Pandas : introduction

La bibliothèque pandas offre des structures de données et des fonctions pour rendre l'utilisation de données rapide, facile et expressive. C'est un des ingrédients critiques qui font de Python un environnement d'analyse de données à la fois puissant et productif.

Par exemple, voici un extrait de données structurées :

drawing

Pandas combine les fonctions, de calcul haute performante, de NumPy et la souplesse des feuilles de calcul et des bases de données relationnelles (comme SQL). Elle apporte une fonctionnalité d'indexation évoluée qui facilite les opérations de transformation et de sélection de sous-ensembles de données.

In [ ]:
```python
# comme n'importe quelle librarire, il faut commencer par la charger
# à l'aide de la commande import

import pandas

# maintenat que c'est fait on peut utiliser son contenu
# par exemple vérifier la version installée
pandas.__version__
```

In [ ]:
```python
# et si on lui donne un nom pour faciliter les appels
import pandas as pd
pd.__version__
```

Les structures de données de pandas

Dans cette partie nous allons voir les structures de données principales de pandas et comment les utiliser. Tout d'abord nous devons importer les bibliothèques pandas et numpy

In [ ]:
```python
# important : se rappeler que pandas vient compléter numpy
import numpy as np
import pandas as pd

%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

L'objet Series

La structure Series est un tableau à une seule dimension qui permet de stocker tout type de données (entiers, chaînes de caractères, floats, objets Python, etc.). Contrairement aux listes classiques, une $Series$ possède en plus un index qui contient une étiquette pour chaque élément. La méthode basique pour créer une Series s'utilise comme suit:

In [ ]:
```python
data = ["foo", "foofoo", "foofoofoo"]
index = ["1er element", "2eme element", "3eme element"]

s = pd.Series(data, index=index)
print(s)
```

In [ ]:
```python
print("-"*42)

## On peut aussi utiliser des entiers pour l'index :

data = ["foo", "foofoo", "foofoofoo"]
index = [10, 20, 30]

print('Choisir un index d\'entiers :')
s = pd.Series(data, index=index)
print(s)
```

In [ ]:
```python
print("-*"*20)
print('ou par défaut si on ne passe pas d\'index :')
s = pd.Series(data)
print(s)
```

Notons qu'ici data peut être une liste mais pas seulement, on peut aussi utiliser

- un dictionnaire
- un ndarray (tableau numpy)
- un scalaire
- range

In [ ]:
```python
data=range(120,100,-2)
print(data)

s = pd.Series(data)
print(s)
```

Création d'une Series à partir d'un dictionnaire

Les Series peuvent être instanciées à partir d'un dictionnaire

In [ ]:
```python
data = {'Ah' : 101, 'Oh' : 202, 'Ih' : 303}
pd.Series(data)
```

Note:

- Quand data est un dictionnaire, et qu'un index n'est pas passé en paramètre, l'index correspondra aux clés du dictionnaire.
- Si on passe un index, les valeurs du dictionnaire seront récupérés à partir des étiquettes de l'index. Exemple :

In [ ]:
```python
data = {'a': 0., 'b': 1., 'c': 2.}
pd.Series(data)
```

In [ ]:
```python
data = {'a': 0., 'b': 1., 'c': 2.}
pd.Series(data, index=['b', 'c', 'd', 'a'])
```

Note: NaN (not a number) est le marqueur utilisé par pandas pour indiquer que des données sont manquantes.

Création d'une Series à partir d'un ndarray

Si data est un ndarray, **index** doit être de la même taille que **data**. Si aucun index n'est passé en paramètre lors de la création de la Series, un index par défaut sera créé avec les valeurs [0, ..., len(data) -1].

In [ ]:
```python
s = pd.Series(
    data=np.random.randn(5),
    index=['e', 'b', 'c', 'd', 'a']
)
print(s)
```

In [ ]:
```python
s.items
```

In [ ]:
```python
s.index
```

In [ ]:
```python
s = pd.Series(data=np.random.randn(5))
s
```

Création d'une Series à partir d'un scalaire

Si data est un scalaire, un index doit être passé en paramètre. La valeur sera répétée pour correspondre à la taille de l'index :

In [ ]:
```python
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

Similitudes avec les ndarray

Il y a des similitudes entre les Series et les ndarray, par exemple, on peut passer une Series en paramètres de la plupart des fonctions de numpy. Par contre, les fonctions de slicing vont aussi être appliquées à l'index.

