

Huawei HCCDA-AI certification

Instructor : Fawad Bahadur

Mastering Pandas: From Basics to Expert Level

Powerful Data Analysis & Manipulation with Python



Dated: 14 June 2025

Instructor : Fawad Bahadur

Training Objectives & Outcomes

1

Understand the fundamentals of Pandas Series and DataFrames

2

Perform efficient data cleaning, filtering, and transformation

3

Master advanced techniques like grouping, merging, reshaping, and time series handling

4

Apply Pandas to real-world data analysis, reporting, and machine learning workflows

Introduction to Pandas

What is Pandas?

- Open-source Python library for data manipulation and analysis.
- Provides powerful data structures: Series and DataFrame.
- Built on top of NumPy, designed for working with structured data.

Why Use Pandas?

- Easy handling of missing data.
- Flexible reshaping and pivoting.
- Powerful group-by functionality.
- Supports time series data.

Example:

```
import pandas as pd  
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}  
df = pd.DataFrame(data)  
print(df)
```

Series		Series		DataFrame	
apples		oranges		apples	oranges
0	3	0	0	0	0
1	2	1	3	1	3
2	0	2	7	2	7
3	1	3	2	3	2



Setting Up Pandas in Your Environment

What is Pandas?

- Open-source Python library for data analysis and manipulation
- Built on top of NumPy

Prerequisites-

- Python installed (preferably 3.6 or above)
- pip (Python package installer)

Installation Using pip:

```
pip install pandas
```

Verify Installation:

```
import pandas as pd  
print(pd.__version__)
```



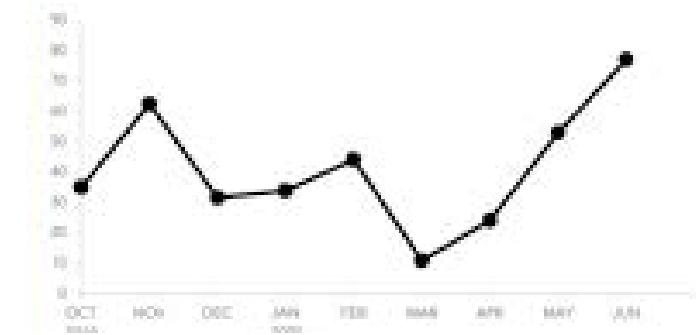
Pandas Series

What is a Series?

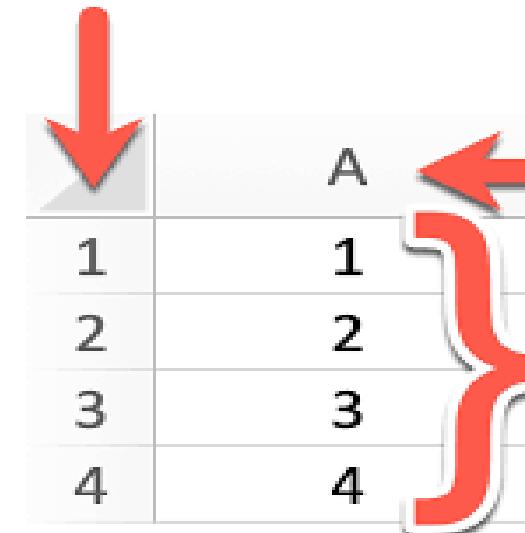
- One-dimensional labeled array.
- Can hold any data type (integers, strings, floats, etc.).
- Labels (index) identify each element.

Creating a Series

```
import pandas as pd  
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])  
print(s)
```



Series Index



A diagram illustrating the components of a Pandas Series. It shows a grid with a single column labeled 'A'. The first row is labeled '1' and contains the value '1'. The second row is labeled '2' and contains the value '2'. The third row is labeled '3' and contains the value '3'. The fourth row is labeled '4' and contains the value '4'. A red arrow points down to the first row, labeled 'Series Index'. A red arrow points left to the column header 'A', labeled 'Series Name'. A red curly brace on the right side of the grid, spanning all four rows, is labeled 'Series Values'.

	A
1	1
2	2
3	3
4	4

**Series
Name**
**Series
Values**

Tip

Series is the building block for
DataFrame columns.

