Pandas

Introduction à Pandas

1. Présentation générale

1. Introduction à Pandas et son rôle dans l'écosystème Python pour l'analyse de données:

- **Définition de Pandas :** Pandas est une bibliothèque open-source Python qui offre des structures de données flexibles pour la manipulation et l'analyse de données. Elle est largement utilisée dans le domaine de la science des données et de l'analyse statistique.
- **Rôle de Pandas :** Pandas facilite le travail avec des données tabulaires et séries temporelles, offrant des outils puissants pour la manipulation, le nettoyage et l'analyse des données.
- **Avantages de Pandas :**
  - Manipulation efficace de données tabulaires.
  - Intégration transparente avec d'autres bibliothèques Python comme NumPy et Matplotlib.
  - Facilité d'importation et d'exportation de données à partir de diverses sources.

2. Installation de Pandas :

- **Installation :** Utiliser la commande `pip install pandas` pour installer Pandas.
- **Importation dans le code :**

```
import pandas as pd
```

2. Structures de données Pandas

1. Les principales structures de données : Series et DataFrame :

- **Series :** Une Serie est une structure unidimensionnelle similaire à un tableau ou une colonne dans une feuille de calcul. Elle peut contenir divers types de données et possède un index qui permet un accès rapide aux éléments.

- **DataFrame :** Un DataFrame est une structure bidimensionnelle qui ressemble à une table de base de données ou à une feuille de calcul Excel. Il est composé de lignes et de colonnes, chaque colonne pouvant être de type différent. Le DataFrame offre une flexibilité pour manipuler et analyser des données tabulaires.

2. Création de Series et DataFrames :

- **Création d'une Serie :**

```
import pandas as pd

# Création d'une Serie à partir d'une liste
my_series = pd.Series([10, 20, 30, 40, 50])
print(my_series)
```

- **Création d'un DataFrame :**

```
import pandas as pd

# Création d'un DataFrame à partir d'un dictionnaire
data = {'Nom': ['Alice', 'Bob', 'Charlie'],
        'Âge': [25, 30, 35],
        'Ville': ['Paris', 'Londres', 'New York']}

df = pd.DataFrame(data)
print(df)
```

- **Création avec index personnalisé :**

```
import pandas as pd

# Création d'une Series avec un index personnalisé
my_series = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c',
'd', 'e'])
print(my_series)
```

- **Création à partir de fichiers :**

```
import pandas as pd
```

```
# Lecture d'un fichier CSV et création d'un DataFrame
df_csv = pd.read_csv('nom_du_fichier.csv')
print(df_csv)
```

3. Importation et exportation de données

1. Importation de données à partir de fichiers CSV, Excel, et autres formats :

- **Importation depuis un fichier CSV :**

```
import pandas as pd

# Importation d'un fichier CSV dans un DataFrame
df_csv = pd.read_csv('nom_du_fichier.csv')
print(df_csv)
```

- **Importation depuis un fichier Excel :**

```
import pandas as pd

# Importation d'une feuille Excel dans un DataFrame
df_excel = pd.read_excel('nom_du_fichier.xlsx',
sheet_name='nom_de_la_feuille')
print(df_excel)
```

- **Importation depuis d'autres formats :**

```
import pandas as pd

# Importation depuis un fichier JSON
df_json = pd.read_json('nom_du_fichier.json')
print(df_json)

# Importation depuis une base de données SQL
from sqlalchemy import create_engine
engine = create_engine('connexion_base_de_donnees')
df_sql = pd.read_sql_query('SELECT * FROM table', engine)
print(df_sql)
```

2. Exportation de données vers différents formats :

- **Exportation vers un fichier CSV :**

```
import pandas as pd

# Exportation d'un DataFrame vers un fichier CSV
df.to_csv('nouveau_fichier.csv', index=False)
```

- **Exportation vers un fichier Excel :**

```
import pandas as pd

# Exportation d'un DataFrame vers un fichier Excel
df.to_excel('nouveau_fichier.xlsx', sheet_name='nom_de_la_feuille',
index=False)
```

- **Exportation vers d'autres formats :**

```
import pandas as pd

# Exportation d'un DataFrame vers un fichier JSON
df.to_json('nouveau_fichier.json', orient='records')

# Exportation vers une base de données SQL
df.to_sql('nom_de_la_table', engine, if_exists='replace',
index=False)
```

Manipulation des données avec Pandas

4. Sélection et indexation

1. Indexation, sélection et filtrage de données :

- **Indexation par position :**

```
import pandas as pd

# Indexation par position
df.iloc[0]  # Première ligne du DataFrame
df.iloc[:, 1]  # Deuxième colonne du DataFrame
```

