



Pandas

created by :
The easylearn academy

DataFrame

- A **DataFrame** is a two dimensional, tabular data structure with labeled rows and columns, similar to a spreadsheet or SQL table.
- It can be thought of as a collection of Series objects sharing the same index.
- **Key Feature:**
 - **Structure:** Rows and columns, both labeled (index for rows, column names for columns).
 - **Heterogeneous Data:** Each column can have a different data type.
 - **Flexible:** Supports operations like filtering, grouping, merging, and reshaping.
 - **Alignment:** Automatically aligns data based on indices and column names.

Series vs DataFrame

Feature	Series	Dataframe
Definition	A one-dimensional labeled array	A two-dimensional labeled data structure (table)
Structure	Like a single column (or row) of data	Like a full table with rows and columns
Dimensions	1D	2D
Index	Single index	Row and column indexes
Columns	Only one, unnamed or with a name	One or more named columns
Data Type	Homogeneous (same type) usually, but can be mixed	Heterogeneous (each column can be a different type)
Creation Example	<code>pd.Series([10, 20, 30])</code>	<code>pd.DataFrame({'a': [10, 20], 'b': [30, 40]})</code>
Use Case	Ideal for a single column or row of data	Ideal for working with full datasets

Create a sample DataFrame

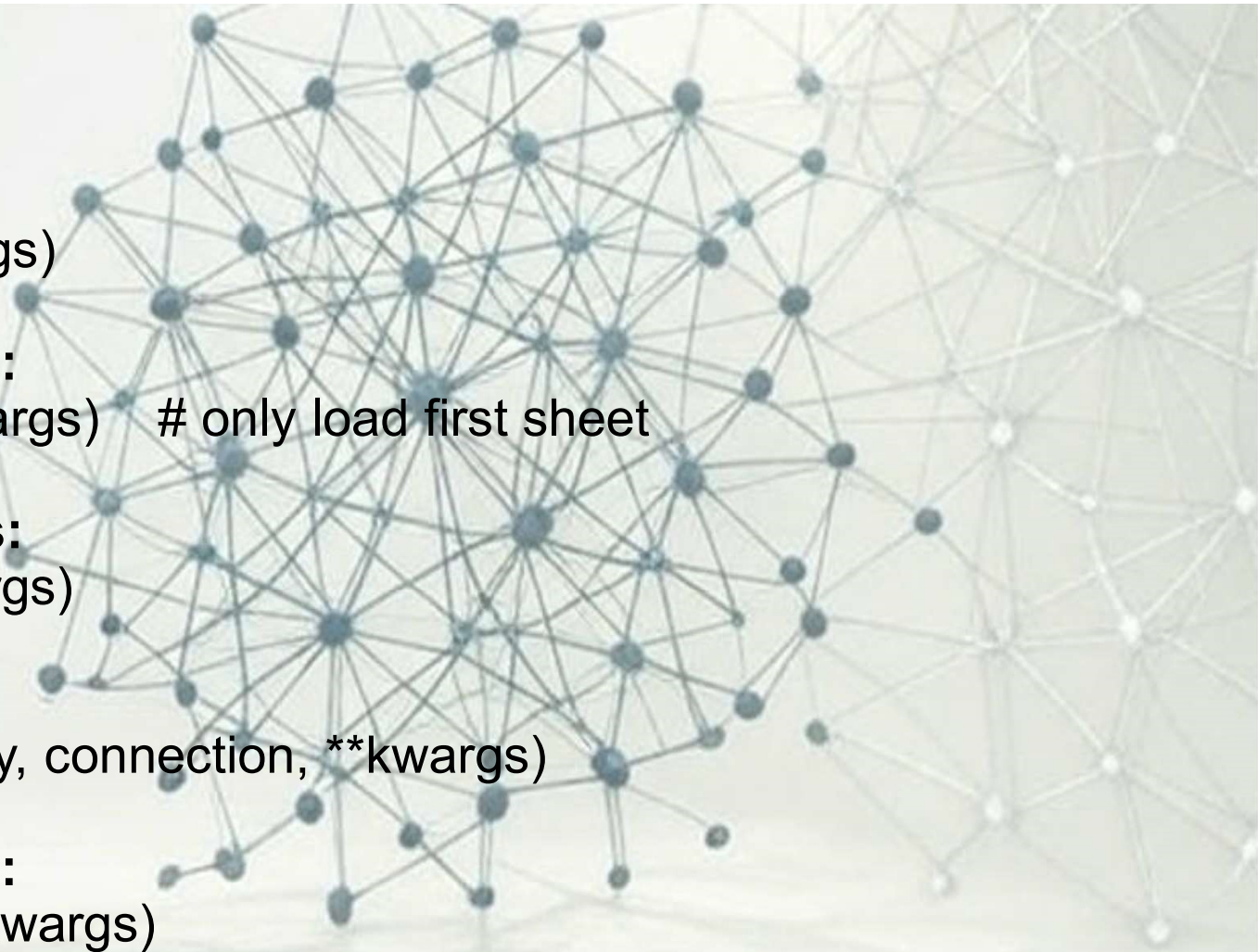
- `import pandas as pd`
- `data = {`
 `'Name': ['Alice', 'Bob', 'Charlie'],`
 `'Age': [25, 30, 35],`
 `'Salary': [50000, 60000, 75000]`
 `}`
- `df = pd.DataFrame(data)`
- `print("Sample DataFrame:")`
- `print(df)`

- **Output:**

	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	75000

Methods for Loading Data:

- **Loading Data from CSV Files**
 - `pd.read_csv(filepath, **kwargs)`
- **Loading Data from Excel Files:**
 - `pd.read_excel(filepath, **kwargs)` # only load first sheet
- **Loading Data from JSON Files:**
 - `pd.read_json(filepath, **kwargs)`
- **Loading Data from SQL:**
 - Databases: `pd.read_sql(query, connection, **kwargs)`
- **Loading Data from HDF5 Files:**
 - `pd.read_hdf(filepath, key, **kwargs)`



Creating DataFrame from Database:

- import pandas as pd
- `df = pd.read_csv(filepath_or_buffer, sep=',', header='infer', names=None, index_col=None, usecols=None, dtype=None, na_values=None, parse_dates=None, encoding=None, ...)`

- **Key parameter:**

- **filepath_or_buffer** (required):

- Specifies the file path (local or URL) or a filelike object (e.g., StringIO) containing the CSV data.

- **Example:**

- `df = pd.read_csv('data.csv')`
 - `df = pd.read_csv('https://example.com/data.csv')`

- **sep**(default: ','):

- Defines the delimiter used in the CSV file (e.g., ',', ';', '\t'). Use this when the file uses a delimiter other than a comma.

- **Example:**

- `df = pd.read_csv('data.txt', sep='\t') # For tabseparated file`

Key parameter of read_csv()

- **Delimiter**(alias for sep):

- Same as sep. Provided for compatibility.

- **Example:**

- `df = pd.read_csv('data.csv', delimiter='|') # For pipe-separated file`

- **header**(default: 'infer'):

- Specifies which row(s) to use as column names.

- **Option:**

- **0**: Use the first row as headers.
- **None**: No header; columns are assigned integer indices (0, 1, 2, ...).
- **List of integers**: Use multiple rows as headers (for MultiIndex).
- **'infer'**: Automatically detect if the first row is a header.

- **Example:**

- `df = pd.read_csv('data.csv', header=None) # No header in the file`

- **Names:**

- List of column names to use. If header=None, this assigns custom column names. If header=0, this overrides the file's header.

- **Example:**

- `df = pd.read_csv('data.csv', header=None, names=['A', 'B', 'C'])`

Key parameter of read_csv()

- **index_col**(default: 'None'):
 - Column(s) to use as the DataFrame's index. Can be a column name, index (integer), or list for MultiIndex.
 - **Example:**
 - `df = pd.read_csv('data.csv', index_col='ID') # Use 'ID' column as index`
- **usecols**(default: 'None'):
 - Specifies a subset of columns to load (by name or index). Reduces memory usage for large files.
 - **Example:**
 - `df = pd.read_csv('data.csv', usecols=['Name', 'Age']) # Load only these columns`
- **dtype**(default: 'None'):
 - Specifies the data type for columns (dictionary or single type). Useful for optimizing memory or ensuring correct types.
 - **Example:**
 - `df = pd.read_csv('data.csv', dtype={'Age': int, 'Score': float})`
- **na_values**(default: 'None'):
 - Defines additional strings to treat as missing values (NaN). Pandas already recognizes values like "", 'NA', 'NaN', etc.
 - **Example:**
 - `df = pd.read_csv('data.csv', na_values=['missing', 'N/A'])`

Key parameter of read_csv()

- **keep_default_na**(default: 'True'):

- If 'False', prevents default NaN values (e.g., 'NA', 'NaN') from being parsed as missing.

- **Example:**

```
df = pd.read_csv('data.csv', na_values=['missing'], keep_default_na=False)
```

- **missing_values**(default: 'None'):

- Deprecated in newer versions; use `na_values` instead.

- **skiprows**(default: 'None'):

- Skips specified rows (integer, list of integers, or callable). Useful for skipping metadata or corrupted rows.

- **Example:**

```
df = pd.read_csv('data.csv', skiprows=2) # Skip first two rows
```

- **nrows**(default: 'None'):

- Limits the number of rows to read. Useful for large files when only a sample is needed.

- **Example:**

```
df = pd.read_csv('data.csv', nrows=100) # Read only 100 rows
```

Key parameter of read_csv()

- **encoding**(default: 'None'):

- Specifies the file encoding (e.g., 'utf-8', 'latin1') for non-standard text files.

- **Example:**

```
df = pd.read_csv('data.csv', encoding='latin1') #encoding="utf-8"
```

- **parse_dates**(default: 'False'):

- Columns to parse as datetime. Can be 'True', a list of column names, or a list of lists for combining columns.

- **Example:**

```
df = pd.read_csv('data.csv', parse_dates=['Date']) # Parse 'Date' as datetime
```

- **date_format**(default: 'None'):

- Specifies the format for parsing dates (used with 'parse_dates').

- **Example:**

```
df = pd.read_csv('data.csv', parse_dates=['Date'], date_format='%Y-%m-%d')
```

- **chunksize**(default: 'None'):

- Reads the file in chunks, returning a 'TextFileReader' object for iteration. Useful for very large files.

- **Example:**

```
for chunk in pd.read_csv('data.csv', chunksize=1000):  
    process_chunk(chunk) # Process 1000 rows at a time
```


Key parameter of read_csv()

- **compression**(default: 'infer'):

- Handles compressed files (e.g., 'gzip', 'bz2', 'zip', 'xz'). If 'infer', detects compression from file extension.

- **Example:**

```
df = pd.read_csv('data.csv.gz', compression='gzip')
```

- **skip_blank_lines**(default: 'True'):

- If 'True', skips blank lines; if 'False', treats them as rows with NaN values.

- **Example:**

```
df = pd.read_csv('data.csv', skip_blank_lines=False)
```

- **low_memory**(default: 'True'):

- Processes the file in chunks internally to save memory. Set to 'False' for faster reading if memory is not a constraint.

- **Example:**

```
df = pd.read_csv('data.csv', low_memory=False)
```


Attributes of DataFrame

Attribute	Syntax	Description
index	df.index	Returns the index (row labels) of the DataFrame.
Values	df.values	Returns the underlying data as a NumPy array.
dtype	df.dtype	Returns the data type of each column.
columns	df.columns	Returns the column labels of the DataFrame.
shape	df.shape	Returns a tuple representing the dimensions of the DataFrame (rows, columns).
ndim	df.ndim	Returns the number of dimensions of the DataFrame.
size	df.size	Returns the total number of elements in the DataFrame (rows × columns).
empty	df.empty	Indicates whether the DataFrame is empty (no rows or columns).
axes	df.axes	Returns a list of the row and column axis labels (`[index, columns]`).
T(Transpose)	df.T	Returns the transpose of the DataFrame (swaps rows and columns).