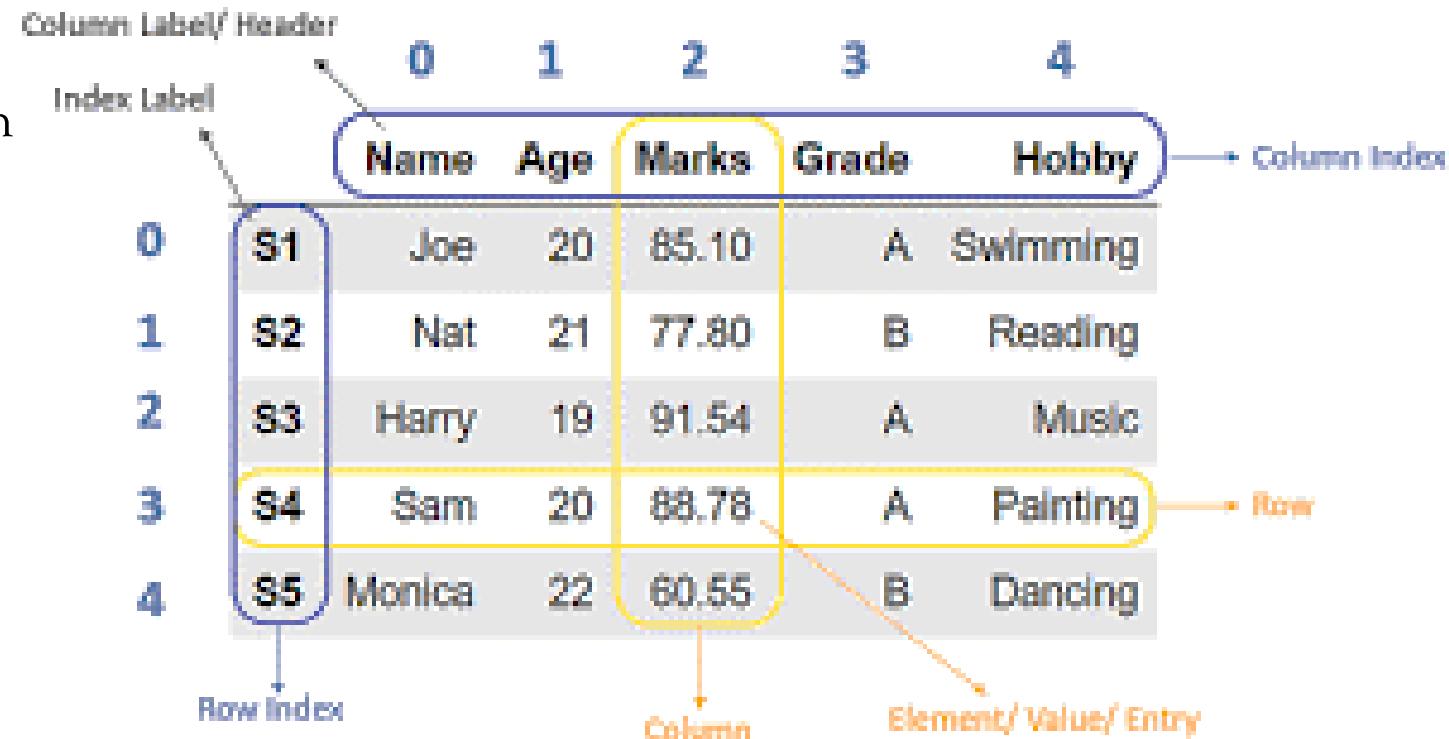
Pandas DataFrame Basics

What is a DataFrame?

- Two-dimensional labeled data structure with columns of potentially different types.
- Like a spreadsheet or SQL table.

Creating a DataFrame

```
import pandas as pd  
data = {  
    'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],  
    'Salary': [50000, 60000, 70000]  
}  
df = pd.DataFrame(data)  
print(df)
```



The diagram illustrates the structure of a Pandas DataFrame. It is a 5x6 grid where the first row and column serve as headers. The columns are labeled 0, 1, 2, 3, 4, and the rows are labeled 0, 1, 2, 3, 4. The header row contains column labels: Name, Age, Marks, Grade, and Hobby. The index row contains index labels: S1, S2, S3, S4, and S5. Arrows point from these labels to their respective positions in the table. The data cells are color-coded: light blue for the first two rows (S1-S2), grey for the third (S3), yellow for the fourth (S4), and light green for the fifth (S5). Labels explain the components: 'Column Label/ Header' points to the column labels; 'Index Label' points to the index labels; 'Column Index' points to the column numbers; 'Row' points to the row number 4; 'Element/ Value/ Entry' points to the cell '60.55'; 'Row Index' points to the index label 'S5'; and 'Column' points to the column 'Marks'.

	0	1	2	3	4
0	S1	Joe	20	85.10	A Swimming
1	S2	Nat	21	77.80	B Reading
2	S3	Harry	19	91.54	A Music
3	S4	Sam	20	88.78	A Painting
4	S5	Monica	22	60.55	B Dancing

Tip

- DataFrame is the core structure for data analysis in Pandas.

Data Selection & Indexing?

