Pandas : Sujets avancés

Pandas : Un manuel complet pour les Data Lovers · https://moncoachdata.com/tutos/pandas-sujets-avances/

Ce tuto sur **Pandas avancé** présente des techniques utiles pour traiter de grands jeux de données et des structures de données complexes : expressions avec `eval()` et `query()`, multi-indexation, traitement des valeurs aberrantes et optimisation de la mémoire.

Calculer avec eval()

La méthode `eval()` évalue une expression écrite sous forme de chaîne de caractères sur les colonnes d'un `DataFrame`. Elle est particulièrement utile pour les gros `DataFrame` et les calculs complexes. Ici, on crée une colonne `C` qui additionne `A` et `B`, puis une colonne `D` à partir d'une expression plus complexe :

```python
import pandas as pd

data = {'A': [1, 2, 3, 4],
        'B': [10, 20, 30, 40]}
df = pd.DataFrame(data)

# Créer une nouvelle colonne C avec eval
df = df.eval('C = A + B')

# Colonne D à partir d'une expression plus complexe
df = df.eval('D = (A * 2) + (B / 3)')
print(df)
```

Python

```
   A   B   C          D
0  1  10  11   5.333333
1  2  20  22  10.666667
2  3  30  33  16.000000
3  4  40  44  21.333333
```

Chaque appel renvoie un nouveau `DataFrame` : affecte le résultat à `df` pour garder la colonne créée.

Filtrer avec query()

La méthode `query()` filtre les lignes d'un `DataFrame` à partir d'une condition écrite sous forme de chaîne de caractères. C'est une manière concise de filtrer les données. Ici, on garde les personnes de plus de 30 ans, puis celles qui cumulent plus de 30 ans et un salaire supérieur à 60 000 :

```python
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'Salary': [50000, 60000, 70000, 80000]}
df = pd.DataFrame(data)

# Filtrer avec une condition
print(df.query('Age > 30'))

# Filtrer avec plusieurs conditions
print(df.query('(Age > 30) and (Salary > 60000)'))
```

Python

```
      Name  Age  Salary
2  Charlie   35   70000
3    David   40   80000
      Name  Age  Salary
2  Charlie   35   70000
3    David   40   80000
```

`eval()` et `query()` sont conçues pour améliorer les performances sur de grands `DataFrame` et des opérations complexes (elles s'appuient sur la bibliothèque `numexpr` si elle est installée). Sur de petits jeux de données, le gain est négligeable : leur intérêt est alors surtout la lisibilité.

Multi-indexation

La multi-indexation (ou indexation hiérarchique) te permet de créer un `DataFrame` avec plusieurs niveaux d'index. Elle est particulièrement utile pour les données aux relations complexes et multidimensionnelles.

```python
data = {'Department': ['HR', 'HR', 'Engineering', 'Engineering'],
        'Employee': ['Alice', 'Bob', 'Charlie', 'David'],
        'Salary': [60_000, 65_000, 80_000, 75_000]}
df = pd.DataFrame(data)

# Créer un index hiérarchique
hierarchical_df = df.set_index(['Department', 'Employee'])
print(hierarchical_df)

# Accéder aux données de Bob, du département RH (HR)
print(hierarchical_df.loc[('HR', 'Bob')])
```

Python

```
                    Salary
Department  Employee
HR          Alice      60000
            Bob        65000
Engineering Charlie    80000
            David      75000
Salary    65000
Name: (HR, Bob), dtype: int64
```

Traiter les valeurs aberrantes

Comme leur nom l'indique, les valeurs aberrantes s'écartent du reste des données. Les traiter est une étape essentielle du prétraitement, car elles peuvent fausser tes analyses et tes modèles statistiques.

Repérer les valeurs aberrantes

Le diagramme en boîte (*boxplot*) permet de les visualiser :

- la **boîte** représente l'intervalle interquartile (IIQ), entre le premier quartile (Q1) et le troisième quartile (Q3). Sa longueur indique la dispersion de la moitié centrale des données ;
- les **moustaches** sont les lignes qui partent de la boîte. En général, elles s'étendent jusqu'à 1,5 fois l'IIQ au-delà de Q1 et de Q3. Tout point situé en dehors est une valeur aberrante *potentielle*.