Index

- **Description:**

- Returns the index (row labels) of the DataFrame.

- **Syntax:**

- `df.index`

- `import pandas as pd`
- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}`
- `df = pd.DataFrame(data)`
- `print(df.index)`
- **Output:**
 - `RangeIndex(start=0, stop=3, step=1)`

Values

- **Description:**

- Returns the underlying data as a NumPy array.

- **Syntax:**

- `df.values`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.values)
```

- **Output:**

- `[['Alice' 25 50000]`
- `['Bob' 30 60000]`
- `['Charlie' 35 75000]]`

Dtype

- **Description:**
 - Returns the data types of each column.
- **Syntax:**
 - `df.dtypes`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.dtypes)
```

- **Output:**
 - Name object
 - Age int64
 - Salary int64
 - dtype: object

columns

- **Description:**

- Returns the column labels of the DataFrame.

- **Syntax:**

- `df.columns`

- `import pandas as pd`
- `data = {`
 - `'Name': ['Alice', 'Bob', 'Charlie'],`
 - `'Age': [25, 30, 35],`
 - `'Salary': [50000, 60000, 75000]``}`
- `df = pd.DataFrame(data)`
- `print(df.columns)`
- **Output:**
 - `Index(['Name', 'Age', 'Salary'], dtype='object')`

Shape

- **Description:**

- Returns the number of dimensions of the DataFrame.

- **Syntax:**

- `df.shape`

- `import pandas as pd`
- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}`
- `df = pd.DataFrame(data)`
- `print(df.shape)`
- **Output:**
 - `(3, 3)`

Ndim

- **Description:**

- Returns the number of dimensions of the DataFrame (always 1 for a DataFrame).

- **Syntax:**

- `df.ndim`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.ndim)
```

- **Output:**

- 2

Size

- **Description:**

- Returns the total number of elements in the DataFrame (rows × columns).

- **Syntax:**

- `df.size`

- `import pandas as pd`
- `data = {`
 - `'Name': ['Alice', 'Bob', 'Charlie'],`
 - `'Age': [25, 30, 35],`
 - `'Salary': [50000, 60000, 75000]``}`
- `df = pd.DataFrame(data)`
- `print(df.size)`
- **Output:**
 - 9

empty

- **Description:**

- Indicates whether the DataFrame is empty (no rows or columns).

- **Syntax:**

- `df.empty`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
```

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

- **Output:**

- `False`

```
• import pandas as pd
• empty_df = pd.DataFrame()
• print(empty_df)
```

- **Output:**

- Empty DataFrame
- Columns: []
- Index: []

axes

- **Description:**

- Returns a list of the row and column axis labels (`[index, columns]`).

- **Syntax:**

- `df.axes`

- `import pandas as pd`
- `data = {`
 - `'Name': ['Alice', 'Bob', 'Charlie'],`
 - `'Age': [25, 30, 35],`
 - `'Salary': [50000, 60000, 75000]``}`
- `df = pd.DataFrame(data)`
- `print(df.axes)`
- **Output:**
 - `[RangeIndex(start=0, stop=3, step=1), Index(['Name', 'Age', 'Salary'], dtype='object')]`

T(Transpose)

- **Description:**

- Returns the transpose of the DataFrame (swaps rows and columns).

- **Syntax:**

- df.T

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.T)
```

- **Output:**

```
•      0    1    2
• Name  Alice  Bob  Charlie
• Age   25   30   35
• Salary 50000 60000 75000
```



Methods of DataFrame

Methods of DataFrame: Data Inspection and Summary

Method	Syntax	Description
DataFrame()	pd.DataFrame()	Create a DataFrame from a dictionary, list, or array.
head(n)	df.head(3)	Returns the first n elements of the DataFrame. (Defaults to 5.)
tail(n)	df.tail(3)	Returns the last n elements of the DataFrame. (Defaults to 5.)
type()	type(df)	Python's builtin type() function to check the type of a Datatable.
describe()	df.describe()	Generates descriptive statistics (count, mean, std, min, quartiles, max) for numeric DataFrame.
info()	df.info()	Provides a summary of the DataFrame, including column names, data types, and non-null counts.
value_counts()	df. value_counts()	Counts unique combinations (typically on Series).

DataFrame()

- **Description:**

- Create a DataFrame from a dictionary, list, or array.

- **Syntax:**

- `pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)`

- **Key Parameters:**

- **data:** Input data (e.g., list, dict, ndarray).
- **index:** Optional index labels (defaults to 0, 1, 2, ...).
- **columns:** The column labels for the DataFrame. Can be a list or array of labels.
- **dtype:** Data type for the Series (e.g., int64, float64).
- **copy:** Whether to copy the input data (default is False).

```
• import pandas as pd
• import numpy as np
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df)
```

- **Output:**

```
•      Name  Age  Salary
•  0  Alice   25   50000
•  1   Bob   30   60000
•  2 Charlie   35   75000
```

head()

- **Description:**

- Returns the **first n elements** of the DataFrame. (Defaults to 5.)

- **Syntax:**

- `df.head(n=5)`

- **Key Parameters:**

- **n**: Number of rows to display (default: 5).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.head())
```

- **Output:**

```
•   Name Age Salary
• 0  Alice  25  50000
• 1   Bob  30  60000
• 2 Charlie  35  75000
```


tail()

- **Description:**

- Returns the last n rows of the DataFrame.

- **Syntax:**

- `df.tail(n=5)`

- **Key Parameters:**

- **n**: Number of rows to display (default: 5).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.tail(2))
```

- **Output:**

- Name Age Salary
- 1 Bob 30 60000
- 2 Charlie 35 75000

type()

- **Description:**

- Python's built-in type() function to check the type of a Series or its elements.

- **Syntax:**

- **type(Series):** Returns the type of the object (pandas.core.frame.DataFrame).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(type(df))
```

- **Output:**

- <class 'pandas.core.frame.DataFrame'>

describe()

- **Description:**

- Generates descriptive statistics (count, mean, std, min, max, quartiles) for numeric columns.

- **Syntax:**

- `df.describe(include='all')`

- **Key Parameters:**

- **include:** Columns to include ('all' for all columns, or specify types like 'np.number' or 'object').

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.describe())
```

- **Output:**

	Age	Salary
• count	3.0	3.000000
• mean	30.0	61666.666667
• std	5.0	12583.057392
• min	25.0	50000.000000
• 25%	27.5	55000.000000
• 50%	30.0	60000.000000
• 75%	32.5	67500.000000
• max	35.0	75000.000000

info()

- **Description:**

- Provides a summary of the DataFrame, including column names, data types, and non-null counts.

- **Syntax:**

- `df.info()`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.info())
```

- **Output:**

- `<class 'pandas.core.frame.DataFrame'>`
- RangeIndex: 3 entries, 0 to 2
- Data columns (total 3 columns):
- # Column Non-Null Count Dtype
- --- ---
- 0 Name 3 non-null object
- 1 Age 3 non-null int64
- 2 Salary 3 non-null int64
- dtypes: int64(2), object(1)
- memory usage: 204.0+ bytes
- None

value_counts()

- **Description:**
 - Returns a Series containing counts of unique values in a DataFrame column or Series. Often used for frequency analysis.
- **Syntax:**
 - `Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)`
- **Key Parameters:**
 - **Normalize:** If True, returns relative frequencies (proportions) instead of counts.
 - **Sort:** If True, sorts by counts in descending order.
 - **Ascending:** If True, sorts in ascending order.
 - **Bins:** Groups numeric data into bins (used for continuous data).
 - **Dropna:** If True, excludes NaN values from counts.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.value_counts())
```

Output:

- Name Age Salary
- Alice 25.0 50000 1
- Bob 30.0 60000 1
- Name: count, dtype: int64

Methods of DataFrame

Method	Syntax	Description
to_csv()	df.to_csv(path, index=True)	Writes the DataFrame to a CSV file.
to_excel()	df.to_excel(path, sheet_name='Sheet1', index=True)	Writes the DataFrame to an Excel file.
read_csv()	pd.read_csv(path, sep=',', encoding='utf-8')	Reads a CSV file into a DataFrame.
read_excel()	pd.read_excel(path, sheet_name=0)	Reads an Excel file into a DataFrame.

to_csv()

- **Description:**

- Writes the DataFrame to a CSV file.

- **Syntax:**

- `df.to_csv(path, index=True)`

- **Key Parameters:**

- **path:** File path or object.
- **index:** If `True`, includes the index.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.to_csv('output.csv')
```

Output:

#open output.csv file to check output

- ,Name,Age,Salary
- 0,Alice,25,50000
- 1,Bob,30,60000
- 2,Charlie,35,75000

to_excel()

- **Description:**
 - Writes the DataFrame to an Excel file.
- **Syntax:**
 - `df.to_excel(path, sheet_name='Sheet1', index=True)`
- **Key Parameters:**
 - **path:** File path.
 - **sheet_name:** Name of the sheet.
 - **index:** If True, includes the index.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.to_excel('output.xlsx')
```

- **Output:** #open output.xlsx file to check output

	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	75000

read_csv()

- **Description:**

- Reads a CSV file into a DataFrame.

- **Syntax:**

- `pd.read_csv(path, sep=',', encoding='utf-8')`

- **Key Parameters:**

- **path:** File path.
- **sep:** Delimiter (default: ',')
- **encoding:** File encoding (e.g., 'utf-8').

- `import pandas as pd`
- `df = pd.read_csv('data.csv')`

- **Output:**

- Load the from data.csv file

read_excel()

- **Description:**

- Reads an Excel file into a DataFrame.

- **Syntax:**

- `pd.read_excel(path, sheet_name=0)`

- **Key Parameters:**

- **path:** File path.
- **sheet_name:** Sheet to read (name or index).

- `import pandas as pd`
- `df = pd.read_excel('data.xlsx')`

Output:

- Load the from exal file

Methods of DataFrame: Data Selection and Filtering

Method	Syntax	Description
loc[]	df.loc[rows, columns]	Access rows and columns by labels or boolean arrays.
iloc[]	df.iloc[rows, columns]	Access rows and columns by integer positions.
at[]	df.at[row_label, column_label]	Fast access to a single value by label.
iat[]	df.iat[row_index, column_index]	Fast access to a single value by integer position.
query(expr)	df.query(expr)	Query the DataFrame using a boolean expression.
filter(items/like/regex)	df.filter(items=None, like=None, regex=None, axis=0)	Subset columns based on names or patterns.
select_dtypes(include/exclude)	df.select_dtypes(include=None, exclude=None)	Select columns by data type.
duplicated()	df.duplicated(subset=None, keep='first')	Identifies duplicate rows.
drop_duplicates()	df.drop_duplicates(subset=None, keep='first', inplace=False, ignore_index=False)	Removes duplicate rows.

loc[]

- **Description:**

- Access rows and columns by labels or boolean arrays.

- **Syntax:**

- `df.loc[rows, columns]`

- **Key Parameters:**

- **rows:** Row labels or boolean array.
- **columns:** Column labels or list of labels.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.loc[df['Age'] > 30, ['Name', 'Age']])
```

- **Output:**

- Name Age
- 2 Charlie 35

iloc[]

- **Description:**
 - Access rows and columns by integer positions.
- **Syntax:**
 - `df.iloc[rows, columns]`
- **Key Parameters:**
 - **rows:** Integer indices or slice.
 - **columns:** Integer indices or slice.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.iloc[0:3, 1:3]
```

```
• Output:
•      Age  Salary
•  0   25   50000
•  1   30   60000
•  2   35  750000
```

at[]

- **Description:**

- Access a single value by row and column label (faster than `loc`).

- **Syntax:**

- `df.at[row_label, column_label]`

- **Key Parameters:**

- **row_label:** Row label.
- **column_label:** Column label.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.at[0, 'Name'])
```

- **Output:**

- 'Alice'

iat[]

- **Description:**

- Access a single value by integer position (faster than `iloc`)

- **Syntax:**

- `df.iat[row_index, column_index]`

- **Key Parameters:**

- **row_index:** Integer row position.
- **column_index:** Integer column position.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.iat[0, 1])
```

- **Output:**

- 25

query()

- **Description:**

- Filters rows using a string expression.

- **Syntax:**

- `df.query(expr)`

- **Key Parameters:**

- **expr:** String expression (e.g., 'age > 30').

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.query('Age > 30 and Salary == 75000'))
```

- **Output:**

- Name Age Salary
- 2 Charlie 35 75000

filter(items/like/regex)

- **Description:**

- Subset the DataFrame rows or columns based on the specified index or column labels.

