



1 Introduction

La bibliothèque **pandas** est très utilisée pour tout ce qui touche au traitement des données. Nous avons vu qu'avec la bibliothèque **csv**, l'enchainement des étapes de traitement peut conduire à des constructions d'instructions fastidieuses et peu lisibles.

Pandas permet d'exprimer de façon plus simple, lisible et concise ce genre de transformations de données. Notez cependant qu'il n'y a rien de magique dans cette bibliothèque et que les commandes vues dans la leçon précédente continuent d'illustrer les traitements de départ.

Mais avec la bibliothèque **pandas**, une représentation adaptée des données permet de rendre le tout plus efficace.

2 Prise en main

→ On commence par importer le module **pandas**.

```
>>> import pandas
```

→ La lecture d'un fichier **csv** se fait ensuite grâce à la commande :

```
>>> villes = pandas.read_csv("villes_point_virgule.csv", delimiter=";", keep_default_na=False)
```

Les deux options sont facultatives mais vont permettre ici deux choses particulièrement intéressantes :

- Une présentation des données plus lisibles (voir image suivante)
- Une gestion des données manquantes (il y en a pas mal dans notre fichier **csv**). Testez la différence !

Toutes les informations sont sur : https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

Affichons la variable **villes** que nous venons de créer :

```
Shell
>>> import pandas
>>> villes = pandas.read_csv("villes_point_virgule.csv", delimiter=";")
>>> villes
   dep nom      cp ...      lat  alt_min  alt_max
0    1 Ozan  1190 ...  46,3833  170.0    205.0
1    1 Cormoranche-sur-Saône  1290 ...  46,2333  168.0    211.0
2    1 Plagne  1130 ...  46,1833  560.0    922.0
3    1 Tossiat  1250 ...  46,1333  244.0    501.0
4    1 Pouillat  1250 ...  46,3333  333.0    770.0
... ...
36695  976 Sada  97640 ... -12,8486     NaN     NaN
36696  976 Tsingoni  97680 ... -12,7897     NaN     NaN
36697  971 Saint-Barthélemy  97133 ...  17,9167     NaN     NaN
36698  971 Saint-Martin  97150 ... -63,0829     NaN     NaN
36699  975 Saint-Pierre-et-Miquelon  97500 ...  1,71819     NaN     NaN
[36700 rows x 12 columns]
```

→ Les données (un extrait ici, étant donnée la masse concernée) sont présentées sous la forme d'un tableau.

Notez la présence d'un indice (valeur de l'index) ajouté en première colonne, afin de rendre la lecture plus aisée.

Remarque : toutes les informations relatives à l'utilisation du module **pandas** se trouvent sur :

https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

Exercice 1 : Tester les commandes suivantes et compléter le tableau.

Commande à saisir	Opération réalisée et/ou observations
>>> villes.head()	affiche les 5 premières lignes
>>> villes.sample(9)	affiche 9 lignes au hasard
>>> villes.columns	retourne la liste des colonnes
>>> villes.dtypes	affiche la liste des derniers types et le type correspondant
>>> villes.info()	fournit un récap d'info sur le DataFrame
>>> villes.shape	donne les dimensions du DataFrame
>>> villes.shape[0]*villes.shape[1]	calcule le nombre de cellules du DataFrame
>>> villes.describe()	affichage de données statistiques

→ On peut facilement ne conserver que les champs qui nous intéressent.

Exemple : si l'on ne veut que les codes postaux, les noms et la population en 2012 des villes :

```
>>> villes[['cp', 'nom', 'nb_hab_2012']]
   cp           nom  nb_hab_2012
0  1190        Ozan      500
1  1290  Cormoranche-sur-Saône     1000
2  1130        Plagne      100
3  1250       Tossiat     1400
4  1250      Pouillat      100
...
36695  97640          Sada    10195
36696  97680        Tsingoni    10454
36697  97133  Saint-Barthélemy     8938
36698  97150       Saint-Martin    36979
36699  97500  Saint-Pierre-et-Miquelon     6080
[36700 rows x 3 columns]
```

→ Dataframes et séries :

Une table lue dans un fichier **csv** est stockée par **pandas** sous la forme d'un **dataframe**, que l'on peut voir comme un tableau de p-uplets nommés.

