Patrick Nsukami

1/12/24

Contents

Figure 1: Pandas logo

Pandas:

Pandas, dérivé de *pan(el)* et *da(ta)s*, est une bibliothèque libre et open source offrant des structures de données et des fonctions pour nous faciliter l'analyse et la manipulation de données tabulaires. Cet outil se veut puissant, flexible et facile à prendre en main.

Pourquoi:

Pandas nous offre les outils pour lire et écrire des données dans différents formats: fichiers texte, CSV, JSON, Microsoft Excel, base de données, HDF5. Et bien plus encore.

Installation:

Il existe différentes manières d'installer Pandas. L'option la plus simple étant d'installer la distribution Anaconda. Vous pouvez également installer Pandas en utilisant `pip` ou `conda`.

Série, DataFrame, MultiIndex:

Pandas nous offre 3 structures de données importantes. Une pour manipuler des vecteurs, une autre pour manipuler des matrices et enfin une dernière pour manipuler des tableaux ayant des dimensions supérieures à 2.

La structure de données Series peut être vue comme étant un tableau à une dimension possédant des étiquettes et capable d'enregistrer des éléments de différents types (int, str, float, objects, etc..).

La structure de données DataFrame peut être vue comme étant un tableau à deux dimensions, possédant des lignes et des colonnes. Les colonnes peuvent être de différents types, le tableau est modifiable, les 2 axes peuvent être étiquettées.

La structure de données MultiIndex nous permet de manipuler des donneés ayant plus de 2 dimensions, mais en utilisant des structures de données sur une (*Series*) ou deux (*DataFrame*) dimensions. Littéralement, grâce à cette structure de données, vous pourrez avoir plus d'une colonne dans la colonne d'index.

La figure suivante nous montre les différentes composantes d'un dataframe. On remarquera que l'objet de type dataframe est un ensemble d'objets de type series.

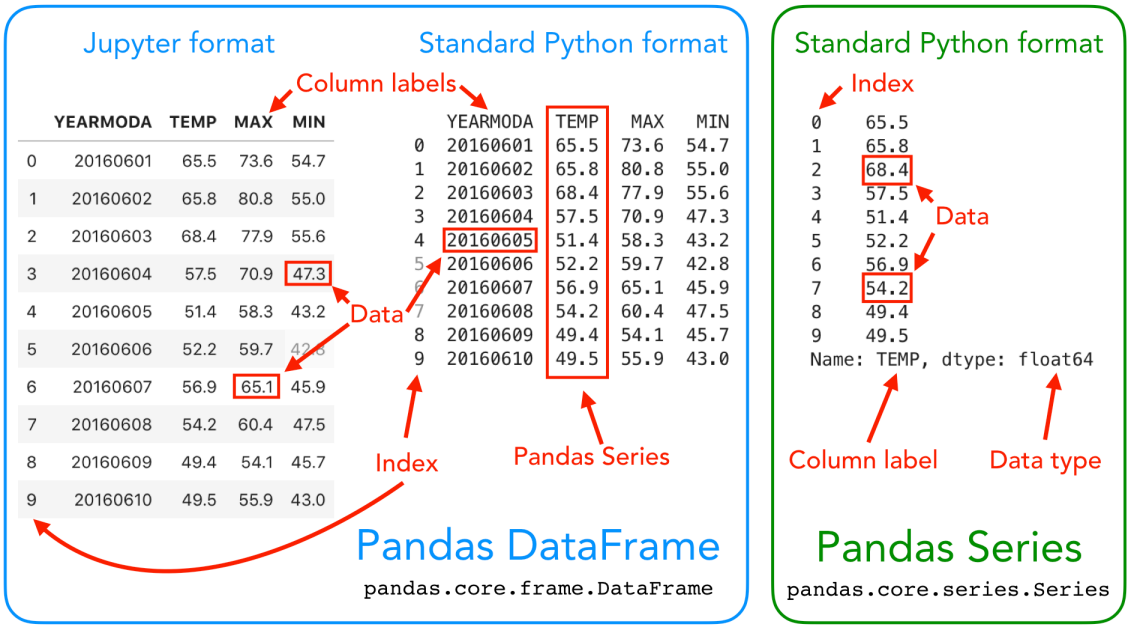

Figure 2: DataFrame

Attributs important à connaître:

