5CS037 - Concepts and Technologies of AI.
Week - 02 - Workshop - 02

Introduction to Exploratory Data Analysis.

Part - 1 - Pandas!!!

Siman Giri

Workshop - 02

November - 10 - 2024

Outline

1. Introduction to Pandas.

1.1 What is pandas?

- Pandas is an open-source add-on modules to python which provides high-performance, easy-to-use data structure, and data analysis tools.

> [pandas] is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.
> -Wikipedia

- The pandas library contains several methods and functions for cleaning, manipulating and analyzing data.
- Though Pandas is built on top of the Numpy package, Numpy is suited for working with homogeneous numerical array data, Pandas is designed for working with tabular or heterogeneous data.

1.2 Why Pandas?

Some of the key features of Pandas are:

1. Supports a variety of data formats, along with their integration and transformation.

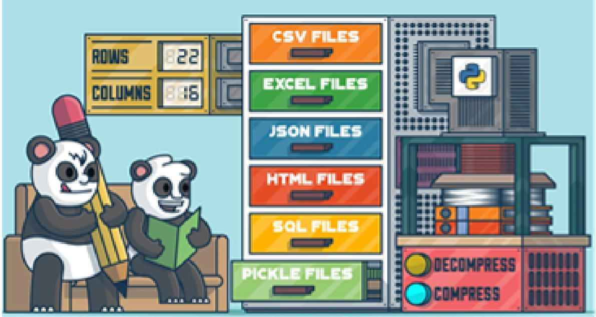

Image by – M.Stojiljkovic – Real Python.

Figure: Various data type supported by Pandas.

1.2.1 Why Pandas? (Continued)

2. Support for Time - Series Data.
   - A sequence of data points indexed in time order (e.g., daily stock prices, temperature readings) are called time series data or simply **temporal data**.
     - Various In-built functions and methods for time based indexing and aggregation simplifies handling, analysis and visualization of temporal data. For example:
     - `pd.to_datetime()`: Converts strings or other formats to date time objects.
     - `pd.date_range()`: Generates sequences of

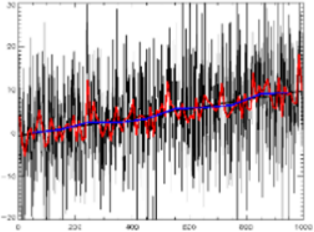

Figure: Plot - Time series Data with Pandas.

1.2.2 Why Pandas? (Continued)

3. Descriptive Statistics with Pandas - Numerical:
   - Pandas simplifies descriptive data analysis with built-in methods and functions, making it easy to calculate statistics like mean, median, and standard deviation. For example:
     - `pd.mean(); pd.median() etc.`

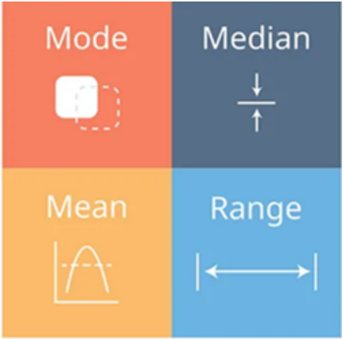

Figure: Describing Data - Numerically.

1.2.2 Why Pandas? (Continued)

3. Descriptive Statistics with Pandas - Graphical:
   - Pandas includes built-in methods for data visualization, making it easy to create plots such as histograms, line charts, and scatter plots directly from DataFrames and Series. For example,
     - you can use `df.hist()` to plot histograms or `df.plot()` for line and bar charts.

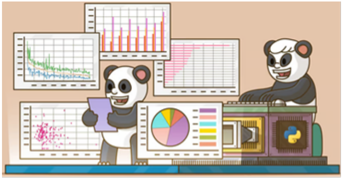

Image by – M.Stojiljkovic – Real Python.

Figure: Describing Data - Graphically.

1.3 How are we going to use Pandas in this Module!

For this module we will use pandas to:

1. Create and Load Data:
    - While we can use Pandas to create our own data, we will primarily use it to load data from various sources, especially in `.csv` format.
    - Additionally, we will also use Pandas to save our outputs in `.csv` files.
2. Data Exploration and Data Wrangling:
   We will use various built-in functions to clean and prepare the data. These may include, but are not limited to, the following:
    - Look for any anomalies, including missing data, inconsistencies, or any other data that seems out of place.
    - perform frequent data operations such as subsetting, filtering, insertion, deletion, and aggregation to prepare the data for analysis.