- **Syntax:**

- `df.filter(items=None, like=None, regex=None, axis=0)`

- **Key Parameters:**

- **items:** List of column or index labels to select (exact matches).
- **like:** String to match in column or index labels (partial matches).
- **regex:** Regular expression to match in column or index labels.
- **axis:** Axis to filter on (0 for index, 1 for columns; default is 0).

filter(items/like/regex)

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, np.nan],
 'Salary': [50000, 60000, 75000]
}
- df = pd.DataFrame(data)
- print(df.filter(items=['Age'], axis=1))
 - **or**
- print(df.filter(items=[0], axis=0))

- **Output:**

- Age
- 0 25.0
- 1 30.0
- 2 NaN

- **or**

- **Output:**

- Name Age Salary
- 0 Alice 25.0 50000

select_dtypes(include/exclude)

- **Description:**

- Select columns from a DataFrame based on specified data types.

- **Syntax:**

- `df.select_dtypes(include=None, exclude=None)`

- **Key Parameters:**

- **include:** Data types to include (e.g., 'int64', 'float64', 'object', or numpy dtype).
- **exclude:** Data types to exclude (e.g., 'int64', 'float64', 'object', or numpy dtype).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, np.nan],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.select_dtypes(include=int, exclude=None))
```

- **Output:**

- Salary
- 0 50000
- 1 60000
- 2 75000

deduplicated()

- **Description:**

- Identifies duplicate rows in a DataFrame and returns a boolean Series where True indicates a duplicate row.

- **Syntax:**

- `df.duplicated(subset=None, keep='first')`

- **Key Parameters:**

- **subset:** Column label(s) to consider for identifying duplicates (default: None, uses all columns).
- **keep:** 'first': Mark duplicates as True except for the first occurrence (default).
- **last:** Mark duplicates as True except for the last occurrence.

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

- **Output:**

- 0 False
- 1 False
- 2 False
- 3 True
- 4 True
- dtype: bool

drop_duplicates()

- **Description:**

- Removes duplicate rows from a DataFrame and returns a new DataFrame with unique rows.

- **Syntax:**

- `df.drop_duplicates(subset=None, keep='first', inplace=False, ignore_index=False)`

- **Key Parameters:**

- **subset:** Column label(s) to consider for identifying duplicates (default: None, uses all columns).
- **keep:** **'first'**: Keep the first occurrence of each duplicate (default). **'last'**: Keep the last occurrence of each duplicate. **False**: Drop all duplicates.
- **inplace:** If True, modifies the DataFrame in place (default: False).
- **ignore_index:** If True, resets the index after dropping duplicates

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
```

```
• print(df.drop_duplicates(keep='first'))
```

- **Output:**

	Name	Age	Salary
• 0	Alice	25	50000
• 1	Bob	30	60000
• 2	Charlie	35	70000

Methods of DataFrame: Data Manipulation

Method	Syntax	Description
<code>copy()</code>	<code>DataFrame.copy(deep=True)</code>	Creates a deep copy of the DataFrame.
<code>assign(**kwargs)</code>	<code>DataFrame.assign(**kwargs)</code>	Adds new columns or modifies existing ones.
<code>pop(column)</code>	<code>DataFrame.pop(item)</code>	Removes and returns a column.
<code>drop(labels, axis)</code>	<code>df.drop(labels, axis=0, inplace=False)</code>	Drops specified rows or columns.
<code>rename(columns/index)</code>	<code>df.rename(columns=None, index=None, inplace=False)</code>	Renames columns or index labels.
<code>replace(to_replace, value)</code>	<code>df.replace(to_replace, value, inplace=False)</code>	Replaces values in the DataFrame.
<code>astype(dtype)</code>	<code>df.astype(dtype)</code>	Converts data types of columns.
<code>mask(cond, other)</code>	<code>DataFrame.mask(cond, other=None, inplace=False)</code>	Replaces values where condition is True.

copy()

- **Description:**

- Creates a deep or shallow copy of the DataFrame.

- **Syntax:**

- df.copy(deep=True)

- **Key Parameters:**

- **Deep:** If True, creates a deep copy (default). If False, creates a shallow copy.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• a=df.copy()
• print(a)
```

- **Output:**

```
•   Name Age Salary
• 0  Alice  25  50000
• 1   Bob  30  60000
• 2 Charlie  35  75000
```

assign()

- **Description:**

- Adds new columns or modifies existing ones in a DataFrame, returning a new DataFrame.

- **Syntax:**

- `df.assign(**kwargs)`

- **Key Parameters:**

- **Kwargs:** Column names and values (can be scalars, lists, or functions).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.assign(number=102106478))
```

- **Output:**

```
•      Name Age Salary  number
•  0  Alice  25  50000  102106478
•  1   Bob  30  60000  102106478
•  2 Charlie  35  75000  102106478
```


pop()

- **Description:**

- Removes and returns a column from the DataFrame.

- **Syntax:**

- `df.pop(item)`

- **Key Parameters:**

- **Item:** Label of the column to remove.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
    'number':[102106478, 102106478, 102106478]
}
• df = pd.DataFrame(data)
• df.pop('number')
• print(df)
```

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

drop()

- **Description:**

- Removes specified rows or columns.

- **Syntax:**

- `df.drop(labels, axis=0, inplace=False)`

- **Key Parameters:**

- **labels:** Row or column labels to drop.
- **axis:** 0 for rows, 1 for columns.
- **inplace:** If `True`, modifies the DataFrame in place.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.drop('Age',axis=1,inplace=True)
• print(df)
```

- **Output:**

```
•      Name  Salary
•  0  Alice  50000
•  1   Bob  60000
•  2 Charlie  75000
```

rename()

- **Description:**

- Renames columns or index labels.

- **Syntax:**

- `df.rename(columns=None, index=None, inplace=False)`

- **Key Parameters:**

- **columns:** Dict of old to new column names.
- **index:** Dict of old to new index labels.
- **inplace:** If `True`, modifies in place.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.rename(columns={'Age': 'age'}))
```

- **Output:**

```
•      Name age Salary
• 0  Alice  25  50000
• 1   Bob  30  60000
• 2 Charlie  35  75000
```


replace()

- **Description:**
 - Replaces values in the DataFrame.
- **Syntax:**
 - `df.replace(to_replace, value, inplace=False)`
- **Key Parameters:**
 - **to_replace:** Value(s) to replace (scalar, list, dict).
 - **value:** Replacement value(s).
 - **inplace:** If `True`, modifies in place.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.replace(25,20))
```

• **Output:**

	Name	Age	Salary
• 0	Alice	20	50000
• 1	Bob	30	60000
• 2	Charlie	35	75000

astype()

- **Description:**
 - Converts data types of columns.
- **Syntax:**
 - `df.astype(dtype)`
- **Key Parameters:**
 - **dtype:** Data type or dict of column-to-type mappings.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• print(df.astype('float64'))
```

```
• Output:
• 0    25.0
• 1    30.0
• 2    35.0
• Name: Age, dtype: float64
```

mask()

- **Description:**

- Replaces values where a condition is True with a specified value.

- **Syntax:**

- `df.mask(cond, other=None, inplace=False)`

- **Key Parameters:**

- **Cond:** Boolean condition or callable.
- **Other:** Value to replace where condition is True.
- **Inplace:** If True, modifies the DataFrame in place.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• mask_df = df.mask(df['Age'] < 28)
• print(mask_df)
```

- **Output:**

- Name Age Salary
- 0 NaN NaN NaN
- 1 Bob 30.0 60000.0
- 2 Charlie 35.0 75000.0

Methods of DataFrame: Data Manipulation & Sorting and Ranking

Method	Syntax	Description
<code>insert(loc, column, value)</code>	<code>df.insert(loc, column, value, allow_duplicates=False)</code>	Inserts a new column at a specific position.
<code>fillna(value/method)</code>	<code>df.fillna(value=None, method=None, inplace=False)</code>	Fills missing values with a value or method (e.g., ffill, bfill).
<code>dropna(axis, how, subset)</code>	<code>df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)</code>	Drops rows or columns with missing values.
<code>truncate(before, after, axis)</code>	<code>df.truncate(before=None, after=None, axis=0, copy=True)</code>	Truncates a DataFrame before and after some index or columns.
<code>convert_dtypes()</code>	<code>df.convert_dtypes()</code>	Converts columns to best possible data types.
<code>sort_values(by, ascending)</code>	<code>df.sort_values(by, ascending=True, inplace=False)</code>	Sorts by values in specified columns.
<code>sort_index(ascending)</code>	<code>df.sort_index(axis=0, ascending=True, inplace=False)</code>	Sorts by index.
<code>rank()</code>	<code>df.rank(axis=0, method='average')</code>	Computes numerical data ranks (1 through n).

insert()

- **Description:**

- Inserts a column at a specified position in the DataFrame.

- **Syntax:**

- `df.insert(loc, column, value, allow_duplicates=False)`

- **Key Parameters:**

- **loc:** Integer index where the column will be inserted.
- **column:** Name of the new column.
- **value:** Values for the column (scalar, list, or array).
- **allow_duplicates:** If True, allows inserting a column with a name that already exists.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.insert(2, 'subject', 'science', allow_duplicates=False)
```

```
• print(df)
• Output:
    Name  Age  subject  Salary
0  Alice   25   science  50000
1   Bob    30   science  60000
2 Charlie   35   science  75000
```

fillna()

- **Description:**

- Fills missing values with a specified value or method.

- **Syntax:**

- `df.fillna(value=None, method=None, inplace=False)`

- **Key Parameters:**

- **value:** Value to fill NA (scalar or dict).
- **method:** 'ffill' or 'bfill' to propagate non-null values.
- **inplace:** If `True`, modifies in place.

```
• import pandas as pd
• import numpy as np
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, np.nan],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• df.fillna(0)
```

- `print(df)`

- **Output:**

- | | Name | Age | Salary |
|---|---------|------|--------|
| 0 | Alice | 25.0 | 50000 |
| 1 | Bob | 30.0 | 60000 |
| 2 | Charlie | 0.0 | 75000 |

dropna()

- **Description:**

- Removes rows or columns containing missing values (NaN).

- **Syntax:**

- `df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **how:** 'any' (drop if any NaN), 'all' (drop if all NaN).
- **thresh:** Minimum number of non-NaN values required to keep the row/column.
- **subset:** Columns to consider for NaN checks.
- **inplace:** If True, modifies the DataFrame in place.