- **Indexation par label :**

```
import pandas as pd

# Indexation par label
df.loc[0, 'Nom']  # Élément à la première ligne et à la colonne 'Nom'
df.loc[:, 'Age']  # Toute la colonne 'Age'
```

- **Sélection conditionnelle :**

```
import pandas as pd

# Sélection conditionnelle
df[df['Age'] > 25]  # Sélection des lignes où l'âge est supérieur à 25
```

- **Filtrage avec des opérations logiques :**

```
import pandas as pd

# Filtrage avec des opérations logiques
df[(df['Age'] > 25) & (df['Ville'] == 'Paris')]
```

- **Utilisation de méthodes spécifiques :**

```
import pandas as pd

# Utilisation de méthodes spécifiques
df.query('Age > 25')  # Équivalent à df[df['Age'] > 25]
```

Les méthodes `iloc` et `loc` sont deux méthodes clés de l'objet DataFrame de Pandas, utilisées pour l'indexation et la sélection de données dans un DataFrame. La principale différence réside dans la façon dont elles utilisent l'index pour effectuer la sélection.

1. `iloc` **(index location) :**
    - Utilise l'indexation basée sur la position.

- Permet d'accéder aux éléments en utilisant des indices entiers pour les lignes et les colonnes.
- Prend des indices entiers ou des plages d'indices entiers comme arguments.

Exemple :

```
import pandas as pd

# Sélection de la première ligne et des deux premières colonnes
df.iloc[0, 0:2]
```

2. `loc` **(label location) :**

   - Utilise l'indexation basée sur les labels (noms).
   - Permet d'accéder aux éléments en utilisant des labels pour les lignes et les colonnes.
   - Prend des labels ou des plages de labels comme arguments.

   Exemple :

```
import pandas as pd

# Sélection de la première ligne et des colonnes 'Nom' à 'Age'
df.loc[0, 'Nom':'Age']
```

Utilisation commune :

- `iloc` est couramment utilisé pour effectuer des sélections basées sur des positions numériques, comme l'extraction de lignes et de colonnes en utilisant des indices numériques.
- `loc` est couramment utilisé pour effectuer des sélections basées sur des labels, notamment lorsqu'on utilise un index personnalisé (non numérique).

Remarque :
Il est important de noter que `iloc` utilise une indexation exclusive pour les plages, tandis que `loc` utilise une indexation inclusive. Cela signifie que lorsqu'on utilise `iloc[0:3]`, cela sélectionne les lignes 0, 1, 2, mais `loc[0:3]` sélectionnera les lignes 0, 1, 2, 3.

En résumé, la principale distinction entre `iloc` et `loc` réside dans la manière dont elles interprètent les indices lors de la sélection de données dans un DataFrame.

2. Opérations de slicing et d'indexation avancée :

- **Slicing sur les lignes et colonnes :**

```
import pandas as pd

# Slicing sur les lignes
df[1:4]  # Sélection des lignes 2 à 4 (excluant la 5ème)

# Slicing sur les colonnes
df.loc[:, 'Nom':'Ville']  # Sélection des colonnes 'Nom' à 'Ville'
```

- **Indexation avancée avec des listes ou des tableaux :**

```
import pandas as pd

# Indexation avancée avec des listes
df.loc[[0, 2, 4], ['Nom', 'Age']]  # Sélection des lignes 1, 3, 5 pour les colonnes 'Nom' et 'Age'

# Indexation avancée avec des conditions
df.loc[df['Age'] > 25, ['Nom', 'Ville']]  # Sélection des lignes où l'âge est supérieur à 25 pour les colonnes 'Nom' et 'Ville'
```

- **Utilisation de la fonction `isin` pour la sélection :**

```
import pandas as pd

# Utilisation de la fonction isin pour la sélection
villes_selection = ['Paris', 'Londres']
df[df['Ville'].isin(villes_selection)]
```

- **Opérations de slicing avec iloc :**

```
import pandas as pd

# Opérations de slicing avec iloc
```

```
df.iloc[1:4, 0:2]  # Sélection des lignes 2 à 4 et des colonnes 1 à 2
```

5. Manipulation des données

1. Modification, suppression et ajout de colonnes :

- **Modification de valeurs dans une colonne :**

```
import pandas as pd

# Modification de valeurs dans une colonne
df['Age'] = df['Age'] + 1  # Incrémentation de l'âge de 1 pour chaque personne
```

- **Modification basée sur une condition :**

```
import pandas as pd

# Modification basée sur une condition
df.loc[df['Ville'] == 'Paris', 'Age'] += 2  # Incrémentation de l'âge de 2 pour les personnes vivant à Paris
```