.ndim	*le nombre d'axes (dimensions).*
.shape	*un tuple d'entiers indiquant le nombre de lignes et de colonnes.*
.size	*le nombre total d'éléments.*
.dtypes	*le type de chaque colonne.*
.index	*les noms de lignes s'il y en a, sinon des entiers.*
.columns	*les noms de colonnes.*
.axes	*les noms de colonnes et noms de lignes en même temps.*
.empty	*l'objet est-il vide ou non?*
.values	*les éléments.*

Méthodes importantes à connaître:

.describe()	*statistiques descriptives.*
.head()	*Renvoie les n premères lignes de la table, 5 par défaut.*
.info()	*Affiche un résumé à propos de la table.*
.tail()	*Renvoie les n dernières lignes de la table.*
.sample()	*Renvoie un échantillon de la table.*
.isnull()	*Detecte les valeurs manquantes (alias de .isna()).*
.nunique()	*Renvoie le nombre d'éléments distincts.*
.memory_usage()	*Renvoie la mémoire utilisée par chaque colonnes.*

1. Sélection:

loc et iloc:

Les fonctions .iloc et .loc nous aident à extraire certaines lignes et/ou certaines colonnes de notre table. La première s'utilise exclusivement avec des numéros d'indices alors que la seconde s'utilise avec des noms (de lignes ou de colonnes) ou bien avec des masques booléens.

Tout ce que nous avons appris concernant **les indices et les slices** reste valable ici. Attention, contrairement à ce que nous avons l'habitude de voir, aucune borne n'est exclue lors d'un slice. Exemple:

```
import numpy as np

df = pd.DataFrame({
    "Nom": ["Hachim", "Mamad", "Ibra", "Ousman", "Abdel"],
    "Rang": ["Colonel", "Colonel", "Capitaine", "Lieutnant", "Lieutnant"],
    "Age": [40, 43, 35, 32, 34],
    "Masse": [82.3, 93.4, 78.8, 89.7, 80.2],
    'Popularité': np.random.random(5).round(2),
}, index=["a", "b", "c", "d", "e"])

# récupération de la cellule sur la 3e ligne et la 1ère colonne
df.iloc[2, 0]

# toute la 2e colonne
df.iloc[:, 1]

# toute la 2e ligne
df.iloc[1]

# la 3e ligne, 2e et 3e colonne
df.iloc[2, 1:]

# selectionner et classer suivant l'ordre croissant ou décroissant
#df.sort_values(by=('Popularité'), ascending=False)
df
```

```
# récupération de la cellule sur la 3e ligne et la 1ère colonne
df.loc["c", "Nom"]

# toute la 2e colonne
df.loc[:, "Rang"] # ou df["Rang"]

# toute la 2e ligne
df.loc["b"]

# la 3e ligne, 2e et 3e col
df.loc["b", "Rang":]
```

Les masques booléens:

Vous pouvez également **selectionner** les valeurs en utilisant ce qu'on appelle les `masques booléens`. Un masque booléen est un objet contenant l'ensemble des valeurs **respectant un certain critère**. Exemple:

```
# ne garder que les enregistrements pour lesquels le Rang est égal à Colonel
mask = df["Rang"] == "Colonel"
df[mask]
```

	Nom	Rang	Age	Masse	Popularité
a	Hachim	Colonel	40	82.3	0.34
b	Mamad	Colonel	43	93.4	0.37

Nous pouvons appliquer plusieurs critères en les combinants avec des opérateurs logiques:

```
# remarquez que le ET logique s'écrit '&', le OU '|', le non '~'
# popularité supérieure à 0.4 ET Age inférieur à 40
mask = (df["Popularité"] > 0.4 ) & (df["Age"] < 40)
df[mask]
```

