* Introduction of pandas:

- Pandas is a Python package that provides data structures and functions for data analysis and manipulation. It's a popular tool for data scientists.
- Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.
- Pandas is an open-source Python library highly regarded for its data analysis and manipulation capabilities. It streamlines the processes of cleaning, modifying, modeling, and organizing data to enhance its comprehensibility and derive valuable insights.
- Pandas is a powerful and open-source Python library.

* Key Data structures:

1) Data Structures:

- Series: A one-dimensional labeled array (similar to a column in Excel).
- DataFrame: A two-dimensional table with labeled rows and columns (like an Excel spreadsheet or SQL table).

2) Data Manipulation:

- Reading and writing data from various file formats (CSV, Excel, SQL, JSON, etc.).
- Filtering, sorting, and grouping data.
- Handling missing data.
- Merging and joining datasets.
- Applying functions to transform data.

1. Series (1D labeled array)

A Series is essentially a one-dimensional array with labels (index).

In [1]:
```python
import pandas as pd

# Creating a Series
data = [10, 20, 30, 40]
s = pd.Series(data, index=['a', 'b', 'c', 'd'])

print(s)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

2. DataFrame (2D labeled table)

A DataFrame is a two-dimensional table (like a spreadsheet) where each column is a Series

In [2]:
```
# DataFrame
import pandas as pd

# Creating a DataFrame
data = {
    "Name": ["Ram", "Shree", "Parth"],
    "Age": [25, 30, 35],
    "City": ["Pune", "Mumbai", "Vashi"]
}

df = pd.DataFrame(data)

print(df)
```

```
    Name  Age    City
0    Ram   25    Pune
1  Shree   30  Mumbai
2  Parth   35   Vashi
```

* Creating Data structures:

In [3]:
```
# Creating a Series with custom index
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

print(s)
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

In [4]:
```
# Creating a DataFrame from a dictionary
data = {
    "Name": ["Saee", "Jiva", "Om"],
    "Age": [25, 30, 35],
    "City": ["kolhapur", "satara", "pune"]
}

df = pd.DataFrame(data)

print(df)
```

```
   Name  Age      City
0  Saee   25  kolhapur
1  Jiva   30    satara
2    Om   35      pune
```

* Data Operations: Indexing & Slicing: df['col'], df.iloc[], df.loc[]

Pandas provides powerful ways to index and slice data in a DataFrame using df['col'], df.loc[], and df.iloc[].

1. Selecting Columns

```
In [29]:
data = {
    "Name": ["Saee", "Jiva", "Om"],
    "Age": [25, 30, 35],
    "City": ["kolhapur", "satara", "pune"]
}

df = pd.DataFrame(data)


print(df["Age"])
```

```
0    25
1    30
2    35
Name: Age, dtype: int64
```

2. Selecting Rows

Using iloc[] (Index-based selection)

- df.iloc[] selects rows by position (starting from 0).

```
In [6]:
data = {
    "Name": ["Saee", "Jiva", "Om"],
    "Age": [25, 30, 35],
    "City": ["kolhapur", "satara", "pune"]
}

df = pd.DataFrame(data)

print(df.iloc[0])
```

```
Name        Saee
Age           25
City    kolhapur
Name: 0, dtype: object
```

3. Selecting Specific Rows & Columns

- Using loc[]

```
In [7]:
data = {
    "Name": ["Saee", "Jiva", "Om"],
    "Age": [25, 30, 35],
    "City": ["kolhapur", "satara", "pune"]
}

df = pd.DataFrame(data)

print(df.loc[0, ["Name", "Age"]])
```

```
Name     Saee
Age        25
Name: 0, dtype: object
```

* Data Cleaning: .isnull(), .dropna(), .fillna()

Pandas provides powerful data cleaning functions like .isnull(), .dropna(), and .fillna() to handle missing values.

1. Detecting Missing Values (.isnull(), .notnull())

- df.isnull() → Returns True where values are missing.
- df.notnull() → Returns False where values are not missing.

In [8]:
```python
import pandas as pd
import numpy as np

data = {
    "Name": ["Ram", "Sham", "Ruhi", "Dev"],
    "Age": [25, np.nan, 35, 40],  # Missing value in Age
    "City": ["Pune", "Sangli", np.nan, "Satara"]  # Missing value in City
}

df = pd.DataFrame(data)

print(df.isnull())  # Check for missing values
```

```
    Name    Age   City
0  False  False  False
1  False   True  False
2  False  False   True
3  False  False  False
```

2. Removing Missing Values (.dropna())

Removes rows or columns with missing values.

- df.dropna(axis=0) → Drops rows with NaN (default).
- df.dropna(axis=1) → Drops columns with NaN.

