Contents

[PANDAS 2](#_Toc210506801)

[SERIES 2](#_Toc210506802)

[CODE EXAMPLE 2](#_Toc210506803)

[BROADCASTING IN PANDAS SERIES 2](#_Toc210506804)

[DATAFRAMES 4](#_Toc210506805)

[CREATING A DATAFRAME 4](#_Toc210506806)

[READ CSV IN DATAFRAME 6](#_Toc210506807)

[WORKING WITH COLUMNS 7](#_Toc210506808)

[WORKING WITH ROWS 8](#_Toc210506809)

[RETERIVE ROW/ROWS FROM DATA FRAME 8](#_Toc210506810)

[APPENDING ROW/ROWS IN DATA FRAME 9](#_Toc210506811)

[CONDITIONAL FILTERING 9](#_Toc210506812)

[FILTER BY SINGLE CONDITION 10](#_Toc210506813)

[apply() METHOD 10](#_Toc210506814)

[MISSING DATA 10](#_Toc210506815)

[GROUPBY OPERATIONS 11](#_Toc210506816)

[EXAMPLE -1 11](#_Toc210506817)

[MULTIINDEX 12](#_Toc210506818)

[EXAMPLE -2 13](#_Toc210506819)

[COMBINING DATAFRAMES 13](#_Toc210506820)

[TEXT AND TIME METHODS 13](#_Toc210506821)

[INPUT AND OUTPUTS 14](#_Toc210506822)

[CSV 14](#_Toc210506823)

[WRITING TO CSV 14](#_Toc210506824)

[HTML 15](#_Toc210506825)

[EXCEL 15](#_Toc210506826)

[SQL DATABASE 15](#_Toc210506827)

# PANDAS

* Open-source library for data analysis. Documentation : <https://pandas.pydata.org/docs/>

# SERIES

A **Pandas Series** is a **one-dimensional labeled array** in Python, capable of holding any data type — integers, strings, floats, etc. Widely used for data analysis and manipulation.

**KEY FEATURES OF A SERIES:**

* **Labeled index**: Each element has an associated label (like a dictionary key).
* **Homogeneous data**: All elements are of the same type.
* **Built on NumPy**: Efficient and fast under the hood.

|  |  |
| --- | --- |
| NUMPY ARRAY | PANDAS SERIES |
| |  |  | | --- | --- | | **Index** | **Data** | | 0 | 1776 | | 1 | 1867 | | 2 | 1821 |  * In numpy array every element has a numeric index. Because of this feature we can be able to retrieve the numpy element using index.   **num\_array = np.array([1, 2, 3])**  **num\_array[0]**   * On the similar lines, by default - every row element in pandas has index.For example, to store the year of independence of countries | |  |  |  | | --- | --- | --- | | **Index** | **Labelle Index** | **Data** | | 0 | USA | 1776 | | 1 | CANADA | 1867 | | 2 | MEXICO | 1821 |  * Pandas Series adds labeled index. Hence, we can grab data using labelled index. * Pandas Series just holding NumPy array internally, * and then adding on that named index. We can grab information through a numeric index or labeled index within a Pandas Series |

## CODE EXAMPLE

|  |  |
| --- | --- |
| import pandas as pd  myindex = ["USA", "Canada", "Mexico"]  mydata = [1776, 1867, 1821]  **myseries = pd.Series(data=mydata, index=myindex)**  print(myseries)  print(myseries["USA"]) 🡪 1776 | **CREATING NAMED INDEX SERIES**  **OUTPUT** USA 1776  Canada 1867  Mexico 1821 |
| ages = {"Alice": 30, "Bob": 25, "Charlie": 35}  age\_series = pd.Series(ages)  print(age\_series)  print(age\_series["Alice"]) 🡪 30 | **CREATING PANDAS SERIES USING DICTIONARY**  Alice 30  Bob 25  Charlie 35 |

