



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

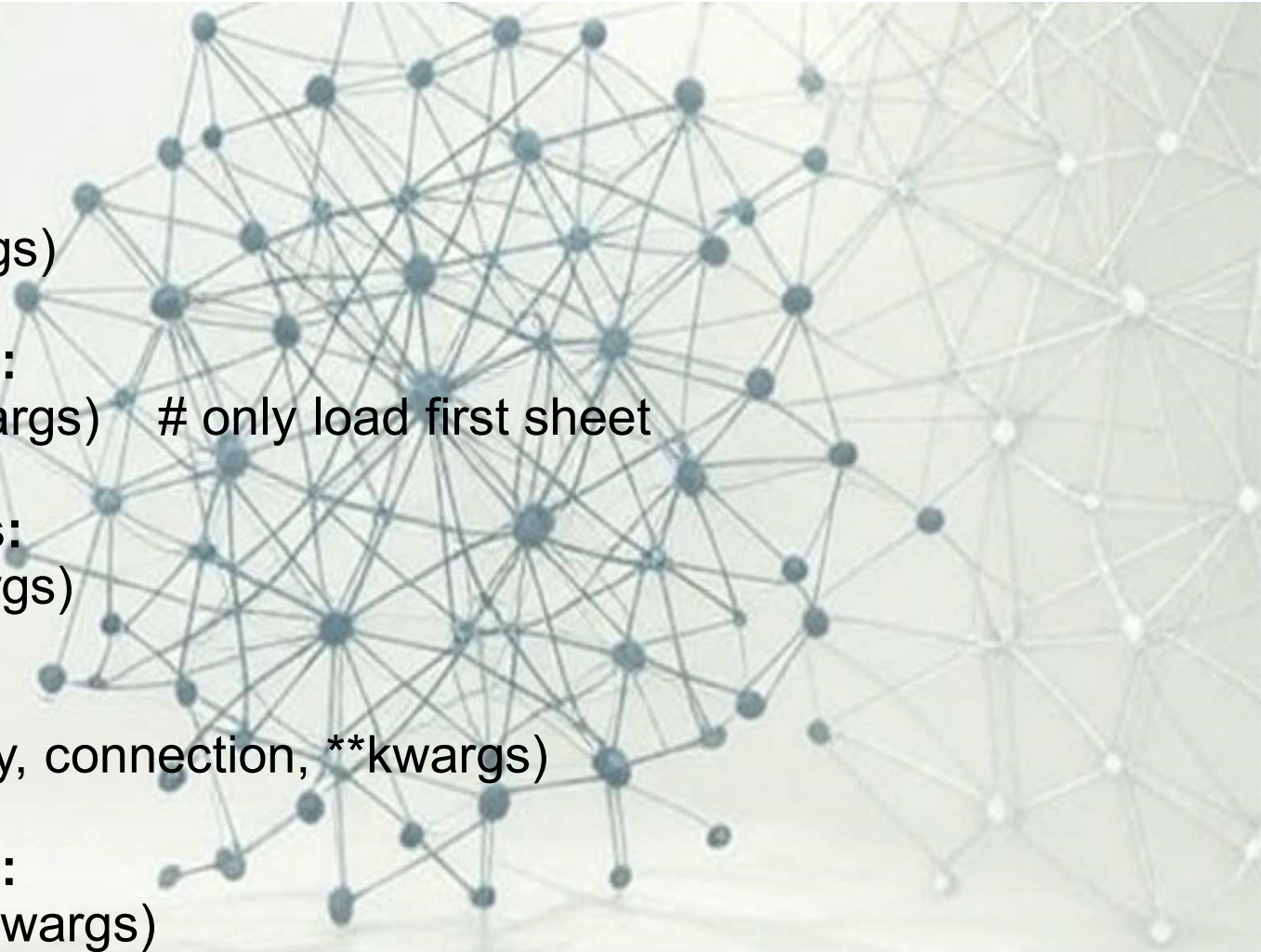
- `import pandas as pd`
- `data = {`
 `'Name': ['Alice', 'Bob', 'Charlie'],`
 `'Age': [25, 30, 35],`
 `'Salary': [50000, 60000, 75000]`
 `}`
- `df = pd.DataFrame(data)`
- `print("Sample DataFrame:")`
- `print(df)`