```
• import pandas as pd
• import numpy as np
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, np.nan],
    'Salary': [50000, 60000, 75000]
}
```

```
• df = pd.DataFrame(data)
• print(df.dropna(axis=0 , inplace=False))
```

- **Output:**

	Name	Age	Salary
0	Alice	25.0	50000
1	Bob	30.0	60000

truncate()

- **Description:**

- Truncates a DataFrame before and/or after specified indices.

- **Syntax:**

- `df.truncate(before=None, after=None, axis=0, copy=True)`

- **Key Parameters:**

- **before:** Index to start truncation.
- **after:** Index to end truncation.
- **axis:** 0 for rows, 1 for columns.
- **copy:** If True, returns a copy.

- `import pandas as pd`

- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}`

- `df = pd.DataFrame(data)`

- `df = df.set_index('Name') # Set 'Name' as index`
- `result = df.truncate(after='Bob', axis=0, copy=True)`

- `print(result)`

- **Output:**

- Age Salary
- Name
- Alice 25 50000
- Bob 30 60000

convert_dtypes()

- **Description:**

- Converts columns to the best possible dtypes using pandas nullable data types.

- **Syntax:**

- df.convert_dtypes()

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
```

```
• df = pd.DataFrame(data)
• df_converted = df.convert_dtypes()
• print(df_converted.dtypes)
```

Or

```
• print(df_converted)
```

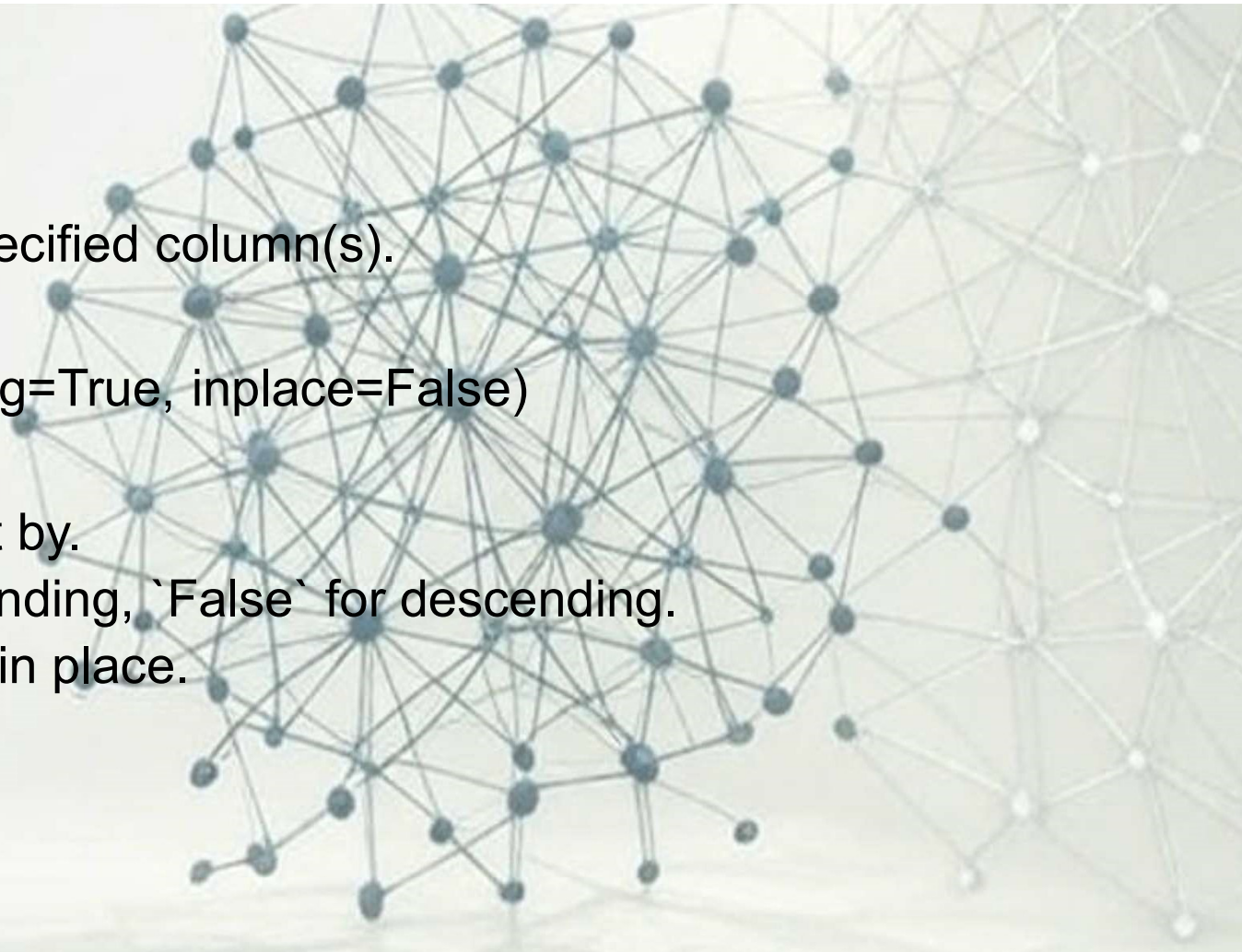
- **Output:**

- Name string[python]
- Age Int64
- Salary Int64
- dtype: object

Or

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

sort_values()

- **Description:**
 - Sorts the DataFrame by specified column(s).
 - **Syntax:**
 - `df.sort_values(by, ascending=True, inplace=False)`
 - **Key Parameters:**
 - **by:** Column name(s) to sort by.
 - **ascending:** `True` for ascending, `False` for descending.
 - **inplace:** If `True`, modifies in place.
- 

sort_values()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'Bob'],
 'Age': [25, 30, 35, 25, 30],
 'Salary': [50000, 60000, 70000, 60000, 70000]
}
- df = pd.DataFrame(data)
- df_sorted = df.sort_values(by='Age', ascending=True, inplace=False)
- print(df_sorted)

Or

- df_multiple_sort = df.sort_values(by=['Age', 'Salary'], ascending=[True, False], inplace=False)
- print(df_multiple_sort)

• Output:

- Name Age Salary
- 0 Alice 25 50000
- 3 Alice 25 60000
- 1 Bob 30 60000
- 4 Bob 30 70000
- 2 Charlie 35 70000

Or

- Name Age Salary
- 3 Alice 25 60000
- 0 Alice 25 50000
- 4 Bob 30 70000
- 1 Bob 30 60000
- 2 Charlie 35 70000

sort_index()

- **Description:**

- Sorts the DataFrame by index

- **Syntax:**

- `df.sort_index(axis=0, ascending=True, inplace=False)`

- **Key Parameters:**

- **axis:** 0 for index, 1 for columns.
- **ascending:** Sort order.
- **inplace:** If `True`, modifies in place.

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

```
• df_sorted = df.sort_index()
• print(df_sorted)
```

- **Output:**

	Age	Salary
Name		# this is index
Alice	25	50000
Alice	25	60000
Bob	30	60000
Bob	30	70000
Charlie	35	70000

rank()

- **Description:**

- Computes numerical ranks (1 through n) for values in each column or row.

- **Syntax:**

- `df.rank(axis=0, method='average', numeric_only=None, na_option='keep', ascending=True, pct=False)`

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.
- **method:** 'average', 'min', 'max', 'first', 'dense' for handling ties.
- **na_option:** 'keep' (keep NaN), 'top', 'bottom'.
- **ascending:** If True, ranks in ascending order.
- **pct:** If True, returns percentile ranks.

- `import pandas as pd`

- `data = {`

- `'Name': ['Alice', 'Bob', 'Charlie'],`

- `'Age': [25, 30, 35],`

- `'Salary': [50000, 60000, 75000]`

- `}`

- `df = pd.DataFrame(data)`

- `df['Salary_Rank'] = df['Salary'].rank(method = 'average', ascending = False)`

- `print(df)`

- **Output:**

- | | Name | Age | Salary | Salary_Rank |
|---|---------|-----|--------|-------------|
| 0 | Alice | 25 | 50000 | 3.0 |
| 1 | Bob | 30 | 60000 | 2.0 |
| 2 | Charlie | 35 | 75000 | 1.0 |

Methods of DataFrame: Grouping and Aggregation

Method	Syntax	Description
agg(func)	df.agg(func)	Aggregates using one or more operations (e.g., mean, sum).
aggregate(func)	df.agg(func)	Alias for `agg`.
groupby(by, axis)	df.groupby(by, axis=0)	Groups data by specified columns for aggregation.
melt(id_vars, value_vars)	df.melt(id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None)	Unpivots a DataFrame from wide to long format.
pivot(index, columns, values)	df.pivot(index=None, columns=None, values=None)	Creates a pivot table.
pivot_table(index, columns, values, aggfunc)	df.pivot_table(values, index, columns=None, aggfunc='mean')	Creates a pivot table with aggregation.
transform(func)	df.transform(func, axis=0, *args, **kwargs)	Applies a function to each group and returns transformed values.

agg()

- **Description:**

- Applies aggregation functions to grouped data.

- **Syntax:**

- `df.agg(func)`

- **Key Parameters:**

- **func:** Function(s) (e.g., 'mean', 'sum', or custom function).

- `import pandas as pd`

- `data = { 'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Salary': [50000, 60000, 75000] }`

- `df = pd.DataFrame(data) # Creating dataframe`

- `print(df) # Use agg() to calculate average and total salary for the Salary column`

- `agg_results = df['Salary'].agg(['mean', 'sum'])`

- `agg_results = agg_results.rename(index={'mean': 'Average_Salary', 'sum': 'Total_Salary'})`

- `print(agg_results) # Rename the columns for better readability`

- **Output:**
Average_Salary 61666.666667
Total_Salary 185000.000000
Name: Salary, dtype: float64

- `agg_results = df.agg({
 'Salary': ['mean', 'sum'], # Calculate mean and sum of Salary
 'Age': ['mean', 'count'] # Calculate mean age and count of employees
})`
- `print("\nAggregated Results:")`
- `print(agg_results)`

- Aggregated Results:
 -
 -
 -
 -
 -
- | | Salary | Age |
|-------|---------------|------|
| mean | 61666.666667 | 30.0 |
| sum | 185000.000000 | NaN |
| count | NaN | 3.0 |

- `import pandas as pd`
- `data = { 'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Salary': [50000, 60000, 75000] }`
- `df = pd.DataFrame(data)`
- `print(df)`
- `agg_results = df.agg({ 'Salary': ['mean', 'sum'], 'Age': ['mean', 'count'] })`
- `agg_flat = {`
- `'Average_Salary': agg_results.loc['mean', 'Salary'], 'Total_Salary': agg_results.loc['sum', 'Salary'],`
- `'Average_Age': agg_results.loc['mean', 'Age'], 'Employee_Count': agg_results.loc['count', 'Age'] }`
- `agg_df = pd.DataFrame([agg_flat])`
- `print(agg_df)`

- **Output: #original**
- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | Alice | 25 | 50000 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |

- **Output: # after Aggregation**
- | | Average_Salary | Total_Salary | Average_Age | Employee_Count |
|---|----------------|--------------|-------------|----------------|
| 0 | 61666.666667 | 185000.0 | 30.0 | 3.0 |

aggregate()

- **Description:**

- Applies aggregation functions to grouped data.

- **Syntax:**

- `df.agg(func)`

- **Key Parameters:**

- **func:** Function(s) (e.g., 'mean', 'sum', or custom function).