Pour accéder à un sous-ensemble du **dataframe**, par exemple à l'enregistrement numéro 10, on utilise la méthode **loc** :

```
Shell
>>> villes.loc[10]
dep          1
nom    Chaveyriat
cp        1660
nb_hab_2010    927
nb_hab_1999    810
nb_hab_2012    900
dens         54
surf        16,87
long      5,06667
lat       46,1833
alt_min       188
alt_max       260
Name: 10, dtype: object
>>>
```

Le nom de la ville s'obtient comme pour un dictionnaire, la clé étant le descripteur :

```
>>> villes.loc[10]['nom']
'Chaveyriat'
```

Ou bien avec cette autre syntaxe plus simple :

```
>>> villes.loc[10, 'nom']
'Chaveyriat'
```

Il est possible de récupérer toutes les valeurs d'une même colonne en remplaçant l'indice de la ligne par le caractère ' : ' et de récupérer toutes les colonnes d'une même ligne en remplaçant le nom du champ par le caractère ' : '.

Exemples :

```
Shell
>>> villes.loc[:, 'nom']
0 Ozan
1 Cormoranche-sur-Saône
2 Plagne
3 Tossiat
4 Pouillat
...
36695 Sada
36696 Tsingoni
36697 Saint-Barthélemy
36698 Saint-Martin
36699 Saint-Pierre-et-Miquelon
Name: nom, Length: 36700, dtype: object
>>>
```

```
Shell x
>>> villes.loc[100, :]
dep 1
nom Château-Gaillard
cp 1500
nb_hab_2010 1818
nb_hab_1999 1370
nb_hab_2012 1700
dens 113
surf 16,06
long 5,3
lat 45,9667
alt_min 222
alt_max 253
Name: 100, dtype: object
>>>
```

→ Il est aussi possible de ne récupérer QUE certaines lignes et certaines colonnes.

Exemple :

```
Shell
>>> villes.loc[[320, 436], ['nom', 'cp']]
      nom    cp
320 Cerdon  1450
436 Happencourt  2480
>>>
```

→ Une série est ce que l'on obtient à partir d'un **dataframe** en ne sélectionnant qu'un seul champ
Deux syntaxes sont possibles mais attention à ne pas confondre la **série villes['nom']** ou **villes.nom**
et le **dataframe** à un seul champ **villes[['nom']]** :

La série, avec ses deux syntaxes possibles pour la commande :

```
Shell
>>> villes['nom']
0 Ozan
1 Cormoranche-sur-Saône
2 Plagne
3 Tossiat
4 Pouillat
...
36695 Sada
36696 Tsingoni
36697 Saint-Barthélemy
36698 Saint-Martin
36699 Saint-Pierre-et-Miquelon
Name: nom, Length: 36700, dtype: object
>>>
```

```
Shell
>>> villes.nom
0 Ozan
1 Cormoranche-sur-Saône
2 Plagne
3 Tossiat
4 Pouillat
...
36695 Sada
36696 Tsingoni
36697 Saint-Barthélemy
36698 Saint-Martin
36699 Saint-Pierre-et-Miquelon
Name: nom, Length: 36700, dtype: object
>>>
```

Le dataframe à un seul champ :

```
Shell
>>> villes[['nom']]
      nom
0 Ozan
1 Cormoranche-sur-Saône
2 Plagne
3 Tossiat
4 Pouillat
...
36695 ...
36696 Sada
36697 Tsingoni
36698 Saint-Barthélemy
36699 Saint-Martin
36699 Saint-Pierre-et-Miquelon
[36700 rows x 1 columns]
>>>
```

```
>>> type(villes['nom'])
<class 'pandas.core.series.Series'>
>>> type(villes.nom)
<class 'pandas.core.series.Series'>
```

```
>>> type(villes[['nom']])
<class 'pandas.core.frame.DataFrame'>
```

3 Exploitation des données

3.1 - Interrogation (extraction de données selon certains critères).

Nous allons reprendre quelques exemples de la leçon précédente, mais en utilisant les commandes de **pandas**.

Exercice 2 et Exercice 3 :

On veut afficher la liste des plus grandes villes françaises (exemple de critère : population supérieure à 150 000 habitants).

Étudions les effets successifs des commandes suivantes :

Commande à saisir	Opération réalisée et/ou observations
<pre>>>> villes[villes.nb_hab_2012>150000]</pre>	dataframe des villes > 150000
<pre>>>> villes[villes.nb_hab_2012>150000].nom</pre>	série de noms des villes > 150000
<pre>>>> villes[villes.nb_hab_2012>150000].nom.unique()</pre>	donne une liste de ces noms, l'obtient à une seule ligne
<i>Et les équivalents (ou pas !) avec la méthode loc :</i>	
<pre>>>> villes.loc[villes['nb_hab_2012']>150000]</pre>	dataframe
<pre>>>> villes.loc[villes['nb_hab_2012']>150000,['nom']]</pre>	dataframe
<pre>>>> villes.loc[villes['nb_hab_2012']>150000,['nom']].unique()</pre>	erreur

Exercice 4 :

Écrire et tester la commande qui permettra d'obtenir ce qui est demandé ci-dessous :

Donner la liste des villes dont l'altitude minimale est négative.