In [9]:
```python
data = {
    "Name": ["Ram", "Sham", "Ruhi", "Dev"],
    "Age": [25, np.nan, 35, 40],  # Missing value in Age
    "City": ["Pune", "Sangli", np.nan, "Satara"]  # Missing value in City
}

df = pd.DataFrame(data)
cleaned_df = df.dropna()
print(cleaned_df)
```

```
  Name   Age    City
0  Ram  25.0    Pune
3  Dev  40.0  Satara
```

3. Filling Missing Values (.fillna())

Replace missing values with a specific value or method.

In [10]:
```
data = {
    "Name": ["Ram", "Sham", "Ruhi", "Dev"],
    "Age": [25, np.nan, 35, 40],  # Missing value in Age
    "City": ["Pune", "Sangli", np.nan, "Satara"]  # Missing value in City
}

df = pd.DataFrame(data)
filled_df = df.fillna("Unknown")
print(filled_df)
```

```
   Name      Age     City
0   Ram     25.0     Pune
1  Sham  Unknown   Sangli
2  Ruhi     35.0  Unknown
3   Dev     40.0   Satara
```

* Renaming: df.rename()

Filtering: df[df['col'] > value]

1. Renaming Columns & Index (df.rename())

- df.rename(columns={"old_name": "new_name"}) → Renames column(s).
- df.rename(index={old_index: new_index}) → Renames row index.

In [11]:
```
data = {
    "Name": ["Ram", "Sham", "Ruhi", "Dev"],
    "Age": [25, np.nan, 35, 40],  # Missing value in Age
    "City": ["Pune", "Sangli", np.nan, "Satara"]  # Missing value in City
}
df = pd.DataFrame(data)

# Rename columns
df = df.rename(columns={"Name": "Full Name", "Age": "Years"})
print(df)
```

```
  Full Name  Years    City
0       Ram   25.0    Pune
1      Sham    NaN  Sangli
2      Ruhi   35.0     NaN
3       Dev   40.0  Satara
```

2. Filtering Data (df[df['col'] > value])

You can filter rows based on conditions.

In [12]:
```
data = {
    "Name": ["Ram", "Sham", "Ruhi", "Dev"],
    "Age": [25, np.nan, 35, 40],  # Missing value in Age
    "City": ["Pune", "Sangli", np.nan, "Satara"]  # Missing value in City
}
df = pd.DataFrame(data)
df = df.rename(columns={"Name": "Full Name", "Age": "Years"})
filtered_df = df[df["Years"] > 28]
print(filtered_df)
```

```
  Full Name  Years    City
2      Ruhi   35.0     NaN
3       Dev   40.0  Satara
```

*Aggregation and Grouping:

1. Grouping Data (df.groupby('col'))

- df.groupby('col') groups data based on a column.
- Combine it with aggregation functions (sum(), mean(), count(), etc.).

In [13]:
```
data = {
    "Department": ["HR", "IT", "IT", "HR", "Finance", "Finance", "IT"],
    "Employee": ["Ram", "Sham", "Ruhi", "Dev","Saee", "Jiva", "Om"],
    "Salary": [50000, 70000, 80000, 52000, 60000, 65000, 72000]
}

df = pd.DataFrame(data)

# Group by 'Department' and calculate total Salary
grouped = df.groupby("Department")["Salary"].sum()
print(grouped)
```

```
Department
Finance    125000
HR         102000
IT         222000
Name: Salary, dtype: int64
```

2. Aggregation Functions (.sum(), .mean(), .count())

- You can apply different aggregation functions to grouped data.

In [14]:
```
data = {
    "Department": ["HR", "IT", "IT", "HR", "Finance", "Finance", "IT"],
    "Employee": ["Ram", "Sham", "Ruhi", "Dev","Saee", "Jiva", "Om"],
    "Salary": [50000, 70000, 80000, 52000, 60000, 65000, 72000]
}

df = pd.DataFrame(data)

agg_df = df.groupby("Department").agg({
    "Salary": ["sum", "mean"],
    "Employee": "count"
})
print(agg_df)
```

```
            Salary          Employee
               sum     mean    count
Department
Finance     125000  62500.0        2
HR          102000  51000.0        2
IT          222000  74000.0        3
```

* Merging and joining:

Merging DataFrames (pd.merge())

pd.merge(df1, df2, on='column') → Combines two DataFrames based on a common column.

Supports different types of joins:

```
* Inner Join (default) → Matches common values.
* Left Join → Keeps all values from the left DataFrame.
* Right Join → Keeps all values from the right DataFrame.
* Outer Join → Keeps all values from both DataFrames.
```

In [15]:
```
df1 = pd.DataFrame({
    "ID": [1, 2, 3],
    "Name": ["Ram", "Sham", "Ruhi"]
})

df2 = pd.DataFrame({
    "ID": [2, 3, 4],
    "Salary": [50000, 60000, 70000]
})

# Inner Join (default) - only matching IDs will be merged
merged_df = pd.merge(df1, df2, on="ID")
print(merged_df)

left_join = pd.merge(df1, df2, on="ID", how="left")
print("left_join",left_join)
```

```
   ID  Name  Salary
0   2  Sham   50000
1   3  Ruhi   60000
left_join    ID  Name   Salary
0   1   Ram      NaN
1   2  Sham  50000.0
2   3  Ruhi  60000.0
```

2. Concatenating DataFrames (pd.concat())

- Stacks DataFrames either vertically (default) or horizontally.
- Useful for combining datasets with the same columns.