	Nom	Rang	Age	Masse	Popularité

2. Modification:

Modification après sélection à l'aide de loc, iloc, mask:

Une des tâches que nous aurons souvent à effectuer en analyse de donnée: modifier des valeurs de notre table. Modifier une valeur consiste à la retrouver, *en utilisant les mécanismes de sélection ci dessus*, puis en effectuant une assignation. Exemple:

```
# modifier toute la colonne
df["Popularité"] = df["Popularité"] * 3

# modifier toute la ligne
df.iloc[2] = ["Ibrahim", "Caporal", 36, 78.1, 3.45]

# modifier une cellule
df.loc["a", "Nom"] = "Assim"

# modifier plusieurs valeurs respectant un critère
mask = (df["Popularité"] < 3 )
df.loc[mask, "Rang"] = "Officier"
```

3. Ajouter (ligne, colonne):

Ajout de ligne (concaténation):

Ajouter une nouvelle ligne est équivalent à concaténer deux dataframes. Exemple:

```
# creation nouvelle ligne
nvl_ligne = pd.DataFrame({
    "Nom": "Nsukami",
    "Rang": "civil",
    "Age": 45,
    "Masse": 90.1,
    "Popularité": 9.8}, index=["p"])

# Ajout d'une nouvelle ligne
df = pd.concat([df, nvl_ligne])
df
```

	Nom	Rang	Age	Masse	Popularité
a	Assim	Officier	40	82.3	1.02
b	Mamad	Officier	43	93.4	1.11
c	Ibrahim	Caporal	36	78.1	3.45
d	Ousman	Officier	32	89.7	0.90
e	Abdel	Officier	34	80.2	1.20
p	Nsukami	civil	45	90.1	9.80

Ajout de colonne:

```
# ajout nouvelle colonne en dernière position:
df["Pays"] = ["Mali", "Guinée", "Burkina", "Wakanda", "Zamunda", "Kongo"]
df
```

	Nom	Rang	Age	Masse	Popularité	Pays
a	Assim	Officier	40	82.3	1.02	Mali
b	Mamad	Officier	43	93.4	1.11	Guinée
c	Ibrahim	Caporal	36	78.1	3.45	Burkina
d	Ousman	Officier	32	89.7	0.90	Wakanda
e	Abdel	Officier	34	80.2	1.20	Zamunda
p	Nsukami	civil	45	90.1	9.80	Kongo

Insertion de colonne:

Nous aurons parfois besoin de réaliser l'insertion d'une colonne, non pas en fin de tableau, mais plutôt à une position spécifique dans le tableau:

```
# ajout nouvelle colonne en dernière position:
df.insert(
    loc=1,
    column="Capitale",
    value=["Bamako", "Conakry", "Burkina", "Wakanda", "Zamunda", "Kongo"]
)
df
```

	Nom	Capitale	Rang	Age	Masse	Popularité	Pays
a	Assim	Bamako	Officier	40	82.3	1.02	Mali
b	Mamad	Conakry	Officier	43	93.4	1.11	Guinée
c	Ibrahim	Burkina	Caporal	36	78.1	3.45	Burkina
d	Ousman	Wakanda	Officier	32	89.7	0.90	Wakanda
e	Abdel	Zamunda	Officier	34	80.2	1.20	Zamunda
p	Nsukami	Kongo	civil	45	90.1	9.80	Kongo

4. Supprimer (ligne, colonne):

Comment supprimer une ligne ? Comment supprimer une colonne ? Dans les deux cas, nous ferons appel à la fonction drop. Exemple:

```
df.drop(index="a", inplace=True) # suppression de la ligne 'a'
df.drop(columns="Masse", inplace=True) # suppresion de la colonne 'Masse'
df
```

	Nom	Capitale	Rang	Age	Popularité	Pays
b	Mamad	Conakry	Officier	43	1.11	Guinée
c	Ibrahim	Burkina	Caporal	36	3.45	Burkina
d	Ousman	Wakanda	Officier	32	0.90	Wakanda
e	Abdel	Zamunda	Officier	34	1.20	Zamunda
p	Nsukami	Kongo	civil	45	9.80	Kongo

