PANDAS

A PYTHON LIBRARY FOR DATA-MANIPULATION.

INTRODUCTION TO PANDAS

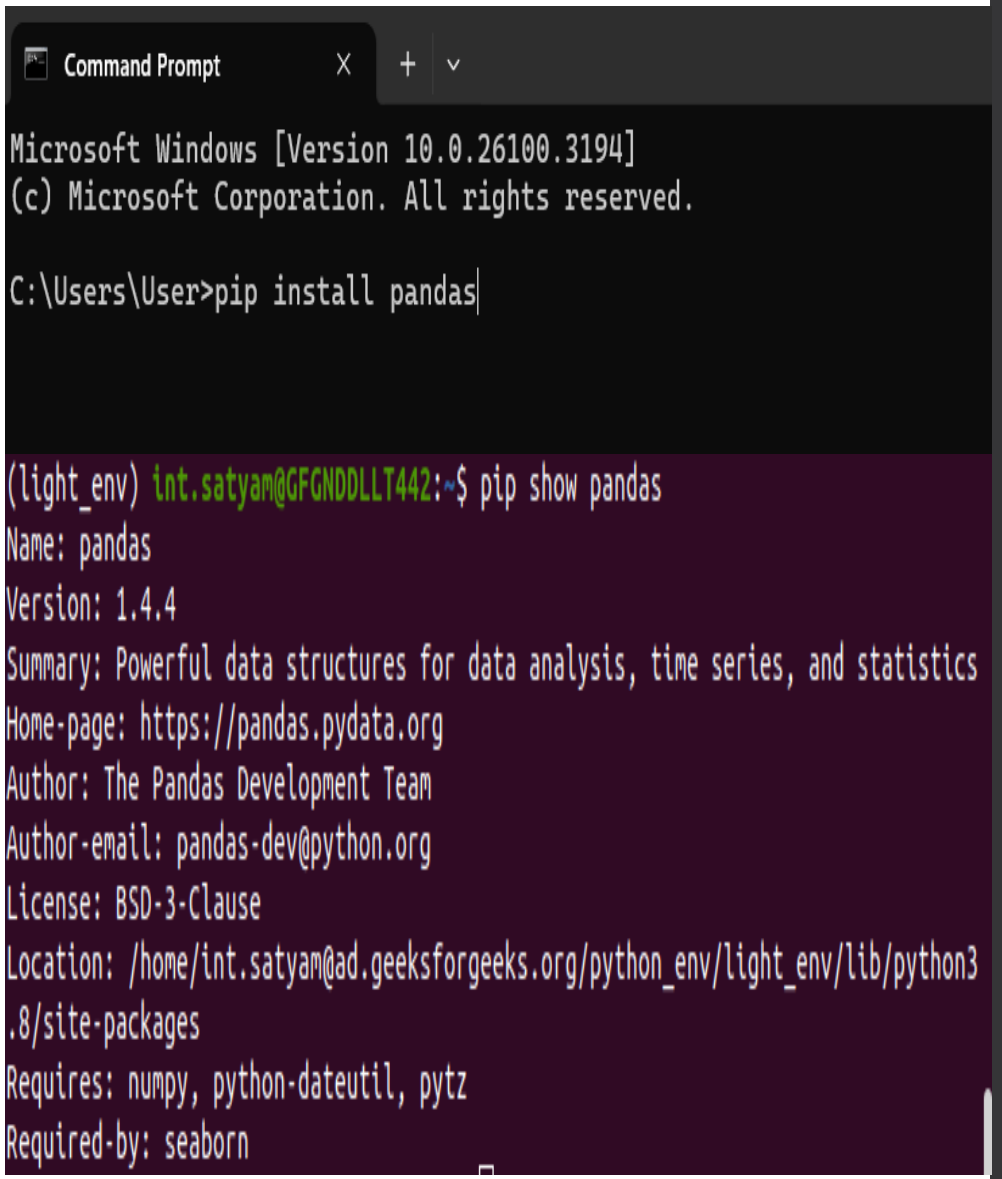
OVERVIEW AND SIGNIFICANCE

- Developed by Wes McKinney (2008) for quantitative analysis.
- Fast, powerful, flexible, easy-to-use data analysis & manipulation tool.
- Pandas is a high-level, open-source library for data analysis and manipulation in Python.
- Built on Python, open-source.
- Name from “Panel Data” handles multi-dimensional data.
- It provides two main data structures: Series (1D) and DataFrame (2D), for handling structured data.
- Supports label-based indexing for intuitive row and column access.
- Offers robust data cleaning tools including dropna(), fillna(), and type conversions.
- Enables easy input/output with CSV, Excel, JSON, HTML, and SQL formats.
- Provides descriptive statistics and summary functions like mean(), sum(), and describe().
- Supports powerful filtering, conditional selection, reshaping, and grouping operations.
- Integrates seamlessly with NumPy, Matplotlib, Seaborn, and Scikit-learn.
- Includes advanced time series features like resampling, shifting, and rolling computations.
- Optimized for performance with C/Cython backend for fast operations on large data.
- Widely used in industry and academia for its flexibility and reliability.
- Essential component of the Python data science ecosystem and commonly used in real-world projects.

INSTALLATION AND SETUP

SETTING UP PANDAS FOR USE

- Install Pandas using pip install pandas.
- Use conda install pandas if working with Anaconda.
- Verify installation with import pandas as pd.
- Check installed version using pd.__version__.
- Update Pandas using pip install --upgrade pandas or conda update pandas.
- Install Jupyter Notebook using pip install notebook or conda install notebook.
- Use Pandas directly in Google Colab without installation.
- Dependencies like NumPy are auto-installed with Pandas.
- Set up a virtual environment using venv or conda create.
- Install specific version using pip install pandas==version_number.
- Test setup with a sample DataFrame: pd.DataFrame({'A': [1, 2]}).



