🐼

Pandas in Python (Part 2)

Created by : Eng. Amjaad Altakhaineh

Pandas is one of the most popular Python libraries for **data manipulation and analysis**. Below, I'll walk you through **everything related to Pandas** in **detailed categories** so you can master this tool!

---

1. Getting Started with Pandas

- **Installation:**

```
pip install pandas
```

- **Import Pandas:**

```
import pandas as pd
```

---

2. Reading and Writing Data (I/O Operations)

3.1 Reading Data

- **CSV Files:**

```
df = pd.read_csv('data.csv')
```

- **Excel Files:**

```
df = pd.read_excel('data.xlsx')
```

3.2 Writing Data

- **CSV:**

```
df.to_csv('output.csv', index=False)
```

- **Excel:**

```
df.to_excel('output.xlsx', index=False)
```

4. Data Inspection and Summary

4.1 Basic Data Summary

```
df.head()      # First 5 rows
df.tail()      # Last 5 rows
df.info()      # DataFrame structure and data types
df.describe()  # Summary statistics for numerical data
```

4.2 Data Types Conversion

```
df['Age'] = df['Age'].astype('float')
df['Date'] = pd.to_datetime(df['Date'])
```

---

5. Indexing and Selecting Data

5.1 Selection by Label (`loc`)

The `loc` method in pandas is a powerful tool for selecting data based on labels of rows and columns in a DataFrame or Series. It provides a way to access data using the explicit labels (names) of rows and columns, making it more intuitive when working with labeled data.

Key Features of `loc` :

1. **Access rows by index label**:
   You can retrieve specific rows using their labels.
2. **Access columns by name**:
   You can specify column names to select one or more columns.
3. **Simultaneously select rows and columns**:

   `loc` allows you to slice and select rows and columns together.

4. **Boolean indexing**:
   You can use conditions to filter data with
   `loc`.

```
df.loc[0:2, ['Name', 'Age']]  # Select rows 0 to 2 and column
s 'Name' and 'Age'
```

5.2 Selection by Position (`iloc`)

The `iloc` method in pandas is used for selecting data by **integer-based** positions (row and column indices). Unlike `loc`, which works with labels, `iloc` relies purely on numerical positions of rows and columns.

Key Features of `iloc`:

1. **Access rows by position**:
   You can retrieve rows using their integer indices.
2. **Access columns by position**:
   You can specify column positions to select one or more columns.
3. **Simultaneously select rows and columns by position**:

   `iloc` allows slicing and selecting both rows and columns together using integer-based indexing.
4. **Boolean indexing**:
   You can combine
   `iloc` with conditions based on row/column positions.

```
df.iloc[0:3, 0:2]  # Select first 3 rows and first 2 columns
```

5.3 Conditional Selection

```
df[df['Age'] > 25]  # Filter rows where Age > 25
```

6. Handling Missing Data

1. Identifying Missing Values

To handle missing values, you first need to identify them.

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {'A': [1, 2, np.nan, 4], 'B': [np.nan, 2, 3, 4], 'C':
[1, 2, 3, 4]}
df = pd.DataFrame(data)

# Check for missing values
print(df.isnull())  # Returns a DataFrame of True/False value
s
print(df.isnull().sum())  # Count missing values per column
print(df.isnull().any())  # Check if any missing value exists
per column
```

2. Dropping Missing Values

You can drop rows or columns with missing values using `dropna`.

```
# Drop rows with missing values
df_dropped_rows = df.dropna()
```

```
# Drop columns with missing values
df_dropped_cols = df.dropna(axis=1)

# Drop rows with missing values in a specific column
df_specific = df.dropna(subset=['A'])

# Drop rows if all values are missing
df_all_na = df.dropna(how='all')

# Drop rows with at least `thresh` non-NA values
df_thresh = df.dropna(thresh=2)
```

3. Filling Missing Values

Use `fillna` to fill missing values with specific values, methods, or statistics.