In [16]:
```
df1 = pd.DataFrame({
    "ID": [1, 2],
    "Name": ["Ram", "Sham"]
})
```

```
df2 = pd.DataFrame({
    "ID": [3, 4],
    "Name": [ "Jiva", "Om"]
})

concat_df = pd.concat([df1, df2])
print(concat_df)
```

```
   ID  Name
0   1   Ram
1   2  Sham
0   3  Jiva
1   4    Om
```

* File I/O:

pd.read_csv(), df.to_csv(), pd.read_excel(), df.to_excel()

Pandas provides powerful File I/O methods to read and write data in various formats, such as CSV and Excel.

1. Reading Data from a File

- Reading a CSV File (pd.read_csv())
- Loads a CSV (Comma-Separated Values) file into a Pandas DataFrame.

In [ ]:
```
import pandas as pd

df = pd.read_csv("data.csv")  # Reads 'data.csv' into a DataFrame
print(df.head())  # Display the first 5 rows
```

2) Writing Data to a File

- Writing to a CSV File (df.to_csv())
- Saves a Pandas DataFrame as a CSV file.

In [ ]:
```
df.to_csv("output.csv", index=False)  # Save without the index column
```

3) Reading an Excel File

- (pd.read_excel())
- Reads data from an Excel file (.xlsx).

In [ ]:
```
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")  # Read specific sheet
print(df.head())
```

4) Writing to an Excel File

- (df.to_excel())
- Saves a DataFrame to an Excel file (.xlsx).

In [ ]:
```
df.to_excel("output.xlsx", sheet_name="Results", index=False)
```

* Time Series:

Datetime: pd.to_datetime(), .resample()

- Time Series Handling in Pandas
- Pandas provides powerful tools for working with datetime data and time series. Key functions include:
- pd.to_datetime() → Convert a column to datetime format.
- .resample() → Aggregate data over a time period (daily, monthly, yearly, etc.).

1. Converting to Datetime (pd.to_datetime())

Converts a column or list of date strings into a datetime format.

In [18]:
```
data = {"Date": ["2023-01-01", "2023-01-02", "2023-01-03"], "Value": [10, 20, 30
df = pd.DataFrame(data)

# Convert 'Date' column to datetime format
df["Date"] = pd.to_datetime(df["Date"])
print(df.dtypes)
```

```
Date     datetime64[ns]
Value             int64
dtype: object
```

2. Resampling Time Series Data (.resample())

Resampling aggregates time-series data over different time intervals (e.g., daily, weekly, monthly).

In [19]:
```
df.set_index("Date", inplace=True)

# Resample to monthly frequency and calculate mean
monthly_df = df.resample("M").mean()
print(monthly_df)
```

```
            Value
Date             
2023-01-31   20.0
```

* Data visualization:

Pandas allows to create various graphs directly from your data using built-in functions. This tutorial covers Pandas capabilities for visualizing data with line plots, area charts, bar plots, and more.

Pandas offers several features that make it a great choice for data visualization:

-

Variety of Plot Types: Pandas supports various plot types including line plots, bar plots, histograms, box plots, and scatter plot

- Customization: Users can customize plots by adding titles, labels, and styling enhancing the readability of the visualizations.
- Handling of Missing Data: Pandas efficiently handles missing data ensuring that visualizations accurately represent the dataset without errors
- Integration with Matplotlib: Pandas integrates with Matplotlib that allow users to create a wide range of static, animated, and interactive plots.lots.

df.plot() for basic plotting.