In [ ]:
```python
s
```

In [ ]:
```python
s[0]
```

In [ ]:
```python
s[:3]
```

In [ ]:
```python
s > s.mean()
```

In [ ]:
```python
s[s > s.mean()]
```

In [ ]:
```python
s[[4, 3, 1]]
```

In [ ]:
```python
np.exp(s)
```

Nous pouvons également appliquer nos propres fonctions en les passant comme paramètres à la méthode apply().

IMPORTANT : la fonction "apply" s'applique sur tous les éléments de la Series un par un. Un exemple :

In [ ]:
```python
# Exemple :  test booléen
def is_greater_than_zero(x):
    return x if x>=0 else 0

print("-"*42)
print(' Avant l\'application de la fct')
print("-"*42)
print(s)
print("-"*42)
print(' Après l\'application de la fct')
print("-"*42)
print(s.apply(is_greater_than_zero)) # Noter qu'il ne faut pas mettre de parenthèses après le nom de votre f
```

In [ ]:
```python
# Exemple : puissance
def to_int_power_two(x):
    return int(x)**2

print("-"*42)
print(' Avant application de la fct')
print("-"*42)

print(s)
print("-"*42)
print(' Après application de la fct')
print("-"*42)
print(s.apply(to_int_power_two))
```

Comme les ndarray, une Series pandas a un attribut dtype (pour data type, c'est le type d'objet contenu dans data).

In [ ]:
```python
s.dtype
```

Vous pouvez aussi convertir une Series (sous condition) en ndarray

In [ ]:
```python
nb_elements=5
ser=pd.Series(np.random.randint(0,nb_elements,nb_elements), index=range(nb_elements))
print(ser)
a= np.array(ser)
print(type(a))
print(a)
```

In [ ]:
```python
data = {'Ah' : 'a', 'Oh' : 20, 'Ih' : 30}
ser=pd.Series(data)
print(ser)
a= np.array(ser)
print(a)
print(a.dtype)
```

Similitudes avec les dictionnaires

Une Series a aussi des similitudes avec les dictionnaires, on peut récupérer ou affecter une valeur à partir d'une clé grâce à l'index

In [ ]:
```python
s = pd.Series(
    data = [-1.07461598, -0.1169448, 2.46739711, -0.1597394, 0.24070523],
    index = ["a", "b", "c", "d", "e"]
)
s['c'] = "Pourquoi suis je là ?"
s['e'] = 42

s
```

In [ ]:
```python
print('e' in s)
```

In [ ]:
```python
## Si on essaie d'accéder à un élément avec un index invalide, une
## exception est lancée

try:
    s["f"]
    print("Ce message ne doit pas s'afficher")
except KeyError:
    print("L'exception a bien été lancée")
```

Opérations vectorielles sur les séries

Lorsque l'on travaille avec des tableaux numpy, on n'a en général pas besoin de faire de boucle. C'est aussi vrai pour les Series pandas. Souvenez-vous, les Series peuvent être passées en paramètre de la plupart des méthodes de numpy qui accèptent un ndarray.

In [ ]:
```python
s = pd.Series(
    data = [-1.07461598, -0.1169448, 2.46739711, -0.1597394, 0.24070523],
    index = ["a", "b", "c", "d", "e"]
)
s
```

In [ ]:
```python
s + s
```

In [ ]:
```python
s * 2
```

In [ ]:
```python
np.exp(s)
```

Une différence importante entre les Series et les ndarray est que les opérations entre Series s'alignent automatiquement sur l'étiquette. Ainsi, vous pouvez écrire des opérations sans que les Series aient nécessairement le même index :

In [ ]:
```python
s[:]+s[:-1]
```

La Series résultant d'une opération entre deux Series dont les index sont différents aura l'union des index des deux Series comme nouveau index. Si une des étiquettes n'est pas présente dans l'une des deux Series, le résultat sera marqué comme NaN (Not a Number $\Rightarrow$ donnée manquante). Ceci permet une plus grande flexibilité mais peut conduire à des résultats inattendus, soyez prudents.

Note: On peut supprimer les index qui contiennent des NaN grâce à la méthode dropna.