- **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   75000
```

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

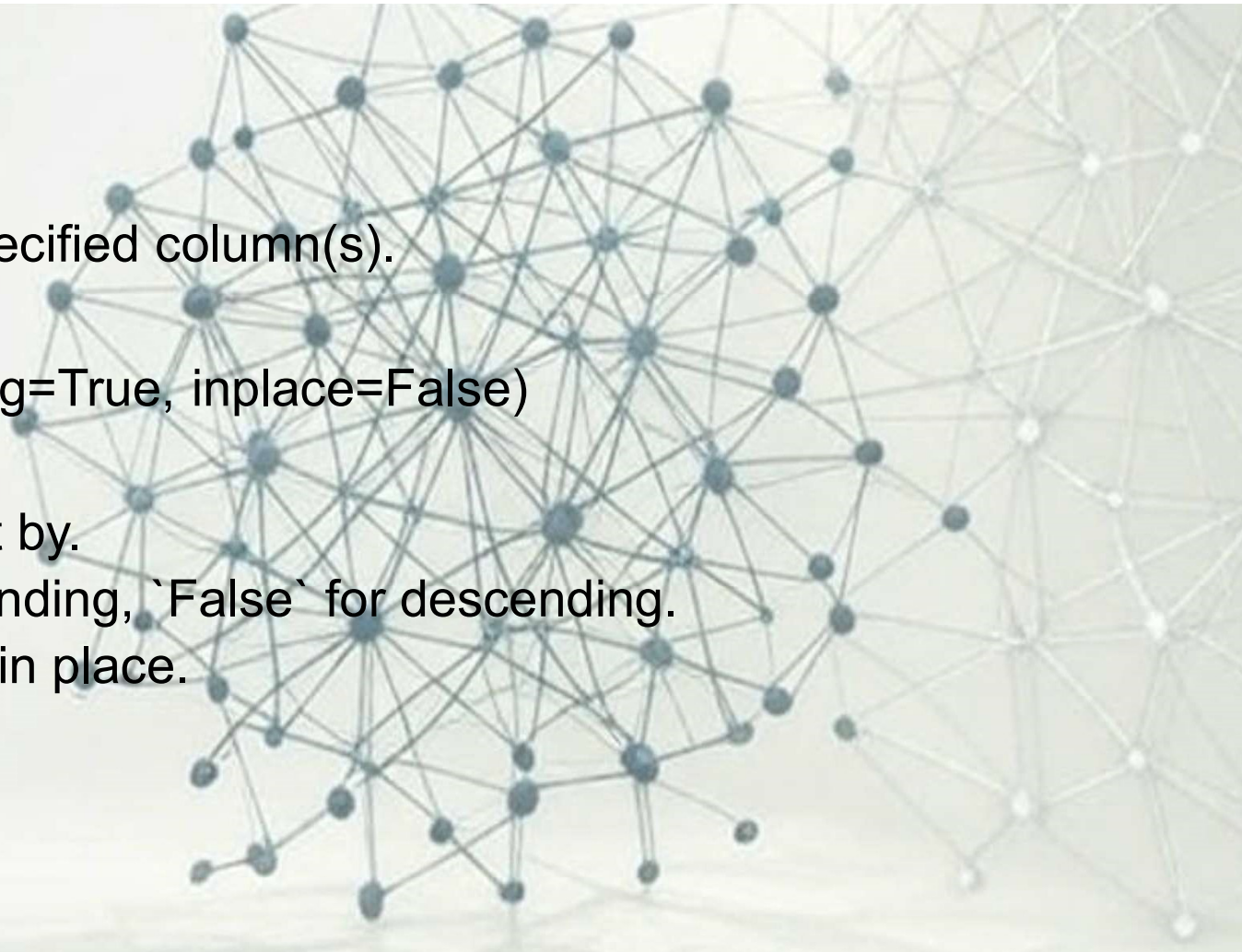
- **Output:**

- 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:**

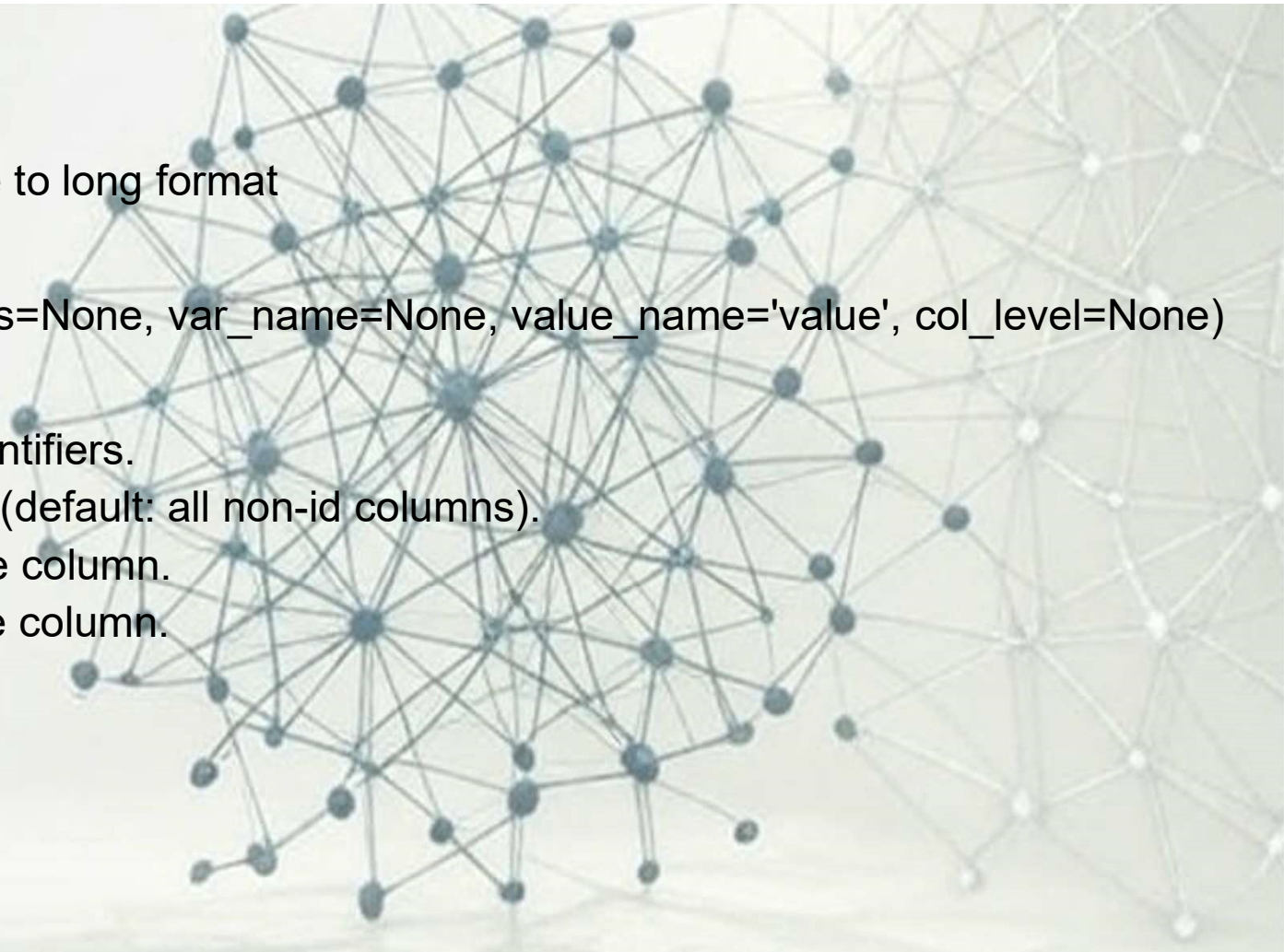
- Unpivots a DataFrame from wide to long format

- **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:**

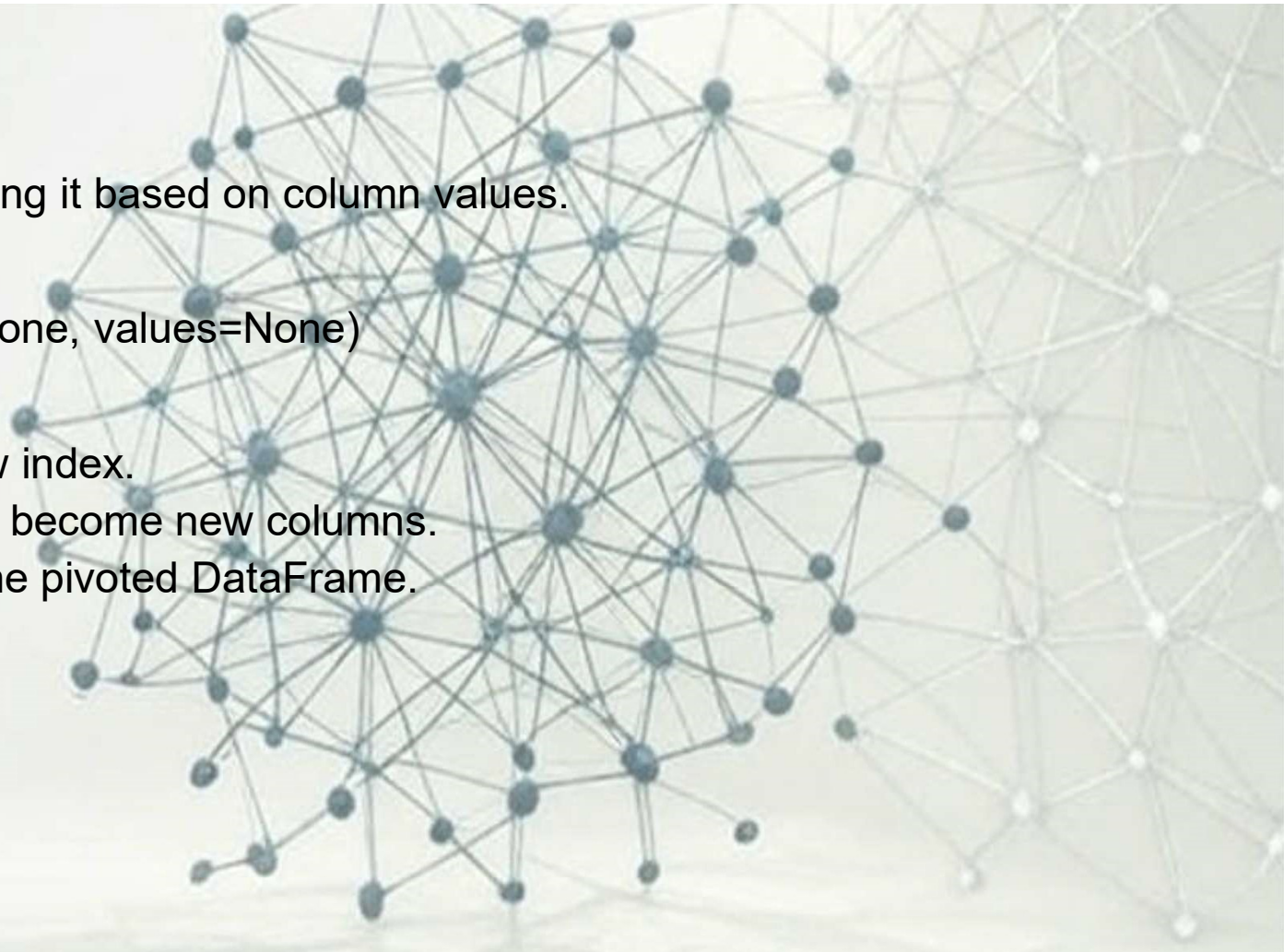
- Reshapes a DataFrame by pivoting it based on column values.

- **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:**

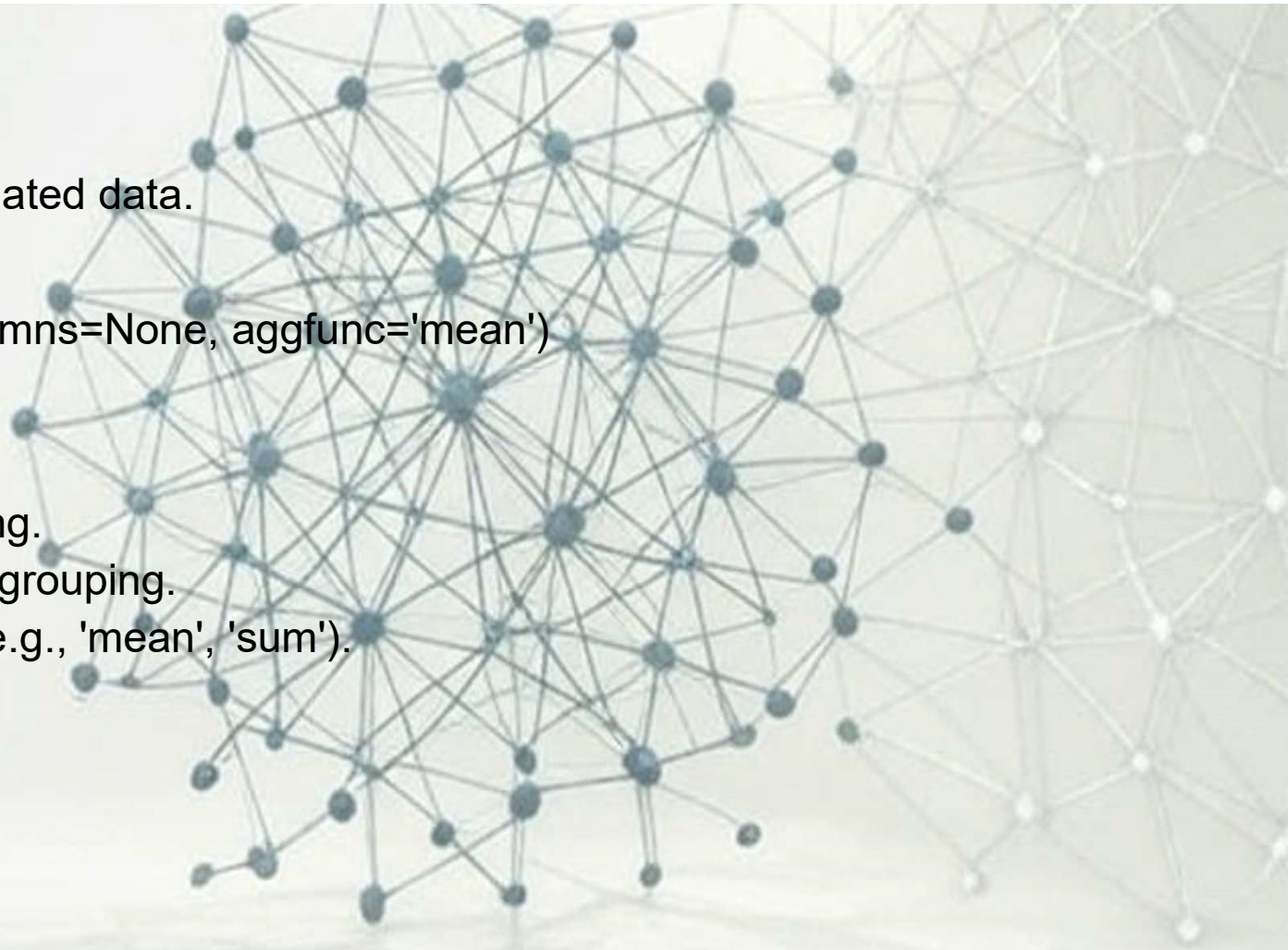
- Creates a pivot table with aggregated data.

- **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:**

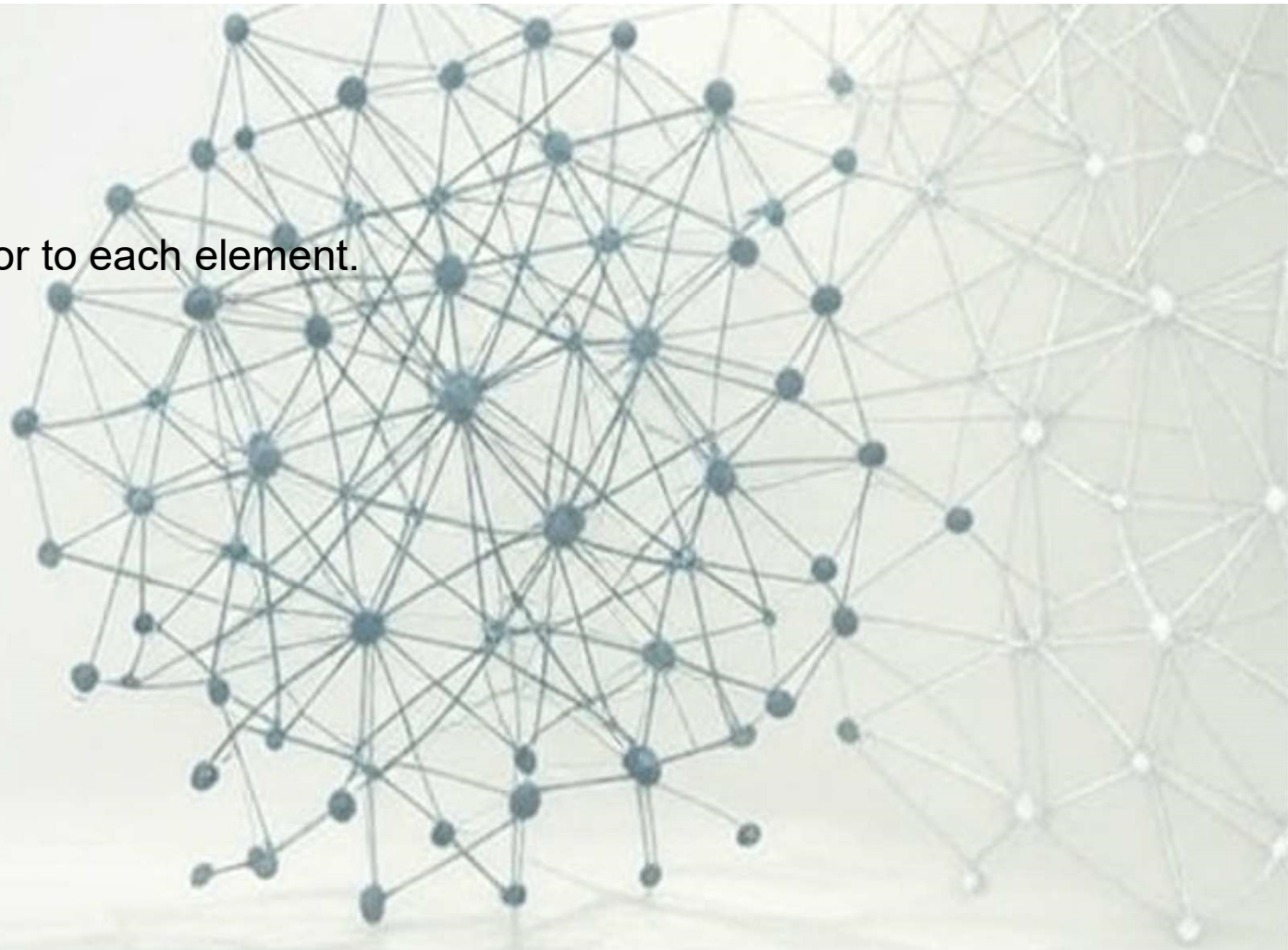
- | | Name | Age | Salary | Normalized_Salary |
|---|---------|-----|--------|-------------------|
| 0 | Alice | 25 | 50000 | 0.0 |
| 1 | Bob | 30 | 60000 | 0.4 |
| 2 | Charlie | 35 | 75000 | 1.0 |

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:**

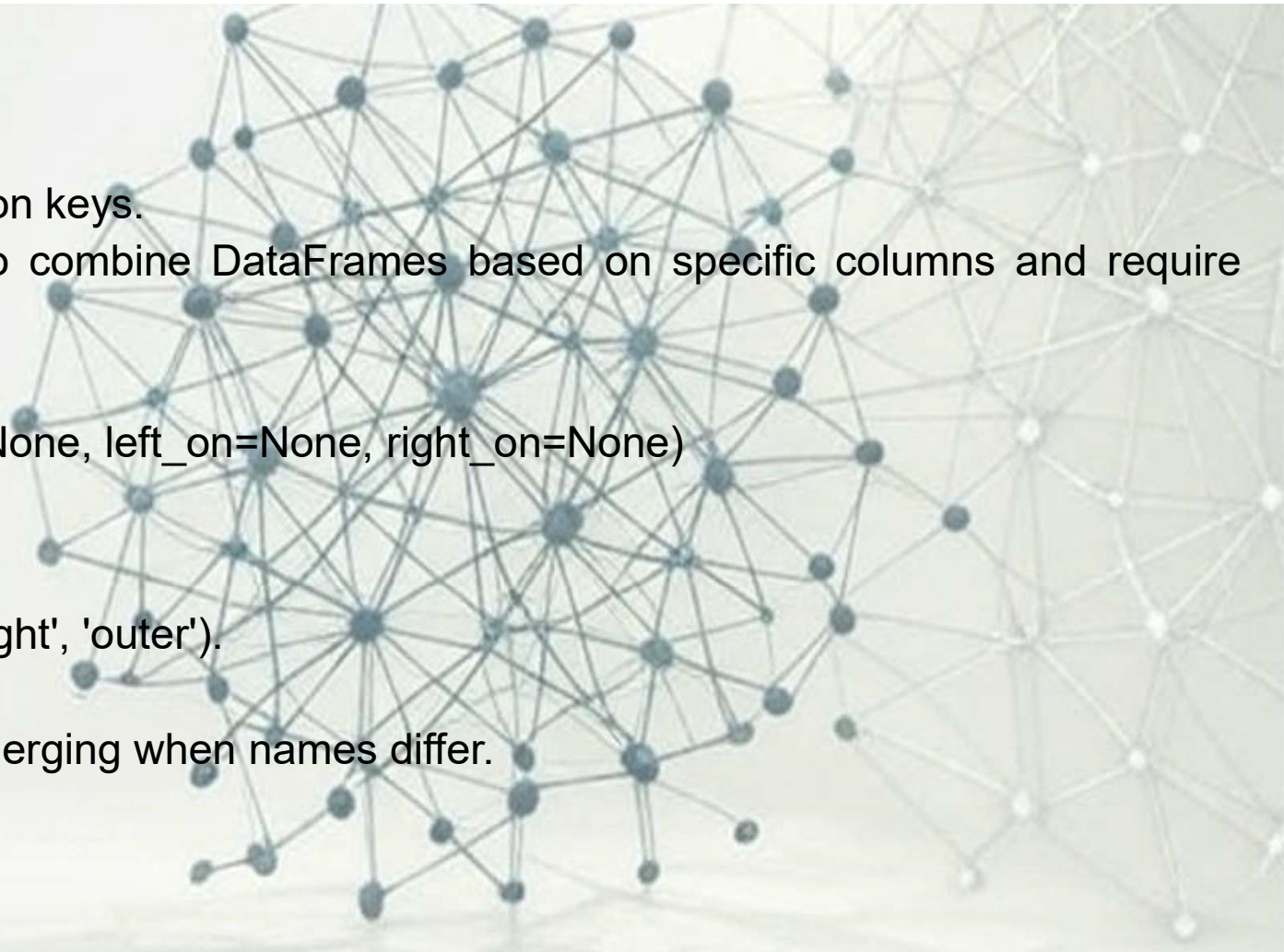
- Merges two DataFrames based on keys.
- Use **merge()** when you need to combine DataFrames based on specific columns and require more complex join operations.

- **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:**

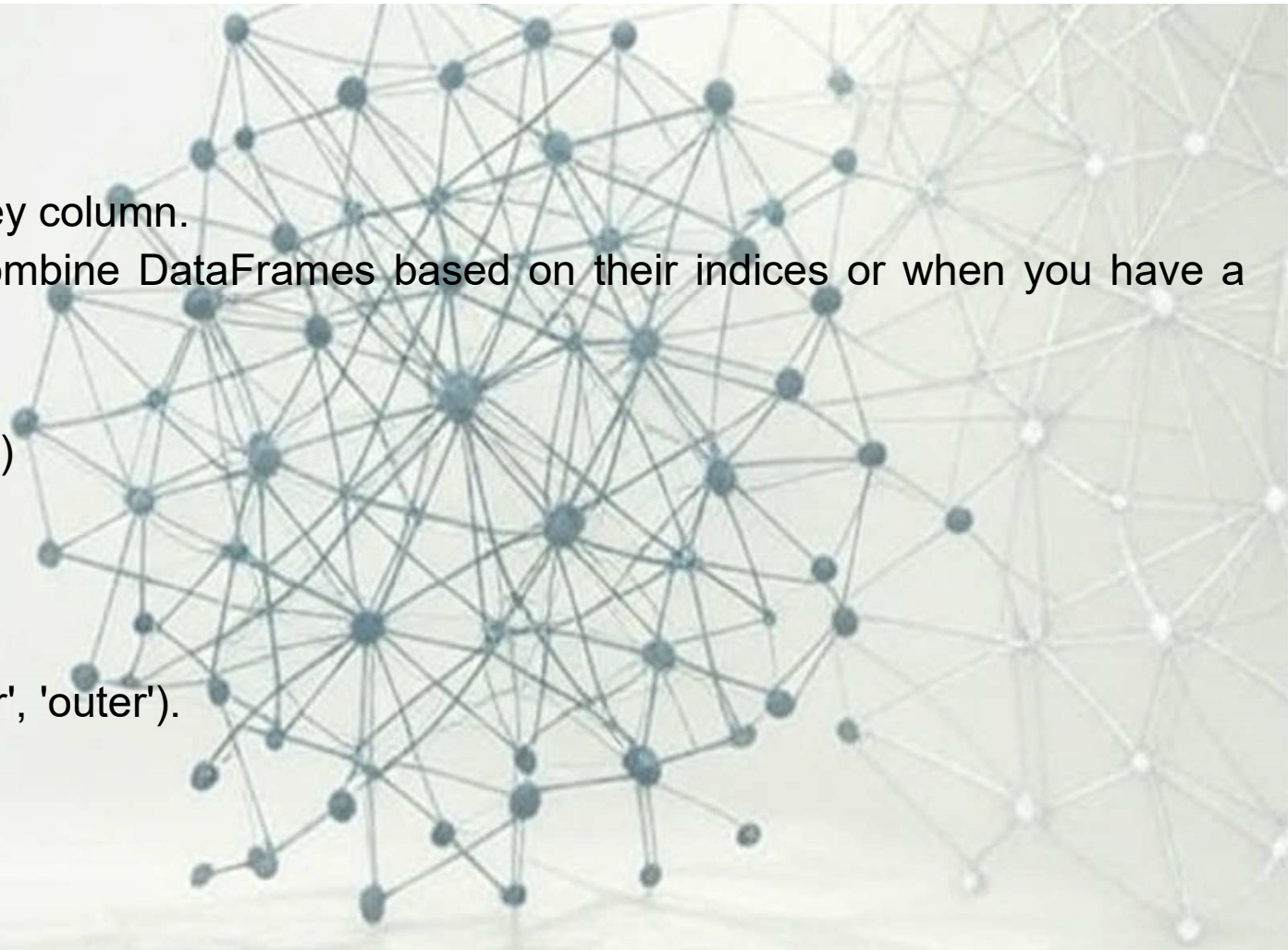
- Joins DataFrames on index or key column.
- Use **join()** when you want to combine DataFrames based on their indices or when you have a simpler joining requirement.

- **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("\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()

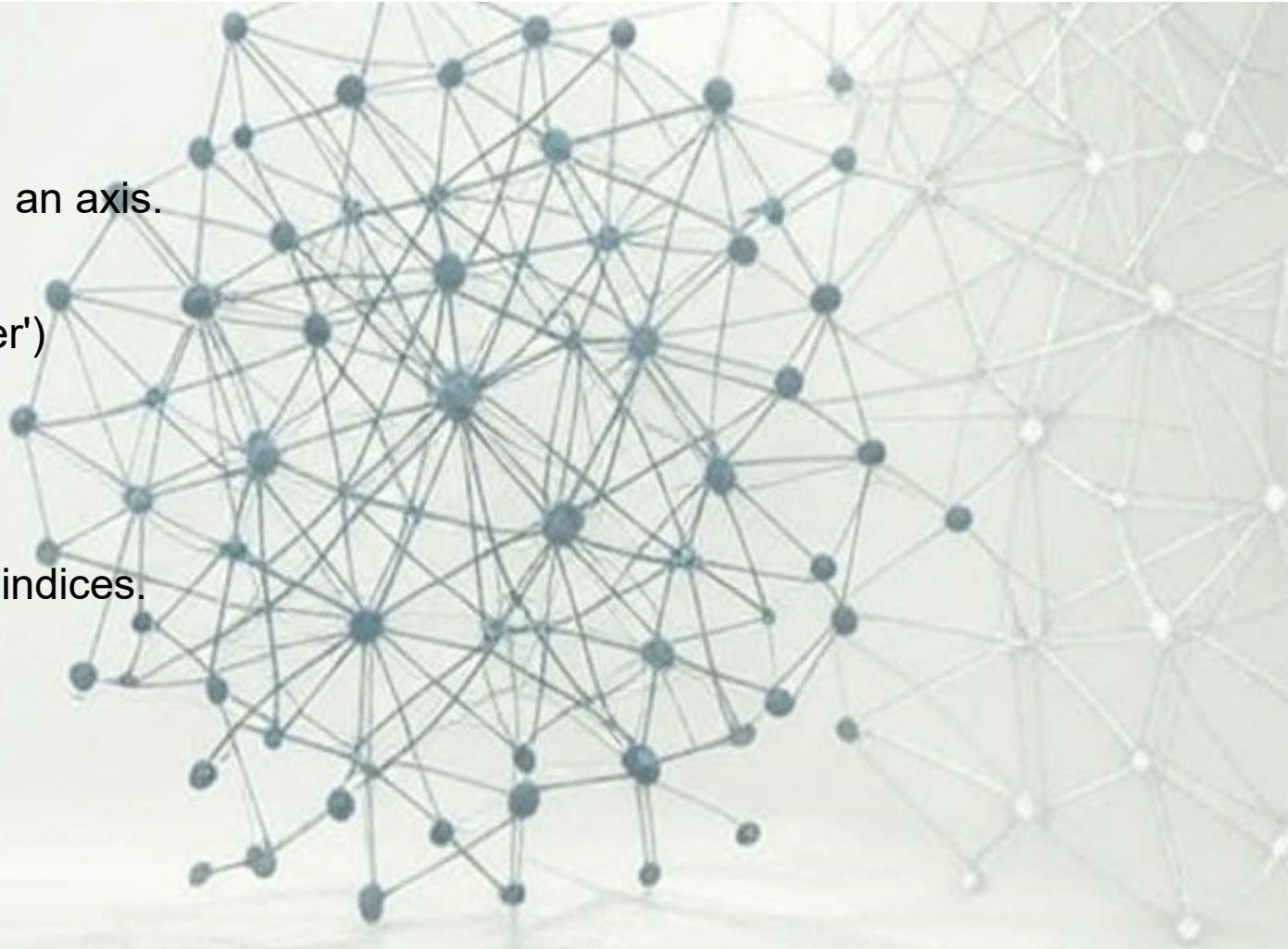
- **# Join df2 to df1 on their indices with left join**
- `result_left = df1.join(df2, how='left')`
- `print("\nLeft Join (All indices from df1 with matching from df2):")`
- `print(result_left)`

- **# 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 |

Methods of DataFrame: Reshaping and Transformation

Method	Syntax	Description
stack()	df.stack(level=-1, dropna=True)	Pivots columns to rows, creating a MultiIndex.
unstack(level)	df.unstack(level=-1, fill_value=None)	Pivots a level of MultiIndex to columns.
transpose()	df.transpose(copy=True)	Transposes rows and columns (alias: `T`).
explode(column)	df.explode(column, ignore_index=False)	Explodes lists or arrays in a column into separate rows.
squeeze()	df.squeeze(axis=None)	Converts a single-column DataFrame to a Series.
mean(axis)	df.mean(axis=0)	Computes the mean of columns or rows.
median(axis)	df.median(axis=0, skipna=True, numeric_only=False)	Computes the median.
mode(axis)	df.mode(axis=0, numeric_only=False, dropna=True)	Computes the mode.

stack()

- **Description:**

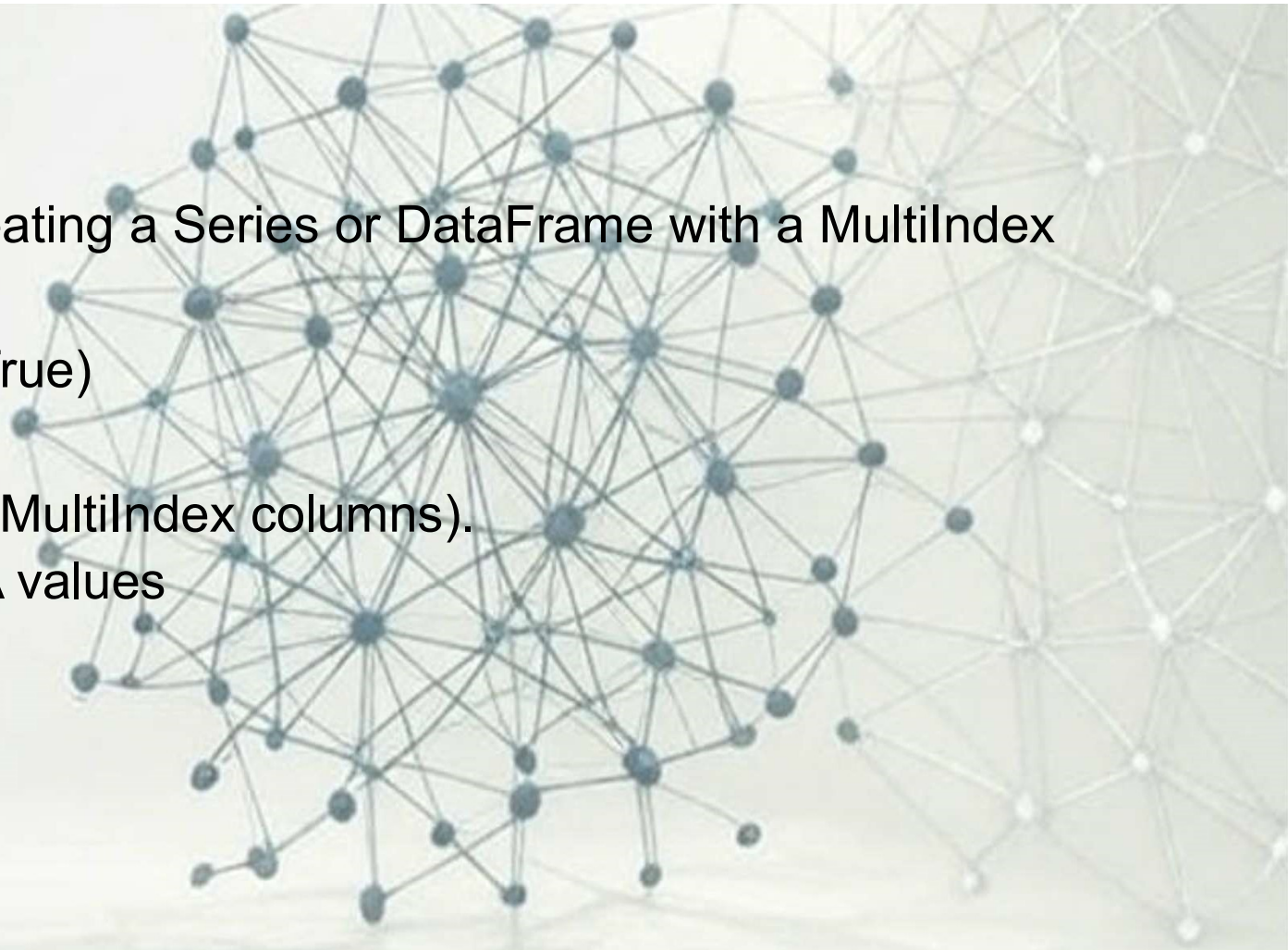
- Pivots columns to rows, creating a Series or DataFrame with a MultiIndex

- **Syntax:**

- `df.stack(level=-1, dropna=True)`

- **Key Parameters:**

- **level:** Level(s) to stack (for MultiIndex columns).
- **dropna:** Drop rows with NA values



stack()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}
- df = pd.DataFrame(data)
- **# Use the stack() function**
- stacked_df = df.set_index('Name').stack()
- **# Rename the columns for clarity**
- stacked_df.columns = ['Name', 'Attribute', 'Value']
- print(stacked_df)
- print("\nresetindex\n", stacked_df.reset_index())

• Output:

- Name
- Alice Age 25
- Salary 50000
- Bob Age 30
- Salary 60000
- Charlie Age 35
- Salary 75000
- dtype: int64
- **resetindex**
- Name level_1 0
- 0 Alice Age 25
- 1 Alice Salary 50000
- 2 Bob Age 30
- 3 Bob Salary 60000
- 4 Charlie Age 35
- 5 Charlie Salary 75000

unstack()

- **Description:**

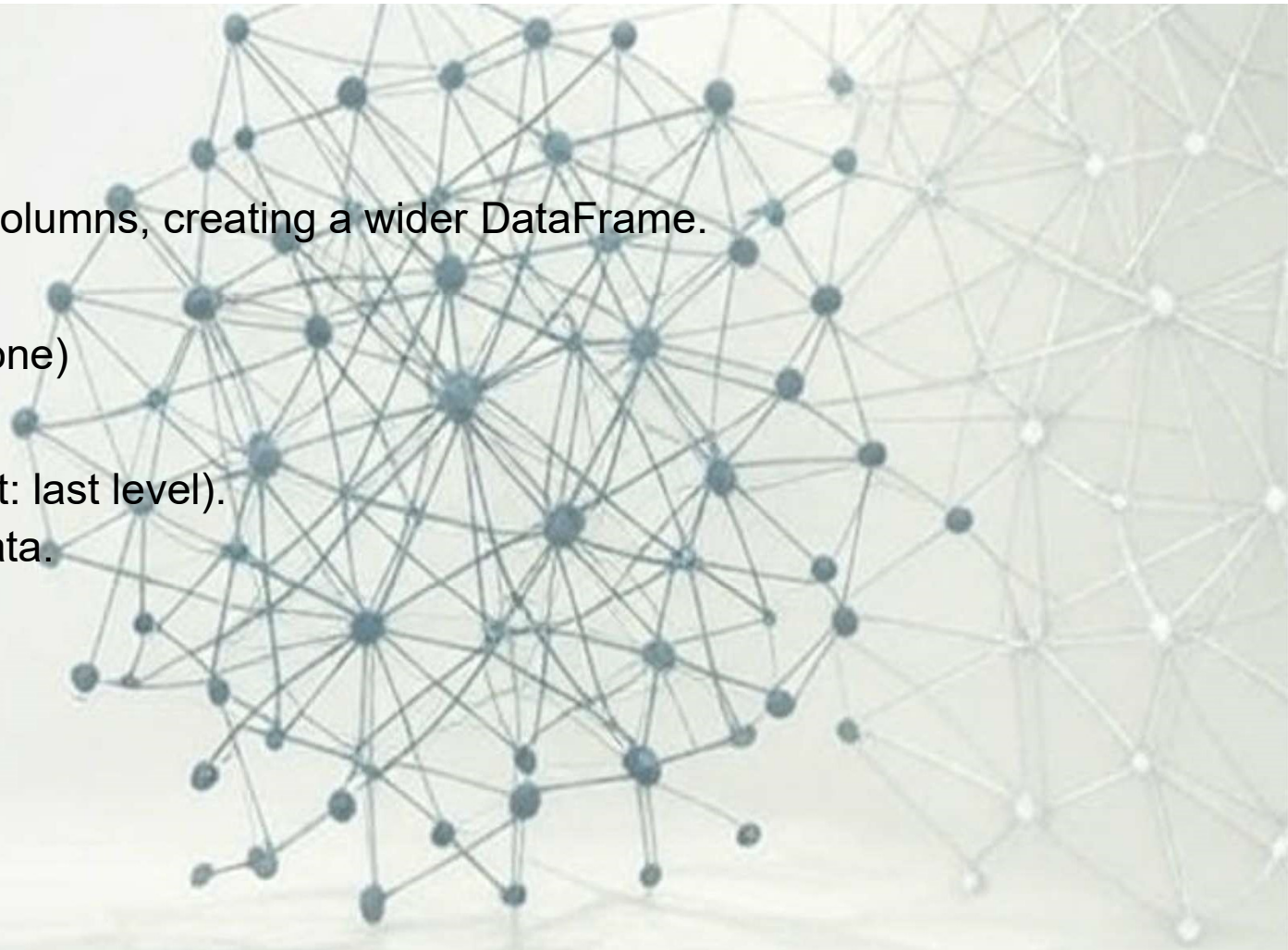
- Pivots a level of a MultiIndex to columns, creating a wider DataFrame.

- **Syntax:**

- `df.unstack(level=-1, fill_value=None)`

- **Key Parameters:**

- **level:** Level(s) to unstack (default: last level).
- **fill_value:** Value to fill missing data.



unstack()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]
}
- df = pd.DataFrame(data)
- **# Stack the DataFrame**
- stacked_df = df.set_index('Name').stack().reset_index()
- stacked_df.columns = ['Name', 'Attribute', 'Value']
- print(stacked_df)
- **# unstack() to pivot back to wide format**
- unstacked_df = stacked_df.set_index(['Name', 'Attribute']).unstack()
- # Reset index to make it a regular DataFrame
- unstacked_df.columns = unstacked_df.columns.droplevel(0)
- # Remove the top level of the column MultiIndex
- unstacked_df.reset_index(inplace=True)
- print('\n', unstacked_df)

• Output:

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

- Attribute Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- **Or if droplevel 1**
- Name Value Value
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000

transpose()

- **Description:**

- Transposes the DataFrame (swaps rows and columns).

- **Syntax:**

- `df.transpose(copy=True)`

- **Key Parameters:**

- **copy:** Return a copy if True.

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

- **Output:**

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

Or

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

explode()

- **Description:**

- Converts each element of a list-like column (e.g., lists, Series, or arrays) into separate rows, replicating the index for each element.

- **Syntax:**

- `df.explode(column, ignore_index=False)`

- **Key Parameters:**

- **column:** The column (or list of columns) containing list-like data to explode.
- **ignore_index:** If **True**, resets the index to default integer index (0 to n-1); if **False**, preserves the original index.

