Pandas

Prof. Pranay Kumar Saha

February 12, 2025

What is Pandas?

- Open-source Python library providing:
  - High-performance data structures
  - Easy-to-use data analysis tools
- Ideal for data manipulation, cleaning, and analysis

Installation

- Install with `pip`:

```
pip install pandas
```

- Or, install with `conda`:

```
conda install pandas
```

Importing Pandas

```
import pandas as pd
import numpy as np

print("Pandas version:", pd.__version__)
print("NumPy version:", np.__version__)
```

Series

- One-dimensional labeled array
- Capable of holding multiple data types
- Labels are the *index*

Creating a Series

```
# From a Python list
data = [10, 20, 30, 40]
labels = ['a', 'b', 'c', 'd']
ser = pd.Series(data, index=labels)
```

Other ways to create a Series

```
# From a NumPy array
arr = np.array([100, 200, 300])
pd.Series(arr)

# From a dictionary
d = {'x': 10, 'y': 20, 'z': 30}
pd.Series(d)
```

DataFrame

- Two-dimensional labeled data structure with columns of different types
- Similar to a spreadsheet or SQL table

Creating a DataFrame

```
data_dict = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Age': [24, 30, 18, 35],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}
df = pd.DataFrame(data_dict)
df
```

Custom Row Labels

```
df_custom_index = pd.DataFrame(
    data_dict,
    index=['row1','row2','row3','row4']
)
df_custom_index
```

Inspecting the DataFrame

```
# Display top rows
df.head(2)

# Display bottom rows
df.tail(2)
```

Data Example

df

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles
Charlie	35	Chicago
David	40	Houston
Eve	45	Phoenix

Data Example

df

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles
Charlie	35	Chicago
David	40	Houston
Eve	45	Phoenix

df.head(2)

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles

Data Example

df

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles
Charlie	35	Chicago
David	40	Houston
Eve	45	Phoenix

df.head(2)

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles

df.tail(2)

Name	Age	City
David	40	Houston
Eve	45	Phoenix

Shape, Info, and Describe

```
print("Shape of df:", df.shape)
print("\nDataFrame Info:\n")
print(df.info())
print("\nDescriptive Statistics:\n")
print(df.describe())
```

Example

Name	Age	Salary
Alice	25	50000
Bob	30	60000
Charlie	35	70000
David	40	80000
Eve	45	90000

- `df.shape: (5, 3)`
- `df.info()`

```
Column    Non-Null Count    Dtype
-------   ---------------   -------
Name      5 non-null        object
Age       5 non-null        int64
Salary    5 non-null        int64
```

`df.describe()`

	Age	Salary
count	5.00000	5.00000
mean	35.00000	70000.00000
std	7.90569	15811.38830
min	25.00000	50000.00000
25%	30.00000	60000.00000
50%	35.00000	70000.00000
75%	40.00000	80000.00000
max	45.00000	90000.00000

Selecting Columns and Rows

```
# Single column
df['Age']

# Multiple columns
df[['Name', 'City']]

# Select rows by label
df_custom_index.loc['row2']

# Select rows by integer location
df.iloc[1]  # second row
```

Filtering

```
# Filter by condition
df[df['Age'] > 25]

# Combine multiple conditions (example)
df[(df['Age'] > 25) & (df['City'] == 'Houston')]
```

Example

Name	Age	Salary
Alice	25	50000
Bob	30	60000
Charlie	35	70000
David	40	80000
Eve	45	90000

Name	Age	City
Bob	30	Los Angeles
Charlie	35	Chicago
David	40	Houston
Eve	45	Phoenix

Name	Age	City
David	40	Houston

Reading Data

```
# Reading a CSV file
df_csv = pd.read_csv('data.csv')

#Reading an excel file
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Reading a JSON file
df_json = pd.read_json('data.json')
```

Writing Data

```python
# Writing to a CSV file
df.to_csv('output.csv', index=False)
# index=False prevents writing the index to the file

# Writing to an excel file
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)

# Writing to a JSON file
df.to_json('output.json', orient='records')
```

Handling Missing Data

```
nan_data = {
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [10, 11, 12, 13]
}
df_nan = pd.DataFrame(nan_data)
df_nan

# Check missing
df_nan.isnull()
```

```
# Drop rows with missing values
df_nan_drop = df_nan.dropna()

# Fill missing
df_nan_fill = df_nan.fillna(0)
```