>>> villes[villes.alt_min < 0].nom.unique()

Donner la liste des villes dont l'altitude maximale est supérieure à 4000 mètres.

>>> villes[villes.alt_max > 4000].nom.unique()

Donner la liste des villes dont la densité est supérieure à 15000 habitants/km².

>>> villes[villes.dens > 15000].nom.unique()

Donner la liste des villes dont la population était de 8000 habitants en 2012.

>>> villes[villes.nb_hab_2012 == 8000].nom.unique()

Quelle est la 100^{ème} ville du tableau dont la population était de 1000 habitants en 2012 ?

>>> villes.loc[villes['nb_hab_2012'] == 1000].nom.unique()

Il s'agit de la ville qui s'appelle *Saint-Étienne*

2997

Important : il est possible de combiner plusieurs facteurs de sélection en utilisant les opérateurs "&" (et) et "|" (ou)

Exercice 5 : déterminer le rôle des commandes ci-dessous

```
>>> villes[(villes.nb_hab_2012>150000)&(villes.alt_min<5)].nom  
2049      Nice  
4439    Marseille  
12678   Bordeaux  
16755   Nantes  
31165   Le Havre  
33676   Toulon  
Name: nom, dtype: object  
>>> type(villes[(villes.nb_hab_2012>150000)&(villes.alt_min<5)].nom)  
<class 'pandas.core.series.Series'>
```

```
>>> villes.loc[(villes['nb_hab_2012']>150000)&(villes['alt_min']<5),['nom']]  
          nom  
2049      Nice  
4439    Marseille  
12678   Bordeaux  
16755   Nantes  
31165   Le Havre  
33676   Toulon  
>>> type(villes.loc[(villes['nb_hab_2012']>150000)&(villes['alt_min']<5),['nom']])  
<class 'pandas.core.frame.DataFrame'>
```

À laquelle de ces deux commandes peut-on en plus appliquer la méthode *unique()* ?

La première car c'est une série

Exercice 6 : Écrire et tester la commande pour chacun des cas suivants.

Trouver le nom et le département de la seule ville de France avec une densité supérieure à 50 hab./km² et une altitude minimale supérieure à 1500 m.

>>> villes[(villes.dens>50)&(villes.alt_min>1500)].
, nom, dens, dept

Déterminer la population moyenne en 2012 des communes situées à une altitude minimale supérieure à 1500 m.

Aide : la méthode *mean()* permet de calculer la moyenne sur l'ensemble des lignes extraites.

>>> villes[villes.alt_min>1500].nb_hab_2012.mean()

Extraire la liste des communes de la région Hauts-de-France dont le point culminant excède 235 m et n'afficher que le département, le nom et l'altitude maximale de chaque commune.

>>> voir "pandas O. py"

Déterminer l'altitude minimale moyenne des communes françaises.

>>> villes.alt_min.mean() ou >>> villes.loc[:, alt_min].mean()

3.2 – Tri

→ Les méthodes ***nlargest*** et ***nsmallest*** permettent de déterminer les plus grands et les plus petits éléments selon un critère donné.

Exercice 7 : Tester les commandes suivantes et compléter le tableau.

Commande à saisir	Opération réalisée et/ou observations
<code>>>> villes.nlargest(10,'nb_hab_2012')</code>	donne les 10 premières villes avec le plus d'hab
<code>>>> villes.nsmallest(10,'alt_min')</code>	donne les 10 dernières villes avec l'altitude minimum
<code>>>> villes.nlargest(50,'alt_min').nsmallest(3,'dens').nom</code>	50 premières villes les plus basses et parmi elles les 3 moins peuplées

→ Le tri d'un dataframe s'effectue à l'aide de la méthode ***sort_values()***

Exercice 8 : Tester les commandes suivantes et compléter le tableau.

Commande à saisir	Opération réalisée et/ou observations
<code>>>> villes.sort_values(by='nb_hab_2012')</code>	
<code>>>> villes.sort_values(by='nb_hab_2012',ascending=False)</code>	
<code>>>> villes.sort_values(by=['nom','dens'], ascending=[True,False])[['nom','cp','dens']]</code>	
<code>>>> villes.sort_values(by='alt_min', ascending=False).nom.unique()[:10]</code>	Trier le dataframe pour trouver la commune avec l'altitude minimum la plus importante de France

4 Manipulation de données

4.1 – Crédation d'un nouveau champ

Il est très facile de créer de nouveaux champs à partir de champs existants.