1.3.1 How are we going to use Pandas in this Module!

For this module we will use pandas to:

3. Data Analysis and Presentation: We will use various tools built into Pandas to process and analyze data, presenting it in graphical or tabular formats. This will help us identify trends, patterns, and correlations, ultimately answering our initial questions.

Image by – M.Stojiljkovic – Real Python.

Figure: Data Exploration with Pandas.

Getting Started with Pandas.
2.Data Structures of Pandas.

2.1 Building blocks of Pandas.

- There are three core components of Pandas library which are Series,DataFrame Data Structure and index.

index

index
0
1
2
3
4
5
6
7
8
9
10

series

name
William
Emma
Sofia
Markus
Edward
Thomas
Ethan
Olivia
Arun
Anika
Paulo

Dataframe

index	name	region	sales	expenses
0	William	East	50000	42000
1	Emma	North	52000	43000
2	Sofia	East	90000	50000
3	Markus	South	34000	44000
4	Edward	West	42000	38000
5	Thomas	West	72000	39000
6	Ethan	South	49000	42000
7	Olivia	West	55000	60000
8	Arun	West	67000	39000
9	Anika	East	65000	44000
10	Paulo	South	67000	45000

Figure: Building Blocks of Pandas.

- In this section briefly describe the functionality of aforementioned each building blocks.
- Start with regular `import statement:`

```
import pandas as pd
```

2.2 Data Structure - Series.

- Series is a one - dimensional labeled array capable of holding any data type {integers, floats, strings, Python Objects, etc.}
- It is a fundamental data structure in pandas and is similar to a list or a column in spreadsheet,but with following additional attributes:
- Key attributes of a series:
  1. Index: Each element in a Series is associated with a unique label, called an index.
  2. Data: The actual data stored in the Series, which can be of any data type.
  3. Homogeneity: By default, all elements in a Series are of the same data type, though it can hold mixed types in some cases.

Figure: Components of a Series.

INDEX	DATA
0	[1, 2]
1	A
2	1
3	(4, 5)
4	{"a": 1}
5	6

2.2.1 Series - Code Example.

- Example Creating a simple series:

```
import pandas as pd
# Creating a simple Series
data = [10, 20, 30, 40]
series = pd.Series(data)
print(series)
```

- Here are some commonly used attributes with `Series` objects:

Attribute	Description	Syntax (showcasing usage)
name	The name of the Series object	series.name = 'My Series'
dtype	The data type of the Series object	series.dtype
shape	Dimensions of the Series object in a tuple of the form (number of rows,)	series.shape
index	The Index object that is part of the Series object	series.index
values	The data in the Series object	series.values
size	The number of elements in the Series object	series.size

Figure: Common attributes of data structure - Series.

2.3 Data Structure - Index.

- An index in Pandas is an integral part of both Series and DataFrame objects.
- An index in Pandas serves as a label for the data, acting as a row identifier or unique tag that enables easy access, manipulation, and efficient alignment of data.
- Types of Index:
  1. Default Index: A numeric range starting from 0:

```python
import pandas as pd
series = pd.Series([10, 20, 30])
print(series.index)
# Output: RangeIndex(start=0, stop=3, step=1)
```

2.3.1 Data Structure - Index.

- Types of Index:
  2. Custom Index: User defined labels.

```python
series = pd.Series([10, 20, 30],index=['a','b','c'])
print(series)
# Output:
# a    10
# b    20
# c    30
```

2.3.2 Data Structure - Index.

- Types of Index:
  3. Datetime Index: Special index for time-series data.

```python
dates = pd.date_range('2023-01-01', periods=3)
series = pd.Series([10, 20, 30], index=dates)
print(series)
# Output:
# 2023-01-01    10
# 2023-01-02    20
# 2023-01-03    30
```

2.3.3 Data Structure - Index.

- Functions Related to Index:
  1. Inspection:
     - `series.index` - Access the index
     - `df.index` - Access the index in Dataframe
  2. Modification:
     - `set_index()` - Assign a new index.
     - `reset_index()` - Reset the index to default.
  3. Alignment: Pandas automatically aligns data based on the index.