5. Les fonctions map, apply, applymap:

Map:

La fonction map nous permet d'appliquer une transformation aux éléments d'un objet de type Series:

```
def f(x):
    return f"{x}, {x.upper()[:3]}"

df["Pays"] = df["Pays"].map(f)
```

```
df
```

	Nom	Capitale	Rang	Age	Popularité	Pays
b	Mamad	Conakry	Officier	43	1.11	Guinée, GUI
c	Ibrahim	Burkina	Caporal	36	3.45	Burkina, BUR
d	Ousman	Wakanda	Officier	32	0.90	Wakanda, WAK
e	Abdel	Zamunda	Officier	34	1.20	Zamunda, ZAM
p	Nsukami	Kongo	civil	45	9.80	Kongo, KON

Apply:

La fonction apply nous permettra d'appliquer une transformation à chaque élément **d'une ligne ou d'une colonne**. Par défaut, la transformation s'applique suivant la première dimension (l'axe 0: lignes). Mais on peut également appliquer la transformation suivant la deuxième dimensions (axe 1: colonnes):

```
df = pd.DataFrame({'A': [1, 2], 'B': [10, 20]})

# appliquer la fonction sum suivant l'axe 0, l'axe des lignes
# çàd: additionner chaque colonne
df.apply(np.sum, axis=0)
```

```
A     3
B    30
dtype: int64
```

```
df = pd.DataFrame({'A': [1, 2], 'B': [10, 20]})

# appliquer la fonction sum suivant l'axe 1, axe des colonnes
# çàd: additionner chaque ligne
df.apply(np.sum, axis=1)
```

```
0    11
1    22
dtype: int64
```

Applymap:

La fonction applymap nous permet d'appliquer une transformation à chaque élément du tableau:

```
df = pd.DataFrame({'A': [1, 2], 'B': [10, 20]})

def f(x):
    return 4*x**3 + 5

# récupérer CHAQUE VALEUR du tableau, l'élever au cube, multiplier par 3 et ajouter 5
df.applymap(f)
```

	A	B
0	9	4005
1	37	32005

6. Les fonctions d'aggrégation:

En analyse de données, **une valeur** peut donner des informations nous permettant de mieux comprendre ce qui se passe dans **un ensemble *relativement large* de données**. Les fonctions d'aggrégation effectuent une opération sur un *ensemble de valeurs* et elles renvoient *une seule valeur.*

Les fonctions d'aggrégations les plus connues sont: sum, count, min, max, average. À cette liste, nous pouvons ajouter: std, var, mode. La liste est non exhaustive, il en existe plusieurs autres.

7. Le concept de group by:

Les fonctions d'aggrégation sont très utiles. Cependant, pour aller encore plus loin dans nos analyses, nous allons utiliser le groupby. Le groupby va nous permettre d'appliquer la fonction d'aggrégation **non pas sur toute la colonne**, mais seulement **sur une partie de la colonne.**