Selecting Columns

```
df['Age']      # Single column (Series)
```

```
df[['Name', 'Age']] # Multiple columns (DataFrame)
```

Selecting Rows

Using .loc (label-based)

```
df.loc[0]      # First row by index label
```

```
df.loc[0:2]    # Rows 0 to 2 inclusive
```

Using .iloc (integer position-based)

```
df.iloc[0]     # First row by position
```

```
df.iloc[0:2]   # Rows 0 and 1
```

Setting Values

```
df.loc[1, 'Salary'] = 65000
```

Additional Selection

- df[df['Age'] > 25] # Rows where Age > 25

The diagram illustrates the selection of data from a DataFrame. On the left, a DataFrame is shown with four rows and three columns: Name, Age, and Department. The rows are indexed 0, 1, 2, and 3. The columns are Name, Age, and Department. Row 0 has values Jim, 26, Sales. Row 1 has values Dwight, 26, Sales. Row 2 has values Angela, 27, Accounting. Row 3 has values Tio, 32, Human Resources. Two red arrows point downwards from the 'Name' and 'Department' column headers to the corresponding columns in the DataFrame. An arrow points from the original DataFrame to a modified version on the right. In the modified DataFrame, the 'Age' column for all rows is highlighted in blue, indicating that the 'Age' column has been selected.

	Name	Age	Department
0	Jim	26	Sales
1	Dwight	26	Sales
2	Angela	27	Accounting
3	Tio	32	Human Resources

→

	Name	Department
0	Jim	Sales
1	Dwight	Sales
2	Angela	Accounting
3	Tio	Human Resources

Select one or more columns

Tip

Use .loc for label indexing and .iloc for positional indexing.

Handling Missing Data?

Detecting Missing Data

```
df.isnull()      # Returns DataFrame of True/False for missing  
df.isnull().sum() # Count missing values per column
```

Changing Data Types

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

Removing Duplicates

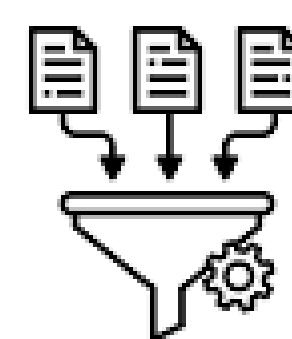
```
df.drop_duplicates(inplace=True)  
df.drop_duplicates(subset=['ID', 'Email'], inplace=True)
```

Tip

Clean and transform your data
to prepare it for analysis.



loan_amnt	term	int_rate	sub_grade	emp_length	home_ownership	annual_inc	loan_status	grade	sub_grade	emp_length	home_ownership	annual_inc	loan_status	int_rate	sub_grade	emp_length	home_ownership	annual_inc	loan_status
36000	36 months	10	C1	10+ years	MORTGAGE	60000	Fully Paid	A	B1	10+ years	MORTGAGE	60000	Fully Paid	10	B2	10+ years	MORTGAGE	60000	Fully Paid
24000	36 months	10	C1	10+ years	MORTGAGE	60000	Fully Paid	B2	C1	10+ years	MORTGAGE	60000	Fully Paid	10	C2	10+ years	MORTGAGE	60000	Fully Paid
120000	36 months	10	B1	10+ years	MORTGAGE	60000	Fully Paid	C1	D1	10+ years	MORTGAGE	60000	Fully Paid	10	D1	10+ years	MORTGAGE	60000	Fully Paid
36000	36 months	10	C1	10+ years	MORTGAGE	60000	Current	A	B1	10+ years	MORTGAGE	60000	Current	10	B2	10+ years	MORTGAGE	60000	Current
10000	36 months	10	D1	3 years	MORTGAGE	70000	Fully Paid	B1	C1	3 years	MORTGAGE	70000	Fully Paid	10	C2	3 years	MORTGAGE	70000	Fully Paid
10000	36 months	10	C1	4 years	MORTGAGE	70000	Fully Paid	B2	D1	4 years	MORTGAGE	70000	Fully Paid	10	D1	4 years	MORTGAGE	70000	Fully Paid
10000	36 months	10	B1	10+ years	MORTGAGE	70000	Fully Paid	C1	A	10+ years	MORTGAGE	70000	Fully Paid	10	B2	10+ years	MORTGAGE	70000	Fully Paid
20000	36 months	10	B1	10+ years	MORTGAGE	80000	Fully Paid	D1	C1	10+ years	MORTGAGE	80000	Fully Paid	10	C2	10+ years	MORTGAGE	80000	Fully Paid
10000	36 months	10	C1	10+ years	MORTGAGE	80000	Fully Paid	B1	D1	10+ years	MORTGAGE	80000	Fully Paid	10	D1	10+ years	MORTGAGE	80000	Fully Paid
10000	36 months	10	D1	10+ years	MORTGAGE	80000	Fully Paid	C1	B1	10+ years	MORTGAGE	80000	Fully Paid	10	B2	10+ years	MORTGAGE	80000	Fully Paid



Data Aggregation & Grouping

GroupBy Basics

- Split data into groups based on column values.
- Apply aggregation functions.