CORE DATA STRUCTURES

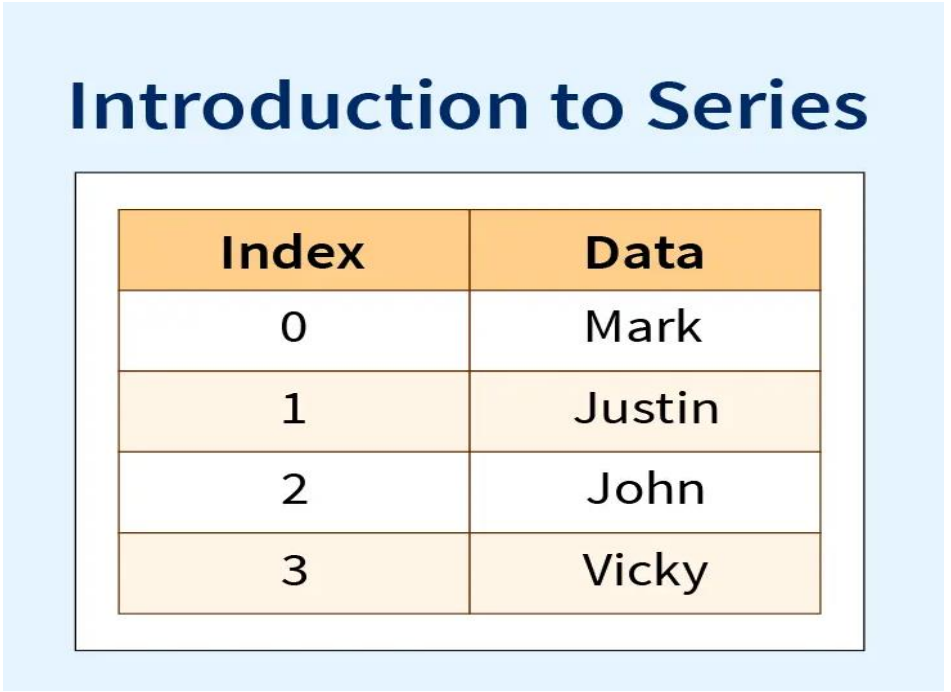
UNDERSTANDING SERIES AND DATA FRAME

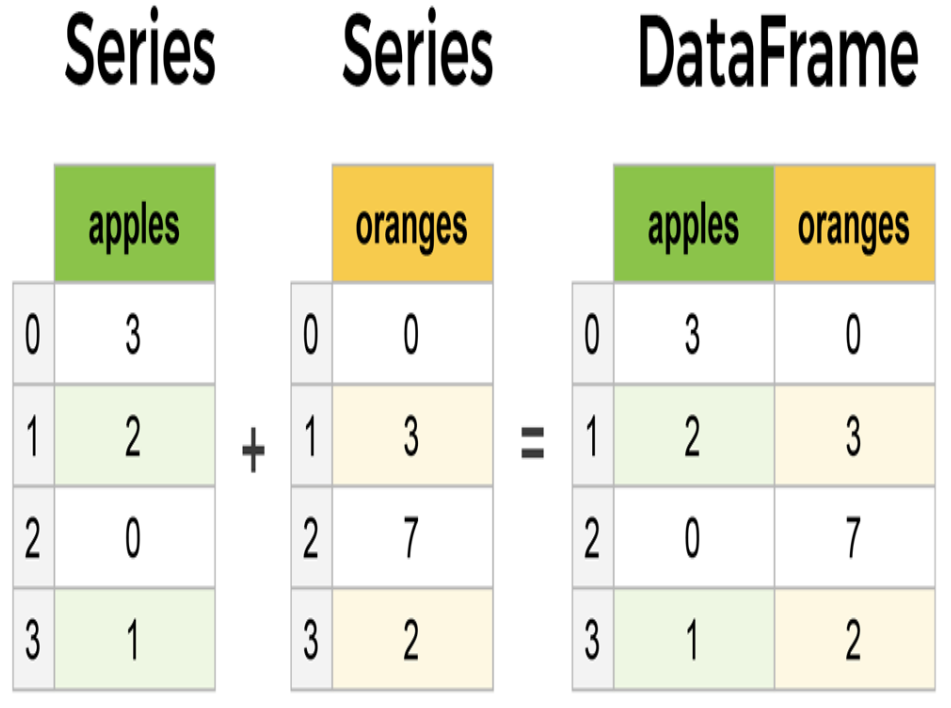
SERIES

- One-dimensional labeled array
- Can hold any data type (int, float, string, etc.)
- Accessed by position or label
- Created using pd.Series(data, index)
- Index is customizable
- Supports NumPy-like operations
- Automatically aligns data on index
- Efficient for single-column operations

DATAFRAMES

- Two-dimensional labeled data structure
- Columns can be of different types
- Created using pd.DataFrame(data, columns)
- Rows and columns are both labeled
- Supports indexing and slicing
- Used for tabular data
- Missing data handled gracefully
- Integrated functions for stats and I/O





CREATING DATA FRAMES

READING & WRITING DATA IN CSV/ EXCEL FILE

I/O - CSV

- Read with pd.read_csv('file.csv')
- Write with df.to_csv('file.csv')
- Supports options like header, nrows
- Handles delimiters and encodings
- Efficient for flat file data exchange
- Reads from local or remote paths
- Allows setting index column
- Commonly used in pipelines

I/O - EXCEL

- Read with pd.read_excel('file.xlsx')
- Write with df.to_excel('file.xlsx')
- Supports multiple sheets
- Use sheet_name parameter
- Requires openpyxl or xlrd engine
- Retains formatting and formulas (partially)
- Useful for business/reporting use cases
- Easily integrates with other Excel tools