Code Example: Access and Reset Index

```
# Access:
print(series.index)
# Set or Reset Index
series.index = ['x', 'y', 'z']  # Series
# For DataFrame
df = pd.DataFrame({'A': [1, 2]}, index=['row1', 'row2'])
df.reset_index(inplace=True)
# Converts the index into a column
```

2.4 Data Structure - DataFrame.

- A DataFrame is a two-dimensional object
  - comprising of tabular data organized in rows and columns
  - individual columns can be of different value types (numeric / string / Boolean etc.)
  - row indices: refers to individual rows (called index, usually integers if not defined otherwise).
  - column indices: refers to name(head) of each columns, if not defined otherwise.
- Each column in a DataFrame is a Series.

Series 1

INDEX	DATA
0	A
1	B
2	C
3	D
4	E
5	F

&

Series 2

INDEX	DATA
0	1
1	2
2	3
3	4
4	5
5	6

&

Series 3

INDEX	DATA
0	[1, 2]
1	A
2	1
3	(4, 5)
4	{"a": 1}
5	6

=

Dataframe

INDEX	SERIES 1	SERIES 2	SERIES 3
0	A	1	[1, 2]
1	B	2	A
2	C	3	1
3	D	4	(4, 5)
4	E	5	{"a": 1}
5	F	6	6

Figure: Components of Dataframe.

Getting Started with Pandas.
3.Building and Modifying DataFrames.

3.1 Creating DataFrames.

- A Pandas DataFrame can be created by converting the in-built python data structures such as lists, dictionaries etc. Example:

```python
#Transforming in-built data structures-DataFrame
#Style-1
import pandas as pd
pd.DataFrame({'Bob': ['I liked it.','It was awful'], 'Sue': ['Pretty good.', 'Bland.']})
#Style-2
pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty good.', 'Bland.']},
             index=['Product A', 'Product B'])
```

- Output for both the styles: Observe the Difference

	Bob	Sue
0	I liked it.	Pretty good.
1	It was awful.	Bland.

Output Style - 1

	Bob	Sue
Product A	I liked it.	Pretty good.
Product B	It was awful.	Bland.

Output Style - 2

Figure: Creating DataFrames with In-built Python Data-structures.

3.2 Loading Data to DataFrames.

- In the real world, a pandas DataFrame will typically be created by loading the datasets from CSV file, Excel file, etc.
  - Pandas provides the `read_csv()` function to read data stored as a csv file into a pandas DataFrame.
  - Pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, ...), each of them with the prefix `read_*`.
- The head/tail/info methods and the dtypes attribute are convenient for a first check.

```
#Importing Data from file
import pandas as pd
# path to your dataset must be given to built in
    read_csv("Your path") function.
dataset = pd.read_csv("/data/Week02/bank.csv")
dataset.head()
dataset.tail()
dataset.info()
# Run the above code and observe the output.
```

3.3 Writing DataFrames to CSV.

- Whereas `read_*` functions are used to read data to pandas, the `to_*` methods are used to store data.
- The `to_csv("path+file name", index=false)` method stores the data as an csv file.
  - `path`: Where you wan to store the created file.
  - `file name`: in the name you want to store the file.
  - `index:boolean`: store the index or not.
- Pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, ...), each of them with the prefix `to_*`.

```
#Importing Data from file
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],'City': ['
    New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data) # creating a DataFrame
#Writing DataFrame to csv.
df.to_csv('output.csv', index=False)
# Run the above code and observe the output.
```

Getting Started with Data Analysis.
Basic Operation on Data.
4.Data Inspection and Exploration.

4.1 First Data Inspection and Exploration.

- {Almost always} Once we load our data into dataframe, following are the basic inspection we perform:
  1. Viewing Data:

methods	Result
df.head()	Displays the first few rows of the DataFrame.
df.tail()	Displays the last few rows.
df.info()	Provides a summary of the DataFrame, including non-null counts and data types.
df.describe()	Generates summary statistics for numerical columns.

Figure: For First Inspection of Data.