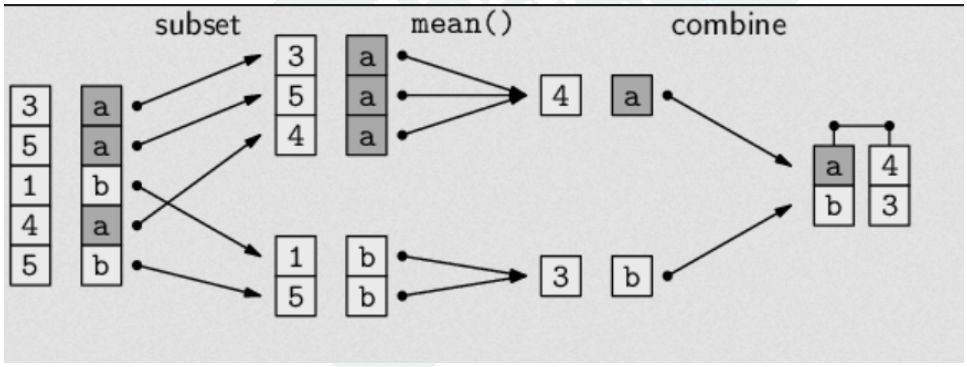

Figure 3: Group by

```
df = pd.DataFrame({"A": ["a", "a", "b", "a", "b"], "B": [3, 5, 1, 4, 5]})
df.groupby("A")["B"].mean()
```

```
A
a    4.0
b    3.0
Name: B, dtype: float64
```

Nous pouvons créer plusieurs groupes avant d'appliquer la fonction d'aggrégation:

```
df = pd.DataFrame({
    "Provenance": ["Burkina", "Senegal", "RDC", "Senegal", "Burkina", "Burkina"],
    "Client": ["Yannick", "Cheikh", "Patrick", "Cheikh", "Moussa", "Moussa"],
    "Total": [4.4, 1.2, 3.5, 2.7, 0.3, 8.6]
})

# grouper suivant la provenance,
# ensuite, calculer le total POUR CHAQUE client
df.groupby(["Provenance", "Client"])["Total"].sum()
```

```
Provenance  Client
Burkina     Moussa     8.9
            Yannick    4.4
RDC         Patrick    3.5
Senegal     Cheikh     3.9
Name: Total, dtype: float64
```

Nous pouvons appliquer plusieurs fonctions d'aggrégation en même temps. Nous pouvons même utiliser une fonction personnalisée pour faire l'aggrégation:

```
df = pd.DataFrame({
    "Provenance": ["Burkina", "Senegal", "RDC", "Senegal", "Burkina", "Burkina"],
    "Client": ["Yannick", "Cheikh", "Patrick", "Cheikh", "Moussa", "Moussa"],
    "Total": [4.4, 1.2, 3.5, 2.7, 0.3, 8.6]
})

def f(seq):
    return sum(seq)**2 + 3

# remarquez cmt, les méthodes sont écrites avec des doubles quotes
# remaquez cmt, la fonction custom est écrite sans les doubles quotes
df.groupby(["Provenance", "Client"])["Total"].aggregate(["max", "min", "sum", f])
```

Provenance	Client	max	min	sum	f
Burkina	Moussa	8.6	0.3	8.9	82.21
	Yannick	4.4	4.4	4.4	22.36
RDC	Patrick	3.5	3.5	3.5	15.25
Senegal	Cheikh	2.7	1.2	3.9	18.21

Conclusion:

En espérant que vous ayez appris deux ou trois choses, sachez que le sujet est beaucoup plus vaste qu'il n'y parait. Il vous est donc vivement recommandé de parcourir les liens suivants afin de parfaire votre apprentissage:

Pour en apprendre plus:

- Page Wikipedia
- Le tutoriel officiel
- Kaggle tutorial
- Introduction aux Series et aux indexes
- DataFrame
- Opérations avancées
- Gestion des dates et des séries temporelles
- Learn Python throught data processing in Pandas
- Introduction to Data Processing in Python with Pandas | SciPy 2019 Tutorial | Daniel Chen
- Solve short hands-on challenges to perfect your data manipulation skills.
- Pandas, join columns of another DataFrame.

Pour aller encore plus loin:

Pandas est devenu "la" bibliothèque à utiliser pour manipuler des données. Pandas est très largement utilisée dans la communauté des développeurs Python. Cependant, lorsque le jeu de données devient *extrêmement massif*, Pandas peut ne plus suffir et produire des erreurs liées à la mémoire. Il existe donc des alternatives à utiliser lorsque l'on manipule des jeux de données dont la taille est importante:

- Polars: Une alternative à Pandas se voulant plus performante.
- Vaex: Autre alternative à Pandas permettant de travailler sur des jeux de données encore plus larges.
- Dask: Alternative à Pandas dont l'objectif est de vous permettre d'executer votre code sur un cluster.