La librairie Pandas : première approche

Table des matières

I. Contexte

Contexte

Pré-requis

- Connaissance de la syntaxe Python
- Connaissance de la programmation orientée objet en Python
- Mathématiques niveau lycée
- De la bonne humeur et de la motivation !

La visualisation de données est un ensemble de méthodes graphiques. Afin de produire une représentation graphique de données, il est nécessaire de manipuler, traiter les données. En Python, on utilise pour cela la librairie Pandas.

Vous devez vous demander pourquoi utiliser encore une autre librairie pour des tableaux.

La réponse est que Pandas permet de manipuler de gros volumes de données de différents formats de façon assez intuitive. Dans ce cours nous allons donc étudier cette librairie en détail et notamment les opérations autour de son objet principal, le `Dataframe`.

II. Toujours plus de tableaux, de Numpy à Pandas

Objectif de ce cours

- Comprendre la différence entre Pandas et Numpy

Pandas est une librairie Python qui permet de manipuler facilement des données. Notamment des données sous forme de tableaux à double entrées avec des étiquettes de variables (des colonnes) et d'individus (des lignes).

Ces tableaux de données sont appelés `DataFrames`. On peut facilement lire et écrire ces `DataFrames` à partir de fichiers comme un fichier Excel par exemple. Il est également assez simple de tracer des graphiques à partir de ces objets `DataFrames` grâce aux librairies Matplotlib et Seaborn.

La première chose à faire est d'importer la librairie.

```
1 import pandas as pd
```

On importe la librairie pandas en utilisant le mot clef `as` qui va nous permettre d'alléger la syntaxe afin d'appeler les fonctions de la librairie de la façon suivante :

```
1 my_function = pd.pandas_function()
```

plutôt que :

```
1 my_function = pandas.pandas_function()
```

Ce n'est sans doute pas grand chose, mais c'est du temps de gagné dans votre journée si vous utilisez souvent les fonctions de Pandas.

```
1 import pandas as pd
2 import numpy as np
3 print('libs loaded')
```

Avertissement : /Users/mac/.pyenv/versions/3.8.0/lib/python3.8/site-packages/pandas/compat/__init__.py:85: UserWarning: Could not import the lzma module. Your installed Python is incomplete. Attempting to use lzma compression will result in a RuntimeError.

warnings.warn(msg)

```
1 libs loaded
```

Avertissement : /Users/mac/.pyenv/versions/3.8.0/lib/python3.8/site-packages/pandas/compat/__init__.py:85: UserWarning: Could not import the lzma module. Your installed Python is incomplete. Attempting to use lzma compression will result in a RuntimeError.

warnings.warn(msg)

III. Objets Pandas : les Series

Objectifs de ce cours

- Comprendre les `Series`

Les `Series` sont un des objets principaux de la librairie Pandas, une `Series` à `n` valeurs peut être vue comme un vecteur de dimension `(1,n)`, ou encore un tableau à une dimension.

Pour construire une `Series`, on utilise la fonction `Series()` de Pandas en lui passant en argument une liste tel que :

```
1 pd.Series([0.25, 0.5, 0.75, 1.0])
```

```
1 l=[0.25, 0.5, 0.75, 1.0]
2 data = pd.Series(l); data
```

Résultat obtenu (Out) :

```
0 0.25
1 0.50
2 0.75
3 1.00
dtype: float64
```

Lors de l'affichage de la `Series` on voit `dtype: float64`.

C'est le type des éléments qui composent la `Series`. On peut le voir notamment avec l'attribut `dtypes`.

```
1 #types
2 data.dtypes
```

Out : dtype('float64')

On peut remarquer aussi une liste de valeurs sur le côté gauche. Ce sont les **index**, c'est-à-dire l'emplacement des valeurs. Ce qui pourrait correspondre aux numéros de lignes et de colonnes sur un logiciel de traitement de données type Excel, Google Sheets, Numbers, etc.

On utilise les commandes `index` pour accéder à l'index et `values` pour les valeurs de la `Series`.

```
1 data.index
```

Out : `RangeIndex(start=0, stop=4, step=1)`

```
1 data.values
```

Out : `array([0.25, 0.5 , 0.75, 1. ])`

On peut aussi stocker différents types de données dans les `Series`, tel que les `string`.

```
1 data_str = pd.Series(['il', 'fait', 'pas trop', "moche aujourd'hui !"]); data_str
```

Out :

```
0 il
1 fait
2 pas trop
3 moche aujourd'hui !
```

```
dtype: object
```

Il est possible d'effectuer des opérations mathématiques élémentaires sur les `Series` tel que l'addition `+, -, *, /`.