```
• import pandas as pd
• data = {
•     'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
•     'Hobbies': [['Reading', 'Hiking'], ['Cooking'], ['Gaming',
• 'Traveling']]
• }
• df = pd.DataFrame(data)
• exploded_df = df.explode('Hobbies')
• print(exploded_df)
```

- **Output:**

```
•   Name  Age  Hobbies
•  0  Alice  25  Reading
•  0  Alice  25  Hiking
•  1   Bob  30  Cooking
•  2  Charlie 35  Gaming
•  2  Charlie 35  Traveling
```


squeeze()

- **Description:**

- Converts a single-column or single-row DataFrame/Series into a scalar or Series.

- **Syntax:**

- `df.squeeze(axis=None)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns (optional, inferred if possible).

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

- **Output:**

- 0 25
- 1 30
- 2 35
- Name: Age, dtype: int64
- <class 'pandas.core.series.Series'>

mean()

- **Description:**

- Computes the mean of values.

- **Syntax:**

- `df.mean(axis=0)`

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.

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

- **Output:**

- Age 30.000000
- Salary 61666.666667
- dtype: float64

- **Or**

- 0 25012.5
- 1 30015.0
- 2 37517.5
- dtype: float64

median()

- **Description:**

- Computes the median of the DataFrame or Series along an axis

- **Syntax:**

- `df.median(axis=0, skipna=True, numeric_only=False)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).
- **numeric_only:** Only include numeric columns.

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

- **Output:**

```
• 0    25012.5
• 1    30015.0
• 2    37517.5
• dtype: float64
```

or

```
• Age      30.0
• Salary   60000.0
• dtype: float64
```


mode()

- **Description:**

- Computes the mode (most frequent value) of the DataFrame or Series along an axis.

- **Syntax:**

- `df.mode(axis=0, numeric_only=False, dropna=True)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **numeric_only:** Only include numeric columns.
- **dropna:** Ignore NA/null values.

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

- **Output:**

- 0 1
- 0 25 50000
- 1 30 60000
- 2 35 75000

• Or

- Age Salary
- 0 25 50000
- 1 30 60000
- 2 35 75000

Methods of DataFrame: Statistical and Mathematical Operations

Method	Syntax	Description
sum(axis)	df.sum(axis=0)	Computes the sum.
prod(axis)	df.prod(axis=0, skipna=True, numeric_only=False)	Computes the product.
min(axis)	df.min()	Returns the minimum.
max(axis)	df.max()	Returns the maximum.
std(axis)	df.std(axis=0)	Computes the standard deviation.
var(axis)	df.var(axis=0, skipna=True, ddof=1, numeric_only=False)	Computes the variance.
cov(other)	df.cov(min_periods=None, ddof=1)	Computes covariance with another Series or DataFrame.
corr(method)	df.corr(method='pearson')	Computes correlation (e.g., Pearson, Spearman).

sum()

- **Description:**

- Computes the sum of values along an axis.

- **Syntax:**

- `df.sum(axis=0)`

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.

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

- **Output:**

- Age 90
- Salary 185000
- dtype: int64

- **Or**

- 0 50025
- 1 60030
- 2 75035
- dtype: int64

prod()

- **Description:**

- Computes the product of values along an axis.

- **Syntax:**

- `df.prod(axis=0, skipna=True, numeric_only=False)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).
- **numeric_only:** Only include numeric columns.