Tous les points en dehors des moustaches ne sont pas forcément des erreurs : ils doivent être examinés. Ils peuvent apporter une information précieuse ou révéler une erreur dans les données. Il est donc essentiel de comprendre le contexte avant de décider comment les traiter.

```python
import matplotlib.pyplot as plt

data = {'Values': [25, 30, 200, 40, 20, 300, 35, 45, 30]}
df = pd.DataFrame(data)

# Diagramme en boîte pour visualiser les valeurs aberrantes
df.boxplot(column='Values')
plt.show()
```

Python

Les remplacer par NaN

Une fois identifiées, tu peux remplacer les valeurs aberrantes par `NaN` pour les exclure des calculs. Utilise `.loc[]` (et non une double indexation du type `df['Values'][masque] = ...`, qui ne fonctionne plus avec Pandas 3) :

```python
import numpy as np

df = pd.DataFrame({'Values': [25, 30, 200, 40, 20, 300, 35, 45, 30]})
threshold = 100  # seuil au-delà duquel une valeur est aberrante

# Remplacer les valeurs aberrantes par NaN
df.loc[df['Values'] > threshold, 'Values'] = np.nan
print(df)
```

Python

```
   Values
0    25.0
1    30.0
2     NaN
3    40.0
4    20.0
5     NaN
6    35.0
7    45.0
8    30.0
```

Les écrêter avec clip()

L'écrêtage (*clipping*) fixe une limite inférieure et une limite supérieure : toutes les valeurs en dehors de cet intervalle sont remplacées par la limite la plus proche. Cela atténue l'impact des valeurs aberrantes sans les supprimer. Ici, 200 et 300 sont remplacés par 100, et -100 par 0 :

```python
data = {'Values': [25, 30, 200, 40, 20, 300, 35, 45, -100]}
df = pd.DataFrame(data)

# Limites inférieure et supérieure
lower_bound = 0
upper_bound = 100

# Ramener les valeurs aux limites
df['Values'] = df['Values'].clip(lower=lower_bound, upper=upper_bound)
print(df)
```

Python

```
   Values
0      25
1      30
2     100
3      40
4      20
5     100
6      35
7      45
8       0
```

Optimiser la mémoire

L'optimisation de la mémoire est utile partout, mais elle devient cruciale avec de grands jeux de données. Pandas propose plusieurs techniques pour réduire la consommation de mémoire sans perdre d'information.

1. Choisir les bons types de données

- Utilise `int8` ou `int16` pour les entiers qui varient dans une petite plage, et `float32` pour les nombres à virgule ;
- Utilise le type `category` pour les colonnes qui ont peu de valeurs uniques : cela réduit la mémoire et accélère certaines opérations.

```python
df = pd.DataFrame({
    'Age': [25, 30, 35, 40] * 25_000,
    'City': ['Paris', 'Lyon', 'Nice', 'Lille'] * 25_000,
})
print(df.memory_usage(deep=True).sum())

df['Age'] = df['Age'].astype('int8')
df['City'] = df['City'].astype('category')
print(df.memory_usage(deep=True).sum())
```

Python

```
6150132
200346
```

2. Utiliser des types « sparse » quand il y a beaucoup de valeurs manquantes

Pandas gère les données éparses (*sparse*), qui ne stockent que les valeurs utiles. Elles conviennent aux colonnes qui contiennent beaucoup de valeurs manquantes (ou de zéros). Les anciennes classes `SparseDataFrame` et `SparseSeries` ont été supprimées : utilise le type `pd.SparseDtype` à la place.

```python
import numpy as np

s = pd.Series([0, 0, 0, 1, 0, 0, 0, 2] * 100_000)
sparse = s.astype(pd.SparseDtype('int64', 0))

print(s.memory_usage())
print(sparse.memory_usage())
```

Python

```
6400132
2400132
```

3. Lire les données par morceaux

Pour de grands fichiers, utilise le paramètre `chunksize` de `read_csv()` afin de les lire par petits morceaux au lieu de tout charger en mémoire en une fois :

```python
total = 0
for chunk in pd.read_csv('big_file.csv', chunksize=100_000):
    total += chunk['Sales'].sum()
print(total)
```

Python

4. Optimiser les opérations GroupBy

Utilise `as_index=False` dans un `groupby` pour que les clés de regroupement restent des colonnes ordinaires, au lieu de devenir l'index :

```python
df = pd.DataFrame({'Class': ['A', 'B', 'A', 'B'], 'Score': [10, 20, 30, 40]})
print(df.groupby('Class', as_index=False)['Score'].mean())
```

Python

```
  Class  Score
0     A   20.0
1     B   30.0
```

5. Libérer les DataFrames inutiles

Libère explicitement la mémoire en supprimant les `DataFrame` et `Series` dont tu n'as plus besoin, avec le mot-clé `del` :

```python
del df
```

Python

Avec ces techniques, tu gères efficacement la mémoire quand tu travailles avec de grands jeux de données, et tu peux traiter des données qui ne tiendraient pas en mémoire autrement.

Conclusion

Dans cette série de tutos Pandas, tu as parcouru un large éventail de sujets : des structures de données et du chargement des données jusqu'aux techniques avancées pour traiter efficacement des jeux de données complexes. N'oublie pas que tout se joue dans la pratique !

Documentation : améliorer les performances avec Pandas et travailler avec de gros jeux de données.

Pour aller plus loin, consulte aussi notre guide des 10 astuces Pandas en Python.

Pour aller plus loin, il faut des projets complets. Le cours Nettoyage de données t'en propose 3, avec `concat`, `merge`, `melt` et regex.


PANDAS