In [ ]:
```python
print(s)
print('-*-'*5)
(s[1:] + s[:-1]).dropna()
```

L'objet DataFrame

La DataFrame est un objet bi-dimentionnel avec des colonnes de types potentiellement différents. On peut voir la DataFrame comme une feuille Excel, une table SQL ou encore un dictionnaire de plusieurs Series.

C'est le type d'objet le plus utilisé avec pandas !!

Tout comme la Series, la DataFrame accepte plusieurs entrées pour sa construction :

```
- Dictionnaire de ndarrays de dimension 1, de listes, de dictionnaires ou de Series
- Un ndarray de dimension 2
- Un objet Series
- Une autre DataFrame
```

En plus des données, nous pouvons passer au constructeur d'une DataFrame un index (pour les lignes) et columns (les étiquettes pour les colonnes).

Si les étiquettes ne sont pas fournies explicitement, elles seront inférées (voir plus bas).

Construction à partir d'un dictionnaire de Series ou de dictionnaires

L'index résultant sera l'union des index des Series. Si il y a plusieurs dictionnaires imbriqués, ils seront d'abord convertis en Series. Si l'argument columns n'est pas renseigné, les colonnes seront la liste des clés des dictionnaires. Exemples :

In [ ]:
```python
d = {
    'one': pd.Series([1., 2., 3.,0], index=['a', 'b', 'c','e']),
    'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)

print(df)
print('-$-'*5)
print(df.dtypes)
```

In [ ]:
```python
2*np.random.rand(2,2)-1
```

In [ ]:
```python
pd.DataFrame(d, index=['c', 'a', 'b','h'])
```

In [ ]:
```python
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

Les étiquettes des lignes et des colonnes peuvent être récupérés grâce à l'attribut index et columns de la DataFrame :

In [ ]:
```python
print(df, end="\n"+42*"-"+"\n")

print("Index :", df.index)
print("Colonnes :", df.columns)
```

A partir d'un dictionnaire de ndarray ou de listes

Les ndarrays doivent être de la même taille. Si un index est passé en paramètre, il doit être de la même taille que les ndarrays.

Si aucun index n'est passé, l'index sera, par défaut, range(n) avec n la longueur des ndarrays. Exemples :

In [ ]:
```python
d = {
    'one': [1., 2., 3., 4.],
    'two': [4., 3., 2., 1.]
}
d
```

In [ ]:
```python
df = pd.DataFrame(d)
df
```

In [ ]:
```python
pd.DataFrame(d, index=['a', 'b', 'c', 'd','h'])
```

Sélectionner, ajouter ou supprimer des colonnes

On peut sélectionner une colonne particulière d'un DataFrame :

In [ ]:
```python
print(df['one']) #colonne qui correspond à l'étiquette one'
print('-+'*10)
print(df[["one","two"] ]) #deux colonnes
```

On peut ajouter des colonnes et faire des opérations quand c'est possible :

In [ ]:
```python
print('Avant \n',df)

print('*'*20)
df['three'] = df['one'] * 2*df['two']+1
df['four'] = 4
df
```

In [ ]:
```python
df['flag'] = df['one'] > 2
df
```

On peut supprimer des colonnes (ou faire un "pop") comme avec les dictionnaires :

In [ ]:
```python
print(df,'\n','*'*20)
del df['flag']

print(df,'\n','*'*20)
three = df.pop('three')

print(df,'\n','*'*20)
```

In [ ]:
```python
print(three)
```

Lorsque l'on insert un scalaire il sera propagé pour remplir toute la colonne:

In [ ]:
```python
df['new'] = 'bar'
df
```

Indexation, sélection

On peut sélectionner des données à une position spécifique dans une DataFrame. Plusieurs manières existent :

Operation	Syntaxe	Résultat
Sélectionner une colonne	df[col]	Series
Sélectionner une ligne par son étiquette	df.loc[label]	Series
Sélectionner la ligne numéro i	df.iloc[i]	Series
Sélectionner plusieurs lignes	df[5:10]	DataFrame
Sélectionner plusieurs ligne avec tableau de booleens*	df[bool_vec]	DataFrame

*) True => la ligne est sélectionnée, False => la ligne est ignorée

In [ ]:
```python
d = {
    'one': [1., 2., 3., 4.],
    'two': [4., 3., 2., 1.]
}
df = pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
print(df)
print('-%-'*5)
print(df.loc["a"])
print('-%-'*5)
print(df.iloc[2:])
```

In [ ]:
```python
df.iloc[2]
```

In [ ]:
```python
df[ df["one"] > 2 ]
```

In [ ]:
```python
df.columns
```

In [ ]:
```python
df.index
```

In [ ]:
```python
pp=df.loc['b', 'two']
print(pp)
```

Manipulation basique de données structurées

Pandas est utilisé principalement pour la manipulation de données structurées, on peut par exemple charger un fichier .CSV ou .XLS/.XSLX directement dans une DataFrame

Pour cela, on utilise la méthode de pandas read_csv pour les fichier CSV et read_excel pour les fichiers XLS/XLSX. Attention, read_csv permet de lire les CSV (où les valeurs sont séparées par une virgule) mais plus généralement tout fichier où les valeurs sont séparées par un caractère (dit séparateur). Dans ce cas il faut préciser le séparateur utilisé commme dans l'exemple suivant:

In [ ]:
```python
#ouvrir le fichier avec un tabuleur avant d'utiliser python
#ouvrir le fichier avec un éditeur de texte pour vérifier visuellement le séparateur