* Note: The keys are case sensitive. Unavailability of key gives “KeyError”

|  |  |
| --- | --- |
| # Imaginary Sales Data for 1st and 2nd Quarters for Global Company  q1 = {"Japan": 80, "China": 450, "India": 200, "USA": 250}  q2 = {"Brazil": 100, "China": 500, "India": 210, "USA": 260} | |
| **PANDA SERIES FROM DICTIONARY** | sales\_q1 = pd.Series(q1)  sales\_q2 = pd.Series(q2) |
| **ALL KEYS FROM SERIES** | sales\_q1.keys()  **O/p : Index(['Japan', 'China', 'India', 'USA'], dtype='object')** |

### BROADCASTING IN PANDAS SERIES

|  |  |
| --- | --- |
| **MULTIPLYING VALUES IN PANDAS SERIES** | **double\_sales = sales\_q1 \* 2**  **print(double\_sales)**  O/P **Japan 160**  **China 900**  **India 400**  **USA 500** |

|  |
| --- |
| import pandas as pd  # Imaginary Sales Data for 1st and 2nd Quarters for Global Company  q1 = {"Japan": 80, "China": 450, "India": 200, "USA": 250}  q2 = {"Brazil": 100, "China": 500, "India": 210, "USA": 260}  sales\_q1 = pd.Series(q1)  sales\_q2 = pd.Series(q2)  total\_sales = sales\_q1 + sales\_q2  print(total\_sales)  total\_add\_sales = sales\_q1.add(sales\_q2, fill\_value=0)  print(total\_add\_sales) |

**EXPLANATION OF THE DIFFERENCE BETWEEN:**

* **total*sales = sales*q1 + sales*q2***
* ***total*add*sales = sales*q1.add(sales*q2, fill*value=0)**

|  |  |
| --- | --- |
| **total\_sales = sales\_q1 + sales\_q2**  This performs **element-wise addition** based on **matching indices (countries)**.   * If a country exists in **both Series**, their values are added. * If a country exists in **only one Series**, the result is **NaN** (missing). | **Result:**  Brazil NaN # Missing in Q1  China 950.0 # 450 + 500  India 410.0 # 200 + 210  Japan NaN # Missing in Q2  USA 510.0 # 250 + 260  dtype: float64 |
| **total\_add\_sales = sales\_q1.add(sales\_q2, fill\_value=0)**  This uses the .add() method with fill\_value=0, which means:   * If a country is **missing in either Series**, treat its value as **0**. * This avoids NaN and gives a **complete result**. | **Result:**  Brazil 100.0 # 0 + 100  China 950.0 # 450 + 500  India 410.0 # 200 + 210  Japan 80.0 # 80 + 0  USA 510.0 # 250 + 260  dtype: float64 |

Key Differences

|  |  |  |
| --- | --- | --- |
| **Feature** | **sales\_q1 + sales\_q2** | **sales\_q1.add(sales\_q2, fill\_value=0)** |
| Missing keys | Result is NaN | Treated as 0 |
| Output completeness | May have missing values | Always complete |
| Use case | When you want to preserve missing data | When you want a full merged result |

When to Use Which?

* Use + when you want to **identify missing data** or preserve gaps.
* Use .add(..., fill\_value=0) when you want to **combine data safely**, assuming missing values are zero.

# DATAFRAMES

* A **DataFrame** is a **two-dimensional, labeled data structure** in Python provided by the **Pandas** library. Think of it like an **Excel spreadsheet** or a **SQL table** — it has **rows and columns**, and each column can hold different types of data.
* **Formal Definition: A group of pandas Series object that share the same index**

A diagram of a graph

AI-generated content may be incorrect.

* This example demonstrates multiple series that share the same index. Each of the three series uses the same index values: USA, Canada, Mexico, and others.
* These series can be combined into a single DataFrame, which retains the shared index and adds multiple columns representing each series.
* A data Frame consists of several series with a common index. Structurally, it is like a spreadsheet at a high-level overview.

Key Features of a DataFrame:

* **Rows and columns** with labels (indexes).
* Can hold **heterogeneous data types** (e.g., integers, strings, floats).
* Built on top of **NumPy** for performance.
* Powerful for **data manipulation**, **cleaning**, and **analysis**.