```
import pandas as pd

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

df = pd.DataFrame(data)

row_product = df.prod(axis=0, numeric_only=True)

print("Product of rows:\n", row_product)

column_product = df.prod(axis=1, numeric_only=True)

print("Product of columns:\n", column_product)
```

- **Output:**

- Product of rows:
 - Age 26250
 - Salary 2250000000000000
 - dtype: int64
- Product of columns:
 - 0 1250000
 - 1 1800000
 - 2 2625000
 - dtype: int64

min()

- **Description:**

- Finds the minimum or maximum values

- **Syntax:**

- `df.min()`

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.

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

- **Output:**

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

max()

- **Description:**

- Finds the minimum or maximum values

- **Syntax:**

- df.max()

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.

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

- **Output:**

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

std()

- **Description:**
 - Computes the standard deviation
- **Syntax:**
 - `df.std(axis=0)`
- **Key Parameters:**
 - **axis:** 0 for columns, 1 for rows.

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

• df = pd.DataFrame(data)
• std_age = df['Age'].std()
• print(df);print("Standard deviation of Age:", std_age)
• std_salary = df['Salary'].std()
• print("Standard deviation of Salary:", std_salary)
```

• **Output:**

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

- Standard deviation of Age: 5.0
- Standard deviation of Salary: 12583.057392117915

var()

- **Description:**

- Computes the variance of values along an axis.

- **Syntax:**

- `df.var(axis=0, skipna=True, ddof=1, numeric_only=False)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).
- **ddof:** Delta degrees of freedom (default 1 for sample variance).
- **numeric_only:** Only include numeric columns.

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

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- Variance of Age: 25.0
- Variance of Salary: 158333333.3333333

cov()

- **Description:**

- Computes pairwise covariance of columns, excluding NA/null values.

- **Syntax:**

- `df.cov(min_periods=None, ddof=1)`

- **Key Parameters:**

- **min_periods:** Minimum number of observations required per pair.
- **ddof:** Delta degrees of freedom (default 1 for sample covariance).

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

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- Covariance between Age and Salary:
62500.0

corr()

- **Description:**

- Computes pairwise correlation of columns

- **Syntax:**

- `df.corr(method='pearson')`

- **Key Parameters:**

- **method:** 'pearson', 'spearman', or 'kendall'

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

• df = pd.DataFrame(data)
• print(df)

• corr_age_salary = df['Age'].corr(df['Salary'])
• print("Correlation between Age and Salary:",
    corr_age_salary)
```

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- Correlation between Age and Salary:
0.993399267798783

Methods of DataFrame: Statistical and Mathematical Operations

Method	Syntax	Description
quantile(q)	df.quantile(q=0.5, axis=0, numeric_only=True)	Computes quantiles.
cummax(axis)	df.cummax(axis=0, skipna=True)	Cumulative maximum.
cummin(axis)	df.cummin(axis=None, skipna=True)	Cumulative minimum.
cumsum(axis)	df.cumsum(axis=0, skipna=True)	Cumulative sum.
cumprod(axis)	df.cumprod(axis=0, skipna=True)	Cumulative product.
diff(periods)	df.diff(periods=1, axis=0)	Computes the difference between consecutive rows.
pct_change(periods)	df.pct_change(periods=1, fill_method='ffill', limit=None, freq=None)	Computes percentage change.
eq(other)	df.eq(other, axis='columns', level=None)	Element-wise equality comparison.

quantile()

- **Description:**

- Computes the quantile (e.g., median, percentiles) along an axis.

- **Syntax:**

- `df.quantile(q=0.5, axis=0, numeric_only=True)`

- **Key Parameters:**

- **q:** Quantile(s) to compute (0 to 1; default 0.5 for median).
- **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)
• median_values = df.quantile(q=0.5, axis=0,
    numeric_only=True)
• print("Median values for each numeric column:\n",
    median_values)
```

- **Output:**

- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- **Median values for each numeric column:**
- Age 30.0
- Salary 60000.0
- Name: 0.5, dtype: float64

cummax()

- **Description:**

- Computes the cumulative maximum of the DataFrame or Series along an axis

- **Syntax:**

- `df.cummax(axis=0, skipna=True)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000] }
• df = pd.DataFrame(data)
• c_max_age = df['Age'].cummax()
• print("Cumulative maximum of Age:\n", c_max_age)
• c_max_salary = df['Salary'].cummax()
• print("Cumulative maximum of Salary:\n", c_max_salary)
```

- **Output:**

- Cumulative maximum of Age:
- 0 25
- 1 30
- 2 35
- Name: Age, dtype: int64
- Cumulative maximum of Salary:
- 0 50000
- 1 60000
- 2 75000
- Name: Salary, dtype: int64

cummin()

- **Description:**

- Computes the cumulative minimum of values in a DataFrame or Series.

- **Syntax:**

- `df.cummin(axis=None, skipna=True)`

- **Key Parameters:**

- **axis:** 0 for columns, 1 for rows.
- **skipna:** If True, skips NaN values.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• c_min_age = df['Age'].cummin()
• print("Cumulative minimum of Age:\n", c_min_age)
• c_min_salary = df['Salary'].cummin()
• print("Cumulative minimum of Salary:\n", c_min_salary)
```

- **Output:**

- Cumulative minimum of Age:
- 0 25
- 1 25
- 2 25
- Name: Age, dtype: int64
- Cumulative minimum of Salary:
- 0 50000
- 1 50000
- 2 50000
- Name: Salary, dtype: int64

cumsum()

- **Description:**

- Computes the cumulative sum of the DataFrame or Series along an axis.

- **Syntax:**

- `df.cumsum(axis=0, skipna=True)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).

```
import pandas as pd

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

df = pd.DataFrame(data)

c_sum_age = df['Age'].cumsum()

print("Cumulative sum of Age:\n", c_sum_age)

c_sum_salary = df['Salary'].cumsum()

print("Cumulative sum of Salary:\n", c_sum_salary)
```

- **Output:**

- **Cumulative sum of Age:**
- 0 25
- 1 55
- 2 90
- Name: Age, dtype: int64
- **Cumulative sum of Salary:**
- 0 50000
- 1 110000
- 2 185000
- Name: Salary, dtype: int64

cumprod()

- **Description:**

- Computes the cumulative product of the DataFrame or Series along an axis.

- **Syntax:**

- `df.cumprod(axis=0, skipna=True)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for columns.
- **skipna:** Exclude NA/null values (default True).

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]
}
• df = pd.DataFrame(data)
• c_product_age = df['Age'].cumprod()
• print("Cumulative product of Age:\n", c_product_age)
• c_product_salary = df['Salary'].cumprod()
• print("Cumulative product of Salary:\n", c_product_salary)
```

- **Output:**

- **Cumulative product of Age:**
- 0 25
- 1 750
- 2 26250
- Name: Age, dtype: int64
- **Cumulative product of Salary:**
- 0 50000
- 1 3000000000
- 2 2250000000000000
- Name: Salary, dtype: int64

diff()

- **Description:**

- Computes the difference between consecutive elements along an axis (e.g., row-to-row or column-to-column).

- **Syntax:**

- `df.diff(periods=1, axis=0)`

- **Key Parameters:**

- **periods:** Number of periods to shift for differencing (default 1).
- **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)

difference_age = df['Age'].diff()

print("Difference of Age:\n", difference_age)

difference_salary = df['Salary'].diff()

print("Difference of Salary:\n", difference_salary)
```

- **Output:**

- **Difference of Age:**
- 0 NaN
- 1 5.0
- 2 5.0
- Name: Age, dtype: float64
- **Difference of Salary:**
- 0 NaN
- 1 10000.0
- 2 15000.0
- Name: Salary, dtype: float64

pct_change()

- **Description:**

- Computes the percentage change between consecutive elements along an axis.

- **Syntax:**

- `df.pct_change(periods=1, fill_method='ffill', limit=None, freq=None)`

- **Key Parameters:**

- **periods:** Number of periods to shift (default 1).
- **fill_method:** Method to fill NA values before calculation (deprecated in newer versions).
- **limit:** Maximum number of consecutive NAs to fill.

```
import pandas as pd

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

df = pd.DataFrame(data)

percentage_change_age = df['Age'].pct_change()

print("Percentage change of Age:\n", percentage_change_age)

percentage_change_salary = df['Salary'].pct_change()

print("Percentage change of Salary:\n", percentage_change_salary)
```

- **Output:**

- **Percentage change of Age:**

- 0 NaN
- 1 0.200000
- 2 0.166667
- Name: Age, dtype: float64

- **Percentage change of Salary:**

- 0 NaN
- 1 0.20
- 2 0.25
- Name: Salary, dtype: float64

eq()

- **Description:**

- Element-wise equality comparison with another DataFrame, Series, or scalar.

- **Syntax:**

- `df.eq(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex

```
import pandas as pd

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

df = pd.DataFrame(data)

age_comparison = df['Age'].eq(30)
salary_comparison = df['Salary'].eq(60000)

print("Age Comparison (Age == 30):", age_comparison)

print("\nSalary Comparison (Salary == 60000):", salary_comparison)
```

- **Output:**

- **Age Comparison (Age == 30):**
- 0 False
- 1 True
- 2 False
- Name: Age, dtype: bool
- **Salary Comparison (Salary == 60000):**
- 0 False
- 1 True
- 2 False
- Name: Salary, dtype: bool

Methods of DataFrame: Missing Data Handling

Method	Syntax	Description
ne(other)	df.ne(other, axis='columns', level=None)	Element-wise not equal.
lt(other)	df.lt(other, axis='columns', level=None)	Element-wise comparisons.
gt(other)	df.gt(other, axis='columns', level=None)	Element-wise greater-than comparison with another DataFrame, Series, or scalar
le(other)	df.le(other, axis='columns', level=None)	Element-wise less-than-or-equal comparison with another DataFrame, Series, or scalar
ge(other)	df.ge(other, axis='columns', level=None)	Element-wise greater-than-or-equal comparison with another DataFrame, Series, or scalar.
isna()	df.isna()	Detects missing values (returns boolean DataFrame).
notna()	df.notna()	Detects non-missing values.
isnull()	df.isnull()	Alias for `isna`.
notnull()	df.notna()	Alias for `notna`.

ne()

- **Description:**

- Element-wise not-equal comparison with another DataFrame, Series, or scalar

- **Syntax:**

- `df.ne(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex.

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

- **Output:**

- **Age Comparison (Age != 30):**
- 0 True
- 1 False
- 2 True
- Name: Age, dtype: bool
- **Salary Comparison (Salary != 60000):**
- 0 True
- 1 False
- 2 True
- Name: Salary, dtype: bool

lt()

- **Description:**

- Element-wise less-than comparison with another DataFrame, Series, or scalar

- **Syntax:**

- `df.lt(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex.

```
import pandas as pd

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

df = pd.DataFrame(data)

age_comparison = df['Age'].lt(30)

salary_comparison = df['Salary'].lt(60000)

print("Age Comparison (Age < 30):\n", age_comparison)

print("Salary Comparison (Salary < 60000):\n", salary_comparison)
```

- **Output:**

- **Age Comparison (Age < 30):**

- 0 True
- 1 False
- 2 False
- Name: Age, dtype: bool

- **Salary Comparison (Salary < 60000):**

- 0 True
- 1 False
- 2 False
- Name: Salary, dtype: bool

lt()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]}
- df = pd.DataFrame(data)
- print(df)
- age_comp_multiple = df['Age'].lt(30) | df['Age'].lt(35)
- salary_comp_multiple = df['Salary'].lt(50000) | df['Salary'].lt(60000)
- print("Age Comparison (Age < 30 or < 35):", age_comp_multiple)
- print("Salary Comparison (Salary < 50000 or < 60000):", salary_comp_multiple)

- **Output:**
- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- **Age Comparison (Age < 30 or < 35):**
- 0 True
- 1 True
- 2 False
- Name: Age, dtype: bool
- **Salary Comparison (Salary < 50000 or < 60000):**
- 0 True
- 1 False
- 2 False
- Name: Salary, dtype: bool

gt()

- **Description:**

- Element-wise greater-than comparison with another DataFrame, Series, or scalar

- **Syntax:**

- `df.gt(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex

```
import pandas as pd

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

df = pd.DataFrame(data)

age_comparison = df['Age'].gt(30)

salary_comparison = df['Salary'].gt(60000)

print("Age Comparison (Age > 30):\n", age_comparison)

print("Salary Comparison (Salary > 60000):\n",
      salary_comparison)
```

- **Output:**

- **Age Comparison (Age > 30):**
- 0 False
- 1 False
- 2 True
- Name: Age, dtype: bool
- **Salary Comparison (Salary > 60000):**
- 0 False
- 1 False
- 2 True
- Name: Salary, dtype: bool

gt()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]}
- df = pd.DataFrame(data)
- print(df)
- **# Multiple GT Comparison**
- age_comp_multiple = df['Age'].gt(30) |
df['Age'].gt(35) **# Second condition is redundant**
- salary_comp_multiple = df['Salary'].gt(50000) |
df['Salary'].gt(60000)
- print("Age Comparison (Age > 30 or > 35):\n",
age_comp_multiple)
- print("Salary Comparison (Salary > 50000 or >
60000):\n", salary_comp_multiple)

- **Output:**
- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- **Age Comparison (Age > 30 or > 35):**
- 0 False
- 1 False
- 2 True
- Name: Age, dtype: bool
- **Salary Comparison (Salary > 50000 or > 60000):**
- 0 False
- 1 True
- 2 True
- Name: Salary, dtype: bool

le()

- **Description:**

- Element-wise less-than-or-equal comparison with another DataFrame, Series, or scalar.

- **Syntax:**

- `df.le(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000] }
• df = pd.DataFrame(data)
• age_comparison = df['Age'].le(30)
• salary_comparison = df['Salary'].le(60000)
• print("Age Comparison (Age ≤ 30):\n", age_comparison)
• print("Salary Comparison (Salary ≤ 60000):\n",
    salary_comparison)
```

- **Output:**

- **Age Comparison (Age ≤ 30):**

- 0 True
- 1 True
- 2 False
- Name: Age, dtype: bool

- **Salary Comparison (Salary ≤ 60000):**

- 0 True
- 1 True
- 2 False
- Name: Salary, dtype: bool

le()

- `import pandas as pd`
- `data = {`
 `'Name': ['Alice', 'Bob', 'Charlie'],`
 `'Age': [25, 30, 35],`
 `'Salary': [50000, 60000, 75000]}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Multiple LE Comparison**
- `age_comp_multiple = df['Age'].le(30) |`
 `df['Age'].le(35)`
- **# Second condition includes all ages**
- `salary_comp_multiple = df['Salary'].le(50000) |`
 `df['Salary'].le(60000)`
- `print("Age Comparison (Age \leq 30 or \leq 35):\n",`
 `age_comp_multiple)`
- `print("Salary Comparison (Salary \leq 50000 or \leq`
 `60000):\n", salary_comp_multiple)`

- **Output:**
- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | Alice | 25 | 50000 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |
- **Age Comparison (Age \leq 30 or \leq 35):**
- | | |
|---|------|
| 0 | True |
| 1 | True |
| 2 | True |
- `Name: Age, dtype: bool`
- **Salary Comparison (Salary \leq 50000 or \leq 60000):**
- | | |
|---|-------|
| 0 | True |
| 1 | True |
| 2 | False |
- `Name: Salary, dtype: bool`

ge()

- **Description:**

- Element-wise greater-than-or-equal comparison with another DataFrame, Series, or scalar.

- **Syntax:**

- `df.ge(other, axis='columns', level=None)`

- **Key Parameters:**

- **other:** DataFrame, Series, or scalar to compare.
- **axis:** 0 for rows, 1 for columns.
- **level:** Level to broadcast in MultiIndex.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]}
• df = pd.DataFrame(data)
• age_comparison = df['Age'].ge(30)
• salary_comparison = df['Salary'].ge(60000)
• print("Age Comparison (Age ≥ 30):\n", age_comparison)
• print("Salary Comparison (Salary ≥ 60000):\n",
    salary_comparison)
```

- **Output:**

- **Age Comparison (Age ≥ 30):**

- 0 False
- 1 True
- 2 True
- Name: Age, dtype: bool

- **Salary Comparison (Salary ≥ 60000):**

- 0 False
- 1 True
- 2 True
- Name: Salary, dtype: bool

ge()

- import pandas as pd
- data = {
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000]}
- df = pd.DataFrame(data)
- print(df)
- **# Multiple GE Comparison Example**
- age_comp_multiple = df['Age'].ge(30) |
df['Age'].ge(35)
- **# Second condition is more restrictive**
- salary_comp_multiple = df['Salary'].ge(50000) |
df['Salary'].ge(60000)
- print("Age Comparison (Age \geq 30 or \geq 35):\n",
age_comp_multiple)
- print("Salary Comparison (Salary \geq 50000 or \geq
60000):\n", salary_comp_multiple)