- **Suppression de colonnes :**

```
import pandas as pd

# Suppression d'une colonne
df.drop('Nom', axis=1, inplace=True)  # Suppression de la colonne 'Nom'
```

- **Ajout d'une nouvelle colonne :**

```
import pandas as pd

# Ajout d'une nouvelle colonne
df['Nouvelle_Colonne'] = [10, 20, 30]  # Ajout d'une colonne avec des valeurs spécifiques
```

- **Création de colonnes basées sur des opérations :**

```
import pandas as pd

# Création de colonnes basées sur des opérations
df['Age_Carre'] = df['Age'] ** 2  # Création d'une colonne avec
l'âge au carré
```

2. Application de fonctions sur les données :

- **fonction `apply` :**
  La fonction `apply` en Pandas est utilisée pour appliquer une fonction donnée le long d'un axe spécifié d'un DataFrame. Elle peut être utilisée pour appliquer des fonctions personnalisées, des fonctions lambda ou des fonctions prédéfinies à chaque élément, chaque ligne ou chaque colonne d'un DataFrame.

  La signature de base de la fonction `apply` est la suivante :

```
DataFrame.apply(func, axis=0, broadcast=None, raw=False,
reduce=None, result_type=None, args=(), **kwds)
```

  - **`func` :** La fonction à appliquer.
  - **`axis` :** L'axe le long duquel appliquer la fonction. `axis=0` applique la fonction sur chaque colonne, et `axis=1` applique la fonction sur chaque ligne.
  - **`raw` :** Si `True`, la fonction est appliquée aux tableaux NumPy sous-jacents plutôt qu'aux objets Pandas, ce qui peut améliorer les performances dans certains cas.
    Voici quelques exemples pour illustrer son utilisation :

```
import pandas as pd

# Exemple d'application d'une fonction lambda pour doubler les
valeurs d'une colonne
df['Age_Doubled'] = df['Age'].apply(lambda x: x * 2)

# Exemple d'application d'une fonction prédéfinie pour calculer la
longueur des noms dans une colonne
df['Nom_Length'] = df['Nom'].apply(len)

# Exemple d'application d'une fonction sur chaque ligne pour
```

```
calculer la somme de deux colonnes
df['Somme_Colonnes'] = df.apply(lambda row: row['Age'] +
row['Nouvelle_Colonne'], axis=1)
```

Ainsi, la fonction `apply` offre une flexibilité pour appliquer des transformations personnalisées sur les données d'un DataFrame, en permettant une manipulation souple et puissante des données.

- **fonction `map` :**
  La fonction `map` en Pandas est utilisée pour substituer chaque valeur dans une série (ou une colonne d'un DataFrame) par une autre valeur, basée sur un mappage spécifié. Elle est principalement utilisée pour remplacer des valeurs dans une colonne en se basant sur une correspondance avec un dictionnaire ou une autre série.

  La signature de base de la fonction `map` est la suivante :

```
Series.map(arg, na_action=None)
```

  - **`arg` :** Une fonction, un dictionnaire ou une série utilisée pour mapper les valeurs.
  - **`na_action` :** Spécifie l'action à effectuer en cas de valeurs manquantes (`None`). Les options sont `'ignore'` pour ignorer les valeurs manquantes et les laisser inchangées, et `'raise'` pour lever une exception en cas de valeurs manquantes.

  Voici quelques exemples pour illustrer son utilisation :

```
import pandas as pd

# Exemple d'utilisation avec un dictionnaire
villes_mapping = {'Paris': 'France', 'Londres': 'Royaume-Uni', 'New
York': 'États-Unis'}
df['Pays'] = df['Ville'].map(villes_mapping)

# Exemple d'utilisation avec une fonction lambda pour transformer
les valeurs d'une colonne
df['Pays_Majuscule'] = df['Pays'].map(lambda x: x.upper())
```

  Dans le premier exemple, la colonne `'Ville'` est transformée en une nouvelle colonne `'Pays'` en utilisant le dictionnaire `villes_mapping`. Dans le

deuxième exemple, une nouvelle colonne `'Pays_Majuscule'` est créée en appliquant une fonction lambda pour mettre en majuscules les valeurs de la colonne `'Pays'`.

La fonction `map` est utile lorsque vous avez besoin de remplacer les valeurs dans une colonne en fonction d'une correspondance spécifiée, et elle offre une manière concise de le faire en utilisant des dictionnaires, des fonctions ou des séries.