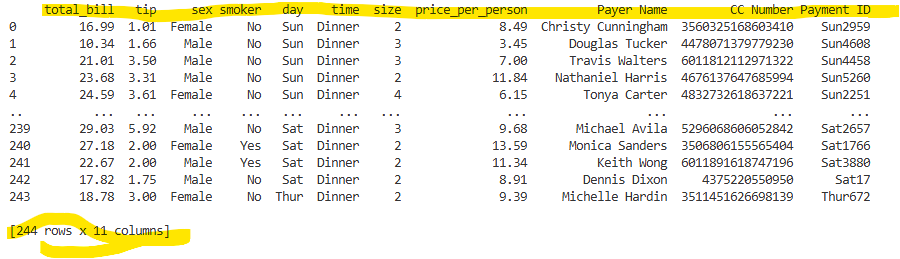
## CREATING A DATAFRAME

|  |  |
| --- | --- |
| ## CREATING THE MATRIX of size **4X3** with numbers between **1-101**  np.random.seed(101)  data = np.random.randint(1, 101, size=(4, 3)) | [[96 12 82]  [71 64 88]  [76 10 78]  [41 5 64]] |
| myindex = ["CA", "NY", "AZ", "TX"]  df = pd.DataFrame(data, index=myindex) | |  | | --- | | 0 1 2 | | **CA** 96 12 82 | | **NY** 71 64 88 | | **AZ** 76 10 78 |   **TX** 41 5 64 |
| mycolumns = ["Jan", "Feb", "Mar"]  df = pd.DataFrame(data, index=myindex, columns=mycolumns)   * This will add **named index and labeled column name** for the data | |  | | --- | | Jan Feb Mar | | CA 96 12 82 | | NY 71 64 88 | | AZ 76 10 78 | | TX 41 5 64 | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| import pandas as pd  # From a dictionary  data = {      'Name': ['Alice', 'Bob', 'Charlie'],      'Age': [25, 30, 35],      'City': ['Delhi', 'Mumbai', 'Bangalore']  }  df = pd.DataFrame(data)  print(df) | **Output:**   |  | | --- | | Name Age City | | 0 Alice 25 Delhi | | 1 Bob 30 Mumbai | | 2 Charlie 35 Bangalore | |

|  |  |  |  |
| --- | --- | --- | --- |
| * When we create pandas dataframe it automatically assigns a **default index-**  if we don’t specify one. * **DEFAULT INDEX**: The default index is a **RangeIndex**: a sequence of integers starting from **0** up to **n-1**, where **n** is the number of rows.   + It looks like: **RangeIndex(start=0, stop=n, step=1)** | | | |
| **import pandas as pd**  **df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie'],'Score': [85, 90, 95]})**  **print(df)** | | Name Score  0 Alice 85  1 Bob 90  2 Charlie 95  0,1, 2 are the default index values. | |
| Why Does Pandas Create an Index?   * The index helps **identify rows** uniquely. * It enables **label-based selection** using `.loc[]`. * It supports **alignment** during operations like merging, joining, and reshaping. | | | |
| Customizing the Index  **We can replace the default index with meaningful labels:**  df = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})  print(df)  df.index = ["s1", "s2"]  df.to\_csv("students.csv") |  | |
| df = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})  df.set\_index("Name", inplace=True)  df.to\_csv("students.csv") |  | |
| **GETTING THE INDEX: We can get the index of the dataframe using df.index** | | |
| Example-1 : When the index is default index of dataframe  OUTPUT  RangeIndex(start=0, stop=2, step=1) | df = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})  **print(df.index)** | |
| Example-2: Explicitly set the index of data frame using set\_index()  OUTPUT  Index(['Alice', 'Bob'], dtype='object', name='Name') | df = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})  df.set\_index("Name", inplace=True)  print(df)  print(df.index)  df.to\_csv("students.csv") | |

