La bibliothèque de la semaine
Gestion des tableaux avec pandas

Lucas Henrion Laurent Bataille

Vendredi 17/04

Les bibliothèques `pandas` et `numpy` permettent d'importer des données ainsi que de les visualiser et les manipuler. Les fanatiques de R ne manqueront pas de remarquer les similitudes offertes par pandas lorsqu'il s'agit de gérer les données. Ces deux bibliothèques ne sont rien de moins que les couteaux suisses de Python et il est commun de les importer de cette manière :

```
1 import pandas as pd
2 import numpy as np
```

Les acronymes `pd` et `np` sont donc à peu près universellement utilisé dans les codes python.

Les données gérées par pandas appartiennent soit à la catégorie `Series`, `DataFrame`.

— `Series` : des tableaux unidimensionnels. La colonne d'une série ne peut recevoir de nom.

— `DataFrame` : des tableaux bidimensionnels où les colonnes et les lignes peuvent recevoir des noms. La notation `df` sera utilisée dans les exemples. Un `DataFrame` peut cependant contenir des types de données plus complexes que des chaînes de caractère et des nombres, il est par exemple tout à fait possible de remplir un DataFrame avec des vecteurs !

Chaque ligne est associée à un index, qui peut être vu un moyen d'identifier chaque donnée encodée dans le DataFrame. L'index est soit un nombre, soit une chaîne de caractère, l'important étant que chaque donnée ait un index unique. L'index des différentes lignes ne correspond pas forcément au positionnement de celle-ci !

Sachez qu'il est possible de classer les données avec une architecture complexe au niveau de l'indexation, il est possible d'associer chaque donnée à un doublet d'index, pourvu que les combinaisons soient distinctes [1].

1. Voir la fonction `pd.MultiIndex`

1 Charger des données - `pd.read_csv()`

Cette fonction est utilisée pour lire un fichier csv. Si on lui égal un nom, alors on peut sauver ce fichier csv et l'utiliser ultérieurement pour d'autres traitements. Bien sur, d'autre formats peuvent être lus en changeant le nom de la fonction (`pd.read_xls`, `pd.read_sql` ...) mais le principe reste le même. Vu que python tente d'être le plus universel possible, il faut souvent définir le séparateur des colonnes, décimales et toute autre info utiles. Plus d'infos dans la documentation de la bibliothèque pandas

Exemple :

```
data=pd.read_csv('methane.csv', delimiter=',',sep='.')
```

2 Sélectionner des données - `[]`, `loc`, `iloc`

— `[]` : sélection d'une colonne via son numéro ou son nom. Utiliser le nom d'une colonne est particulière utile pour des données de grande taille.

```
        # On récupère la colonne "Age"
        dfnew = df["Age"]
        # On récupère les colonnes "Age", "College", "Salary"
        dfnew = df[["Age", "College", "Salary"]]

```

— `loc` : sélection de lignes sur base de l'index (qui peut ne pas être un entier ou ordonné de façon conventionnelle)

```
    # formalisme orienté objet : df est un objet de la classe panda
    déjà existant...
    # On récupère la ligne concernant Avery Bradley
    dfnew = df.loc["Avery Bradley"]
    # On récupère les lignes à propos d'Avery Bradley et R.J. Hunter
    dfnew = df.loc[["Avery Bradley", "R.J. Hunter"]]

```

— `iloc` : sélection de lignes sur base de la position ou d'un filtre sur les valeurs d'un champ du DataFrame (arguments : slice, position de ligne, liste de booléens de la même dimension que le DataFrame)

```
    # Sélection des 4e et 5e lignes, recadrafe sur la 2e et 3e
    colonne
    dfnew = df.iloc [[3, 4], [1, 2]]
    # On récupère les lignes concernant des meneurs (point guard)
    dfnew   = df.iloc [data["Position"]=="PG",:]