- **Output:**
- Name Age Salary
- 0 Alice 25 50000
- 1 Bob 30 60000
- 2 Charlie 35 75000
- **Age Comparison (Age \geq 30 or \geq 35):**
- 0 False
- 1 True
- 2 True
- Name: Age, dtype: bool
- **Salary Comparison (Salary \geq 50000 or \geq 60000):**
- 0 True
- 1 True
- 2 True
- Name: Salary, dtype: bool

isna()

- **Description:**

- Detects missing values, returning a boolean DataFrame

- **Syntax:**

- df.isna()

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

- **Output:**

```
• Name Age Salary
• 0 Alice 25.0 50000.0
• 1 Bob 30.0 NaN
• 2 NaN NaN NaN
• 3 Charlie 35.0 75000.0
• Name column isna():
• 0 False
• 1 False
• 2 True
• 3 False
• Name: Name, dtype: bool
```

- **Output:**

```
• Age column isna():
• 0 False
• 1 False
• 2 True
• 3 False
• Name: Age, dtype: bool
• Salary column isna():
• 0 False
• 1 True
• 2 True
• 3 False
• Name: Salary, dtype: bool
```


isna()

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

- print("\nEntire DataFrame isna() check:")
- print(df.isna())

- print("\nCount of missing values per column:")
- print(df.isna().sum())

- **Output:**
- Name Age Salary
- 0 Alice 25.0 50000.0
- 1 Bob 30.0 NaN
- 2 NaN NaN NaN
- 3 Charlie 35.0 75000.0
- **Entire DataFrame isna() check:**
- Name Age Salary
- 0 False False False
- 1 False False True
- 2 True True True
- 3 False False False
- **Count of missing values per column:**
- Name 1
- Age 1
- Salary 2
- dtype: int64

notna()

- **Description:**

- Detects non-missing values, returning a boolean DataFrame

- **Syntax:**

- df.notna()

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

- **Output:**

```
•      Name  Age  Salary
•  0  Alice  25.0  50000.0
•  1   Bob  30.0    NaN
•  2   NaN  NaN    NaN
•  3  Charlie  35.0  75000.0
• Name column notna():
•  0    True
•  1    True
•  2   False
•  3    True
• Name: Name, dtype: bool
```

- **Output:**

```
• Age column notna():
•  0    True
•  1    True
•  2   False
•  3    True
• Name: Age, dtype: bool
• Salary column notna():
•  0    True
•  1   False
•  2   False
•  3    True
• Name: Salary, dtype: bool
```

notna()

- import pandas as pd
- import numpy as np
- data = {
 'Name': ['Alice', 'Bob', np.nan, 'Charlie'],
 'Age': [25, 30, np.nan, 35],
 'Salary': [50000, np.nan, np.nan, 75000]
}
- df = pd.DataFrame(data)
- print(df)
- print("\nEntire DataFrame notna() check:")
- print(df.notna())
- print("\nCount of non-missing values per column:")
- print(df.notna().sum())

- **Output:**
- Name Age Salary
- 0 Alice 25.0 50000.0
- 1 Bob 30.0 NaN
- 2 NaN NaN NaN
- 3 Charlie 35.0 75000.0
- **Entire DataFrame notna() check:**
- Name Age Salary
- 0 True True True
- 1 True True False
- 2 False False False
- 3 True True True
- **Count of non-missing values per column:**
- Name 3
- Age 3
- Salary 2
- dtype: int64

isnull()

- **Description:**

- Detects missing values, returning a boolean DataFrame

- **Syntax:**

- df.isnull()

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

- **Output:**

```
• Name Age Salary
• 0 Alice 25.0 50000.0
• 1 Bob 30.0 NaN
• 2 NaN NaN NaN
• 3 Charlie 35.0 75000.0
• Name column isnull():
• 0 False
• 1 False
• 2 True
• 3 False
• Name: Name, dtype: bool
```

- **Output:**

```
• Age column isnull():
• 0 False
• 1 False
• 2 True
• 3 False
• Name: Age, dtype: bool
• Salary column isnull():
• 0 False
• 1 True
• 2 True
• 3 False
• Name: Salary, dtype: bool
```

notnull()

- **Description:**

- Detects non-missing values, returning a boolean DataFrame

- **Syntax:**

- df.notnull()

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

- **Output:**

```
•      Name  Age  Salary
•  0  Alice  25.0  50000.0
•  1   Bob  30.0    NaN
•  2  NaN  NaN    NaN
•  3  Charlie  35.0  75000.0
• Name column notnull():
•  0    True
•  1    True
•  2   False
•  3    True
• Name: Name, dtype: bool
```

- **Output:**

```
• Age column notnull():
•  0    True
•  1    True
•  2   False
•  3    True
• Name: Age, dtype: bool
• Salary column notnull():
•  0    True
•  1   False
•  2   False
•  3    True
• Name: Salary, dtype: bool
```

Methods of DataFrame: String Operations & Time Series Operations

Method	Syntax	Description
str.lower()	df['column'].str.lower()	Converts strings in a Series to lowercase
str.upper()	df['column'].str.upper()	Converts strings in a Series to uppercase
str.strip()	df['column'].str.strip(to_strip=None)	Removes leading and trailing whitespace from strings in a Series.
str.contains()	df['column'].str.contains(pat, case=True, na=None, regex=True)	Tests if a pattern or regex is contained within a string Series.
str.split()	df['column'].str.split(pat=None, n=-1, expand=False)	Splits strings in a Series on a delimiter into a list or DataFrame
resample(rule)	df.resample(rule, axis=0, closed='right', label='right', convention='end', kind=None, on=None, level=None)	Resamples time-series data (e.g., daily to monthly).
shift(periods, axis)	df.shift(periods=1, freq=None, axis=0, fill_value=None)	Shifts data by a number of periods.

str.lower()

- **Description:**

- Converts strings in a Series to lowercase.

- **Syntax:**

- `df['column'].str.lower()`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]}
• df = pd.DataFrame(data)
• print(df)
• df['Name_Lower'] = df['Name'].str.lower()
• print("DataFrame with lowercase names:")
• print(df)
• # Case-insensitive filtering to find names containing 'bob'
• search_term = 'bOb'
• matches = df[df['Name'].str.lower().str.contains(search_term.lower())]
• print("\nCase-insensitive search for 'bob':")
• print(matches)
```

- **Output:**

```
•      Name Age Salary
• 0  Alice  25  50000
• 1   Bob  30  60000
• 2 Charlie 35  75000
• DataFrame with lowercase names:
•      Name Age Salary Name_Lower
• 0  Alice  25  50000    alice
• 1   Bob  30  60000     bob
• 2 Charlie 35  75000   Charlie
• Case-insensitive search for 'bob':
•      Name Age Salary Name_Lower
• 1   Bob  30  60000     bob
```

str.upper()

- **Description:**

- Converts strings in a Series to uppercase

- **Syntax:**

- `df['column'].str.upper()`

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000]}
• df = pd.DataFrame(data)
• print(df)
• df['Name_Upper'] = df['Name'].str.upper()
• print("DataFrame with uppercase names:")
• print(df)
• # Case-insensitive filtering using uppercase
• search_term = 'bOb'
• matches = df[df['Name'].str.upper().str.contains(search_term.upper())]
• print("\nCase-insensitive search for 'BOB':")
• print(matches)
```

- **Output:**

```
•      Name Age Salary
• 0  Alice  25  50000
• 1   Bob  30  60000
• 2 Charlie  35  75000
• DataFrame with uppercase names:
•      Name Age Salary Name_Upper
• 0  Alice  25  50000    ALICE
• 1   Bob  30  60000     BOB
• 2 Charlie  35  75000   CHARLIE
• Case-insensitive search for 'BOB':
•      Name Age Salary Name_Upper
• 1  Bob  30  60000     BOB
```

str.strip()

- **Description:**

- Removes leading and trailing whitespace from strings in a Series.

- **Syntax:**

- `df['column'].str.strip(to_strip=None)`

- **Key Parameters:**

- **to_strip:** Specific characters to strip (default: whitespace).

```
• import pandas as pd
• data = {
    'Name': [' Alice ', ' Bob ', 'Charlie ', ' Diana'],
    'Age': [25, 30, 35, 28],
    'Salary': [50000, 60000, 75000, 65000] }
• df = pd.DataFrame(data)
• df['length'] = df['Name'].str.len()
• print(df)
• df['Name_clean'] = df['Name'].str.strip()
• df['length_clean'] = df['Name_clean'].str.len()
• print(df)
```

- **Output:**

```
•      Name Age Salary length
• 0  Alice  25  50000     9
• 1   Bob  30  60000     5
• 2 Charlie 35  75000     9
• 3  Diana 28  65000     7
•      Name Age Salary length Name_clean length_clean
• 0  Alice  25  50000     9    Alice           5
• 1   Bob  30  60000     5     Bob           3
• 2 Charlie 35  75000     9   Charlie           7
• 3  Diana 28  65000     7    Diana           5
```


str.replace()

- **Description:**

- Replaces occurrences of a specified substring or pattern in string columns with a new value. Works on Series with string dtype or object columns containing strings.

- **Syntax:**

- `df.str.replace(pat, repl, n=-1, case=True, regex=False)`

- **Key Parameters:**

- **pat:** String or regex pattern to match.
- **repl:** String or function to replace matched patterns.
- **n:** Number of replacements to make (default: -1, replace all).
- **case:** If True, performs case-sensitive matching; if False, case-insensitive (applies when `regex=False`).
- **regex:** If True, treats `pat` as a regular expression; if False, treats it as a literal string.

str.replace()

- `import pandas as pd`
- `data = {`
 - `'Name': ['Alice-Smith', 'Bob_Jones', 'Charlie Brown', 'Diana.Miller'],`
 - `'Age': [25, 30, 35, 28],`
 - `'Phone': ['(555)123-4567', '555.789.1234', '555 987 6543', '555-456-7890'],`
 - `'Email': ['alice@old.com', 'bob@old.com', 'charlie@old.com', 'diana@old.com']``}`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Basic character replacement**
- `df['Name_Clean'] = df['Name'].str.replace(r'[-_.]', ' ', regex=True)`
- `print("\nReplace special characters with spaces:")`
- `print(df[['Name', 'Name_Clean']])`

- **Output:**

- Name Age Phone Email
- 0 Alice-Smith 25 (555)123-4567 alice@old.com
- 1 Bob_Jones 30 555.789.1234 bob@old.com
- 2 Charlie Brown 35 555 987 6543 charlie@old.com
- 3 Diana.Miller 28 555-456-7890 diana@old.com

- **Replace special characters with spaces:**

- Name Name_Clean
- 0 Alice-Smith Alice Smith
- 1 Bob_Jones Bob Jones
- 2 Charlie Brown Charlie Brown
- 3 Diana.Miller Diana Miller

str.replace()

- **# Phone number standardization**
- `df['Phone_Clean'] = df['Phone'].str.replace(r'[()\-\ .]', '', regex=True)`
- `print("\nStandardized phone numbers:")`
- `print(df[['Phone', 'Phone_Clean']])`
- **# Domain replacement in emails**
- `df['Email_New'] = df['Email'].str.replace('@old.com', '@new.org')`
- `print("\nUpdated email domains:")`
- `print(df[['Email', 'Email_New']])`
- **# Case-insensitive replacement**
- `df['Name_NoBob'] = df['Name'].str.replace('bob', 'Robert', case=False)`
- `print("\nCase-insensitive name replacement:")`
- `print(df[['Name', 'Name_NoBob']])`

- **Output:**
- **Standardized phone numbers:**
- | | Phone | Phone_Clean |
|---|---------------|-------------|
| 0 | (555)123-4567 | 5551234567 |
| 1 | 555.789.1234 | 5557891234 |
| 2 | 555 987 6543 | 5559876543 |
| 3 | 555-456-7890 | 5554567890 |
- **Updated email domains:**
- | | Email | Email_New |
|---|-----------------|-----------------|
| 0 | alice@old.com | alice@new.org |
| 1 | bob@old.com | bob@new.org |
| 2 | charlie@old.com | charlie@new.org |
| 3 | diana@old.com | diana@new.org |
- **Case-insensitive name replacement:**
- | | Name | Name_NoBob |
|---|---------------|---------------|
| 0 | Alice-Smith | Alice-Smith |
| 1 | Bob_Jones | Robert_Jones |
| 2 | Charlie Brown | Charlie Brown |
| 3 | Diana.Miller | Diana.Miller |

str.replace()

- **# Multi-pattern replacement**
- `df['Name_Final'] = (df['Name']
 .str.replace(r'[-_]', ' ', regex=True) # Replace separators
 .str.replace(r'\s+', ' ', regex=True) # Collapse spaces
 .str.title()) # Title case`
- `print("\nMulti-step name cleaning:")`
- `print(df[['Name', 'Name_Final']])`
- **# Replacement with captured groups**
- `df['Last_First'] = df['Name'].str.replace(r'(\w+)[-_](\w+)', r'\2, \1',
 regex=True)`
- `print("\nLast, First format:")`
- `print(df[['Name', 'Last_First']])`

- **Output:**

- Multi-step name cleaning:
- | | Name | Name_Final |
|-----|---------------|---------------|
| • 0 | Alice-Smith | Alice Smith |
| • 1 | Bob_Jones | Bob Jones |
| • 2 | Charlie Brown | Charlie Brown |
| • 3 | Diana.Miller | Diana Miller |
- Last, First format:
- | | Name | Last_First |
|-----|---------------|----------------|
| • 0 | Alice-Smith | Smith, Alice |
| • 1 | Bob_Jones | Jones, Bob |
| • 2 | Charlie Brown | Brown, Charlie |
| • 3 | Diana.Miller | Diana.Miller |

str.contains()

- **Description:**

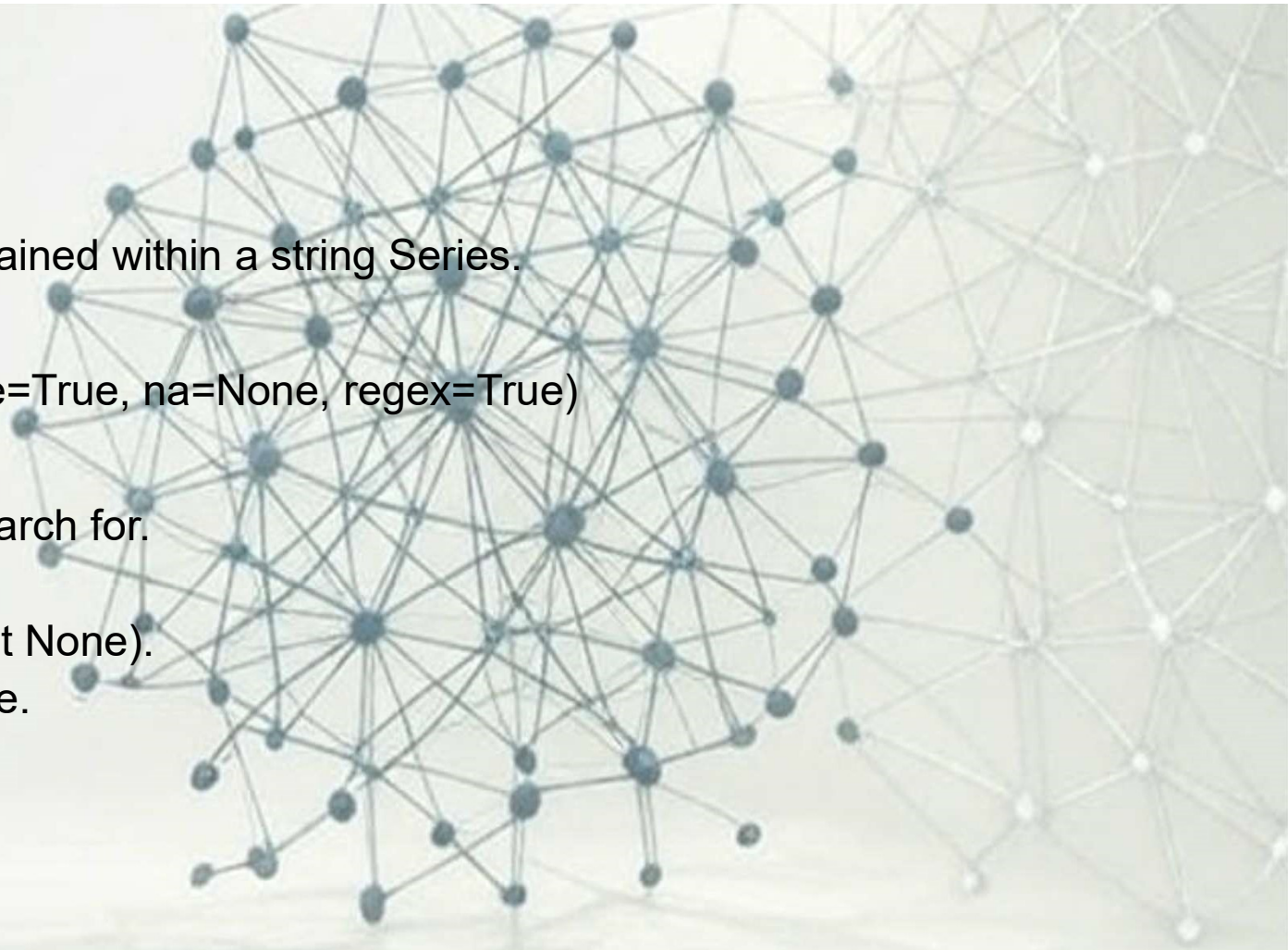
- Tests if a pattern or regex is contained within a string Series.