Common Aggregation Functions

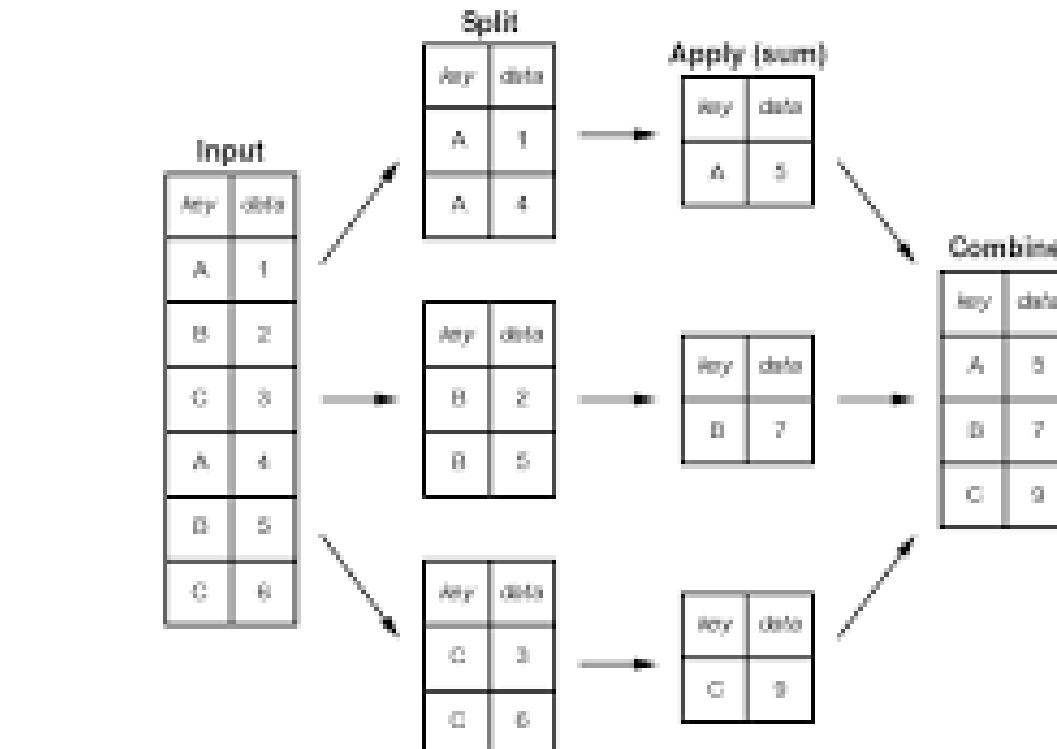
- .sum()
- .mean()
- .count()
- .min()
- .max()

Example

```
grouped = df.groupby('AgeGroup')
print(grouped['Income'].mean())
```

Multiple Aggregations

```
grouped['Income'].agg(['mean', 'max', 'min'])
```



Tip

GroupBy is powerful for summarizing data by categories.

Concatenation in Pandas

Purpose

Used to stack DataFrames or Series vertically or horizontally along an axis.

Functionality

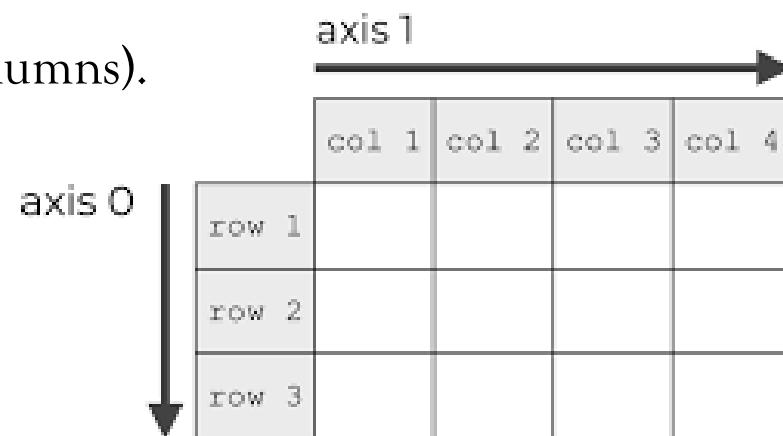
- Combines DataFrames by appending them along a specified axis (rows or columns).
- Does not perform any relational operations or key-based joining.
- Can handle DataFrames with different columns or indices.
- Can result in duplicate indices if not handled properly.