Par exemple, pour calculer le dénivelé maximum de chaque commune, il suffit d'exécuter :

```
>>> villes['deniv'] = villes.alt_max - villes.alt_min
```

4.2 – Fusion de tables

Pour comprendre le principe de la fusion de tables, nous allons partir de deux nouveaux fichiers CSV :

- Villes_HDF1.csv qui ne contient que le nom des villes, leur code postal et leur département (02-59-60-62-80)
 - Villes_HDF2.csv qui contient tous les champs sauf les codes postaux.

On commence par lire les deux tables dans les variables **villes1** et **villes2**. Pensez à spécifier le délimiteur ';'.

```
>>> villes1= pandas.read_csv ("villes - HDPI.csv", delimiter =";")
```

```
>>> villes2= / / - / / "villes-HDF2 cov": / / / /
```

Puis exéutez les commandes

```
>>> villes1.columns  
>>> villes2.columns
```

Quels noms portent les champs qui contiennent les noms des villes dans villes1 et villes2 ? an nom el name

Pour fusionner les deux tables villes1 et villes2 et récupérer l'ensemble des données dans une seule et même table, que nous nommerons villes3, nous allons effectuer une **jointure**, c'est-à-dire faire correspondre deux champs :

Le champ 'nom' de villes1

et le champ 'name' de villes2.

```
>>> villes3=pandas.merge(villes1,villes2, left_on='nom',right_on='name')
```

Puis exécutez la commande

```
>>> villes3.columns
```

En procédant comme nous l'avons fait, il y a eu un conflit entre les champs des deux tables. On voit que les tables initiales contiennent toutes les deux un champ 'dep', d'où les suffixes _x et _y pour marquer la référence à la première ou à la seconde table initiale.

Pour rendre cela plus lisible, nous allons dans une table villes4:

- ne garder que les colonnes de villes2 qui nous intéressent (le nom, le département et la population de 2012)
 - renommer la colonne de département pour éviter la collision avec le champ de même nom de villes1

```
>>> villes4=villes2[['name','dep','nb_hab_2012']].rename(columns={'dep':'depart'})  
>>> villes4.columns
```

```
Shell
>>> villes4=villes2[['name','dep','nb_hab_2012']].rename(columns={'dep':'depart'})
>>> villes4.columns
Index(['name', 'depart', 'nb_hab_2012'], dtype='object')
>>>
```

Et c'est cette nouvelle table que nous allons fusionner avec la table villes1 :

```
Shell >>> villes5=pandas.merge(villes1,villes4, left_on='nom',right_on='name')
>>> villes5.columns
Index(['dep', 'nom', 'cp', 'name', 'depart', 'nb_hab_2012'], dtype='object')
>>>
```

→ Cette fois, plus de conflit car les deux champs de données identiques portent un nom différent.

On ne garde que les colonnes qui nous intéressent et on fixe leur ordre :

```
Shell
>>> villes6=villes5[['nom','dep','cp','nb_hab_2012']]
>>> villes6.columns
Index(['nom', 'dep', 'cp', 'nb_hab_2012'], dtype='object')
>>>
```

On constate que les doublons de noms de ville génèrent un ajout de lignes dans le fichier.

Pour remédier à cela, au moment de la fusion lorsque l'on génère villes5, on peut utiliser deux champs plutôt qu'un.

```
Shell
>>> villes1 = pandas.read_csv("F:/NSI/villes_HDF1.csv", delimiter=";")
>>> villes2 = pandas.read_csv("F:/NSI/villes_HDF2.csv", delimiter=";")
>>> villes4=villes2[['name','dep','nb_hab_2012']].rename(columns={'dep':'depart'})
>>> villes5=pandas.merge(villes1,villes4,left_on=['nom','dep'],right_on=['name','depart'])
>>> villes6=villes5[['nom','dep','cp','nb_hab_2012']]
>>> villes6.sort_values(by='nom')
   nom    dep      cp  nb_hab_2012
1097  Abancourt  59  59265        400
1531  Abancourt  60  60220        700
3304  Abbeville  80  80132      24100
1468 Abbeville-Saint-Lucien  60  60480        500
349   Abbécourt   2   2300        500
...
   ...    ...
602   Evergnicourt   2   2190        500
2821  Évin-Malmaison  62  62141      4500
1568   Évricourt    60  60310        200
3053  Euf-en-Ternois  62  62130        300
809    Eulilly      2   2160        300
[3836 rows x 4 columns]
>>>
```

Mission accomplie !

Si l'on souhaite enregistrer le dataframe obtenu dans un fichier CSV, on peut utiliser les commandes d'ouverture en écriture (et de fermeture) utilisés pour les fichiers texte et utiliser la commande

```
villes6.to_csv(index=False,sep=',')
```

pour obtenir toutes les lignes du fichier CSV.

5 Conclusion

La bibliothèque **pandas** 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 comme s'il s'agissait de p-uplets 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 entre les instructions **pandas** et celles utilisées pour 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.