  2. Checking Dimensions:
     - `df.shape` - Returns the dimensions of the DataFrame as a tuple(rows, columns).
  3. Selecting a Column:
     - use `df['column_name'] to acess a column as a series.`
  4. Selecting Rows:
     - `iloc[]` - Selects rows by numerical index.
     - `loc[]` - Selects rows by labels {index or condtions}.

4.1.1 First Data Inspection and Exploration - Sample Code.

```python
import pandas as pd
# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)
# View the first two rows
print(df.head(2))
# View the last row
print(df.tail(1))
# DataFrame information
print(df.info())
# Summary statistics
print(df.describe())
# Check dimensions of the DataFrame
print(f"The DataFrame has {df.shape[0]} rows and {df.shape
    [1]} columns.")
```

4.1.2 First Data Inspection and Exploration - Sample Code.

```python
import pandas as pd
# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)
# Access the 'Age' column
print(df['Age'])
# Select rows by numerical index
print(df.iloc[0])  # First row
# Select rows by condition
print(df.loc[df['Age'] > 30])  # Rows where Age > 30
```

4.2 Filtering and Modifying DataFrame.

1. Filtering Rows and Columns: Filtering helps you select rows or columns based on conditions or labels.
   - Filter Rows By Condition:

```python
import pandas as pd
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
})
# Filter rows where Age > 28
filtered_rows = df[df['Age'] > 28]
print(filtered_rows)
```

   - Select Specific Columns:

```python
# Select only 'Name' and 'Salary' columns
selected_columns = df[['Name', 'Salary']]
print(selected_columns)
```

4.2.1 Filtering and Modifying DataFrame.

2. Dropping Rows or Columns: The `drop()` method is used to remove rows or columns.
   - Drop a columns

```
# Drop the 'Salary' column
df_without_salary = df.drop(columns=['Salary'])
print(df_without_salary)
```

   - Drop a Row

```
# Drop the row with index 1 (Bob)
df_without_row = df.drop(index=1)
print(df_without_row)
```

3. Add a new column: You can add new columns directly in dataframe:

```
# Add a new column for Bonus
df['Bonus'] = df['Salary'] * 0.1
print(df)
```

Getting Started with Data Analysis.
Data Wrangling.
5.Some Common Data Cleaning Operations.

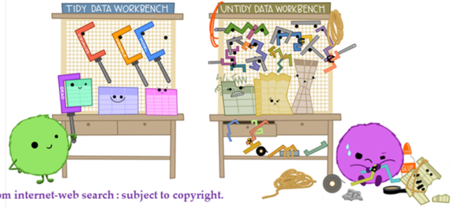

Fig: Tidying the Data.

Fig: Making Sense of Data.

5.1 Cleaning Our Data - Handling Missing Data.

Data Wrangling can comprise of any operation we perform on our data to clean and transform out data, such that further action of model building could be performed.Following are some of the most common action we perform to clean and validate our data.

- Handling Missing Values:
  - Identifying a Missing Values: If any of the rows has a missing values in your dataframe, it is represented as `NaN: Not a Number`.Thus we can use following functionality od pandas to inspect and identify the null values:
    - Missing values in a Pandas DataFrame can be identified with the `dataset.isnull()` method.
    - Total number of missing values in each column can be found using syntax `dataset.isnull().sum()`.
    - The easiest fix for handling missing data might be using `dataset.dropna()` methods, which drops the observation that even have a single missing value.
      {But use with cautions as it will reduce the amount of data we have.}

5.1.1 Cleaning Our Data - Filling Missing Values.

- Filling Missing Values:
  The techniques for filling the missing values are known as Data Imputation Techniques and Several do exits.The best way to impute the data will depend on the problem, and the assumptions taken. Below we present few techniques:
  1. Naive Method: Filling the missing value of a column by coping the value of the previous non-missing observation.
     - Syntax: `dataset.fillna(method = "ffill")`
  2. Imputing with the mean/median/constant:
     Missing values in the column can be imputed(filled) using the mean/median/constant of the non-missing values in the column.{constant can be any values such as 0
     - Syntax: `dataset.column.fillna(dataset.column.mean())`