Use Case

When you want to combine DataFrames with similar structures or append new data to existing DataFrames without considering common keys.

Code

```
pd.concat([df1, df2], axis=0) # Vertical (rows)  
pd.concat([df1, df2], axis=1) # Horizontal (columns)
```



Merge in Pandas

Purpose

Used to combine DataFrames based on common columns or indices, similar to SQL joins.

Functionality

- Merges two DataFrames based on specified columns or indices.
- Supports different types of joins: inner, outer, left, right.
- Can handle complex merging scenarios with multiple keys.
- Can result in duplicate columns if not handled properly.

Use Case

When you want to combine DataFrames based on related columns, similar to joining tables in a database.

Code

```
pd.merge(df1, df2, on='key', how='inner') # Inner join  
pd.merge(df1, df2, on='key', how='left') # Left join
```

Join Type	Syntax	Returns
Inner	how='inner'	Only matching rows
Left	how='left'	All left rows + matching right
Right	how='right'	All right rows + matching left
Outer	how='outer'	All rows from both

Tip

Use merging and joining to combine related datasets effectively.

Join in Pandas

Purpose

A specialized version of merge() optimized for joining DataFrames based on their indices or key columns.

Functionality

- Combines two DataFrames based on indices or specified key columns
- By default, performs a left join, but other types can be specified
- Can be more efficient than merge() when joining on indices.

Use Case

When you want to combine DataFrames based on their indices or specific key columns, similar to joining tables in a database but with a focus on index-based joins.

Code

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

Tip

Use merging and joining to combine related datasets effectively.

Working with Time Series Data

DateTime Index

Use `pd.to_datetime()` to convert columns to datetime.

```
df['Date'] = pd.to_datetime(df['Date'])  
df.set_index('Date', inplace=True)
```

Resampling

Aggregate time series data at different frequencies.

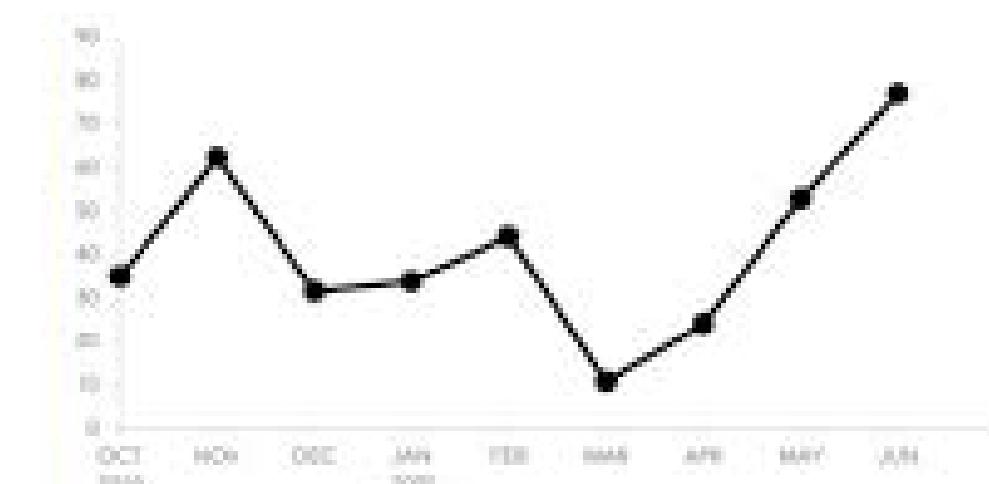
```
df.resample('M').mean() # Monthly average
```

Time-based Selection

```
df.loc['2023-01-01':'2023-01-31']
```

Shifting Data

```
df['PrevDay'] = df['Value'].shift(1)
```



Tip

Pandas makes time series manipulation intuitive and powerful.

Handling Text Data in Pandas

String Methods

- Use .str accessor for vectorized string operations.

```
df['Name'].str.lower()      # Convert to lowercase  
df['Name'].str.contains('a') # Check if 'a' in string
```

Extracting Substrings

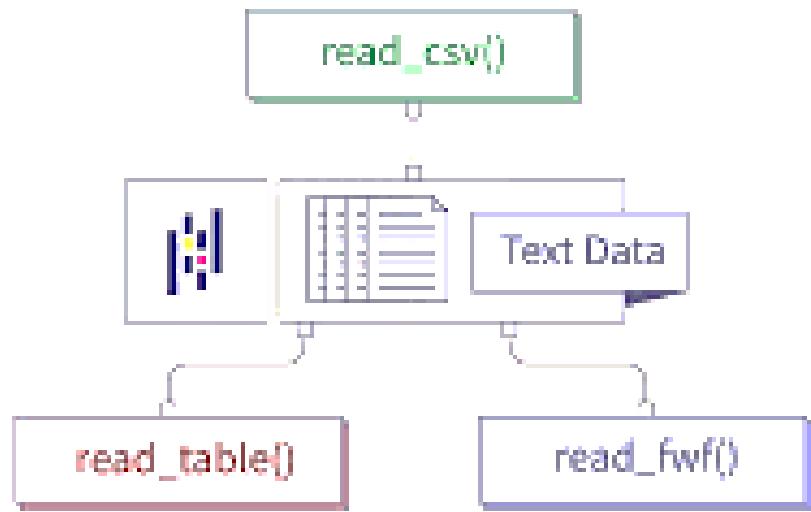
```
df['Initial'] = df['Name'].str[0]
```