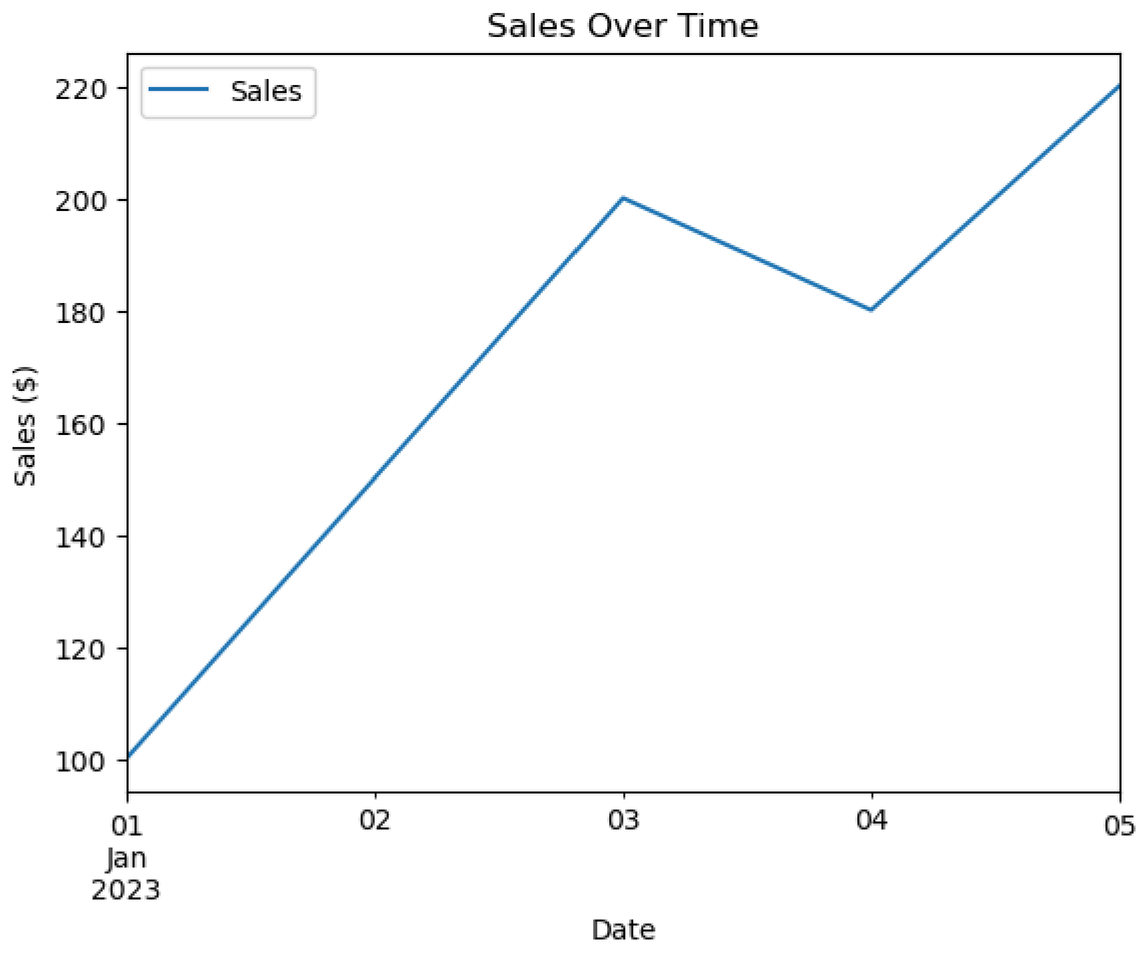
- df.plot() for Basic Plotting in Pandas
- df.plot() is a built-in Pandas function that allows for quick data visualization using Matplotlib. It provides various chart types to help understand data trends easily.

In [20]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Sample data
data = {"Date": pd.date_range(start="2023-01-01", periods=5, freq="D"),
        "Sales": [100, 150, 200, 180, 220]}

df = pd.DataFrame(data)
df.set_index("Date", inplace=True)

# Plot sales over time
df.plot(title="Sales Over Time", ylabel="Sales ($)")
plt.show()
```



* Condition Selection: And ,or

"Condition Selection" refers to the process of choosing specific criteria or conditions within a program or analysis to filter data or determine which actions to take, essentially making decisions based on whether certain conditions are met; "And" and "Or" are logical operators used within these conditions, where "And" requires all specified conditions to be true for the outcome to be true, while "Or" only needs one of the conditions to be true to achieve a true outcome.

1. Using AND (&)

Both conditions on either side of the "And" operator must be true for the overall condition to be true.

In [27]:
```
data = {"Product": ["A", "B", "C", "D"],
        "Sales": [100, 200, 150, 250],
        "Profit": [20, 50, 30, 80]}

df = pd.DataFrame(data)

# Apply AND condition
filtered_df = df[(df["Sales"] > 150) & (df["Profit"] > 30)]
print(filtered_df)
```

```
  Product  Sales  Profit
1       B    200      50
3       D    250      80
```

2. Using OR (|)

Only one of the conditions on either side of the "Or" operator needs to be true for the overall condition to be true.

In [26]:
```
data = {"Product": ["A", "B", "C", "D"],
        "Sales": [100, 200, 150, 250],
        "Profit": [20, 50, 30, 80]}

df = pd.DataFrame(data)

filtered_df = df[(df["Sales"] > 150) | (df["Profit"] > 30)]
print(filtered_df)
```

```
  Product  Sales  Profit
1       B    200      50
3       D    250      80
```

In [ ]:
```
```