- `import pandas as pd`
- `data = {`
 `'Name': ['Alice', 'Bob', 'Charlie'],`
 `'Age': [25, 30, 35],`
 `'Salary': [50000, 60000, 75000]`
 `}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **For use case check `agg()` above example**

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

groupby()

- **Description:**

- Groups data by column(s) for aggregation

- **Syntax:**

- `df.groupby(by, axis=0)`

- **Key Parameters:**

- **by:** Column(s) or function to group by.
- **axis:** Axis to group along (default: 0).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Age': [25, 30, 35, 40, 28, 33],
    'Salary': [50000, 60000, 75000, 58000, 62000, 72000],
    'Department': ['HR', 'Finance', 'Finance', 'HR', 'IT', 'IT']
}
• df = pd.DataFrame(data)
• print(df)
```

- **Output:**

	Name	Age	Salary	Department
• 0	Alice	25	50000	HR
• 1	Bob	30	60000	Finance
• 2	Charlie	35	75000	Finance
• 3	David	40	58000	HR
• 4	Eva	28	62000	IT
• 5	Frank	33	72000	IT

- grouped = df.groupby('Department').agg({
 'Salary': 'mean', 'Age': 'mean', 'Name': 'count' **# count of employees per department**
 }).rename(columns={'Salary': 'Average_Salary', 'Age': 'Average_Age', 'Name': 'Employee_Count'})
- print(grouped)
- salary_sums = df.groupby('Department')['Salary'].sum().sort_values(ascending=False)
- print("\nTotal Salary by Department:")
- print(salary_sums)

	Average_Salary	Average_Age	Employee_Count
--	----------------	-------------	----------------

Department

- | | | | |
|-----------|---------|------|---|
| • Finance | 67500.0 | 32.5 | 2 |
| • HR | 54000.0 | 32.5 | 2 |
| • IT | 67000.0 | 30.5 | 2 |

• Department #Total Salary by Department:

- | | |
|-----------|--------|
| • Finance | 135000 |
| • IT | 134000 |
| • HR | 108000 |
- Name: Salary, dtype: int64

melt()

- **Description:**

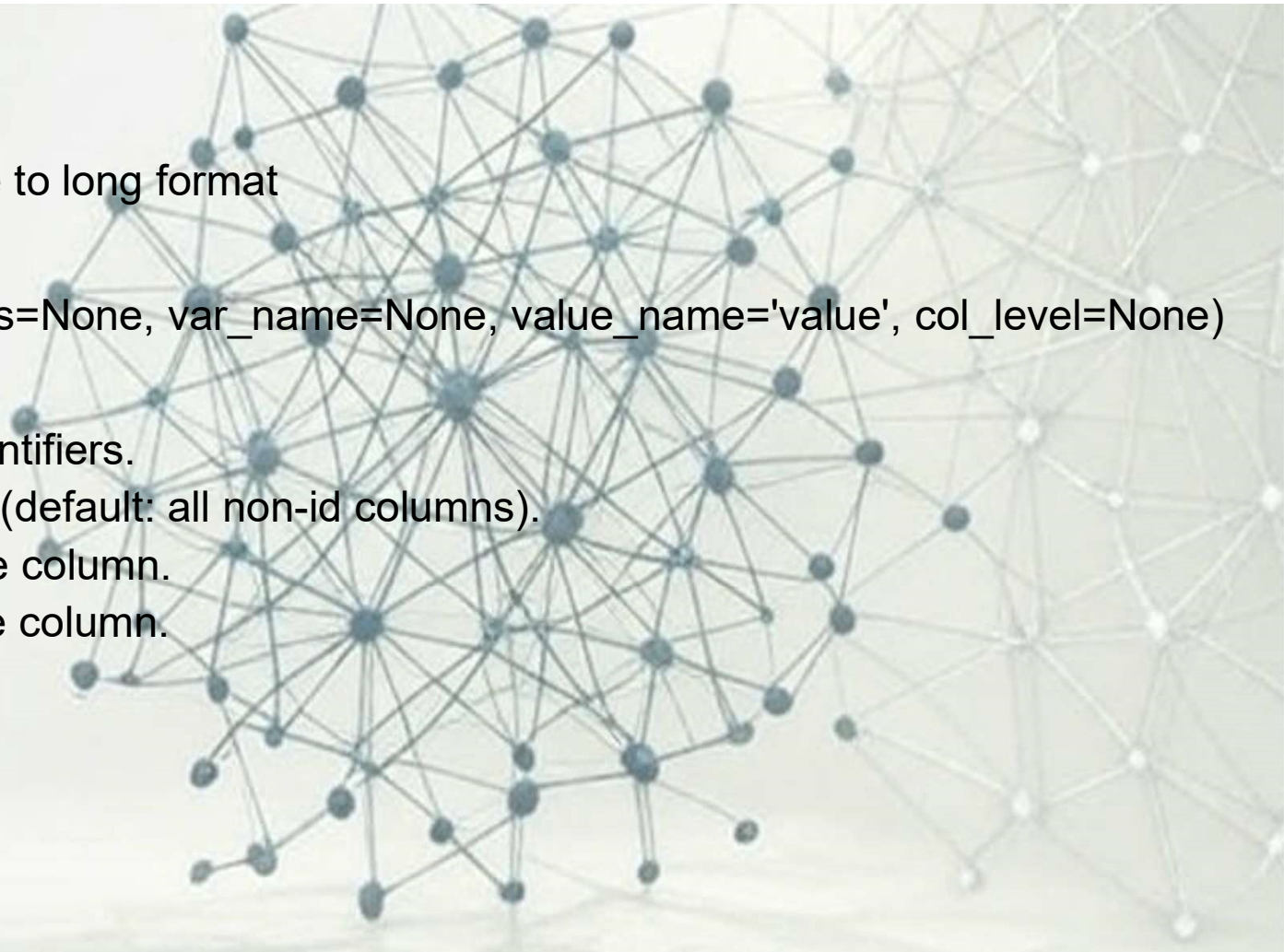
- Unpivots a DataFrame from wide to long format

- **Syntax:**

- `df.melt(id_vars=None, value_vars=None, var_name=None, value_name='value', col_level=None)`

- **Key Parameters:**

- **id_vars:** Columns to keep as identifiers.
- **value_vars:** Columns to unpivot (default: all non-id columns).
- **var_name:** Name for the variable column.
- **value_name:** Name for the value column.



- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}
- df = pd.DataFrame(data)
- print("Original DataFrame:")
- print(df)
- **#Melt the DataFrame to long format**
- df_melted = df.melt(id_vars='Name',
 var_name='Attribute', value_name='Value')
- print("\nMelted DataFrame:")
- print(df_melted)

- **Output:**

- Original DataFrame:

	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	75000
- Melted DataFrame:

	Name	Attribute	Value
0	Alice	Age	25
1	Bob	Age	30
2	Charlie	Age	35
3	Alice	Salary	50000
4	Bob	Salary	60000
5	Charlie	Salary	75000

pivot()

- **Description:**

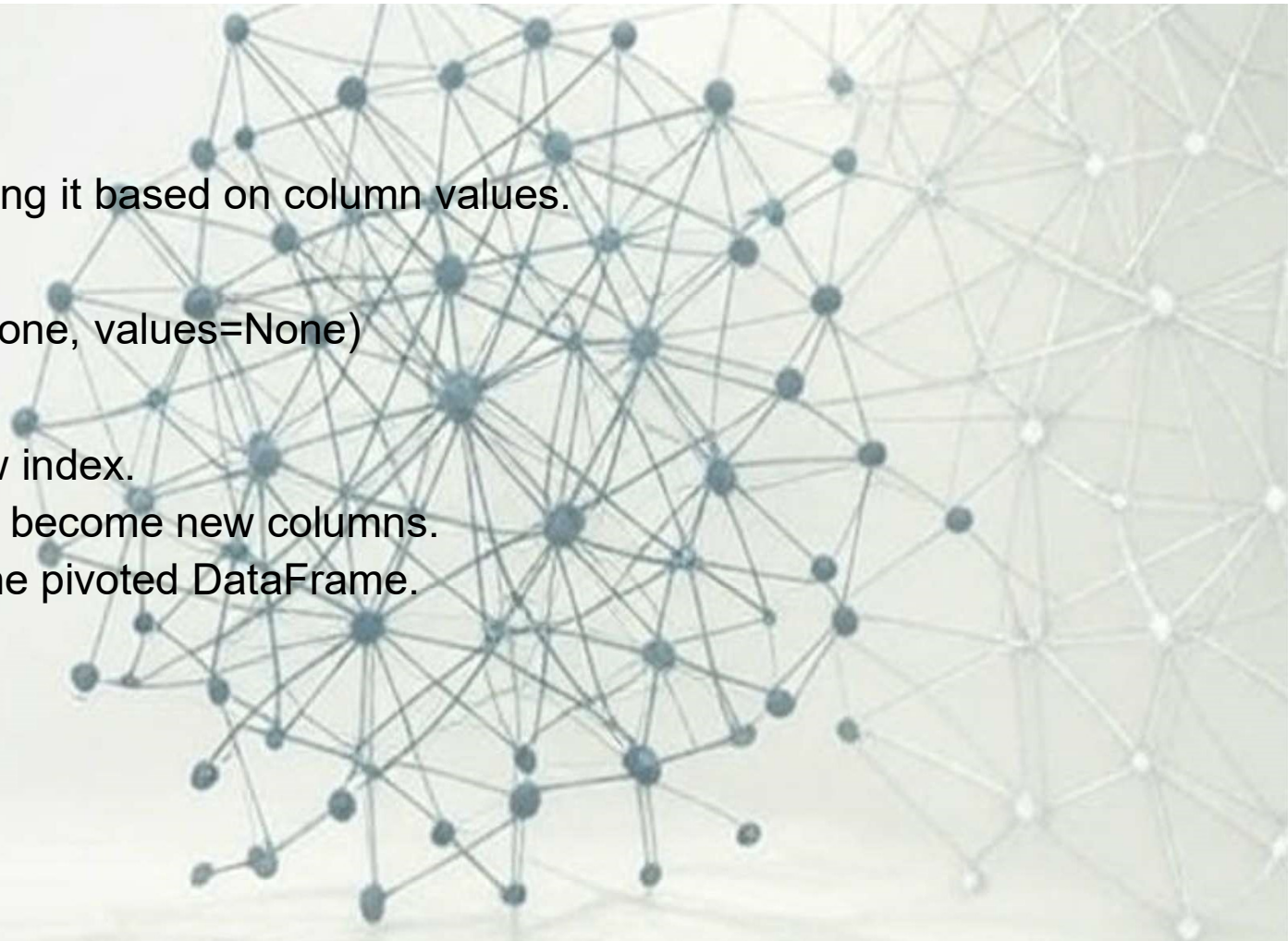
- Reshapes a DataFrame by pivoting it based on column values.

- **Syntax:**

- `df.pivot(index=None, columns=None, values=None)`

- **Key Parameters:**

- **index:** Column to use as the new index.
- **columns:** Column whose values become new columns.
- **values:** Column(s) to populate the pivoted DataFrame.



- import pandas as pd
- data = {
 - 'Name': ['Alice', 'Bob', 'Charlie'],
 - 'Age': [25, 30, 35],
 - 'Salary': [50000, 60000, 75000]
- df = pd.DataFrame(data)
- print(df)
- **# Melt the DataFrame to long format**
- df_melted = df.melt(id_vars='Name',
 - var_name='Attribute', value_name='Value')
- print("Melted DataFrame:")
- print(df_melted)
- **# Pivot the melted DataFrame back to wide format**
- df_pivoted = df_melted.pivot(index='Name',
 - columns='Attribute', values='Value')
- print("\nPivoted DataFrame:")
- print(df_pivoted)

- **Output:**
 - Name Age Salary
 - 0 Alice 25 50000
 - 1 Bob 30 60000
 - 2 Charlie 35 75000
 - **Melted DataFrame:**
 - Name Attribute Value
 - 0 Alice Age 25
 - 1 Bob Age 30
 - 2 Charlie Age 35
 - 3 Alice Salary 50000
 - 4 Bob Salary 60000
 - 5 Charlie Salary 75000
 - **Pivoted DataFrame:**
 - Attribute Age Salary