Splitting Strings

```
df['Name'].str.split(' ')
```

Replacing Text

```
df['Name'] = df['Name'].str.replace('Alice', 'Alicia')
```



```
read_csv()
```

```
read_table()
```

```
read_fwf()
```

Tip

Text processing in Pandas is efficient and easy with .str methods.

Pivot Tables in Pandas

What is Pivot Tables

- A powerful tool to summarize, aggregate, and analyze data.
- Rearranges data for better readability and insights.

Syntax

```
df.pivot_table(values, index, columns, aggfunc)
```

Parameters

- values: Column(s) to aggregate
- index: Rows
- columns: Columns
- aggfunc: Aggregation function (default is mean)

Dataset

Name	Gender	Age
John	Male	45
Samantha	Female	0
Stephen	Male	4
Joe	Female	38
Emily	Female	12
Tom	Male	43



Pivot Table

Gender	%Gender	Age Group	Count
Male	50%	>18 years	2
		<18 years	1
Female	50%	>18 years	1
		<18 years	2

Crosstabs in Pandas

What is Crosstab?

- A function to compute a simple cross-tabulation of two (or more) factors.
- Used to analyze relationships between categorical variables.

Syntax

```
pd.crosstab(index, columns)
```

Parameters

- index: Series (rows)
- columns: Series (columns)
- Optional: values, aggfunc, margins, etc.

	반	성별
0	A	남
1	A	여
2	A	여
3	B	여
4	B	남
5	B	남



	남	여
A	1	2
B	2	1

```
pd.crosstab(df['반'], c
```

df

Reading & Writing Data with Pandas

Reading Data

CSV files

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

Excel files

```
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
```

JSON files

```
df = pd.read_json('data.json')
```

Writing Data

To CSV

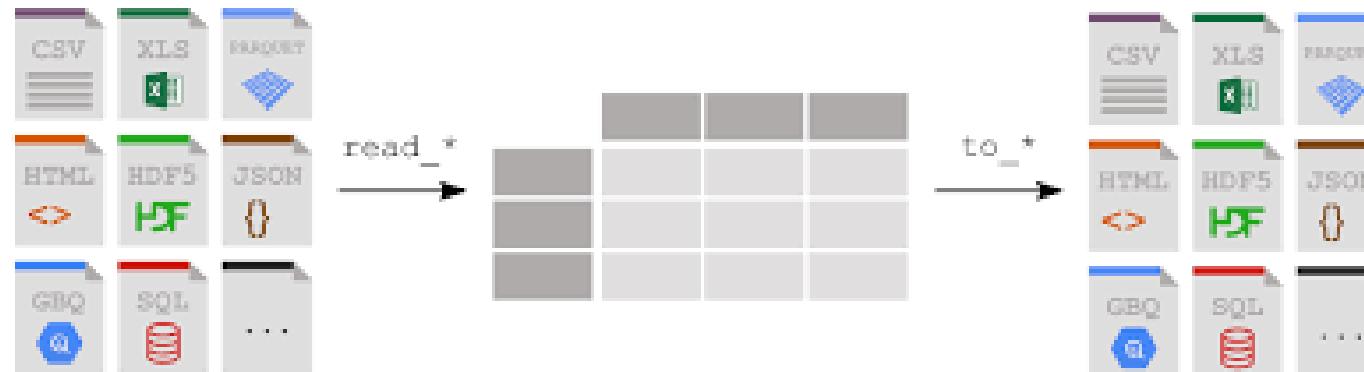
```
df.to_csv('output.csv', index=False)
```

To Excel

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

To JSON

```
Df.to_json('output.json', index=False)
```



Categorical Data in Pandas

What are Categorical Variables?

- Variables with a fixed number of possible values (categories).
- Saves memory and improves performance.