Attention toutefois aux `NaN` et à bien utiliser ces opérations sur des `Series` de même type !

```
s1 = pd.Series([1, 2, 3], ['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], ['a', 'd', 'c'])
plus = s1 + s2
moins = s1 - s2
fois = s1 * s2
div = s1 / s2;div
```

Out :

```
a 0.25
b NaN
c 0.50
d NaN
dtype: float64
```

Attention

Attention aussi à la dimension / `shape` de vos données.

IV. Objets Pandas : les DataFrames

Objectifs de ce cours

- Comprendre les DataFrames

Le `Dataframe` se comporte comme un dictionnaire dont les clefs sont les noms des colonnes et les valeurs sont des `Series`.

On peut les créer à partir d'un array Numpy ou bien d'un dictionnaire.

```
ar = np.array([[1.1, 2, 3.3, 4], [2.7, 10, 5.4, 7], [5.3, 9, 1.5, 15]])
df = pd.DataFrame(ar, index = ['a1', 'a2', 'a3'], columns = ['A', 'B', 'C', 'D']); df
```

Résultat obtenu (Out) :

	A	**B**	**C**	**D**
a1	1.1	2.0	3.3	4.0
a2	2.7	10.0	5.4	7.0
a3	5.3	9.0	1.5	15.0

On peut voir que certaines valeurs sont **en gras**.

Ce sont l'index et les colonnes un peu comme les `Series`, on peut y accéder via les attributs `index` et `columns`. Les deux objets sont des objets `pandas.core.indexes.base.Index`.

```
df.index
```

Out : `Index(['a1', 'a2', 'a3'], dtype='object')`

```
df.columns
```

Out : `Index(['A', 'B', 'C', 'D'], dtype='object')`

Comme pour les `Series`, on peut le réindexer pour changer l'ordre des lignes et/ou des colonnes, ou n'en récupérer que certaines.

```
df.reindex(columns = ['C', 'B', 'A'], index = ['a2', 'a3'])
```

Out :

	C	B	A
a2	5.4	10.0	2.7
a3	1.5	9.0	5.3

Concept de dimension d'un DataFrame

La dimension c'est la taille de votre tableau.

C'est un concept important qui peut s'avérer problématique quand on cherche à effectuer des opérations sur des tableaux de différentes *shape*.

Fondamental

Liste des principales commandes :

- `df.shape` : renvoie la dimension du dataframe sous forme (nombre de lignes, nombre de colonnes),
- On peut aussi faire `len(df)` pour avoir le nombre de lignes, ou également `len(df.index)`,
- on peut aussi faire `len(df.columns)` pour avoir le nombre de colonnes,
- `df.memory_usage()` : donne une série avec la place occupée par chaque colonne, `sum(df.memory_usage())` donne la mémoire totale occupée.

```
df.shape
```

Out : `(3, 4)`

On prendra le `DataFrame df` suivant pour la suite des exemples.

```
dico = {'A': [1.1, 2.7, 5.3],
        'B': [2, 10, 2],
        'C': [3.3, 5.4, 1.5],
        'D': [4, 7, 15]}
df = pd.DataFrame(dico, index = ['a1', 'a2', 'a3']); df
```

Out :

	A	B	C	D
a1	1.1	2	3.3	4
a2	2.7	10	5.4	7
a3	5.3	2	1.5	15

Manipuler un DataFrame

Pour renvoyer la `Series` correspondant à la ligne `l` d'index `index` on utilise la fonction `loc['index']`. Par exemple pour l'index `a2` :

```
df.loc['a2']
```

Out :

```
A 2.7
B 10.0
C 5.4
D 7.0
Name: a2, dtype: float64
```

On peut aussi renvoyer un `dataframe` avec un sous-ensemble de lignes et de colonnes :

```
1 df.loc[['a2', 'a3'], ['A', 'C']]
```

Out :