	A	B	C	D	E	F	G	H	I
1	Date	Temperatu	Dew Point	Humidity	Sea Level F	visibility m	wind speec	cloud cove	event
2									
3	1/1/2024	38	23	52	30.34	10	8	5	
4	1/2/2024	36	18	47	23.23	10	7	3	
5	1/3/2024	31	21	46	43.34	10	8	1	
6	1/4/2024	38	9	42	34.54	10	9	3	
7	1/5/2024	32	2	41	41.04	10	5	0	
8	1/6/2024	25	4	35	44.311	10	4	3	
9	1/7/2024	34	11	45	47.582	10	2	8	Rain

```
# Write to CSV
df.to_csv('sample_data.csv', index=False)  # Saves without the index column
print(" CSV file written: sample_data.csv")
```

```
CSV file written: sample_data.csv
```

```
[ ]  # Write to Excel
     df.to_excel('products.xlsx', index=False, sheet_name='Inventory')
     print("Excel file written: products.xlsx")
```

```
Excel file written: products.xlsx
```

VIEWING YOUR DATA

- df.head() – Returns the first 5 rows of the DataFrame.
- df.tail() – Returns the last 5 rows of the DataFrame.
- df.head(n) / df.tail(n) – Displays the first or last n rows.
- df.shape – Shows the number of rows and columns as a tuple.
- df.columns – Lists all column names in the DataFrame.
- df.index – Displays the row index labels of the DataFrame.
- df.dtypes – Returns data types of each column.
- df.info() – Gives a summary of columns, types, and non-null counts.
- df.describe() – Provides statistical summary for numeric columns.
- df.values – Returns the data as a 2D NumPy array.
- df.sample(n) – Randomly selects n rows from the DataFrame.
- df.to_string() – Converts the DataFrame into a plain text string.

```
# First 5 rows
print(" df.head():\n", df.head())
```

```
 df.head():
       Name  Age  Salary Department
0    Alice   25   50000         HR
1      Bob   30   60000    Finance
2  Charlie   35   70000         IT
3    David   40   80000         HR
4      Eva   28   62000         IT
```

```
# Last 5 rows
print("\n df.tail():\n", df.tail())
```

```
 df.tail():
       Name  Age  Salary Department
1      Bob   30   60000    Finance
2  Charlie   35   70000         IT
3    David   40   80000         HR
4      Eva   28   62000         IT
5    Frank   22   45000    Finance
```

```
# Shape of the DataFrame (rows, columns)
print(" df.shape:", df.shape)
```

```
df.shape: (6, 4)
```

```
[ ] # Column names
print(" df.columns:", df.columns.tolist())
```

```
df.columns: ['Name', 'Age', 'Salary', 'Department']
```

```
# Data types of each column
print(" df.dtypes:\n", df.dtypes)
```

```
 df.dtypes:
 Name          object
Age            int64
Salary         int64
Department    object
dtype: object
```

DROPPING, SORT AND RANK

DROPPING

Dropping refers to removing rows or columns from a DataFrame using the .drop() method. This is commonly used when you want to clean or simplify your dataset by removing unnecessary or unwanted data.

- Use .drop() to remove rows/columns
- Specify axis=0 for rows
- Specify axis=1 for columns
- Use inplace=True to modify original
- Accepts label or list of labels
- Handles missing labels with errors='ignore'
- Can chain with other DataFrame methods
- Common for cleaning data

SORT AND RANK

Sorting and ranking are used to organize data either by values or index and to assign ranks based on order. These functions help analyze and prioritize data effectively.

- .sort_values() sorts by column values
- .sort_index() sorts by index
- Use ascending=False to reverse order
- Rank with .rank()
- Handles ties with ranking methods
- Specify axis for row/column sorting
- Sort multiple columns using a list
- Maintains index by default

```
# Drop a column (e.g. 'Department')
df_dropped_col = df.drop('Department', axis=1)
print("After dropping 'Department' column:\n", df_dropped_col)
```

```
After dropping 'Department' column:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   35   70000
3    David   40   80000
```

```
[ ] # Drop a row by index label
    df_dropped_row = df.drop(2)  # Drops the row with index 2 (Charlie)
    print("After dropping row at index 2:\n", df_dropped_row)
```

```
After dropping row at index 2:
      Name  Age Department  Salary
0  Alice   25         HR   50000
1    Bob   30    Finance   60000
3  David   40         HR   80000
```

```
[ ] # Handle missing labels safely
    df_safe = df.drop(['NonExistent'], axis=1, errors='ignore')
    print("Dropping non-existent column safely:\n", df_safe)
```

```
Dropping non-existent column safely:
      Name  Age Department  Salary
0    Alice   25         HR   50000
1      Bob   30    Finance   60000
2  Charlie   35         IT   70000
3    David   40         HR   80000
```