Converting to Category

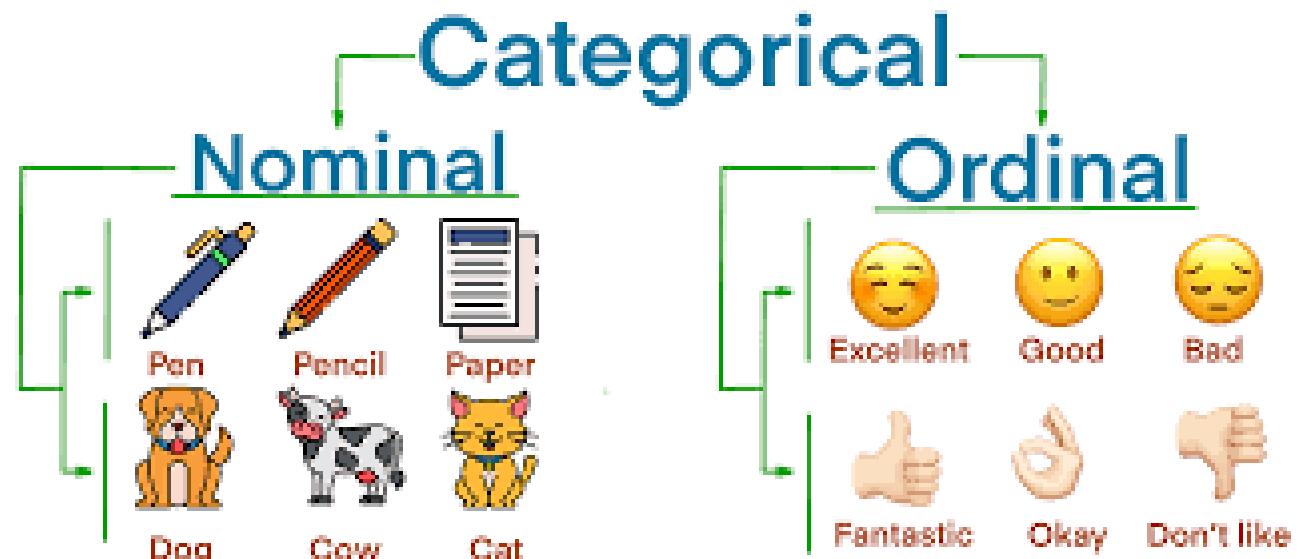
```
df['Gender'] = df['Gender'].astype('category')
```

Benefits

- Efficient storage.
- Faster operations.
- Useful for grouping and analysis.

Categories and Codes

```
print(df['Gender'].cat.categories)  
print(df['Gender'].cat.codes)
```



Tip

Use categorical dtype for columns
with repeated values.

Efficient Function Application in Pandas

Using .apply()

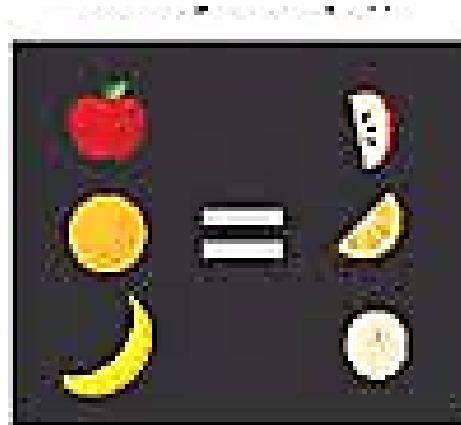
Apply a function along axis of DataFrame or Series.

```
df['Income_K'] = df['Income'].apply(lambda x: x / 1000)
```

Using .map()

Map values of a Series according to an input correspondence.

```
df['AgeGroup'] = df['Age'].map(lambda x: 'Adult' if x >= 18 else  
'Minor')
```



Using Vectorized Operations

Prefer built-in vectorized functions for speed.

Tip

Vectorized and built-in functions outperform custom Python loops.

Optimizing Pandas Performance

Use Vectorized Operations

Avoid loops
use built-in Pandas/NumPy functions.

Use Categorical Data Type

Reduce memory and speed up operations on repeated values.

Downcast Numeric Types

```
df['Age'] = pd.to_numeric(df['Age'], downcast='integer')
```

Use .query() and .eval()

Faster filtering and evaluation.
`df.query('Age > 30 & Income > 50000')`

Avoid Chained Indexing

Use .loc instead for safe and efficient assignment.

Tip

Efficient coding drastically improves speed on large data..



Pandas Integration with Other Libraries



NumPy

Pandas is built on NumPy arrays, enabling fast numerical operations.

```
import numpy as np  
arr = np.array([1, 2, 3])  
df = pd.DataFrame(arr)
```

Matplotlib & Seaborn

For data visualization.

```
df['Age'].plot(kind='hist')  
import seaborn as sns  
sns.boxplot(data=df, x='Age')
```



Pandas works seamlessly with the Python data ecosystem.