	A	**C**
a2	2.7	5.4
a3	5.3	1.5

On peut aussi utiliser `iloc[:]` qui fonctionne sur le même principe que les tableaux Numpy avec le caractère `:` pour afficher certaines lignes.

```
1 df.iloc[0:2]
```

Out :

	A	**B**	**C**	**D**
a1	1.1	2	3.3	4
a2	2.7	10	5.4	7

On peut bien sûr combiner les fonctions `loc` et `iloc` afin d'effectuer une sélection plus fine.

```
1 df.loc[:,['A', 'D']].iloc[0:2]
```

Out :

	A	**D**
a1	1.1	4
a2	2.7	7

Complément

Plus d'opérations avec `at` et `iat` :

- `df.loc[:,['A', 'C']]` : toutes les lignes et seulement les colonnes A et C,
- `df.loc['a2', 'C']` : accès à la valeur de la ligne a2 et de la colonne C : 5.4,
- `df.at['a2', 'C']` : autre façon recommandée d'accéder à la valeur de la ligne a2 et de la colonne C : 5.4,
- On peut aussi faire une affectation pour changer la valeur : `df.at['a2', 'C'] = 7`,
- On peut aussi utiliser des indices numériques : `df.at[0, 1]` (ou même un mélange des deux).

Pour accéder à un sous-ensemble du Dataframe avec les numéros des lignes et colonnes :

- `df.iloc[1]` : renvoie la deuxième ligne,
- `df.iloc[1:3,[0, 2]]` : renvoie le Dataframe avec les lignes 1 à 3 exclues, et les colonnes numéros 0 et 2,

- `df.iloc[:,2:4]` : renvoie toutes les lignes et les colonnes 2 à 4 exclues,
- `df.iloc[1,2]` : renvoie la valeur à la ligne 2 et la colonne 3,
- `df.iat[1,2]` : renvoie la valeur à la ligne 2 et la colonne 3, mais c'est la façon recommandée d'accéder aux valeurs,
- On peut aussi faire une affectation pour changer la valeur : `df.iat[1, 2] = 7`.

Itération sur un Dataframe

Quand on boucle sur un `Dataframe`, on boucle sur les noms des colonnes et non sur les valeurs :

```
dico = {'A': [1.1, 2.7, 5.3],
        'B': [2, 10, 2],
        'C': [3.3, 5.4, 1.5],
        'D': [4, 7, 15]}
df = pd.DataFrame(dico, index = ['a1', 'a2', 'a3'])
for x in df:
  print(x) # affiche le nom de la colonne
```

```
A
B
C
D
```

Si on veut accéder à une colonne, par exemple la colonne `A`, il faut utiliser les `[]`.

```
df['A']
```

Out :

```
a1 1.1
a2 2.7
a3 5.3
Name: A, dtype: float64
```

Afin d'itérer sur un `DataFrame` on peut utiliser la méthode `iterrow()` qui nous renvoie l'index et la ligne, il faut donc être précis quant à l'utilisation de cette dernière.

```
for index,row in df.iterrows():
    print(index, row)
```

```
a1 A 1.1
B 2.0
C 3.3
D 4.0
Name: a1, dtype: float64
a2 A 2.7
B 10.0
C 5.4
D 7.0
Name: a2, dtype: float64
a3 A 5.3
B 2.0
```

```
C 1.5
D 15.0
Name: a3, dtype: float64
```

On peut donc manipuler les `Series` et les `index` de notre `DataFrame`, ici les variables `row` et `index` tel que :

```
for index, row in df.iterrows():
    print(index,':', row['A'],row['D'])
```

```
a1 : 1.1 4.0
a2 : 2.7 7.0
a3 : 5.3 15.0
```

Cette méthode est assez pratique pour rechercher des valeurs particulières.

```
value=2.7
for index, row in df.iterrows():
    print(row['A'],row['D'])
    if row['A']==value:
        print('do something with this value')
        break
```

```
1.1 4.0
2.7 7.0
do something with this value
```

Informations d'un DataFrame

L'avantage en passant par les `DataFrame` c'est qu'ils possèdent plein de méthodes pratiques comme renvoyer un tableau donnant des statistiques descriptives sur les valeurs numériques de notre tableau.