## READ CSV IN DATAFRAME



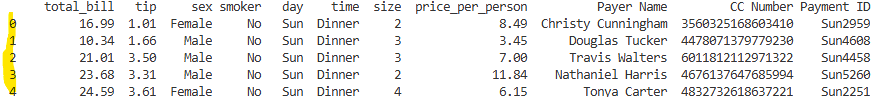
|  |  |
| --- | --- |
| **READ CSV IN DATAFRAME** | **df = pd.read\_csv("tips.csv")** |
| **LIST OF COLUMN NAME**  **(X-AXIS)** | **print(df.columns)**  **OUTPUT**  **Index(['total\_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size', 'price\_per\_person', 'Payer Name', 'CC Number', 'Payment ID'], dtype='object')** |
| **INDEX**  **(Y-AXIS)** | **print(df.index)**  For a dataframe - pandas assigns a RangeIndex starting from 0. So, the output will typically look like:  **RangeIndex(start=0, stop=244, step=1)**  This means:   * The index starts at 0. * Goes up to (but not including) 244. * In steps of 1. |
| **Dataframe Head** | **df.head(n)**   * **n**: (optional) The number of rows to return. Default is 5.(i.e. no value provided it shows top 5 rows) or returns **first n rows** of the dataframe. * It is useful for **quick inspection** of the data structure, column names, and sample values. |
| **Dataframe Tail** | **df.tail(n)**   * **n** *(optional)*: Number of rows to return from the end of the DataFrame. If not specified, it returns the last **5 rows** by default. * The df.tail() function in **pandas** is used to display the **last few rows** of a DataFrame. |
| **META DATA OF DATAFRAME**   * **Object represents the string datatype** | **df.info()** |
| **STATISTICAL SUMMARY OF DATAFRAME**  A new DataFrame showing statistics like:   * **count: Number of non-null entries** * **mean: Average value** * **std: Standard deviation** * **min: Minimum value** * **25%: First quartile (Q1)** * **50%: Median (Q2)** * **75%: Third quartile (Q3)** * **max: Maximum value** | * **The df.describe() function provides a statistical summary of the DataFrame’s numeric columns by default.** |

## WORKING WITH COLUMNS

|  |  |  |  |
| --- | --- | --- | --- |
| ACCESSING SINGLE COLUMN | SYNTAX: df[‘column\_name’]  **EXAMPLE : df["total\_bill"]**  *Note: The* ***type(df["total\_bill"]) is <class 'pandas.core.series.Series'>*** | | |
| ACCESSING MULTIPLE COLUMN | **column\_name\_list = ["total\_bill", "tip", "size"]**  **df[column\_name\_list]**  **OR**  **df[["total\_bill", "tip", "size"]]** | |  |
| **CREATING A NEW COLUMN** : Let’s say we want to add a new column in the dataframe**: tip\_percentage**  Note : If the column with same name exists - the data will be overridden | | | |
| df["tip\_percentage"] = (df["tip"] / df["total\_bill"]) \* 100  print(df.head()) | | | |
| **ROUNDING OFF**  df["tip\_percentage"] **= np.round**((df["tip"] / df["total\_bill"]) \* 100, 2)  print(df.head()) | | | |
| **DROPPING COLUMN(S)**   * The df.drop() function in **pandas** is used to **remove rows or columns** from a DataFrame. * **Syntax: df.drop(labels, axis=0 or 1, inplace=False)**   **🔹 Parameters:**   * **labels**: The row or column names to drop. * **axis**:   + 0 → drop rows   + 1 → drop columns * **inplace**:   + False (default) → returns a new DataFrame   + True → modifies the original DataFrame   **Examples:** | | | |
| **DROP A COLUMN** | | df=df.drop('tip', axis=1) | |
| **DROP MULTIPLE COLUMNS** | | df=df.drop(['tip', 'size'], axis=1) | |
| **DROP A ROW BY INDEX** | | df=df.drop(0, axis=0) | |
| **DROP MULTIPLE ROWS** | | df=df.drop([0, 1, 2], axis=0) | |
| **DROP IN-PLACE (MODIFIES ORIGINAL)** | | df=df.drop('tip', axis=1) | |

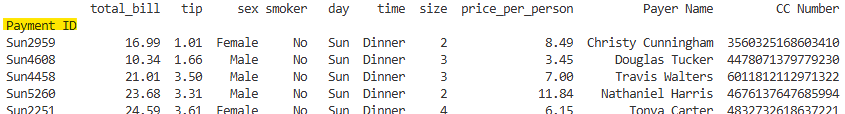
## WORKING WITH ROWS

* By default pandas assigns a index column to dataframe rows, which is unique to the row .
* It act as primary key for the dataframe.