df = pd.read_csv("data/chipotle.tsv", sep="\t")
print(type(df))
```

In [ ]:
```python
df1 = pd.read_csv("data/food.csv", sep='\t')
print(type(df1))
df.head(5)
```

In [ ]:
```python
## On peut afficher les dimensions (nombre de lignes et de colonnes)
## avec l'attibut shape (comme avec numpy)
print('la taille :',df.shape) ## (nb lignes, nb colonnes)
print('*'*40)
print('Avec :',df.shape[0],' lignes') ## (nb lignes, nb colonnes)
print('*'*40)
print('Avec :',df.shape[1],' colonnes') ## (nb lignes, nb colonnes)
print('*'*40)
```

In [ ]:
```python
## La commande df.head(n) permet d'afficher uniquement les n premiers éléments
# car la taille de la dataframe est grande avec 4622 lignes
df.head(6) # les 6 premières lignes de 0 à 5 = 6-1
```

In [ ]:
```python
## De même df.tail(n) affiche les n derniers éléments
df.tail(3)
```

In [ ]:
```python
# La commande df.describe() permet d'obtenir, en une seule commande,
# les statistiques des colonnes (UNIQUEMENT pour les colonnes de type numérique (float, int, etc.))
df.describe()
```

Pratique

Exercice 1

1- Créer une Series avec comme index le nom des départements en Auvergne et pour les données, le nombre d'habitants du département correspondant.

2- Créer une Series avec comme index le nom des départements en Auvergne et pour les données, la surface en km^2 du département correspondant.

3- Créer une dataframe indexée par les départements d'Auvergne et qui contient :

```
- nombre d'habitants
- surface
- code postal
- date de création
- nombre d'arrondissements
- nombre de cantons
- nombre de communes
```

Exercice 2

1- Avec la commande pd.read_csv ouvrez le fichier food.csv et assignez le à une dataframe appelée food. Attention le nom de l'extension est trompeur, vérifiez bien le séparateur utilisé (commande cat, grep, ou un éditeur).

2- Afficher les 6 premières lignes

3- Quel est le nombre d'observations (== de lignes) dans le jeu de données ?

4- Afficher le nombre de colonnes

5- Afficher les noms des colonnes

6- Quel est le nom de la 105ème colonne ?

7- Quel est le type d'objet présent à la 105ème colonne ?

8- Quel est l'index du dataset (base de données) ?

9- Quel est le nom du produit à la 19ème ligne ?

Exercice 3 : At Chipotle, How Many Calories Do People Really Eat? (NYT)

Chipotle Mexican Grill est une chaîne de restauration rapide américaine, spécialisée dans la cuisine tex-mex.

1- Avec la commande pd.read_csv ouvrir le fichier chipotle.tsv dans une dataframe df_Chipotle

2- Afficher les 7 premières lignes. Afficher les noms des colonnes

3- Créer une nouvelle colonne dans la dataframe contenant le prix de vente en float à partir de la colonne "item_price".

4- Afficher les lignes d'index impair

5- Afficher les lignes dont le prix du produit est supérieur au prix moyen (p_m)

6- Afficher les lignes dont le prix du produit est supérieur au prix moyen des produits + la déviation standard / sqrt(N) où N est le nombre d'observations :

$$prix \geq \frac{p_m \times \sigma}{\sqrt(N)}$$

In [ ]:
```python

```