- **Syntax:**

- `df['column'].str.contains(pat, case=True, na=None, regex=True)`

- **Key Parameters:**

- **pat:** String or regex pattern to search for.
- **case:** Case-sensitive if True.
- **na:** Value to return for NA (default None).
- **regex:** Treat `pat` as regex if True.



str.contains()

- import pandas as pd
- data = {
 'Name': ['Alice Johnson', 'Bob Smith', 'Charlie Brown', 'Diana Williams', 'Eva Smithson'],
 'Department': ['Marketing', 'HR', 'IT Support', 'Finance', 'HR Management'],
 'Email': ['alice.j@company.com', 'bob.smith@company.com', 'charlie@company.com', 'diana.w@company.org', 'eva@company.net']
}
- df = pd.DataFrame(data)
- print("Original DataFrame:")
- print(df)
- **# Basic contains check**
- hr_employees = df[df['Department'].str.contains('HR')]
- print("\n Employees in HR departments:")
- print(hr_employees)

• Output:

• Original DataFrame:

	Name	Department	Email
• 0	Alice Johnson	Marketing	alice.j@company.com
• 1	Bob Smith	HR	bob.smith@company.com
• 2	Charlie Brown	IT Support	charlie@company.com
• 3	Diana Williams	Finance	diana.w@company.org
• 4	Eva Smithson	HR Management	eva@company.net

• Employees in HR departments:

	Name	Department	Email
• 1	Bob Smith	HR	bob.smith@company.com
• 4	Eva Smithson	HR Management	eva@company.net

str.contains()

- **# Case-insensitive search**
- `smith_employees = df[df['Name'].str.contains('smith', case=False)]`
- `print("\n Employees with 'smith' in name (case-insensitive):")`
- `print(smith_employees)`
- **# Regex pattern matching**
- `email_pattern = r'\.org$' # Ends with .org`
- `org_emails = df[df['Email'].str.contains(email_pattern, regex=True)]`
- `print("\n Employees with .org email addresses:")`
- `print(org_emails)`
- **# Multiple patterns**
- `management_pattern = r'Management|Support'`
- `special_depts = df[df['Department'].str.contains(management_pattern)]`
- `print("\n Employees in management or support roles:")`
- `print(special_depts)`

- **Output:**
- **Employees with 'smith' in name (case-insensitive):**
- | | Name | Department | Email |
|---|--------------|---------------|-----------------------|
| 1 | Bob Smith | HR | bob.smith@company.com |
| 4 | Eva Smithson | HR Management | eva@company.net |
- **Employees with .org email addresses:**
- | | Name | Department | Email |
|---|----------------|------------|---------------------|
| 3 | Diana Williams | Finance | diana.w@company.org |
- **Employees in management or support roles:**
- | | Name | Department | Email |
|---|---------------|---------------|---------------------|
| 2 | Charlie Brown | IT Support | charlie@company.com |
| 4 | Eva Smithson | HR Management | eva@company.net |

str.contains()

- **# Creating new boolean columns**
- `df['Has_Middle_Initial'] = df['Name'].str.contains(r'\w+ \w+ \w+')`
- `df['Uses_Company_Domain'] = df['Email'].str.contains(r'@company\.'`
- `print("\n DataFrame with new boolean columns:")`
- `print(df)`

- **Output:**

- DataFrame with new boolean columns:

	Name	Department	Email	Has_Middle_Initial	Uses_Company_Domain
0	Alice Johnson	Marketing	alice.j@company.com	False	True
1	Bob Smith	HR	bob.smith@company.com	False	True
2	Charlie Brown	IT Support	charlie@company.com	False	True
3	Diana Williams	Finance	diana.w@company.org	False	True
4	Eva Smithson	HR Management	eva@company.net	False	True

str.contains()

- **# Counting matches**

- `dot_in_name_count = df['Name'].str.contains(r'\.').sum()`
- `print(f"\n Number of names containing dots: {dot_in_name_count}")`

- **# Complex pattern with groups**

- `initial_pattern = r'^(\w+)\.(\w+)@'`
- `initial_matches = df[df['Email'].str.contains(initial_pattern)]`
- `print("\n Employees with first.last email pattern:")`
- `print(initial_matches)`

- **Output:**

- Number of names containing dots: 0
- Employees with first.last email pattern:

	Name	Department	Email	Has_Middle_Initial	Uses_Company_Domain
0	Alice Johnson	Marketing	alice.j@company.com	False	True
1	Bob Smith	HR	bob.smith@company.com	False	True
3	Diana Williams	Finance	diana.w@company.org	False	True

str.split()

- **Description:**

- Splits strings in a Series on a delimiter into a list or DataFrame

- **Syntax:**

- `df['column'].str.split(pat=None, n=-1, expand=False)`

- **Key Parameters:**

- **pat:** Delimiter or regex pattern (default: whitespace).
- **n:** Maximum number of splits.
- **expand:** Return DataFrame if True, list if False.

- `import pandas as pd`

- `data = {`

- `'Full_Name': ['Alice Marie Johnson', 'Bob Lee Smith', 'Charlie Brown', 'Diana Williams', 'Eva Grace Smithson'],`

- `'Department': ['Marketing|Sales', 'HR|Recruiting', 'IT Support', 'Finance|Accounting', 'HR Management'],`

- `'Email': ['alice.j@company.com', 'bob.smith@company.com', 'charlie@company.com', 'diana.w@company.org', 'eva@company.net'],`

- `'Address': ['123 Main St, New York, NY', '456 Oak Ave, Boston, MA', '789 Pine Rd, Chicago, IL', '101 Elm St, San Francisco, CA', '202 Maple Dr, Seattle, WA'] }`

- `df = pd.DataFrame(data)`

- **# code will continue in next slid**

- `print(df)`
- `#split on whitespace (default)`
- `df[['First', 'Middle', 'Last']] = df['Full_Name'].str.split(expand=True)`
- `print("\nName split into columns (default whitespace):")`
- `print(df[['Full_Name', 'First', 'Middle', 'Last']])`

• **Output:**

	Full_Name	Department	Email	Address
• 0	Alice Marie Johnson	Marketing Sales	alice.j@company.com	123 Main St, New York, NY
• 1	Bob Lee Smith	HR Recruiting	bob.smith@company.com	456 Oak Ave, Boston, MA
• 2	Charlie Brown	IT Support	charlie@company.com	789 Pine Rd, Chicago, IL
• 3	Diana Williams	Finance Accounting	diana.w@company.org	101 Elm St, San Francisco, CA
• 4	Eva Grace Smithson	HR Management	eva@company.net	202 Maple Dr, Seattle, WA

• **Name split into columns (default whitespace):**

	Full_Name	First	Middle	Last
• 0	Alice Marie Johnson	Alice	Marie	Johnson
• 1	Bob Lee Smith	Bob	Lee	Smith
• 2	Charlie Brown	Charlie	Brown	None
• 3	Diana Williams	Diana	Williams	None
• 4	Eva Grace Smithson	Eva	Grace	Smithson

Use case

- **# Split with specific delimiter**

- `df[['Dept1', 'Dept2']] = df['Department'].str.split('|', expand=True)`
- `print("\nDepartment split on pipe character:")`
- `print(df[['Department', 'Dept1', 'Dept2']])`

- **# Split with max splits parameter**

- `df[['Street', 'City_State']] = df['Address'].str.split(',', n=1, expand=True)`
- `print("\n Address split with maxsplit=1:")`
- `print(df[['Address', 'Street', 'City_State']])`

- **# Split and keep original column**

- `df['Name_Parts'] = df['Full_Name'].str.split()`
- `print("\n Name parts as list:")`
- `print(df[['Full_Name', 'Name_Parts']])`

- **Output: “ For output run this code”**

Use case

- **# Split with regex pattern**
 - `df['Email_Parts'] = df['Email'].str.split(r'[@\.]', regex=True)`
 - `print("\n Email split with regex pattern:")`
 - `print(df[['Email', 'Email_Parts']])`
- **# Accessing split elements directly**
 - `df['First_Initial'] = df['Full_Name'].str.split().str[0].str[0]`
 - `df['Last_Name'] = df['Full_Name'].str.split().str[-1]`
 - `print("\n Extracted name components:")`
 - `print(df[['Full_Name', 'First_Initial', 'Last_Name']])`
- **# Split and count elements**
 - `df['Name_Word_Count'] = df['Full_Name'].str.split().str.len()`
 - `print("\n7. Name word counts:")`
 - `print(df[['Full_Name', 'Name_Word_Count']])`
- **Output: “ For output run this code”**

resample()

- **Description:**

- Resamples time-series data to a new frequency (e.g., daily to monthly).

- **Syntax:**

- `df.resample(rule, axis=0, closed='right', label='right', convention='end', kind=None, on=None, level=None)`

- **Key Parameters:**

- **rule:** Frequency string (e.g., 'D' for daily, 'M' for monthly).
- **closed:** Which side of interval is closed ('right' or 'left').
- **label:** Which bin edge to label ('right' or 'left').
- **on:** Column to use for time index (if not index).

```
• import pandas as pd
• import numpy as np
• date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
• data = {
    'Sales': np.random.randint(100, 500, size=(len(date_rng))),
    'Expenses': np.random.randint(50, 300, size=(len(date_rng))) }
• df = pd.DataFrame(data, index=date_rng)
• print("Original DataFrame:")
• print(df)
```

- **# Resample to weekly frequency and sum**
- `weekly_summary = df.resample('W').sum()`
- `print("\nWeekly summary (sum of Sales and Expenses):")`
- `print(weekly_summary)`

- **Output:**

	Sales	Expenses
--	-------	----------

- | | | |
|--------------|-----|-----|
| • 2025-01-01 | 330 | 259 |
| • 2025-01-02 | 379 | 237 |
| • 2025-01-03 | 124 | 97 |
| • 2025-01-04 | 335 | 267 |
| • 2025-01-05 | 167 | 218 |
| • 2025-01-06 | 199 | 199 |
| • 2025-01-07 | 142 | 189 |
| • 2025-01-08 | 258 | 291 |
| • 2025-01-09 | 126 | 196 |
| • 2025-01-10 | 467 | 208 |

- **Output:**

- **Weekly summary (sum of Sales and Expenses):**

- | | Sales | Expenses |
|--------------|-------|----------|
| • 2025-01-05 | 1335 | 1078 |
| • 2025-01-12 | 1192 | 1083 |

Use Case

- **# Resample to monthly frequency and calculate mean**
- `monthly_average = df.resample('M').mean()`
- `print("\nMonthly average (mean of Sales and Expenses):")`
- `print(monthly_average)`

- **# Resample to daily frequency and calculate the maximum**
- `daily_max = df.resample('D').max()`
- `print("\nDaily maximum (max of Sales and Expenses):")`
- `print(daily_max)`

- **# Resample with a custom aggregation function**
- `custom_resample = df.resample('2D').agg({'Sales': 'sum', 'Expenses': 'mean'})`
- `print("\nCustom resampling (2-day sum of Sales and mean of Expenses):")`
- `print(custom_resample)`

- **Output:: “ For output run this code”**

shift()

- **Description:**

- Shifts data by a specified number of periods along an axis.

- **Syntax:**

- `df.shift(periods=1, freq=None, axis=0, fill_value=None)`

- **Key Parameters:**

- **periods:** Number of periods to shift (positive or negative).
- **freq:** Frequency for time-based shifting (e.g., 'D' for days).
- **axis:** 0 for rows, 1 for columns.
- **fill_value:** Value to fill new NA values.

```
• import pandas as pd
• import numpy as np
• date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
• data = {
    'Sales': np.random.randint(100, 500, size=(len(date_rng))),
    'Expenses': np.random.randint(50, 300, size=(len(date_rng)))
}
• df = pd.DataFrame(data, index=date_rng)
• print("Original DataFrame:")
• print(df)
```

- **Output:**

- Sales Expenses

• 2023-01-01	372	60
• 2023-01-02	222	75
• 2023-01-03	325	52
• 2023-01-04	269	276
• 2023-01-05	363	205
• 2023-01-06	457	154
• 2023-01-07	106	184
• 2023-01-08	193	256
• 2023-01-09	313	181
• 2023-01-10	292	282

shift()

- **# Shift Sales and Expenses by 1 day**
- `df['Sales_Shifted'] = df['Sales'].shift(1)`
- `df['Expenses_Shifted'] = df['Expenses'].shift(1)`
- `print("\n DataFrame with Sales and Expenses shifted by 1 day:")`
- `print(df)`

- **Output:**

- **DataFrame with Sales and Expenses shifted by 1 day:**

	Sales	Expenses	Sales_Shifted	Expenses_Shifted
• 2023-01-01	132	148	NaN	NaN
• 2023-01-02	255	194	132.0	148.0
• 2023-01-03	115	126	255.0	194.0
• 2023-01-04	229	210	115.0	126.0
• 2023-01-05	216	73	229.0	210.0
• 2023-01-06	341	208	216.0	73.0
• 2023-01-07	176	122	341.0	208.0
• 2023-01-08	316	106	176.0	122.0
• 2023-01-09	314	278	316.0	106.0
• 2023-01-10	209	83	314.0	278.0

shift()

- **# Calculate daily change in Sales and Expenses**

- `df['Sales_Change'] = df['Sales'] - df['Sales_Shifted']`
- `df['Expenses_Change'] = df['Expenses'] - df['Expenses_Shifted']`
- `print("\n Daily change in Sales and Expenses:")`
- `print(df[['Sales', 'Sales_Shifted', 'Sales_Change', 'Expenses', 'Expenses_Shifted', 'Expenses_Change']])`

- **# Shift by multiple periods (e.g., 2 days)**

- `df['Sales_Shifted_2'] = df['Sales'].shift(2)`
- `df['Expenses_Shifted_2'] = df['Expenses'].shift(2)`
- `print("\n DataFrame with Sales and Expenses shifted by 2 days:")`
- `print(df)`

- **# Using shift with a negative value to look ahead**

- `df['Sales_Shifted_Neg'] = df['Sales'].shift(-1)`
- `print("\n DataFrame with Sales shifted forward by 1 day:")`
- `print(df[['Sales', 'Sales_Shifted_Neg']])`

Methods of DataFrame: Time Series Operations

Method	Syntax	Description
asfreq(freq)	df.asfreq(freq, method=None, fill_value=None)	Converts time series to specified frequency.
to_period(freq)	df['column'].to_period(freq=None)	Converts to a period index.
to_timestamp()	df['column'].to_timestamp(freq=None, how='start')	Converts a PeriodIndex or Series to a DatetimeIndex or datetime Series
dt.year	df['column'].dt.year	Extracts the year from a datetime Series (accessed via `dt` accessor).
dt.month	df['column'].dt.month	Extracts the month component (1–12) from a datetime Series.
dt.day	df['column'].dt.day	Extracts the day of the month (1–31) from a datetime Series.
dt.hour	df['column'].dt.hour	Extracts the hour component (0–23) from a datetime Series.
dt.minute	df['column'].dt.minute	Extracts the minute component (0–59) from a datetime Series.
dt.isocalendar().week	df['column'].dt.week or df['column'].dt.isocalendar().week	Extracts the week number (1–53) of the year from a datetime Series.
dt.quarter	df['column'].dt.quarter	Extracts the quarter (1–4) of the year from a datetime Series.
dt.strftime	df['column'].dt.strftime(format)	Converts datetime Series to strings using a specified format.

asfreq()

- **Description:**

- Likely a typo; no such method exists in pandas. Possibly meant `asfreq()` for frequency conversion on time-series data.

- **Syntax:**

- `df.asfreq(freq, method=None, fill_value=None)`

- **Key Parameters:**

- **freq:** Frequency string (e.g., 'D' for daily, 'M' for monthly).
- **method:** Fill method for missing data ('ffill', 'bfill', or None).
- **fill_value:** Value to use for missing data if 'method' is None.