****

Rather than using the default row-index, If we have any other column which is unique that can act as row index

**df = df.set\_index("Payment ID")**

Note: Since we are assigning back to “df” -it will change the original dataframe

Note we can reset back to default row index using **df = df.reset\_index()**

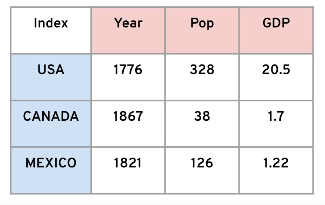
### RETERIVE ROW/ROWS FROM DATA FRAME

|  |  |
| --- | --- |
| **USING INDEX** | Retrieving specific row : **df.iloc[0]** 🡪 Retrieve the 1st row |
| Retrieve multiple rows: **df.iloc[0:4]** |
| **USING LABELLED NAMED** | Retrieve specific row with a given label index **: df.loc["Sun4608"]** |
| Retrieve multiple rows with labelled index **indexed\_list = ["Sun4608", "Sun2959", "Sun5260"]**  **print(df.loc[indexed\_list])** |

### APPENDING ROW/ROWS IN DATA FRAME

# CONDITIONAL FILTERING

* Conditional filtering allows us to select “rows” based on condition on “column”



* In above dataframe
  + The columns are called “Features”[Year(of independence), population & GDP]
  + The rows are called “Instance” of data

CONDITION: FIND THE COUNTRY WHOSE POLPULATION IS GREATER THAN 50

|  |  |
| --- | --- |
| A table with numbers and a few different colored squares  AI-generated content may be incorrect. |  |
| Final dataframe after applying the the condition | |

* To find countries with a population greater than 50, we can directly compare the population column to 50 in Pandas.
* This comparison is broadcast across each value in the column, returning a Boolean series—True where the population exceeds 50 and False otherwise.
* We then use this Boolean series to filter and select the rows where the condition holds true.

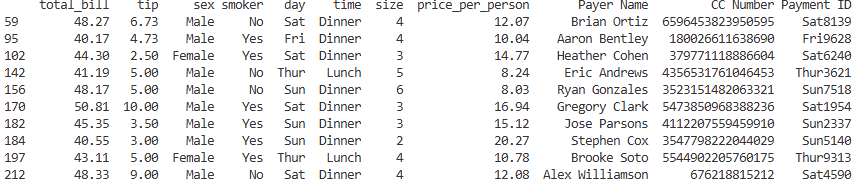
## FILTER BY SINGLE CONDITION

**CONDITION:** Select the row where total\_bill is more than 40

A screenshot of a computer screen

AI-generated content may be incorrect.

df = df[df["total\_bill"] > 40]



* The dataframe will return only those rows where **df["total\_bill"] > 40** is **true**

|  |  |
| --- | --- |
| Dataframe where “sex” column is Male | df = df[df["sex"] == "Male"] |
| **AND CONDITION** | df = df[(df["total\_bill"] > 40) **&** (df["sex"] == "Male")]  ***Select the rows in dataframe where total\_bill > 40 and sex is ‘Male’*** |
| **OR CONDITION** | df[(df["total\_bill"] > 40) | (df["sex"] == "Male")]  Select the rows in dataframe where total bill >40 or sex is ‘Male’ |
| **FILTER BY LIST** | weekend = ["Sat", "Sun"]  df = df[df["day"].**isin**(weekend)]  Filter the data where day is either Saturday or Sunday |

# apply() METHOD

* apply() method call to apply any custom python function along an axis of a DataFrame or to elements of a Series

# MISSING DATA

# GROUPBY OPERATIONS

A diagram of a number

AI-generated content may be incorrect.