Replacing Values

```
df_replace = df_nan.fillna(-1)
df_replace.replace(-1, 99)
```

Concatenation

```
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'col1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['D', 'E'], 'col1': [4, 5]})

df_concat = pd.concat([df1, df2], axis=0)
df_concat
```

Concatenation

```
pd.concat([df1, df2], axis=0)
```

key	col1
A	1
B	2
C	3
D	4
E	5

```
pd.concat([df1, df2], axis=1)
```

	key	col1	key	col1
0	A	1	D	4
1	B	2	E	5
2	C	3	NaN	NaN

```
pd.concat([df1.add_suffix('_df1'), df2.add_suffix('_df2')], axis=1)
```

Merge (like SQL Join)

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2'], 'A': [1, 2, 3]})
right = pd.DataFrame({'key': ['K1', 'K2', 'K3'], 'B': [4, 5, 6]})

df_merge = pd.merge(left, right, on='key', how='inner')
df_merge
```

Join types: `inner`, `outer`, `left`, `right`.

Example: Inner Join

left

key	A
K0	1
K1	2
K2	3

right

key	B
K1	4
K2	5
K3	6

inner Join

key	A	B
K1	2	4
K2	3	5

Example: Left Join

left

key	A
K0	1
K1	2
K2	3

right

key	B
K1	4
K2	5
K3	6

left Join

	key	A	B
0	K0	1	NaN
1	K1	2	4.0
2	K2	3	5.0

Example: Right Join

left

key	A
K0	1
K1	2
K2	3

right

key	B
K1	4
K2	5
K3	6

right Join

	key	A	B
0	K1	2.0	4
1	K2	3.0	5
2	K3	NaN	6

Example: Outer Join

left

key	A
K0	1
K1	2
K2	3

right

key	B
K1	4
K2	5
K3	6

outer Join

	key	A	B
0	K0	1.0	NaN
1	K1	2.0	4.0
2	K2	3.0	5.0
3	K3	NaN	6.0

Grouping and Aggregation

```
sales_data = {
    'Store': ['A', 'A', 'B', 'B', 'C', 'C'],
    'Product': ['Apples', 'Oranges', 'Apples',
                'Oranges', 'Apples', 'Oranges'],
    'Revenue': [100, 150, 80, 120, 90, 160]
}

df_sales = pd.DataFrame(sales_data)
df_sales.groupby('Store').agg({'Revenue': 'sum'})

# Group by multiple columns
df_sales.groupby(['Store', 'Product']).agg({'Revenue': 'mean'})
```

Example

Store	Product	Revenue
A	Apples	100
A	Oranges	150
B	Apples	80
B	Oranges	120
C	Apples	90
C	Oranges	160

Store	Revenue
A	250
B	200
C	250

Store	Product	Revenue
A	Apples	100.0
A	Oranges	150.0
B	Apples	80.0
B	Oranges	120.0
C	Apples	90.0
C	Oranges	160.0

Example

Store	Product	Revenue
A	Apples	100
A	Apples	120
A	Oranges	150
B	Apples	80
B	Apples	90
B	Oranges	120
C	Apples	90
C	Oranges	160

```
Store    Product    Revenue
  A      Apples       110.0  # (100+120)/2
  A      Oranges      150.0
  B      Apples        85.0  # (80+90)/2
  B      Oranges      120.0
  C      Apples        90.0
  C      Oranges      160.0
```

Apply / Map

- `df['column'].apply(function)` applies a function to every element in that column
- `df.apply(function, axis=1)` applies a function across rows

Pandas Date

`pd.date_range()` is a flexible function in Pandas that allows creating a wide variety of date (and time) ranges.

Basic Usage

Example: Creating a sequence of 5 dates starting from January 1, 2025, with a daily frequency:

```
import pandas as pd
dates = pd.date_range(start='2025-01-01', periods=5, freq='D')
```

This results in a `DatetimeIndex` containing Pandas `Timestamp` objects.

Common Frequencies in `pd.date_range()`

- **Daily ('D')** –One day apart.
- **Business Days ('B')** –Weekdays only.
- **Weekly ('W')** –Defaults to weeks ending on Sundays (can be customized).
- **Monthly Start ('MS')** –First day of each month.
- **Monthly End ('M')** –Last day of each month.

Quarterly and Yearly Frequencies

- **Quarter End ('Q')** –Last day of each quarter.
- **Quarter Start ('QS')** –First day of each quarter.
- **Year End ('A' or 'Y')** –Last day of the year.
- **Year Start ('AS' or 'YS')** –First day of the year.

Example:

```
pd.date_range(start='2025-01-01', periods=5, freq='Q')
pd.date_range(start='2025-01-01', periods=5, freq='A')
```

Sub-Daily Frequencies

- **Hourly ('H')** –One-hour intervals.
- **Minutes ('T' or 'min')** –One-minute intervals.
- **Seconds ('S')** –One-second intervals.
- **Milliseconds ('L')**, Microseconds ('U'), Nanoseconds ('N').

Example:

```
pd.date_range(start='2025-01-01', periods=5, freq='H')
pd.date_range(start='2025-01-01 00:00', periods=5, freq='T')
```

Time Zone Aware Dates

Creating a timezone-aware date range:

```
pd.date_range(start='2025-01-01', periods=5, freq='D', tz='UTC')
```

This ensures the dates have a specified time zone.

Time series in Pandas

We will:

- Create a DataFrame with a datetime index.
- Resample the data.
- Compute a rolling average.

Reading a CSV File

Use `pd.read_csv()` to load a CSV file into a Pandas DataFrame.

```
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('your_file.csv')
```

This loads the CSV, but the date column is still a string.

Converting the Date Column

Convert a date column named `"date"` from a string to a datetime format.

```
df['date'] = pd.to_datetime(df['date'])
```

If the date format is specific (e.g., ”DD/MM/YYYY”), specify the format explicitly:

```
df['date'] = pd.to_datetime(df['date'], format='%d/%m/%Y')
```

Setting the Date Column as the Index

It’s often useful to set the date column as the index for time-series operations.

```
df.set_index('date', inplace=True)
```

This allows easier filtering, grouping, and resampling by date.

Parsing Dates While Reading CSV

Pandas can automatically parse the date column when reading the CSV:

```
import pandas as pd

# Parse the 'date' column as datetime and set it as the index
df = pd.read_csv('your_file.csv', parse_dates=['date'], index_col='date')
```

This method is efficient and avoids manual conversion.

Creating a Time Series DataFrame

```python
import pandas as pd
import numpy as np

# Create a date range for 100 days starting from January 1, 2025
dates = pd.date_range(start='2025-01-01', periods=100, freq='D')

# Create a DataFrame with random data and set the date range as the index
df = pd.DataFrame({
    'value': np.random.randn(100)
}, index=dates)

# Display the first few rows
print(df.head())
```

Small Data Example

Below is a small example of creating a time series DataFrame with 5 days of data.

```
import pandas as pd

# Create a small dataset for 5 days
dates = pd.date_range(start='2025-01-01', periods=5, freq='D')
data = {'value': [0.5, -0.3, 1.2, 0.0, -1.0]}
df_small = pd.DataFrame(data, index=dates)

print(df_small)
```

Output: Small Data Example

When you run the above code, you will get the following output:

```
            value
------------------
2025-01-01  0.5
2025-01-02  -0.3
2025-01-03  1.2
2025-01-04  0.0
2025-01-05  -1.0
```

Resampling and Rolling Mean

```
# Resample the data to a monthly frequency and calculate the mean
monthly_mean = df.resample('M').mean()
print(monthly_mean)

# Calculate a 7-day rolling mean
df['rolling_mean'] = df['value'].rolling(window=7).mean()
print(df.head(10))
```

Plotting

- Pandas has extensive support for time-series data (date-range generation, frequency conversion, resampling).
- For quick plotting, you can do:

```
df_sales['Revenue'].plot()
```

- Remember to import `matplotlib.pyplot` (and possibly run `%matplotlib inline` in Jupyter).

Summary

- Introduction to Pandas
- Data Structures: Series and DataFrame
- Basic Operations: Selection, Filtering, Indexing
- Data Import/Export
- Data Cleaning (Missing Values, Replacing)
- Merging, Joining, and Concatenating
- Grouping and Aggregation
- Advanced Operations (Apply, Pivot, Time-Series, Plotting)

For more details, check out the official documentation:

`https://pandas.pydata.org/docs/`

Pandas

Thank You for Listening!