Si on veut avoir des informations sur les variables catégorielles il faut ajouter l'option `include = 'all'` tel que :

```
df.describe(include = 'all')
```

Par ailleurs, si on veut des informations sur les variables qui composent le `DataFrame` on utilise la méthode `info()`.

```
df.describe()
```

Out :

	A	B	C	D
count	3.000000	3.000000	3.000000	3.000000
mean	3.033333	4.666667	3.400000	8.666667
std	2.119748	4.618802	1.951922	5.686241
min	1.100000	2.000000	1.500000	4.000000
25%	1.900000	2.000000	2.400000	5.500000
50%	2.700000	2.000000	3.300000	7.000000
75%	4.000000	6.000000	4.350000	11.000000
max	5.300000	10.000000	5.400000	15.000000

On peut aussi faire appel aux fonctions `numpy` pour explorer plus en détail notre `DataFrame`.

```
df.A.mean()
```

Out : `3.033333333333333`

```
df.A.sort_values()
```

Out :

```
a1 1.1
a2 2.7
a3 5.3
Name: A, dtype: float64
```

```
df.sort_values(by='A')
```

Out :

	A	B	C	D
a1	1.1	2	3.3	4
a2	2.7	10	5.4	7
a3	5.3	2	1.5	15

```
df.B.value_counts()
```

Out :

```
2 2
10 1
Name: B, dtype: int64
```

Filtrer et regrouper vos DataFrames

Filtrer avec `[]`

On peut utiliser les opérateurs mathématiques de base afin de filtrer un `DataFrame`. Cela consiste à renvoyer un "sous `DataFrame`" avec seulement les lignes et colonnes où la condition est vérifiée.

```
df[df['A'] > 2]
```

Out :

	A	B	C	D
a2	2.7	10	5.4	7
a3	5.3	2	1.5	15

Grouper avec `groupby()`

En science de données on a souvent besoin de grouper ces données par catégories, là est tout l'objet de la fonction `groupby(by='column')` qui prend comme paramètre la colonne souhaitée du `DataFrame` sur lequel on applique le `groupby`.

```
prix = pd.read_csv('../data/prices.csv'); prix.head()
```

Out :

	Unnamed : 0	ID	day	created	available	local_currency	local_price	min_nights
0	3502203	13170204	2018-12-29	2018-09-26 19:30:07.000+0000	True	EUR	23	30
1	307902	27132220	2018-11-11	2018-09-26 19:40:24.000+0000	False	EUR	177	2
2	3861426	28426717	2018-11-07	2018-09-26 19:38:04.000+0000	False	EUR	73	3
3	1333025	20583341	2018-10-26	2018-09-26 15:09:46.000+0000	False	EUR	175	2
4	3804978	3800777	2019-01-02	2018-09-27 06:06:25.000+0000	False	EUR	68	3

```
1 len(prix.ID.unique()), len(prix)
```

Out : (11749, 300000)

```
1 prix_id = prix.groupby(by='ID')
```

```
1 prix_id
```

Out : <pandas.core.groupby.generic.DataFrameGroupBy object at 0x1192a7dc0>

```
1 prix_id.head().describe()
```

Out :

	Unnamed: 0	ID	local_price	min_nights
count	5.872400e+04	5.872400e+04	58724.000000	58724.000000
mean	2.371990e+06	1.535633e+07	188.829150	7.979123
std	1.374572e+06	9.215156e+06	238.120669	24.372943

min	3.700000e+01	5.396000e+03	9.000000	1.000000
25%	1.179962e+06	6.472251e+06	76.000000	2.000000
50%	2.371228e+06	1.662908e+07	135.000000	3.000000
75%	3.561800e+06	2.369410e+07	230.000000	4.000000
max	4.748363e+06	2.885198e+07	10003.000000	365.000000

Les objets `pandas.core.groupby.generic.DataFrameGroupBy` sont des objets particuliers, on utilise la fonction `get_group()` pour accéder à un certain groupe parmi les groupes filtrés sous forme de `DataFrame`. On peut donc appliquer les méthodes que l'on connaît.

```
1 prix_id.get_group(27188781)
```

Out :

	Unnamed: 0	ID	day	created	available	local_currency	local_price	min_nights
177624	327995	27188781	2018-09-11	2018-09-26 19:37:07.000+0000	False	EUR	75	2
224946	2118181	27188781	2018-08-28	2018-09-26 19:37:07.000+0000	True	EUR	72	2

```
1 prix_id.get_group(27188781).local_price.mean()
```

Out : `73.5`