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')`
- `data = {`
 `'Sales': [200, 220, np.nan, 250, 300, np.nan, 400, 450, 500,`
 `550]`
 `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`

- **Output:**

	Sales
2023-01-01	200.0
2023-01-02	220.0
2023-01-03	NaN
2023-01-04	250.0
2023-01-05	300.0
2023-01-06	NaN
2023-01-07	400.0
2023-01-08	450.0
2023-01-09	500.0
2023-01-10	550.0

asfreq()

- **# Using asfreq to change frequency to business days (B)**
- `business_days = df.asfreq('B')`
- `print("\n DataFrame with frequency set to business days (B):")`
- `print(business_days)`

- **Output:**

DataFrame with frequency set to business days (B):

Sales

2023-01-02	220.0
2023-01-03	NaN
2023-01-04	250.0
2023-01-05	300.0
2023-01-06	NaN
2023-01-09	500.0
2023-01-10	550.0

Using asfreq to fill missing values with NaN

- `daily_data = df.asfreq('D')`
- `print("\n DataFrame with daily frequency (NaN for missing days):")`
- `print(daily_data)`

Using asfreq with method to forward fill missing values

- `forward_filled = df.asfreq('D', method='ffill')`
- `print("\n DataFrame with daily frequency (forward filled):")`
- `print(forward_filled)`

Using asfreq with method to backward fill missing values

- `backward_filled = df.asfreq('D', method='bfill')`
- `print("\n DataFrame with daily frequency (backward filled):")`
- `print(backward_filled)`

Using asfreq to downsample to weekly frequency

- `weekly_data = df.asfreq('W')`
- `print("\n DataFrame downsampled to weekly frequency:")`
- `print(weekly_data)`

to_period()

- **Description:**

- Converts a datetime Series or Index to a period (e.g., monthly, yearly).

- **Syntax:**

- `df['column'].to_period(freq=None)`

- **Key Parameters:**

- **freq:** Period frequency (e.g., 'M' for monthly, 'Y' for yearly).

```
• import pandas as pd
• import numpy as np
• date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
• data = {
    'Sales': np.random.randint(100, 500, size=(len(date_rng))),
    'Expenses': np.random.randint(50, 300, size=(len(date_rng)))
}
• df = pd.DataFrame(data, index=date_rng)
• print(df)
```

- **Output:**

	Sales	Expenses
• 2023-01-01	432	105
• 2023-01-02	270	267
• 2023-01-03	207	276
• 2023-01-04	438	260
• 2023-01-05	140	223
• 2023-01-06	369	56
• 2023-01-07	370	92
• 2023-01-08	373	265
• 2023-01-09	397	145
• 2023-01-10	494	226

- **# Convert the index to a monthly period**
- `df['Month'] = df.index.to_period('M')`
- `print("\nDataFrame with Month period:")`
- `print(df)`

- **Output:**

- **DataFrame with Month period:**

	Sales	Expenses	Month
2023-01-01	363	199	2023-01
2023-01-02	153	268	2023-01
2023-01-03	433	290	2023-01
2023-01-04	466	144	2023-01
2023-01-05	145	94	2023-01
2023-01-06	477	60	2023-01
2023-01-07	296	104	2023-01
2023-01-08	440	277	2023-01
2023-01-09	187	147	2023-01
2023-01-10	364	136	2023-01

- **# Group by the monthly period and sum the Sales and Expenses**
- `monthly_summary = df.groupby('Month').sum()`
- `print("\nMonthly summary (sum of Sales and Expenses):")`
- `print(monthly_summary)`

- **Output:**

Monthly summary (sum of Sales and Expenses):

	Sales	Expenses
Month		
2023-01	3324	1719

to_period()

- **# Convert the index to a quarterly period**
- `df['Quarter'] = df.index.to_period('Q')`
- `print("\n DataFrame with Quarter period:")`
- `print(df)`
- **# Group by the quarterly period and calculate the mean**
- `quarterly_summary = df.groupby('Quarter').mean()`
- `print("\n Quarterly summary (mean of Sales and Expenses):")`
- `print(quarterly_summary)`
- **# Convert the index to a yearly period**
- `df['Year'] = df.index.to_period('Y')`
- `print("\n DataFrame with Year period:")`
- `print(df)`
- **# Group by the yearly period and calculate the total**
- `yearly_summary = df.groupby('Year').mean()`
- `print("\n Yearly summary (sum of Sales and Expenses):")`
- `print(yearly_summary)`

to_timestamp()

- **Description:**

- Converts a PeriodIndex or Series to a DatetimeIndex or datetime Series.

- **Syntax:**

- `df['column'].to_timestamp(freq=None, how='start')`

- **Key Parameters:**

- **freq:** Frequency for timestamp (e.g., 'D' for daily).
- **how:** Placement within period ('start', 'end').

```
• import pandas as pd
• import numpy as np
• date_rng = pd.date_range(start='2023-01-01', end='2023-01-10',
    freq='D')
• data = {
    'Sales': np.random.randint(100, 500, size=(len(date_rng))),
    'Expenses': np.random.randint(50, 300, size=(len(date_rng)))
}
• df = pd.DataFrame(data, index=date_rng)
• print(df)
```

- **Output:**

	Sales	Expenses
• 2023-01-01	232	132
• 2023-01-02	135	276
• 2023-01-03	437	179
• 2023-01-04	456	251
• 2023-01-05	213	259
• 2023-01-06	187	188
• 2023-01-07	154	113
• 2023-01-08	402	137
• 2023-01-09	184	241
• 2023-01-10	207	285

- **# Convert the index to a monthly period**
- `df['Month'] = df.index.to_period('M')`
- `print("\n DataFrame with Month period:")`
- `print(df)`

- **# Convert the Month period back to timestamps**
- `df['Month_Timestamp'] = df['Month'].dt.to_timestamp()`
- `print("\n DataFrame with Month converted back to timestamps:")`
- `print(df)`

- **# Convert the index to a quarterly period**
- `df['Quarter'] = df.index.to_period('Q')`
- `print("\n DataFrame with Quarter period:")`
- `print(df)`

- **Output: “only for first case”**
 - **DataFrame with Month period:**
 -
- | | Sales | Expenses | Month |
|------------|-------|----------|---------|
| 2023-01-01 | 232 | 132 | 2023-01 |
| 2023-01-02 | 135 | 276 | 2023-01 |
| 2023-01-03 | 437 | 179 | 2023-01 |
| 2023-01-04 | 456 | 251 | 2023-01 |
| 2023-01-05 | 213 | 259 | 2023-01 |
| 2023-01-06 | 187 | 188 | 2023-01 |
| 2023-01-07 | 154 | 113 | 2023-01 |
| 2023-01-08 | 402 | 137 | 2023-01 |
| 2023-01-09 | 184 | 241 | 2023-01 |
| 2023-01-10 | 207 | 285 | 2023-01 |

- **# Convert the Quarter period back to timestamps**

- `df['Quarter_Timestamp'] = df['Quarter'].dt.to_timestamp()`
- `print("\nDataFrame with Quarter converted back to timestamps:")`
- `print(df)`

- **# Convert the index to a yearly period**

- `df['Year'] = df.index.to_period('Y')`
- `print("\nDataFrame with Year period:")`
- `print(df)`

- **# Convert the Year period back to timestamps**

- `df['Year_Timestamp'] = df['Year'].dt.to_timestamp()`
- `print("\nDataFrame with Year converted back to timestamps:")`
- `print(df)`

dt.year

- **Description:**

- Extracts the year from a datetime Series (accessed via `dt` accessor).

- **Syntax:**

- `df['column'].dt.year`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-01-5', freq='D')`
- `data = {`
- `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
- `'Expenses': np.random.randint(50, 300, size=(len(date_rng)))`
- `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`
- `# Extract the year from the index`
- `df['Year'] = df.index.year`
- `print("\nDataFrame with Year extracted:")`
- `print(df)`

- **Output:**

- **Original DataFrame:**

	Sales	Expenses
2023-01-01	303	286
2023-01-02	149	280
2023-01-03	447	191
2023-01-04	380	85
2023-01-05	146	283

- **DataFrame with Year extracted:**

	Sales	Expenses	Year
2023-01-01	303	286	2023
2023-01-02	149	280	2023
2023-01-03	447	191	2023
2023-01-04	380	85	2023
2023-01-05	146	283	2023

dt.year()

Group by Year and calculate the total Sales and Expenses

- `yearly_summary = df.groupby('Year').sum()`
- `print("\nYearly summary (sum of Sales and Expenses):")`
- `print(yearly_summary)`

Extracting year from a specific column if it were a datetime column

- `df['Date'] = df.index`
- `df['Year_From_Date'] = df['Date'].dt.year`
- `print("\nDataFrame with Year extracted from Date column:")`
- `print(df[['Date', 'Year_From_Date']])`

• Output:

• Yearly summary (sum of Sales and Expenses):

• Sales Expenses

• Year

• 2023 1425 1125

• DataFrame with Year extracted from Date column:

	Date	Year_From_Date
•	2023-01-01	2023
•	2023-01-02	2023
•	2023-01-03	2023
•	2023-01-04	2023
•	2023-01-05	2023

dt.month

- **Description:**

- Extracts the month component (1–12) from a datetime Series.

- **Syntax:**

- `df['column'].dt.month`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-01-05', freq='D')`
- `data = {`
 - `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
 - `'Expenses': np.random.randint(50, 300, size=(len(date_rng)))`
 - `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`

Extract the month from the index

- `df['Month'] = df.index.month`
- `print("\nDataFrame with Month extracted:")`
- `print(df)`

- **Output:**

- | | Sales | Expenses |
|------------|-------|----------|
| 2023-01-01 | 257 | 65 |
| 2023-01-02 | 255 | 67 |
| 2023-01-03 | 129 | 152 |
| 2023-01-04 | 379 | 277 |
| 2023-01-05 | 162 | 225 |

- **DataFrame with Month extracted:**

- | | Sales | Expenses | Month |
|------------|-------|----------|-------|
| 2023-01-01 | 257 | 65 | 1 |
| 2023-01-02 | 255 | 67 | 1 |
| 2023-01-03 | 129 | 152 | 1 |
| 2023-01-04 | 379 | 277 | 1 |
| 2023-01-05 | 162 | 225 | 1 |

dt.month

- **# Group by Month and calculate the total Sales and Expenses**
- `monthly_summary = df.groupby('Month').sum()`
- `print("\nMonthly summary (sum of Sales and Expenses):")`
- `print(monthly_summary)`

- **# Extracting month from a specific column if it were a datetime column**
- `df['Date'] = df.index`
- `df['Month_From_Date'] = df['Date'].dt.month`
- `print("\nDataFrame with Month extracted from Date column:")`
- `print(df[['Date', 'Month_From_Date']])`

- **Output:**
- **Monthly summary (sum of Sales and Expenses):**
- | | Sales | Expenses |
|-------|-------|----------|
| Month | | |
| 1 | 1617 | 707 |

- **DataFrame with Month extracted from Date column:**
- | | Date | Month_From_Date | |
|--|------------|-----------------|---|
| | 2023-01-01 | 2023-01-01 | 1 |
| | 2023-01-02 | 2023-01-02 | 1 |
| | 2023-01-03 | 2023-01-03 | 1 |
| | 2023-01-04 | 2023-01-04 | 1 |
| | 2023-01-05 | 2023-01-05 | 1 |

dt.day

- **Description:**

- Extracts the day of the month (1–31) from a datetime Series.

- **Syntax:**

- `df['column'].dt.day`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-01-05', freq='D')`
- `data = {`
- `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
- `'Expenses': np.random.randint(50, 300, size=(len(date_rng)))`
- `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`
- **# Extract the day from the index**
- `df['Day'] = df.index.day`
- `print("\nDataFrame with Day extracted:")`
- `print(df)`

- **Output:**

- | | Sales | Expenses |
|------------|-------|----------|
| 2023-01-01 | 484 | 101 |
| 2023-01-02 | 281 | 165 |
| 2023-01-03 | 445 | 159 |
| 2023-01-04 | 269 | 68 |
| 2023-01-05 | 312 | 239 |
- **DataFrame with Day extracted:**
- | | Sales | Expenses | Day |
|------------|-------|----------|-----|
| 2023-01-01 | 484 | 101 | 1 |
| 2023-01-02 | 281 | 165 | 2 |
| 2023-01-03 | 445 | 159 | 3 |
| 2023-01-04 | 269 | 68 | 4 |
| 2023-01-05 | 312 | 239 | 5 |

dt.day

- import pandas as pd
- import numpy as np
- date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
- data = {
- 'Sales': np.random.randint(100, 500, size=(len(date_rng))),
- 'Expenses': np.random.randint(50, 300, size=(len(date_rng)))
- }
- df = pd.DataFrame(data, index=date_rng)
- print(df)
- **# Extract the day from the index**
- df['Day'] = df.index.day
- print("\nDataFrame with Day extracted:")
- print(df)

- **Output:**
- **Daily summary (sum of Sales and Expenses):**
- Sales Expenses
- **Day**
- 1 484 101
- 2 281 165
- 3 445 159
- 4 269 68
- 5 312 239
- **DataFrame with Day extracted from Date column:**
- Date Day_From_Date
- 2023-01-01 2023-01-01 1
- 2023-01-02 2023-01-02 2
- 2023-01-03 2023-01-03 3
- 2023-01-04 2023-01-04 4
- 2023-01-05 2023-01-05 5

dt.hour

- **Description:**

- Extracts the hour component (0–23) from a datetime Series.

- **Syntax:**

- `df['column'].dt.hour`

```
• import pandas as pd
• import numpy as np
• date_rng = pd.date_range(start='2023-01-01 00:00', end='2023-01-01 05:00', freq='H')
• data = {
•     'Sales': np.random.randint(100, 500, size=(len(date_rng))),
•     'Expenses': np.random.randint(50, 300, size=(len(date_rng)))
• }
• df = pd.DataFrame(data, index=date_rng)
• print(df)
• # Extract the hour from the index
• df['Hour'] = df.index.hour
• print("\nDataFrame with Hour extracted:")
• print(df)
```

- **Output:**

	Sales	Expenses
2023-01-01 00:00:00	161	54
2023-01-01 01:00:00	122	195
2023-01-01 02:00:00	418	194
2023-01-01 03:00:00	154	157
2023-01-01 04:00:00	407	176
2023-01-01 05:00:00	341	183

- **DataFrame with Hour extracted:**

	Sales	Expenses	Hour
2023-01-01 00:00:00	161	54	0
2023-01-01 01:00:00	122	195	1
2023-01-01 02:00:00	418	194	2
2023-01-01 03:00:00	154	157	3
2023-01-01 04:00:00	407	176	4
2023-01-01 05:00:00	341	183	5

dt.hour

- **# Group by Hour and calculate the total Sales and Expenses**
- `hourly_summary = df.groupby('Hour').sum()`
- `print("\nHourly summary (sum of Sales and Expenses):")`
- `print(hourly_summary)`
- **# Extracting hour from a specific column if it were a datetime column**
- `df['Date'] = df.index`
- `df['Hour_From_Date'] = df['Date'].dt.hour`
- `print("\nDataFrame with Hour extracted from Date column:")`
- `print(df[['Date', 'Hour_From_Date']])`

- **Output:**
- **Hourly summary (sum of Sales and Expenses):**
- Sales Expenses
- Hour
- 0 161 54
- 1 122 195
- 2 418 194
- 3 154 157
- 4 407 176
- 5 341 183
- **DataFrame with Hour extracted from Date column:**
- Date Hour_From_Date
- 2023-01-01 00:00:00 2023-01-01 00:00:00 0
- 2023-01-01 01:00:00 2023-01-01 01:00:00 1
- 2023-01-01 02:00:00 2023-01-01 02:00:00 2
- 2023-01-01 03:00:00 2023-01-01 03:00:00 3
- 2023-01-01 04:00:00 2023-01-01 04:00:00 4
- 2023-01-01 05:00:00 2023-01-01 05:00:00 5

dt.minute()

- **Description:**

- Extracts the minute component (0–59) from a datetime Series.

