pays.nlargest(10, 'area')
pays.nsmallest(10, 'population')
```

Le tri d'un dataframe s'effectue à l'aide de la méthode `sort_values`, comme par exemple :

```
villes.sort_values(by='population')
```

On peut trier selon plusieurs critères, en spécifiant éventuellement les monotonies. Ainsi, pour classer par continent puis par superficie décroissante (avec une sélection pertinente de champs) :

```
pays.sort_values(by=['continent', 'area'], ascending=[True, False])[
    ['continent', 'name', 'area']]
```

3. Manipulation de données

Après ce survol des méthodes de base pour extraire et ordonner les données contenue dans une table, nous allons pour finir voir quelques méthodes de manipulation de tables.

3.1. Création d'un nouveau champ

Il est très facile de créer de nouveaux champs à partir d'anciens. Par exemple, pour calculer la densité de chaque pays, il suffit d'exécuter :

```
pays['density'] = pays.population / pays.area
```

Les séries peuvent être utilisées avec le module `numpy`. Ainsi, on peut réaliser une carte des villes utilisant la projection de Mercator en effectuant :

```
villes['projection_y'] = numpy.arcsinh(numpy.tan(villes.latitude * numpy.pi / 180))
villes.plot.scatter(x='longitude', y='projection_y')
```

3.2. Fusion de tables

Dans la table des pays, la capitale est indiquée par un numéro… qui correspond au champ **`id`** de la table des villes. Pour récupérer le nom de la capitale de chaque pays, nous allons fusionner les tables en effectuant une *jointure*. Ainsi, nous allons faire correspondre le champ `capital` de `pays` et le champ **`id`** de `villes`. Cela se fait à l'aide de la fonction `merge` :

```
pandas.merge(pays, villes, left_on='capital', right_on='id')
```

Cependant, en procédant ainsi, il va y avoir un conflit entre les champs des deux tables. Cela apparaît en listant les champs de la table obtenue :

```
>>> pandas.merge(pays, villes, left_on='capital', right_on='id').columns
Index(['iso', 'name_x', 'area', 'population_x', 'continent', 'currency_code',
       'currency_name', 'capital', 'id', 'name_y', 'latitude', 'longitude',
       'country', 'population_y'],
      dtype='object')
```

On voit que des tables initiales contiennent toutes les deux des champs `name` et `population`, d'où les suffixes `_x` et `_y` pour marquer la référence à la première table initiale ou à la seconde.
Pour rendre cela plus lisible, nous allons :

1. ne garder que les colonnes de `ville` qui nous intéressent, ici l'identifiant et le nom ;
2. renommer ces colonnes pour éviter les collisions avec les champs de `pays` :

```
villes[['id', 'name']].rename(columns={'id': 'capital', 'name': 'capital_name'})
```

Et c'est cette nouvelle table que nous allons fusionner avec la table `pays` (dont nous ne garderons pas toutes les colonnes non plus) :

```
pays_et_capitales = pandas.merge(
    pays[['iso', 'name', 'capital', 'continent']],
    villes[['id', 'name']].rename(
        columns={'id': 'capital', 'name': 'capital_name'}),
    on='capital')
```

La liste des pays d'Océanie et leurs capitales s'obtient alors facilement :

```
pays_et_capitales[pays_et_capitales.continent == 'OC']
```

4. Conclusion

La bibliothèque pandas que nous avons présentée dans ce document est un outil intéressant pour s'initier à la manipulation de données. En particulier, le rôle central qu'y jouent les *dataframes* permet de manipuler les enregistrements quasiment comme s'il s'agissait de p-uplet nommés.
Cette approche permet aussi de préparer la transition avec le programme de Terminale et le chapitre sur les bases de données. En effet, bien que ce thème apporte des problématiques spécifiques, et bien que les syntaxes diffèrent grandement entre des instructions pandas et une requête SQL, il existe de nombreux points communs entre les deux approches concernant la façon dont les données sont représentées et peuvent être exploitées et manipulées.