 - **Name**
 - Alice 25 50000
 - Bob 30 60000
 - Charlie 35 75000

pivot_table()

- **Description:**

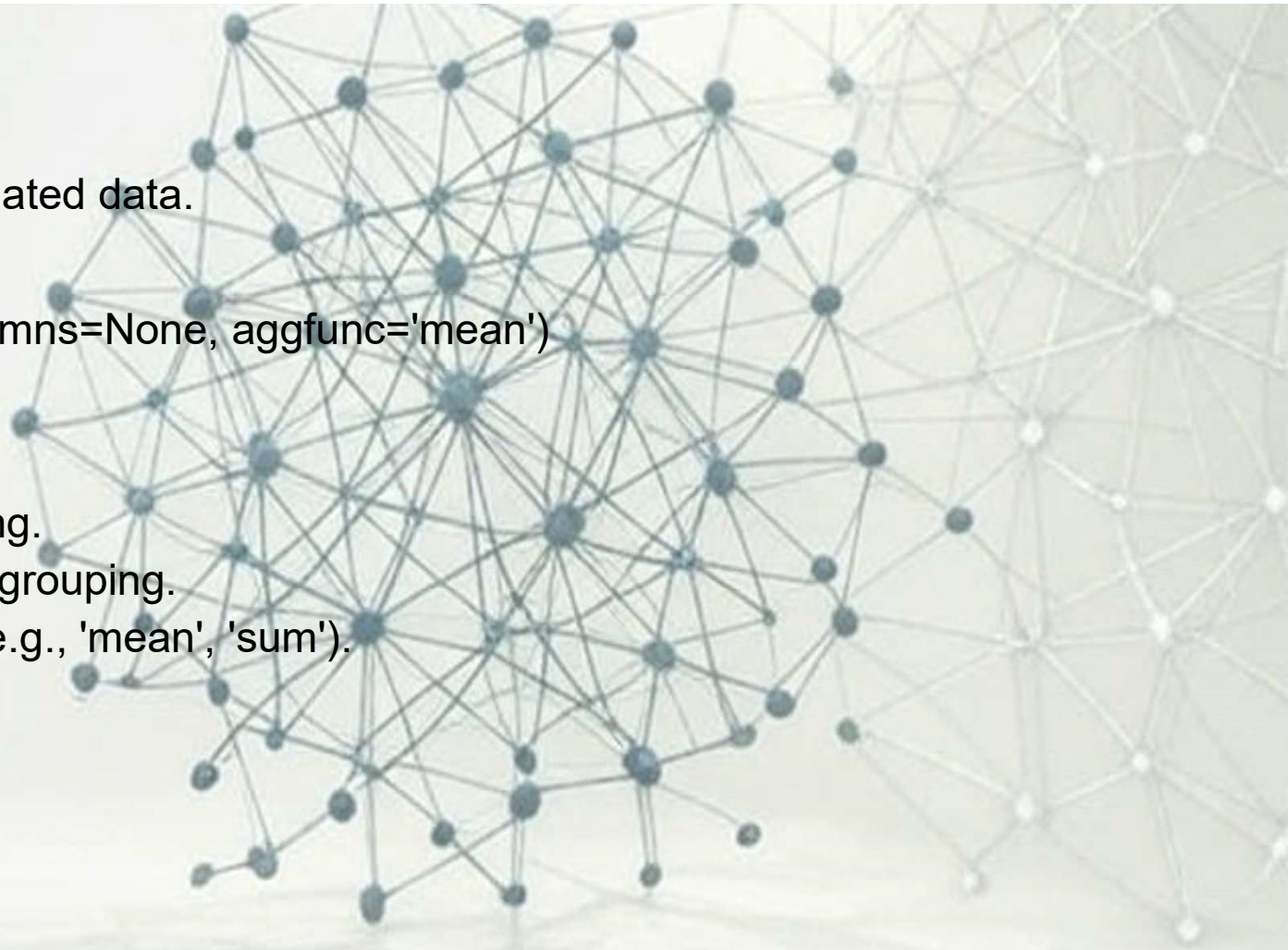
- Creates a pivot table with aggregated data.

- **Syntax:**

- `df.pivot_table(values, index, columns=None, aggfunc='mean')`

- **Key Parameters:**

- **values:** Column to aggregate.
- **index:** Column(s) for row grouping.
- **columns:** Column(s) for column grouping.
- **aggfunc:** Aggregation function (e.g., 'mean', 'sum').



- `import pandas as pd`
- `data = {`
 - `'Name': ['Alice', 'Bob', 'Charlie'],`
 - `'Age': [25, 30, 35],`
 - `'Salary': [50000, 60000, 75000]``}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Use pivot_table to summarize average Age and Salary**
- `pivot_table_result = df.pivot_table(index='Name',`
`values=['Age', 'Salary'], aggfunc='mean')`
- `print("\nPivot Table DataFrame:")`
- `print(pivot_table_result)`

- **Output:**

- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | Alice | 25 | 50000 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |
- Pivot Table DataFrame:
- | | Age | Salary |
|---------|------|---------|
| Name | | |
| Alice | 25.0 | 50000.0 |
| Bob | 30.0 | 60000.0 |
| Charlie | 35.0 | 75000.0 |

transform()

- **Description:**

- Applies a function to each element or group in a DataFrame or Series.

- **Syntax:**

- `df.transform(func, axis=0, *args, **kwargs)`

- **Key Parameters:**

- **func:** Function to apply (e.g., lambda, numpy function).
- **axis:** 0 for rows, 1 for columns.



- `import pandas as pd`
- `data = {`
- `'Name': ['Alice', 'Bob', 'Charlie'],`
- `'Age': [25, 30, 35],`
- `'Salary': [50000, 60000, 75000]`
- `}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Normalize salary using transform**
- `df['Normalized_Salary'] =`
`df['Salary'].transform(lambda x: (x - x.min()) / (x.max()`
`- x.min()))`
- `print("\nDataFrame with Normalized Salary:")`
- `print(df)`

- **Output:**

- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | Alice | 25 | 50000 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |

- **DataFrame with Normalized Salary:**

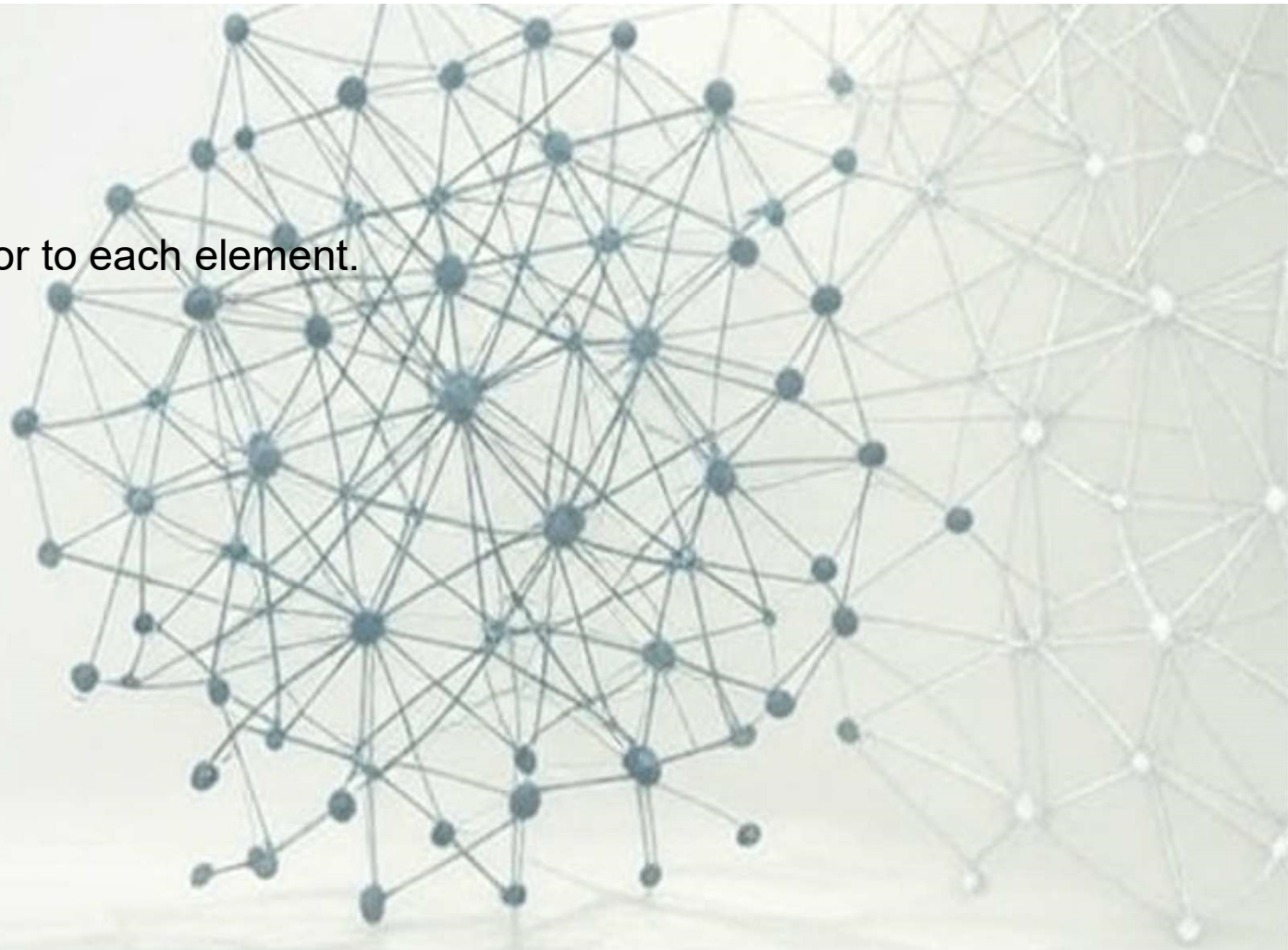
- | | Name | Age | Salary | Normalized_Salary |
|---|---------|-----|--------|-------------------|
| 0 | Alice | 25 | 50000 | 0.0 |
| 1 | Bob | 30 | 60000 | 0.4 |
| 2 | Charlie | 35 | 75000 | 1.0 |

Methods of DataFrame: Merging and Joining & Reshaping and Transformation

Method	Syntax	Description
<code>apply(func, axis)</code>	<code>df.apply(func, axis=0)</code>	Applies a function along an axis (rows or columns).
<code>rolling(window)</code>	<code>df.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0)</code>	Provides rolling window calculations (e.g., mean, sum).
<code>expanding(min_periods)</code>	<code>df.expanding(min_periods=1, axis=0)</code>	Provides expanding window calculations.
<code>ewm(com/span/halflife)</code>	<code>df.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True, ignore_na=False)</code>	Provides exponentially weighted moving calculations.
<code>merge(right, how, on)</code>	<code>df.merge(right, how='inner', on=None, left_on=None, right_on=None)</code>	Merges with another DataFrame or Series.
<code>join(other, on, how)</code>	<code>df.join(other, on=None, how='left')</code>	Joins with another DataFrame on index or columns.
<code>concat(objs, axis)</code>	<code>pd.concat(objs, axis=0, join='outer')</code>	Concatenates multiple DataFrames (use <code>`pd.concat`</code>).

apply()

- **Description:**
 - Applies a function along an axis or to each element.
- **Syntax:**
 - `df.apply(func, axis=0)`
- **Key Parameters:**
 - **func:** Function to apply.
 - **axis:** 0 for columns, 1 for rows.



apply()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}
- df = pd.DataFrame(data)
- print(df)
- **# Define a function to calculate 12% tax on salary**
- def calculate_tax(salary):
 return salary * 0.12
- **# Apply the function to the 'Salary' column**
- df['Tax'] = df['Salary'].apply(calculate_tax)