RETRIEVING BASIC INFO AND SUMMARY

RETRIEVING BASIC INFO

- .shape returns (rows, columns)
- .columns lists column labels
- .index shows row labels
- .dtypes returns data types
- .info() gives memory usage and structure
- .count() returns non-NA counts
- Helps in exploring new datasets
- Efficient for large data introspection

RETRIEVING SUMMARY

- .sum(), .mean(), .std() for basic stats
- .min(), .max() for range values
- .idxmin(), .idxmax() return index positions
- .describe() gives summary of numeric columns
- .value_counts() for categorical frequency
- .median(), .mode() for central tendency
- Works on numeric columns by default
- Use axis to control row/column operation

```
## .shape – Dimensions
# Returns a tuple with number of rows and columns

print("Shape (rows, columns):", df.shape)
```

```
Shape (rows, columns): (5, 4)
```

```
[ ] ## .columns – Column labels
    print(" Column Names:", df.columns.tolist())
```

```
 Column Names: ['Name', 'Age', 'Salary', 'Department']
```

```
[ ] ##  .index – Row labels
    print("Index Labels:", df.index)
```

```
Index Labels: RangeIndex(start=0, stop=5, step=1)
```

```
[ ] ## .dtypes – Data types of each column

    print("Data Types:\n", df.dtypes)
```

```
Data Types:
 Name          object
Age           float64
Salary        float64
Department     object
dtype: object
```

DATA ALIGNMENT

INTERNAL DATA ALIGNMENT

It is an internal mechanism in Pandas that ensures data from different Series or DataFrames is aligned correctly by index or column labels during operations. This allows for clean, efficient, and consistent operations even when shapes or labels differ.

- Aligns data by index during operations
- Automatically fills missing values with NaN
- Internal alignment simplifies merging
- Use .add(), .sub(), etc. with fill_value
- Preserves data consistency across frames
- Useful for combining mismatched data
- Handles reindexing cleanly
- Supports arithmetic with Series/DataFrames

ARITHMETIC OPERATIONS WITH FILL METHODS

- Use df.add(df2, fill_value=0)
- Handles missing values during math ops
- Other methods: .sub(), .mul(), .div()
- Prevents NaN propagation
- Controls behavior of arithmetic on NA
- Ideal for data with different shapes
- Preserves alignment logic
- Use with broadcasting for matrix-like ops

```
# Fill missing with 0 during addition
result_filled = df1.add(df2, fill_value=0)
print("df1.add(df2, fill_value=0):\n", result_filled)
```

```
df1.add(df2, fill_value=0):
      A     B
x   1.0   4.0
y  12.0  35.0
z  23.0  46.0
```

```
series = pd.Series([100, 200], index=['A', 'B'])
added = df1 + series
print("df1 + Series:\n", added)
```

```
df1 + Series:
      A    B
x  101  204
y  102  205
z  103  206
```

```
df1 = pd.DataFrame({
    'Math': [90, 80, 70],
    'English': [85, 75, 65]
}, index=['Ali', 'Sara', 'John'])

df2 = pd.DataFrame({
    'Math': [88, 78],
    'Science': [92, 81]
}, index=['Sara', 'John'])

print("df1:\n", df1)
print("\ndf2:\n", df2)
```

```
df1:
        Math  English
Ali      90       85
Sara     80       75
John     70       65

df2:
        Math  Science
Sara     88       92
John     78       81
```

THANK YOU