  {Please check python documentation for more such imputations techniques}

5.1.2 Handling Missing values - Code Example.

- Add some Missing Values:

```
import pandas as pd
from sklearn.datasets import load_iris
import numpy as np
iris = load_iris() # Load the Iris dataset
iris_df = pd.DataFrame(data=np.c_[iris['data'], iris['
    target']], columns=iris['feature_names'] + ['target'
    ])
np.random.seed(42) # Introduce missing values randomly
mask = np.random.rand(*iris_df.shape) < 0.1 # 10%
iris_df[mask] = np.nan
print("Missing Values in Iris Dataset:")
print(iris_df.isnull().sum())
```

5.1.3 Filling Missing values - Code Example.

- Filling Missing Values:

```
# Filling missing values with forward fill (ffill), mean
    , median, and 0
iris_df_ffill = iris_df.ffill()
iris_df_mean = iris_df.fillna(iris_df.mean())
iris_df_median = iris_df.fillna(iris_df.median())
iris_df_zero = iris_df.fillna(0)
# Expand iris_df with filled columns
iris_df_expanded = pd.concat([iris_df, iris_df_ffill.
    add_suffix('_ffill'), iris_df_mean.add_suffix('_mean
    '),iris_df_median.add_suffix('_median'),iris_df_zero
    .add_suffix('_zero')], axis=1)
# Display the head of the expanded DataFrame
print("\nDataset after Filling Missing Values:")
print(iris_df_expanded.head())
```

5.1 Cleaning Our Data - Some other common operations.

Besides handling missing values we also perform following operations regularly as a part of data cleaning operations.

1. Trimming Whitespaces: Removing extra spaces from text columns.
   Syntax: `df['column_name'] = df['column_name'].str.strip()`

```
df = pd.DataFrame({'Name': ['Alice ', 'Bob '], 'Age': [25, 30]})
df['Name'] = df['Name'].str.strip()
```

2. Changing datatypes with `df.astype()`
   Syntax: `df['column_name'] = df['column_name'].astype(int)`

```
df = pd.DataFrame({'Age': ['25', '30', '35']})
# Change 'Age' column data type to integer
df['Age'] = df['Age'].astype(int)
print(df)
```

5.1.1 Cleaning Our Data - Some other common operations.

3. Renaming Columns with `df.rename()`.

```
# Rename columns
df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25,
    30]})
df = df.rename(columns={'Name': 'Full Name', 'Age': '
    Years'})
print(df)
```

4. Removing Duplicates with `df.drop_duplicates()`.

```
# Remove duplicate rows
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Alice'], '
    Age': [25, 30, 25]})
df = df.drop_duplicates()
print(df)
```

5.2 Cleaning Our Data - Data Transformations.

- Data Transformation is the process of converting raw data into a format that is suitable for analysis.
  - This involves modifying, reshaping, and enriching the data to improve its quality, structure, and usability for various analytical or machine learning tasks.
- Disclaimer!!! The tasks performed during data transformation depend on the nature of the data, type of analytical project and the goals of the analysis. The following slides highlight some of the most commonly performed data transformation operations.

5.2.1 Data Transformations - Reshaping.

1. Reshaping: Rearranging data structure to adapt to different analytical needs.
    - Pivoting: Pivoting transforms data from a long format to a wide format. Use `df.pivot()` to reorganize your data based on specific values.

```python
import pandas as pd
# Sample DataFrame
data = {'Date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
        'City': ['Kathmandu', 'Pokhara', 'Kathmandu', 'Pokhara'],
        'Temperature': [15, 18, 16, 19]}
df = pd.DataFrame(data)
# Pivot: Reshape data to show cities as columns
pivoted_df = df.pivot(index='Date', columns='City', values='Temperature')
print(pivoted_df)
```

5.2.1 Data Transformations - Reshaping.

1. Reshaping: Rearranging data structure to adapt to different analytical needs.
   - Melting: Melting converts data from wide format to long format using `pd.melt()`.

```
    # Melt: Convert wide data back to long format
melted_df = pd.melt(pivoted_df.reset_index(), id_vars=['Date'],
                    var_name='City', value_name='Temperature')
print(melted_df)
```

5.2.2 Data Transformations - Scaling.

2. Data Scaling or {Min - Max}: Min-Max scaling (also known as feature scaling or min-max normalization) is a technique used to scale and center the values of a feature in a specific range, usually between 0 and 1.