- print("\nDataFrame with Tax column:")
- print(df)

• Output:

- **Original DataFrame:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

- **DataFrame with Tax column:**

- Name Age Salary Tax
- 0 Alice 25 50000 6000.0
- 1 Bob 30 60000 7200.0
- 2 Charlie 35 75000 9000.0

rolling()

- **Description:**

- Provides rolling window calculations (e.g., moving average) over a specified window.

- **Syntax:**

- `df.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0)`

- **Key Parameters:**

- **window:** int or time-based (**number of observations** used for each calculation).
- **min_periods:** Minimum observations for valid result.
- **center:** Center the window (default False).
- **win_type:** Window type (e.g., 'boxcar', 'triang').

- `import pandas as pd`

- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}`

- `df = pd.DataFrame(data)`

- `df['Rolling_Avg_Salary'] =
df['Salary'].rolling(window=2).mean()`

- `print(df)`

- **Output:**

- | | Name | Age | Salary | Rolling_Avg_Salary |
|---|---------|-----|--------|--------------------|
| 0 | Alice | 25 | 50000 | NaN |
| 1 | Bob | 30 | 60000 | 55000.0 |
| 2 | Charlie | 35 | 75000 | 67500.0 |

expanding()

- **Description:**

- Provides expanding window calculations (e.g., cumulative sum, mean) over an axis.

- **Syntax:**

- `df.expanding(min_periods=1, axis=0)`

- **Key Parameters:**

- **min_periods:** Minimum observations for valid result.
- **axis:** 0 for rows, 1 for columns.

- `import pandas as pd`

- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}`

- `df = pd.DataFrame(data)`

- `print(df)`

- `df['Expanding_Mean_Salary'] =
df['Salary'].expanding().mean()`

- `print(df)`

- **Output:**

- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | Alice | 25 | 50000 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |
- | | Name | Age | Salary | Expanding_Mean_Salary |
|---|---------|-----|--------|-----------------------|
| 0 | Alice | 25 | 50000 | 50000.000000 |
| 1 | Bob | 30 | 60000 | 55000.000000 |
| 2 | Charlie | 35 | 75000 | 61666.666667 |

ewm()

- **Description:**

- Provides exponentially weighted moving calculations (e.g., mean, std).

- **Syntax:**

- `df.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True, ignore_na=False)`

- **Key Parameters:**

- **com, span, halflife, alpha:** Define weighting decay.
- **min_periods:** Minimum observations for valid result.
- **adjust:** Adjust weights for initial periods.
- **ignore_na:** Ignore missing values.

- **Common Uses of ewm()**

- **Smoothing Time Series Data:** To reduce noise and highlight trends.
- **Volatility Measurement:** In finance, to measure the volatility of stock prices.
- **Forecasting:** To create forecasts based on recent trends.

- `import pandas as pd`
- `data = {`
`'Name': ['Alice', 'Bob', 'Charlie'],`
`'Age': [25, 30, 35],`
`'Salary': [50000, 60000, 75000]`
`}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Calculate exponentially weighted moving average of Salary**
- **# The 'span' parameter controls the decay; a smaller span gives more weight to recent observations**
- `df['EWM_Salary'] =`
`df['Salary'].ewm(span=2).mean()`
- `print("\nDataFrame with Exponentially Weighted Moving Average Salary:")`
- `print(df)`

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

- **DataFrame with Exponentially Weighted Moving Average Salary:**

- Name Age Salary EWM_Salary
- 0 Alice 25 50000 50000.000000
- 1 Bob 30 60000 57500.000000
- 2 Charlie 35 75000 69615.384615

merge()

- **Description:**

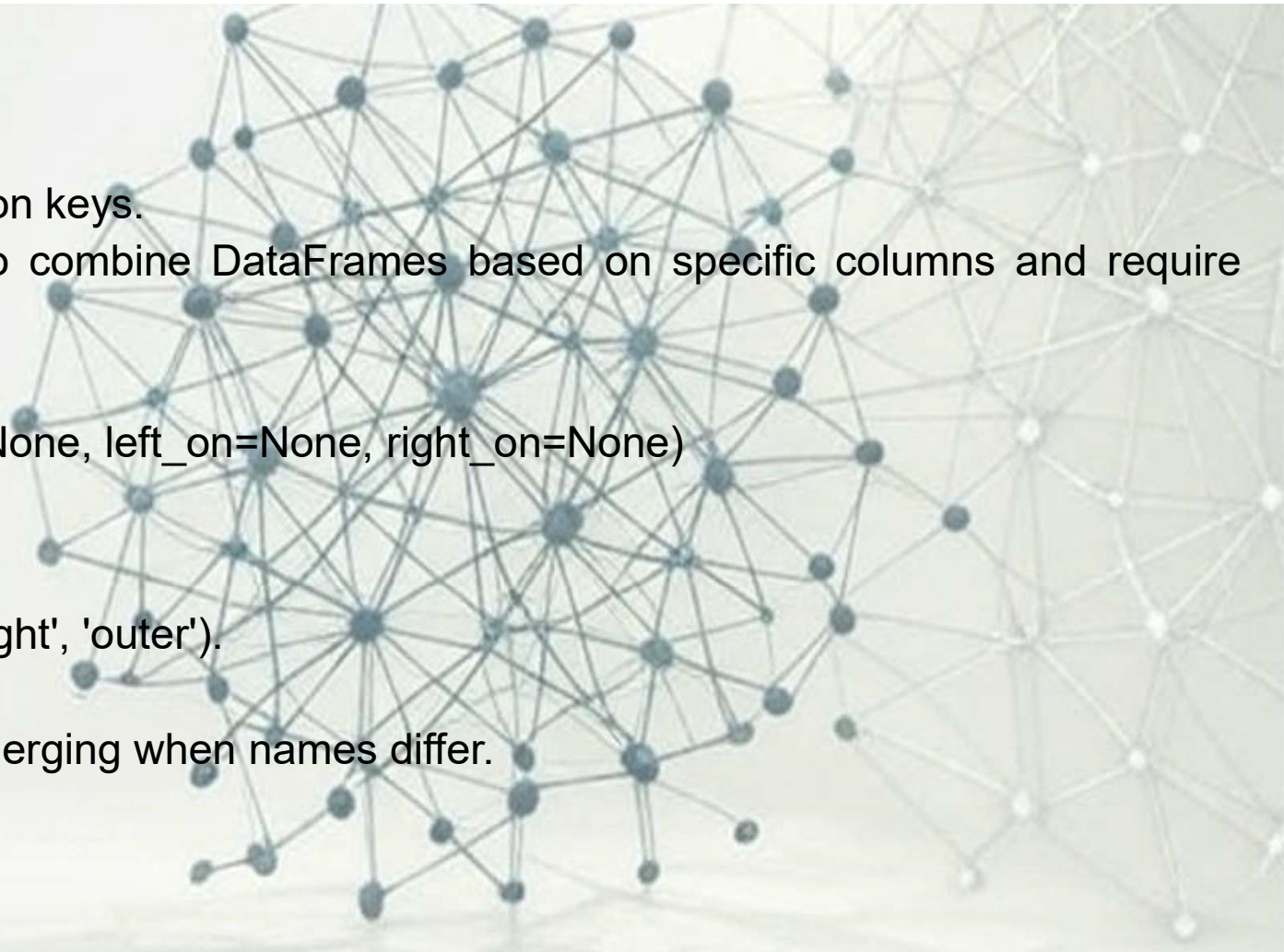
- Merges two DataFrames based on keys.
- Use **merge()** when you need to combine DataFrames based on specific columns and require more complex join operations.

- **Syntax:**

- `df.merge(right, how='inner', on=None, left_on=None, right_on=None)`

- **Key Parameters:**

- **right:** DataFrame to merge with
- **how:** Merge type ('inner', 'left', 'right', 'outer').
- **on:** Column(s) to join on.
- **left_on, right_on:** Columns for merging when names differ.



merge()