Analyse et traitement des données

6. Agrégation et regroupement

1. Utilisation de la méthode `groupby` pour l'agrégation :

La méthode `groupby` en Pandas est utilisée pour regrouper les données en fonction de certaines caractéristiques, puis appliquer des opérations d'agrégation sur chaque groupe résultant. Cela permet de réaliser des analyses statistiques et des calculs sur des sous-ensembles spécifiques de données.

Exemple :
Supposons que nous ayons un DataFrame représentant des données de ventes avec des informations sur les produits et les montants de ventes. Nous pouvons utiliser `groupby` pour regrouper les données par produit et calculer la somme totale des ventes pour chaque produit.

```
import pandas as pd

# Création d'un DataFrame de ventes
data = {'Produit': ['A', 'B', 'A', 'B', 'A', 'B'],
        'Ventes': [100, 150, 200, 120, 180, 130]}

df_ventes = pd.DataFrame(data)

# Utilisation de groupby pour agréger les ventes par produit
ventes_par_produit = df_ventes.groupby('Produit')
['Ventes'].sum().reset_index()
print(ventes_par_produit)
```

Dans cet exemple, nous avons regroupé les données par la colonne 'Produit' en utilisant `groupby`, puis nous avons appliqué la fonction d'agrégation `sum` pour obtenir la somme totale des ventes par produit.

2. Application de fonctions d'agrégation :

Après avoir utilisé `groupby` pour regrouper les données, nous pouvons appliquer différentes fonctions d'agrégation pour obtenir des statistiques sur chaque groupe. Les fonctions couramment utilisées incluent `sum`, `mean`, `count`, `min`, `max`, etc.

Exemple :
En reprenant l'exemple précédent, nous pourrions vouloir connaître la vente moyenne par produit en utilisant la fonction `mean`.

```
# Utilisation de groupby pour agréger les ventes par produit avec la moyenne
ventes_moyennes_par_produit = df_ventes.groupby('Produit')['Ventes'].mean().reset_index()
print(ventes_moyennes_par_produit)
```

Dans cet exemple, la fonction `mean` a été appliquée pour obtenir la vente moyenne par produit.

Fusion et jointure de DataFrames

1. Combinaison de plusieurs DataFrames :

La combinaison de plusieurs DataFrames est souvent nécessaire lorsqu'on travaille avec des ensembles de données provenant de différentes sources ou avec des relations complexes entre les données. Les opérations de fusion et de jointure permettent de combiner ces ensembles de données en fonction de clés communes.

Exemple :
Imaginons deux DataFrames, l'un contenant des informations sur les étudiants et l'autre sur les cours auxquels ils sont inscrits. Nous pourrions fusionner ces DataFrames pour obtenir une vue complète des étudiants et de leurs cours.

```
import pandas as pd

# Création du DataFrame des étudiants
df_etudiants = pd.DataFrame({'ID_Etudiant': [1, 2, 3],
                             'Nom': ['Alice', 'Bob', 'Charlie']})

# Création du DataFrame des cours
df_cours = pd.DataFrame({'ID_Etudiant': [1, 2, 3],
```

```
                            'Cours': ['Math', 'Physique', 'Chimie']})

# Fusion des DataFrames sur la clé 'ID_Etudiant'
df_complet = pd.merge(df_etudiants, df_cours, on='ID_Etudiant')
print(df_complet)
```

Dans cet exemple, les deux DataFrames ont été fusionnés sur la clé `'ID_Etudiant'`, créant ainsi un nouveau DataFrame qui combine les informations des étudiants et des cours.

2. Utilisation de méthodes comme `merge` et `concat` :

- La méthode `merge` est utilisée pour effectuer des opérations de fusion (jointure) entre deux DataFrames.
- La fonction `concat` est utilisée pour concaténer (empiler) plusieurs DataFrames le long d'un axe donné.

Exemple :
Nous pourrions concaténer plusieurs DataFrames contenant les résultats de différents examens pour obtenir une vue complète des performances des étudiants.

```
# Création de deux DataFrames avec des résultats d'examens différents
df_exam1 = pd.DataFrame({'ID_Etudiant': [1, 2, 3], 'Résultat_Exam1':
[85, 90, 78]})
df_exam2 = pd.DataFrame({'ID_Etudiant': [1, 2, 3], 'Résultat_Exam2':
[92, 88, 95]})

# Concaténation des DataFrames le long des colonnes
df_resultats_finaux = pd.concat([df_exam1, df_exam2], axis=1)
print(df_resultats_finaux)
```

Dans cet exemple, les deux DataFrames `df_exam1` et `df_exam2` ont été concaténés le long des colonnes, créant ainsi un DataFrame combiné avec les résultats des deux examens.