```

- iloc - référence
- Indexing and Selecting Data with Pandas

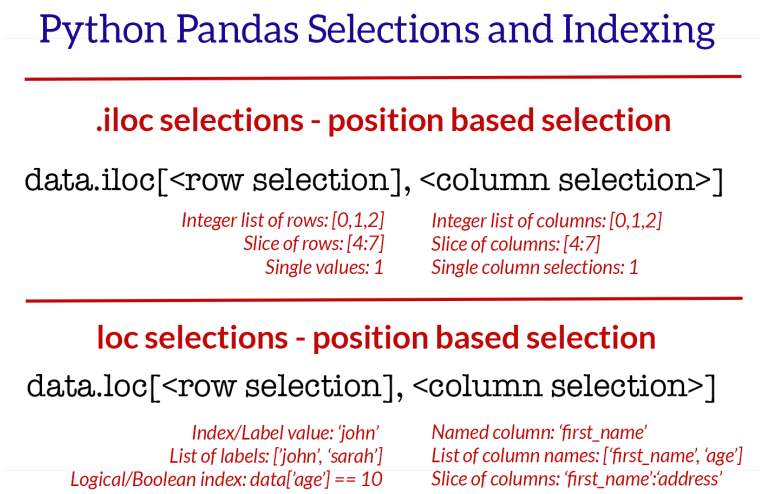

FIGURE 1 – *Exemple utilisation iloc-loc*

3 Appliquer de façon répétitive une opération à tous les éléments d'un DataFrame

3.1 Pandas - `apply`

apply permet de répéter une opération élément par élément, à l'échelle d'un DataFrame ou le long d'une dimension d'un DataFrame.

```
    # Racine carrée de tous les éléments
    newdf = df.apply(np.sqrt)
    # Somme de chaque colonne
    newdf = df.apply(np.sum,axis=0)
```

3.2 Numpy - `where`

La fonction `np.where()` est fort utile car elle permet d'identifier des valeur et éventuellement de les modifier. Par exemple, elle peut servir à détecter des valeurs aberrante, à chercher une valeur particulière .....

```
a = np.array([[0, 1, 2],
              [0, 2, 4],
              [0, 3, 6]])
a1 = np.where(a < 4, a, -1)  # -1 is broadcast
array([[ 0,  1,  2],
       [ 0,  2, -1],
       [ 0,  3, -1]])
```

Le code suivant nous permet d'identifier toute valeur supérieur à 4 et automatiquement lui attribuer la valeur -1. Plus d'information sur la documentation de `numpy` au sujet de la fonction `where`

4 Appliquer un filtre glissant sur des données - `rolling`

La fonction `rolling` est dédiée aux `Series`. Le but est d'appliquer une opération répétitive impliquant plusieurs pas de temps consécutif.

Pour travailler de cette manière, il est nécessaire de définir la fenêtre temporelle que l'on souhaite utiliser ainsi que la taille de fenêtre minimale si des données sont absente du jeu de données. Par défaut on applique souvent une somme, une moyenne ou une médiane.

```
    # Moyenne mobile centrée sur cinq points
    dataM = data.rolling(window=5,center=True, min_periods=1).mean()
```

5 Retirer des données et fusionner des jeux de données

Nettoyer un set de données ou simplement se débarrasser de données ou de colonnes non-pertinentes pour une analyse statistique est une opération routinière. La fonction drop offre la possibilité de réaliser cette opération sur un objet en une seule ligne de code.

```
    # Suppression des colonnes B et C
    df.drop(columns=['B', 'C'])
    # Suppression des deux premières lignes
    df.drop(index=[0, 1])
```

Fusionner des jeux de données peut-être réaliser via la fonction `pd.concat`

```
    df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3'],
                        'C': ['C0', 'C1', 'C2', 'C3'],
                        'D': ['D0', 'D1', 'D2', 'D3']},
                       index=[0, 1, 2, 3])
   df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                        'B': ['B4', 'B5', 'B6', 'B7'],
                        'C': ['C4', 'C5', 'C6', 'C7'],
                        'D': ['D4', 'D5', 'D6', 'D7']},
                       index=[4, 5, 6, 7])
    df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                        'B': ['B8', 'B9', 'B10', 'B11'],
                        'C': ['C8', 'C9', 'C10', 'C11'],
                        'D': ['D8', 'D9', 'D10', 'D11']},
                       index=[8, 9, 10, 11])
    # Liste de DataFrames
    frames = [df1, df2, df3]
    # On précise l'axe et on reset les index pour ne pas que deux données
     possèdent le même identifiant
    result = pd.concat(frames,axis=0,ignore_index=True)

```

— Documentation de la bibliothèque `pandas` au sujet de l'assemblage de set de données

— Documentation de la bibliothèque `pandas` à propos de `drop`