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

```python
import pandas as pd
from sklearn.datasets import load_iris
iris = load_iris() # Load the Iris dataset
iris_df = pd.DataFrame(data=iris['data'], columns=iris['feature_names'])
# Min-Max Scaling using Pandas
iris_minmax_scaled = (iris_df - iris_df.min()) / (iris_df.max() - iris_df.min())
print("Original Iris DataFrame:")
print(iris_df.head())
print("\nMin-Max Scaled Iris DataFrame:")
print(iris_minmax_scaled.head()) # Display scaled data
```

5.2.2 Data Transformations - Handling Categorical Variable.

- Encoding: Converting categorical data into numerical formats for compatibility with models.
  1. Ordinal or Label Encoding:
     - Ordinal encoding is used for categorical data with a meaningful order or ranking.
     - Each category is assigned a numerical value based on its order.
     - Example: Low, Medium, High can be encoded as 1, 2, 3.

```
import pandas as pd
# Sample DataFrame with ordinal categories
df = pd.DataFrame({'Category': ['Low', 'Medium', 'High', 'Low', 'High']})
# Ordinal encoding using map
ordinal_mapping = {'Low': 1, 'Medium': 2, 'High': 3}
df['Category_Ordinal'] = df['Category'].map(ordinal_mapping)
print(df)
```

5.2.2 Data Transformations - Handling Categorical Variable.

2. One Hot Encoding:
   - In one-hot encoding, each category is represented as a binary vector (0 or 1) in which all elements are zero except for the index that corresponds to the category.
   - If there are n categories, each category is represented by a vector of length n with all zeros except for a 1 at the index corresponding to the category.

```
import pandas as pd
df_municipalities = pd.DataFrame({'Municipality': ['
    Kathmandu', 'Bhaktapur', 'Lalitpur', 'Madhyapur Thimi',
    'Kirtipur']})
one_hot_encoding = pd.get_dummies(df_municipalities['
    Municipality'], prefix='Municipality')
df_encoded = pd.concat([df_municipalities, one_hot_encoding
    ], axis=1)
print(df_encoded)# Display the result
```

5.3 Merging and Joining DataFrames.

- Pandas provides powerful tools to combine datasets. Two commonly used operations are Concatenation and Merging.
  1. Concatenation with `pd.concat()`: Concatenation combines DataFrames either vertically (row-wise) or horizontally (column-wise). It does not require a common key. Syntax:
     - `pd.concat([df1, df2], axis=0)` Combine row-wise (default)
     - `pd.concat([df1, df2], axis=1)` Combine column-wise

```python
import pandas as pd
# Sample DataFrames
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
# Row-wise concatenation
combined_rows = pd.concat([df1, df2], axis=0)
print("Row-wise concatenation:")
print(combined_rows)
# Column-wise concatenation
combined_cols = pd.concat([df1, df2], axis=1)
print("\nColumn-wise concatenation:")
print(combined_cols)
```

5.4 Merging and Joining DataFrames.

2. Merging with `pd.merge()`: Merging combines DataFrames based on a common key or column. It is similar to SQL joins.
   Types of joins:
   - Inner Join: Keeps only matching rows.
   - Outer Join: Includes all rows, filling missing values where no match is found.
   - Left Join: All rows from the left DataFrame and matching rows from the right.
   - Right Join: All rows from the right DataFrame and matching rows from the left.

   Syntax:

```
pd.merge(df1, df2, on='key_column', how='join_type')
```

5.4.1 Merging - Code Example.

```
# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob'
    , 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Score': [85, 90, 88]})
# Inner join
inner_merged = pd.merge(df1, df2, on='ID', how='inner')
print("Inner Join:")
print(inner_merged)
# Left join
left_merged = pd.merge(df1, df2, on='ID', how='left')
print("\nLeft Join:")
print(left_merged)
# Outer join
outer_merged = pd.merge(df1, df2, on='ID', how='outer')
print("\nOuter Join:")
print(outer_merged)
```

Towards Worksheet - 2.

Thank You.