```


# Fill with a constant value
df_filled_constant = df.fillna(0)

# Fill with the mean of the column
df_filled_mean = df.fillna(df.mean())

# Fill with forward fill (propagate last valid value forward)
df_filled_ffill = df.fillna(method='ffill')

# Fill with backward fill (propagate next valid value backwar
d)
df_filled_bfill = df.fillna(method='bfill')

# Fill using interpolation
df_interpolated = df.interpolate()

# Fill with a dictionary (specific values per column)
```

```
df_filled_dict = df.fillna({'A': 0, 'B': df['B'].mean()})
```

4. Replacing Missing Values

Replace missing values using `replace`.

```
python
Copy code
# Replace np.nan with a specific value
df_replaced = df.replace(to_replace=np.nan, value=99)
```

5. Imputing Missing Values with scikit-learn

For advanced imputations, scikit-learn provides `SimpleImputer`.

```

from sklearn.impute import SimpleImputer

# Replace missing values with the mean ( Numerical data point
s)
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=
df.columns)

# Replace missing values with the mean ( catogrical data poin
ts)
imputer = SimpleImputer(strategy='most_frequent')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=
df.columns)
```

6. Using Masking

Masking provides custom handling for missing values.

```
# Replace missing values conditionally
df_masked = df.mask(df.isnull(), other=-1)
```

7. Detecting Non-Missing Values

Use `notnull` to detect non-missing values.

```
print(df.notnull())  # Returns a DataFrame of True/False valu
es
```

8. Custom Missing Value Indicators

Handle specific values as missing (e.g., `-999`, `0`).

```
# Treat -999 as missing
df_custom_na = pd.DataFrame({'A': [1, -999, 3, 4]})
df_custom_na = df_custom_na.replace(-999, np.nan)
```

9. Checking for Missing Data Before Operations

Some operations automatically handle missing data, but it is good practice to preprocess.

```
# Fill missing values before summing
df_sum = df.fillna(0).sum()
```

---

7. Data Cleaning and Transformation

7.1 Renaming Columns

```
df.rename(columns={'Name': 'FullName'}, inplace=True)
```

7.2 Removing Duplicates

```
df.drop_duplicates(inplace=True)
```

7.3 String Operations

```
df['Name'] = df['Name'].str.upper()  # Convert names to upper
case
```

---

8. Merging, Joining, and Concatenation

8.1 Concatenation

```
df_combined = pd.concat([df1, df2], axis=0)  # Vertical stack
```

8.2 Merging

```
df_merged = pd.merge(df1, df2, on='ID', how='inner')  # Merge
```

```
on 'ID'
```

8.3 Joining

```
df_joined = df1.join(df2, how='outer')
```

9. GroupBy and Aggregation

9.1 Grouping Data

```
grouped = df.groupby('Department')['Salary'].sum()
```

9.2 Aggregating Data

```
df.groupby('Department').agg({'Salary': 'mean', 'Age': 'ma
x'})
```

10. Sorting and Ranking

10.1 Sorting Values

```
df.sort_values(by='Age', ascending=False, inplace=True)
```

10.2 Ranking

```
df['Rank'] = df['Salary'].rank(ascending=False)
```

---

11. Exporting Data to Other Formats

- **CSV:**

  ```
  df.to_csv('data.csv')
  ```

- **Excel:**

  ```
  df.to_excel('data.xlsx')
  ```

- **JSON:**

  ```
  df.to_json('data.json')
  ```

---

12. Performance Optimization in Pandas

15.1 Use `categorical` Data Types for Memory Optimization

```
df['Category'] = df['Category'].astype('category')
```

15.2 Use `vectorized` operations instead of loops

```
df['New_Column'] = df['Old_Column'] * 2  # Faster than loops
```

16. Pandas in Machine Learning

- **Feature Engineering:** Use `groupby`, `merge`, and `pivot` to generate new features.
- **Handling Missing Data:** Use `fillna` and `dropna`.
- **Scaling Data:** Pandas can preprocess data before sending it to ML models using `StandardScaler` from `sklearn`.