```
• import pandas as pd
• df1 = pd.DataFrame({
•     'EmployeeID': [101, 102, 103, 104],
•     'Name': ['Alice', 'Bob', 'Charlie', 'David'],
•     'Department': ['HR', 'Finance', 'IT', 'Marketing']
• })
• df2 = pd.DataFrame({
•     'EmployeeID': [102, 103, 104, 105],
•     'Salary': [60000, 75000, 58000, 52000],
•     'JoiningYear': [2015, 2016, 2017, 2018]
• })
• # Merge DataFrames on 'EmployeeID' with inner join (default)
• result_inner = pd.merge(df1, df2,
• on='EmployeeID')
• print("Inner Join (Only matching EmployeeID):")
• print(result_inner)
```

• Output:

```
• EmployeeID  Name Department
• 0      101  Alice      HR
• 1      102   Bob  Finance
• 2      103 Charlie    IT
• 3      104  David Marketing
```

```
• EmployeeID Salary JoiningYear
• 0      102  60000      2015
• 1      103  75000      2016
• 2      104  58000      2017
• 3      105  52000      2018
```

• Inner Join (Only matching EmployeeID):

```
• EmployeeID  Name Department Salary JoiningYear
• 0      102   Bob      Finance  60000      2015
• 1      103 Charlie      IT      75000      2016
• 2      104  David      Marketing  58000      2017
```

merge()

- **# Merge DataFrames on 'EmployeeID' with left join**

- `result_left = pd.merge(df1, df2, on='EmployeeID', how='left')`
- `print("\nLeft Join (All from df1 with matching from df2):")`
- `print(result_left)`

- **# Merge DataFrames on 'EmployeeID' with outer join**

- `result_outer = pd.merge(df1, df2, on='EmployeeID', how='outer')`
- `print("\nOuter Join (All from both dfs):")`
- `print(result_outer)`

- **Output:**

- **Left Join (All from df1 with matching from df2):**

	EmployeeID	Name	Department	Salary	JoiningYear
•	0	101	Alice	HR	NaN
•	1	102	Bob	Finance	60000.0
•	2	103	Charlie	IT	75000.0
•	3	104	David	Marketing	58000.0

- **Outer Join (All from both dfs):**

	EmployeeID	Name	Department	Salary	JoiningYear
•	1	102	Bob	Finance	60000.0
•	2	103	Charlie	IT	75000.0
•	3	104	David	Marketing	58000.0
•	4	105	NaN	NaN	52000.0

join()

- **Description:**

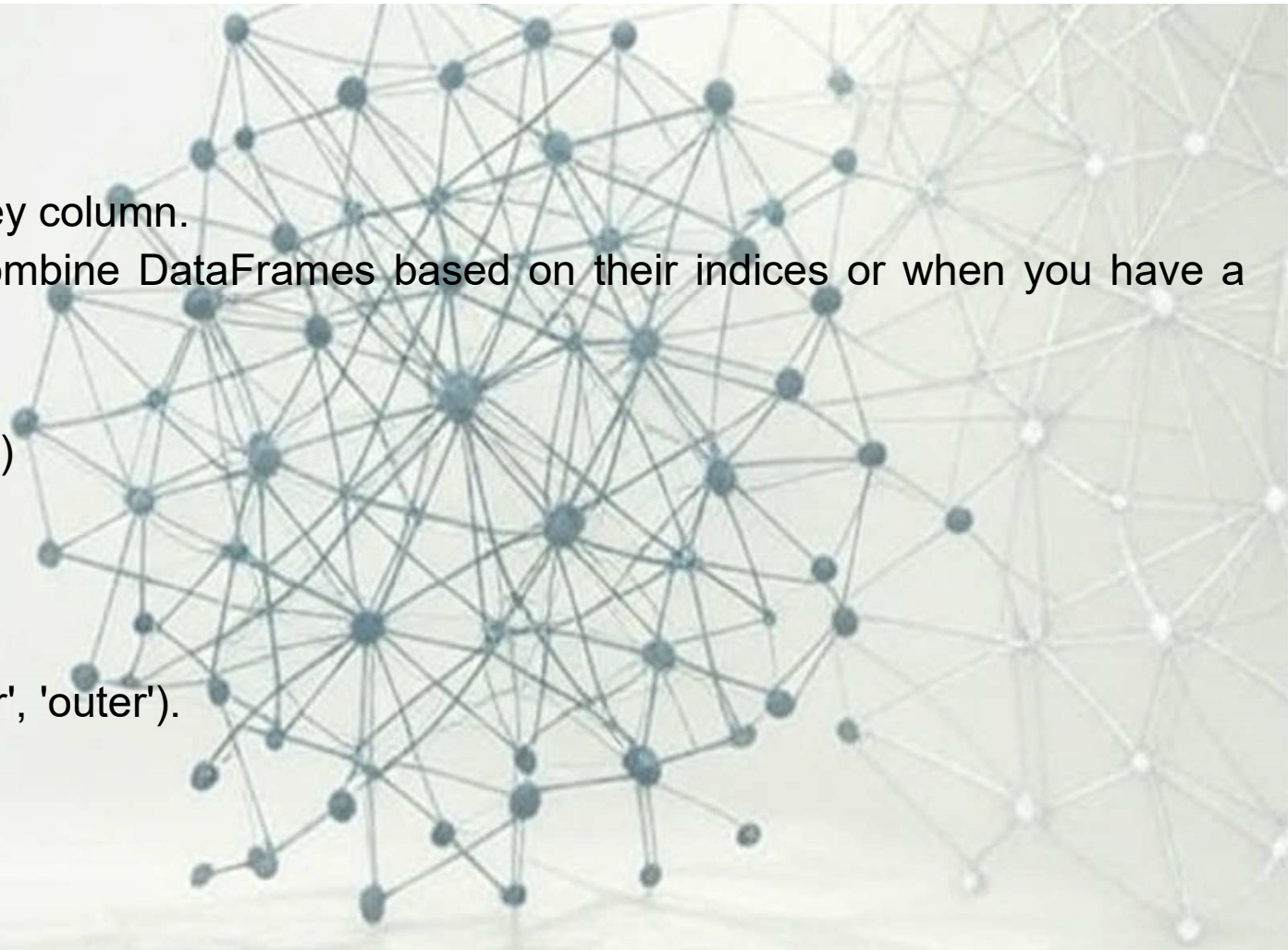
- Joins DataFrames on index or key column.
- Use **join()** when you want to combine DataFrames based on their indices or when you have a simpler joining requirement.

- **Syntax:**

- `df.join(other, on=None, how='left')`

- **Key Parameters:**

- **other:** DataFrame to join.
- **on:** Column or index to join on.
- **how:** Join type ('left', 'right', 'inner', 'outer').



join()

- import pandas as pd
- df1 = pd.DataFrame({
 'Name': ['Alice', 'Bob', 'Charlie', 'David'],
 'Department': ['HR', 'Finance', 'IT', 'Marketing']
}, index=[101, 102, 103, 104])
- print('\n',df1)
- df2 = pd.DataFrame({
 'Salary': [60000, 75000, 58000, 52000],
 'JoiningYear': [2015, 2016, 2017, 2018]
}, index=[102, 103, 104, 105])
- print('\n',df2)
- **# Join df2 to df1 on their indices with inner join**
- result_inner = df1.join(df2, how='inner')
- print("\n\nInner Join (Only matching indices):")
- print(result_inner)

- **Output:**
 - Name Department
 - 101 Alice HR
 - 102 Bob Finance
 - 103 Charlie IT
 - 104 David Marketing
 - Salary JoiningYear
 - 102 60000 2015
 - 103 75000 2016
 - 104 58000 2017
 - 105 52000 2018
- Inner Join (Only matching indices):
 - Name Department Salary JoiningYear
 - 102 Bob Finance 60000 2015
 - 103 Charlie IT 75000 2016
 - 104 David Marketing 58000 2017

join()

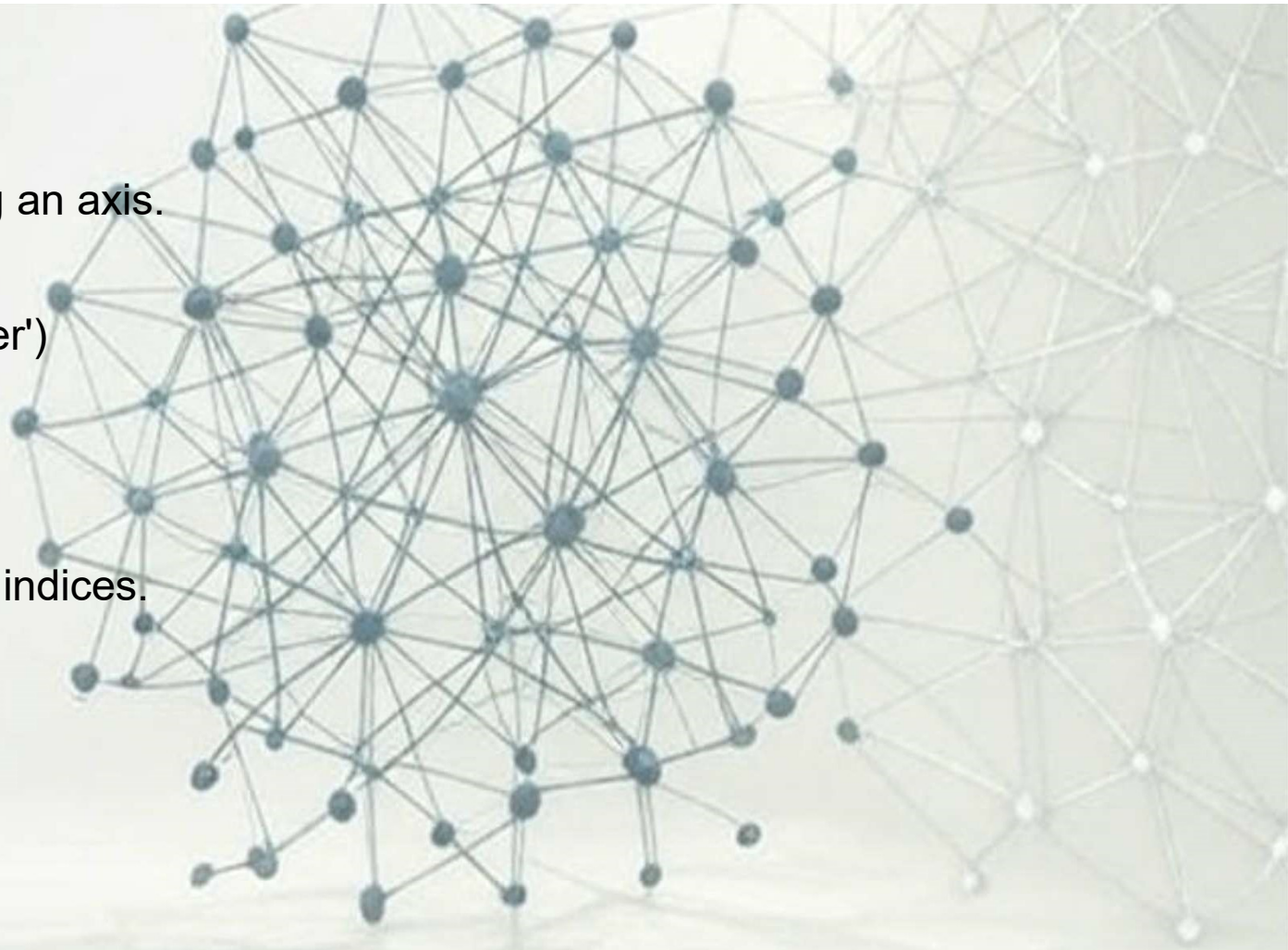
- **# Join df2 to df1 on their indices with left join**
- `result_left = df1.join(df2, how='left')`
- `print("\nLeft Join (All indices from df1 with matching from df2):")`
- `print(result_left)`

- **# Join df2 to df1 on their indices with outer join**
- `result_outer = df1.join(df2, how='outer')`
- `print("\nOuter Join (All indices from both dfs):")`
- `print(result_outer)`

- **Output:**
- **Left Join (All indices from df1 with matching from df2):**
- | | Name | Department | Salary | JoiningYear |
|-----|---------|------------|---------|-------------|
| | Alice | HR | NaN | NaN |
| 101 | Alice | HR | NaN | NaN |
| 102 | Bob | Finance | 60000.0 | 2015.0 |
| 103 | Charlie | IT | 75000.0 | 2016.0 |
| 104 | David | Marketing | 58000.0 | 2017.0 |
- **Outer Join (All indices from both dfs):**
- | | Name | Department | Salary | JoiningYear |
|-----|---------|------------|---------|-------------|
| | Alice | HR | NaN | NaN |
| 101 | Alice | HR | NaN | NaN |
| 102 | Bob | Finance | 60000.0 | 2015.0 |
| 103 | Charlie | IT | 75000.0 | 2016.0 |
| 104 | David | Marketing | 58000.0 | 2017.0 |
| 105 | NaN | NaN | 52000.0 | 2018.0 |

concat()

- **Description:**
 - Concatenates DataFrames along an axis.
- **Syntax:**
 - `pd.concat(objs, axis=0, join='outer')`
- **Key Parameters:**
 - **objs:** List of DataFrames
 - **axis:** 0 for rows, 1 for columns.
 - **join:** 'outer' or 'inner' for handling indices.



concat()

- `import pandas as pd`
- `df1 = pd.DataFrame({
 'EmployeeID': [101, 102, 103],
 'Name': ['Alice', 'Bob', 'Charlie'] })`
- `df2 = pd.DataFrame({
 'EmployeeID': [104, 105],
 'Name': ['David', 'Eva']
})`
- `df3 = pd.DataFrame({
 'EmployeeID': [106, 107],
 'Name': ['Frank', 'Grace']
})`
- **# Concatenate DataFrames along rows (axis=0)**
- `result_rows = pd.concat([df1, df2, df3], axis=0)`
- `print("Concatenated DataFrames along rows:")`
- `print(result_rows)`

- **Output:**
- **Concatenated DataFrames along rows:**

	EmployeeID	Name
• 0	101	Alice
• 1	102	Bob
• 2	103	Charlie
• 0	104	David
• 1	105	Eva
• 0	106	Frank
• 1	107	Grace

concat()

- **# Concatenate DataFrames along columns (axis=1)**
- **# For this, we need to create DataFrames with the same number of rows**
- ```
df4 = pd.DataFrame({
 'Department': ['HR', 'Finance', 'IT',
 'Marketing', 'Sales', 'Support']
})
```
- ```
result_columns = pd.concat(  
    [result_rows.reset_index(drop=True), df4],  
    axis=1)
```
- ```
print("\nConcatenated DataFrames along
columns:")
```
- ```
print(result_columns)
```

- **Output:**
- **Concatenated DataFrames along columns:**
- | | EmployeeID | Name | Department |
|-----|------------|---------|------------|
| • 0 | 101 | Alice | HR |
| • 1 | 102 | Bob | Finance |
| • 2 | 103 | Charlie | IT |
| • 3 | 104 | David | Marketing |
| • 4 | 105 | Eva | Sales |
| • 5 | 106 | Frank | Support |
| • 6 | 107 | Grace | NaN |