Scikit-learn

For machine learning pipelines.



```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test =  
train_test_split(df.drop('target', axis=1), df['target'])
```

SQLAlchemy

Read/write data from SQL databases.



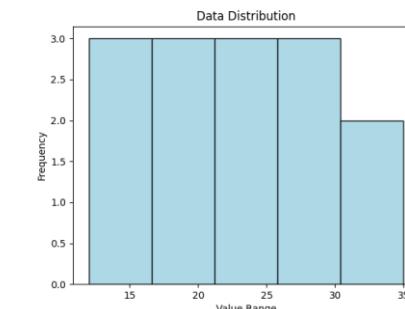
```
from sqlalchemy import create_engine  
engine = create_engine('sqlite:///memory:')  
df.to_sql(name='name', engine)
```

Tip

Data Visualization Using Pandas

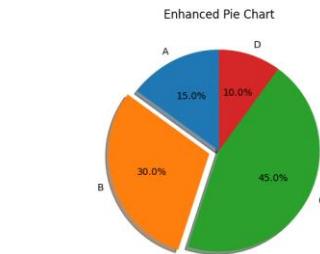
Basic Plotting

```
df['Age'].plot(kind='hist', bins=20, title='Age Distribution')
```



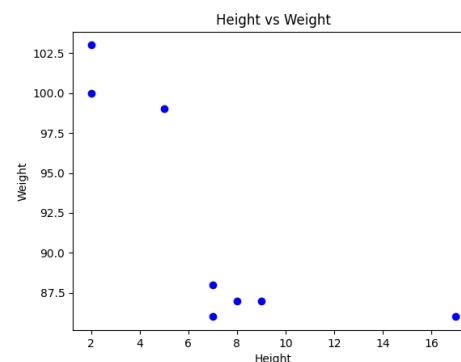
Line Plot

```
df.plot(x='Date', y='Sales', kind='line')
```



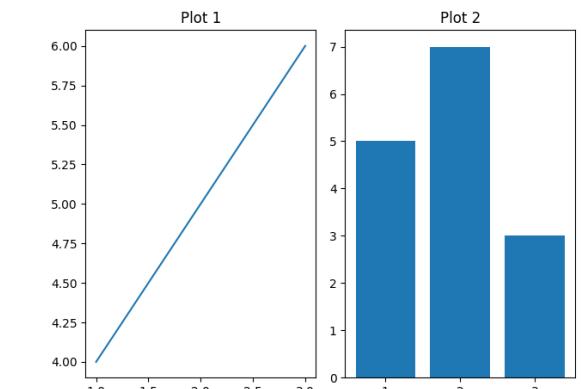
Bar Plot

```
df['Category'].value_counts().plot(kind='bar')
```



Scatter Plot

```
df.plot.scatter(x='Age', y='Income')
```

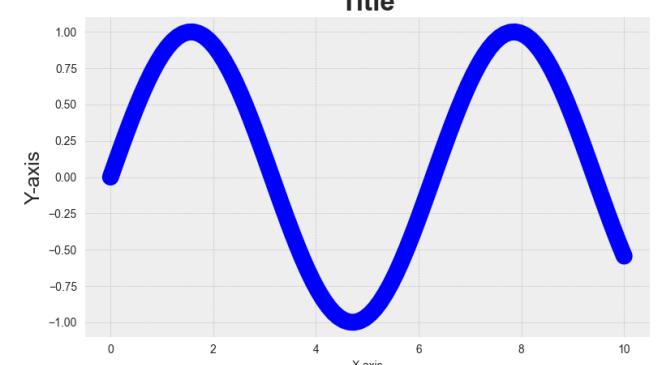


Using matplotlib and seaborn for Enhanced Visualization

```
sns.boxplot(x='Category', y='Income', data=df)
```

Tip

Pandas integrates well with visualization libraries for exploratory analysis.



Instructor : Fawad Bahadur

Exporting Data & Automation in Pandas

Export Data

To CSV

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

To Excel

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

To JSON

```
df.to_json('output.json', orient='records')
```



Converting Between NumPy and Tensors

TensorFlow and PyTorch tensors can be converted to/from NumPy arrays seamlessly.

Tip

Automating pandas workflows saves time and reduces errors.

Automate Tasks with Scripts

- Schedule scripts using cron (Linux) or Task Scheduler (Windows).
- Use Python scripts to automate data cleaning and reporting.

Example: Save filtered data daily

```
filtered = df[df['Age'] > 30]
```

```
filtered.to_csv('filtered_data.csv', index=False)
```



Summary

Summary

Covered Pandas basics to advanced topics:

- Data structures & indexing
- Data cleaning & manipulation
- Grouping, merging & reshaping
- Time series & categorical data
- Performance optimization & visualization
- Integration & automation

Q & A





THANK
YOU