- **Syntax:**

- `df['column'].dt.minute`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01 01:10', end='2023-01-01 01:15', freq='min')`
- `data = {`
 `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
 `'Expenses': np.random.randint(50, 300, size=(len(date_rng))) }`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`
- **# Extract the minute from the index**
- `df['Minute'] = df.index.minute`
- `print("\nDataFrame with Minute extracted:")`
- `print(df)`

- **Output:**

	Sales	Expenses
2023-01-01 01:10:00	153	216
2023-01-01 01:11:00	424	239
2023-01-01 01:12:00	375	122
2023-01-01 01:13:00	389	231
2023-01-01 01:14:00	487	171
2023-01-01 01:15:00	245	96

- **DataFrame with Minute extracted:**

	Sales	Expenses	Minute
2023-01-01 01:10:00	153	216	10
2023-01-01 01:11:00	424	239	11
2023-01-01 01:12:00	375	122	12
2023-01-01 01:13:00	389	231	13
2023-01-01 01:14:00	487	171	14
2023-01-01 01:15:00	245	96	15

- **# Group by Minute and calculate the total Sales and Expenses**

- `minute_summary = df.groupby('Minute').sum()`
- `print("\nMinute summary (sum of Sales and Expenses):")`
- `print(minute_summary)`

- **# Extracting minute from a specific column if it were a datetime column**

- `df['Date'] = df.index`
- `df['Minute_From_Date'] = df['Date'].dt.minute`
- `print("\nDataFrame with Minute extracted from Date column:")`
- `print(df[['Date', 'Minute_From_Date']])`

- **Output:**

- **Minute summary (sum of Sales and Expenses):**

- Sales Expenses

- **Minute**

- 10 153 216
- 11 424 239
- 12 375 122
- 13 389 231
- 14 487 171
- 15 245 96

- **DataFrame with Minute extracted from Date column:**

- Date Minute_From_Date
- 2023-01-01 01:10:00 2023-01-01 01:10:00 10
- 2023-01-01 01:11:00 2023-01-01 01:11:00 11
- 2023-01-01 01:12:00 2023-01-01 01:12:00 12
- 2023-01-01 01:13:00 2023-01-01 01:13:00 13
- 2023-01-01 01:14:00 2023-01-01 01:14:00 14
- 2023-01-01 01:15:00 2023-01-01 01:15:00 15

dt.week()

- **Description:**

- Extracts the week number (1–53) of the year from a datetime Series. Note: As of pandas 2.0, `dt.week` is deprecated; use `dt.isocalendar().week` instead.

- **Syntax:**

- `df['column'].dt.isocalendar().week`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2022-12-31', end='2023-01-05', freq='D')`
- `data = {`
 - `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
 - `'Expenses': np.random.randint(50, 300, size=(len(date_rng)))`
 - `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print("Original DataFrame:")`
- `print(df)`
- **# Extract the ISO week number from the index**
- `df['Week'] = df.index.isocalendar().week`
- `print("\nDataFrame with Week number extracted:")`
- `print(df)`

- **Output:**

- | | Sales | Expenses |
|------------|-------|----------|
| 2022-12-31 | 199 | 86 |
| 2023-01-01 | 183 | 199 |
| 2023-01-02 | 316 | 246 |
| 2023-01-03 | 451 | 276 |
| 2023-01-04 | 223 | 95 |
| 2023-01-05 | 207 | 147 |

- **DataFrame with Week number extracted:**

- | | Sales | Expenses | Week |
|------------|-------|----------|------|
| 2022-12-31 | 199 | 86 | 52 |
| 2023-01-01 | 183 | 199 | 52 |
| 2023-01-02 | 316 | 246 | 1 |
| 2023-01-03 | 451 | 276 | 1 |
| 2023-01-04 | 223 | 95 | 1 |
| 2023-01-05 | 207 | 147 | 1 |

- **# Group by Week and calculate the total Sales and Expenses**

- `weekly_summary = df.groupby('Week').sum()`
- `print("\nWeekly summary (sum of Sales and Expenses):")`
- `print(weekly_summary)`

- **# Extracting week from a specific column if it were a datetime column**

- `df['Date'] = df.index`
- `df['Week_From_Date'] = df['Date'].dt.isocalendar().week`
- `print("\nDataFrame with Week number extracted from Date column:")`
- `print(df[['Date', 'Week_From_Date']])`

- **Output:**

- **Weekly summary (sum of Sales and Expenses):**

- Sales Expenses
- Week
- 1 1197 764
- 52 382 285

- **DataFrame with Week number extracted from Date column:**

- Date Week_From_Date
- 2022-12-31 2022-12-31 52
- 2023-01-01 2023-01-01 52
- 2023-01-02 2023-01-02 1
- 2023-01-03 2023-01-03 1
- 2023-01-04 2023-01-04 1
- 2023-01-05 2023-01-05 1

dt.quarter()

- **Description:**

- Extracts the quarter (1–4) of the year from a datetime Series.

- **Syntax:**

- `df['column'].dt.quarter`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='ME')`
- `data = {`
 - `'Sales': np.random.randint(1000, 5000, size=(len(date_rng))),`
 - `'Expenses': np.random.randint(500, 3000, size=(len(date_rng)))``}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`

- **Output:**

	Sales	Expenses
• 2023-01-31	4787	783
• 2023-02-28	2962	634
• 2023-03-31	2208	1026
• 2023-04-30	4175	2695
• 2023-05-31	3139	2181
• 2023-06-30	4143	670
• 2023-07-31	3344	2071
• 2023-08-31	4722	2203
• 2023-09-30	4878	1307
• 2023-10-31	1629	964
• 2023-11-30	4878	1513
• 2023-12-31	2505	1511

dt.quarter()

- **# Extract the quarter from the index**
- `df['Quarter'] = df.index.quarter`
- `print("\nDataFrame with Quarter extracted:")`
- `print(df)`
- **#Group by Quarter and calculate the total Sales and Expenses**
- `quarterly_summary = df.groupby('Quarter').sum()`
- `print("\nQuarterly summary (sum of Sales and Expenses):")`
- `print(quarterly_summary)`
- **# Extracting quarter from a specific column if it were a datetime column**
- `df['Date'] = df.index`
- `df['Quarter_From_Date'] = df['Date'].dt.quarter`
- `print("\nDataFrame with Quarter extracted from Date column:")`
- `print(df[['Date', 'Quarter_From_Date']])`

- **Output:**
- **DataFrame with Quarter extracted:**
- | | Sales | Expenses | Quarter |
|------------|-------|----------|---------|
| 2023-01-31 | 4787 | 783 | 1 |
| 2023-02-28 | 2962 | 634 | 1 |
| 2023-03-31 | 2208 | 1026 | 1 |
| 2023-04-30 | 4175 | 2695 | 2 |
| 2023-05-31 | 3139 | 2181 | 2 |
| 2023-06-30 | 4143 | 670 | 2 |
| 2023-07-31 | 3344 | 2071 | 3 |
| 2023-08-31 | 4722 | 2203 | 3 |
| 2023-09-30 | 4878 | 1307 | 3 |
| 2023-10-31 | 1629 | 964 | 4 |
| 2023-11-30 | 4878 | 1513 | 4 |
| 2023-12-31 | 2505 | 1511 | 4 |
- **“For other output run code”**

dt.strftime()

- **Description:**

- Converts datetime Series to strings using a specified format.

- **Syntax:**

- `df['column'].dt.strftime(format)`

- `import pandas as pd`
- `import numpy as np`
- `date_rng = pd.date_range(start='2023-01-01', end='2023-01-05', freq='D')`
- `data = {`
 - `'Sales': np.random.randint(100, 500, size=(len(date_rng))),`
 - `'Expenses': np.random.randint(50, 300, size=(len(date_rng)))`
 - `}`
- `df = pd.DataFrame(data, index=date_rng)`
- `print(df)`

- **Output:**

	Sales	Expenses
• 2023-01-01	164	128
• 2023-01-02	174	63
• 2023-01-03	263	223
• 2023-01-04	172	126
• 2023-01-05	345	282

dt.strftime()

- **# Format the index as a string in the desired format**
- `df['Formatted_Date'] = df.index.strftime('%Y-%m-%d')`
- `print("\nDataFrame with Formatted Date:")`
- `print(df)`
- **# Format a specific column if it were a datetime column**
- `df['Date'] = df.index`
- `df['Formatted_Date_From_Column'] = df['Date'].dt.strftime('%d-%m-%Y')`
- `print("\nDataFrame with Formatted Date from Date column:")`
- `print(df[['Date', 'Formatted_Column']])`

- **Output:**
- **DataFrame with Formatted Date:**
- | | Sales | Expenses | Formatted_Date |
|------------|-------|----------|----------------|
| 2023-01-01 | 164 | 128 | 2023-01-01 |
| 2023-01-02 | 174 | 63 | 2023-01-02 |
| 2023-01-03 | 263 | 223 | 2023-01-03 |
| 2023-01-04 | 172 | 126 | 2023-01-04 |
| 2023-01-05 | 345 | 282 | 2023-01-05 |
- **DataFrame with Formatted Date from Date column:**
- | | Date | Formatted_Column |
|------------|------------|------------------|
| 2023-01-01 | 2023-01-01 | 01-01-2023 |
| 2023-01-02 | 2023-01-02 | 02-01-2023 |
| 2023-01-03 | 2023-01-03 | 03-01-2023 |
| 2023-01-04 | 2023-01-04 | 04-01-2023 |
| 2023-01-05 | 2023-01-05 | 05-01-2023 |

Methods of DataFrame: . Other Methods

Method	Syntax	Description
pipe(func)	df.pipe(func, *args, **kwargs)	Applies a function to the DataFrame for chaining.
eval(expr)	df.eval(expr, inplace=False)	Evaluates a string expression on the DataFrame.
where(cond, other)	df.where(cond, other=np.nan, inplace=False, axis=None)	Replaces values where condition is False.
clip(lower, upper)	df.clip(lower=None, upper=None, axis=None, inplace=False)	Trims values at specified thresholds.
nunique(axis)	df.nunique(axis=0, dropna=True)	Counts unique values.

pipe()

- **Description:**

- Applies a function to the DataFrame, allowing method chaining.

- **Syntax:**

- `df.pipe(func, *args, **kwargs)`

- **Key Parameters:**

- **func:** Function to apply; takes DataFrame as first argument.
- **args, kwargs:** Additional arguments for the function.

```
• 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
```

pipe()

- **# Define a function to calculate a salary increase**
- `def increase_salary(df, increase_percentage):`
- `df['Salary'] = df['Salary'] * (1 + increase_percentage / 100)`
- `return df`
- **# Define a function to filter employees by age**
- `def filter_by_age(df, min_age):`
- `return df[df['Age'] >= min_age]`
- **# Use of df.pipe()**
- `result = (# Increase salary by 10%`
 `df.pipe(increase_salary, increase_percentage=10)`
 `.pipe(filter_by_age, min_age=30)`
 `# Filter employees aged 30 and above`
`)`
- `print("Resulting DataFrame after salary increase and filtering:")`
- `print(result)`

- **Output:**
- **Resulting DataFrame after salary increase and filtering:**
- | | Name | Age | Salary |
|---|---------|-----|---------|
| 1 | Bob | 30 | 66000.0 |
| 2 | Charlie | 35 | 82500.0 |

eval()

- **Description:**
 - Evaluates a string expression on DataFrame columns.
- **Syntax:**
 - `df.eval(expr, inplace=False)`
- **Key Parameters:**
 - **expr:** String expression (e.g., 'col1 + col2').
 - **inplace:** Modify DataFrame in place if True.

```
• import pandas as pd
• data = {
    'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
    'Salary': [50000, 60000, 75000] }
• df = pd.DataFrame(data)
• print(df)
• # Use df.eval() to create a new column for total compensation
• df['Bonus'] = [5000, 7000, 8000] # Adding a Bonus column
• # Using eval to calculate Total Compensation
• df['Total_Compensation'] = df.eval('Salary + Bonus')
• print("DataFrame with Total Compensation:")
• print(df)
```

```
• Output:
•   Name Age Salary
• 0  Alice  25  50000
• 1   Bob   30  60000
• 2 Charlie  35  75000
• DataFrame with Total Compensation:
•   Name Age Salary Bonus Total_Compensation
• 0  Alice  25  50000  5000          55000
• 1   Bob   30  60000  7000          67000
• 2 Charlie  35  75000  8000          83000
```

where()

- **Description:**
 - Replaces values where a condition is False with a specified value or another DataFrame/Series.
- **Syntax:**
 - `df.where(cond, other=np.nan, inplace=False, axis=None)`
- **Key Parameters:**
 - **cond:** Boolean condition (True keeps original value).
 - **other:** Value to replace where `cond` is False.
 - **inplace:** Modify DataFrame in place if True.

```
• import pandas as pd
• 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

where()

- **# Use df.where() to filter the DataFrame based on a condition**
- `filtered_df = df.where(df['Salary'] > 55000)`
- `print(df)`
- `print("\nFiltered DataFrame (where Salary > 55000):")`
- `print(filtered_df)`
- **# Replace NaN with a specific value (e.g., 0)**
- `df_with_zero = df.where(df['Salary'] > 55000, other=0)`
- `print("\nFiltered DataFrame with NaN replaced by 0:")`
- `print(df_with_zero)`

- **Output:**
- **Filtered DataFrame (where Salary > 55000):**
- | | Name | Age | Salary |
|---|---------|------|---------|
| 0 | NaN | NaN | NaN |
| 1 | Bob | 30.0 | 60000.0 |
| 2 | Charlie | 35.0 | 75000.0 |
- **Filtered DataFrame with NaN replaced by 0:**
- | | Name | Age | Salary |
|---|---------|-----|--------|
| 0 | 0 | 0 | 0 |
| 1 | Bob | 30 | 60000 |
| 2 | Charlie | 35 | 75000 |

clip()

- **Description:**

- Trims values at specified lower and upper bounds.

- **Syntax:**

- `df.clip(lower=None, upper=None, axis=None, inplace=False)`

- **Key Parameters:**

- **lower:** Minimum threshold; values below are set to this.

- `data = {
 'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
 'Salary': [50000, 60000, 75000], 'Bonus': [2000, 3000, 4000] }`
- `df = pd.DataFrame(data)`
- `print(df)`
- **# Use df.clip() to limit the Salary to a range**
- `clipped_df = df.copy()`
- `clipped_df['Salary'] = clipped_df['Salary'].clip(lower=55000, upper=65000)`
- `print("\nDataFrame after clipping Salary between 55000 and 65000:")`
- `print(clipped_df)`

- **Output:**

- | | Name | Age | Salary | Bonus |
|---|---------|-----|--------|-------|
| 0 | Alice | 25 | 50000 | 2000 |
| 1 | Bob | 30 | 60000 | 3000 |
| 2 | Charlie | 35 | 75000 | 4000 |
- **DataFrame after clipping Salary between 55000 and 65000:**
- | | Name | Age | Salary | Bonus |
|---|---------|-----|--------|-------|
| 0 | Alice | 25 | 55000 | 2000 |
| 1 | Bob | 30 | 60000 | 3000 |
| 2 | Charlie | 35 | 65000 | 4000 |

clip()

- **# Clip both Salary and Bonus columns**
- `clipped_multi_df = df.copy()`
- `print(df)`
- `clipped_multi_df[['Salary', 'Bonus']] = clipped_multi_df[['Salary', 'Bonus']].clip(lower=55000, upper=65000)`
- `print("\nDataFrame after clipping Salary and Bonus between 55000 and 65000:")`
- `print(clipped_multi_df)`

- **# Clip both Salary and Bonus columns**
- `clipped_df = df.copy()`
- `clipped_df[['Salary', 'Bonus']] = clipped_df[['Salary', 'Bonus']].clip(lower=[55000, 2000], upper=[65000, 4000])`
- `print("\nDataFrame after clipping Salary and Bonus:")`
- `print(clipped_df)`

Output:

- | | Name | Age | Salary | Bonus |
|---|---------|-----|--------|-------|
| 0 | Alice | 25 | 50000 | 2000 |
| 1 | Bob | 30 | 60000 | 3000 |
| 2 | Charlie | 35 | 75000 | 4000 |
- **DataFrame after clipping Salary and Bonus between 55000 and 65000:**
- | | Name | Age | Salary | Bonus |
|---|---------|-----|--------|-------|
| 0 | Alice | 25 | 55000 | 55000 |
| 1 | Bob | 30 | 60000 | 55000 |
| 2 | Charlie | 35 | 65000 | 55000 |
- **DataFrame after clipping Salary and Bonus:**
- | | Name | Age | Salary | Bonus |
|---|---------|-----|--------|-------|
| 0 | Alice | 25 | 55000 | 2000 |
| 1 | Bob | 30 | 60000 | 3000 |
| 2 | Charlie | 35 | 65000 | 4000 |

nunique()

- **Description:**

- Counts unique values along an axis.

- **Syntax:**

- `df.nunique(axis=0, dropna=True)`

- **Key Parameters:**

- **axis:** 0 for rows, 1 for column
- **dropna:** Exclude NA/null values (default True).

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

- **Output:**

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

nunique()

- **# Use df.nunique() to count unique values in each column**

- `unique_counts = df.nunique()`
- `print("\nNumber of unique values in each column:")`
- `print(unique_counts)`

- **# Count unique values while excluding NaN values**

- `unique_excluding_nan = df.nunique(dropna=True)`
- `print("\nNumber of unique values in each column (excluding NaN):")`
- `print(unique_excluding_nan)`

- **Output:**

- **Number of unique values in each column:**

- Name 3
- Age 3
- Salary 3
- dtype: int64

- **Number of unique values in each column (excluding NaN):**

- Name 3
- Age 3
- Salary 3
- dtype: int64