* The groupby() function is a powerful tool used for **grouping data** and performing operations on each group independently. It’s commonly used for **data aggregation**, **transformation**, and **filtering**.
* The groupby() method splits the data into groups based on some criteria. It follows a **split-apply-combine** strategy:

1. **Split** the data into groups.
2. **Apply** a function to each group independently.
3. **Combine** the results into a data structure.

**Basic Syntax :** df.groupby('column\_name')

We can group by:

* A single column
* Multiple columns
* A function applied to a column

## EXAMPLE -1

|  |  |
| --- | --- |
| **CREATE DATAFRAME** | import pandas as pd  data = {      'Department': ['Sales', 'Sales', 'HR', 'HR', 'IT', 'IT'],      'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],      'Salary': [50000, 60000, 45000, 47000, 70000, 72000]  }  df = pd.DataFrame(data) |
| **Grouping by Department and Calculating Average Salary** | avg\_department\_salary = df.groupby("Department")["Salary"].mean() |
| **Multiple Aggregations** | agg\_department\_salary = df.groupby("Department")["Salary"].**agg(["mean", "sum", "max"])** |

## MULTIINDEX

* A MultiIndex (**also known as a hierarchical index)** allows us to index and organize data across multiple levels.
* It’s especially useful when working with grouped data, pivot tables, or multi-dimensional datasets.

Creating a MultiIndex with groupby()

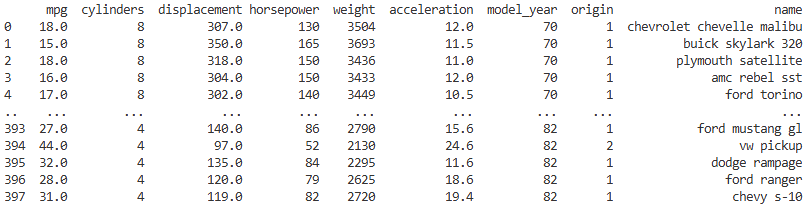
|  |  |
| --- | --- |
| import pandas as pd  data = {      'Department': ['Sales', 'Sales', 'HR', 'HR', 'IT', 'IT'],      'Team': ['A', 'B', 'A', 'B', 'A', 'B'],      'Salary': [50000, 60000, 45000, 47000, 70000, 72000]  }  df = pd.DataFrame(data)  grouped = df.groupby(['Department', 'Team']).mean() |  |

**Here, Department and Team form a MultiIndex on the rows.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
| ACCESSING DATA IN MULTIINDEX | | **sales\_grouped = grouped.loc["Sales"]** | |
| GROUPING BY MULTIPLE COLUMNS RETURNS MULTIPLE INDEX AS TUPLE  **avg\_mpg\_by\_cylinders = df.groupby(["model\_year", "cylinders"])["mpg"].mean()**  **index = avg\_mpg\_by\_cylinders.index**   * Each tuple represents a unique combination of **model\_year** and **cylinders**. * For example – 70 is the outer index and 4 is the inner index   *Multiindex help in slicing or filtering data based on combinations of model\_year and cylinders.* | | | A screenshot of a math test  AI-generated content may be incorrect. |
| **Access MPG of both 70 and 71 (outer Index)**  **mpg\_70\_71 = avg\_mpg\_by\_cylinders.loc[[70, 71]]** | | |
| **Access MPG for a Specific Combination** | avg\_mpg\_by\_cylinders.loc[(70, 4)]  The outer and inner index is passed as tuble | |
| **Access All MPG Values for a Specific Year** | avg\_mpg\_by\_cylinders.loc[72] | |
| **Access All MPG Values for 4-Cylinder Cars Across All Years** | avg\_mpg\_by\_cylinders.xs(4, level='cylinders') | |
| **Filter for 6-Cylinder Cars with MPG > 18**  mpg\_6cyl\_high = avg\_mpg\_by\_cylinders.loc[avg\_mpg\_by\_cylinders.index.get\_level\_values('cylinders') == 6]  mpg\_6cyl\_high = mpg\_6cyl\_high[mpg\_6cyl\_high > 18] | | |

## EXAMPLE -2

|  |  |
| --- | --- |
| READING A CSV DATA IN DATAFRAME | import pandas as pd  df = pd.read\_csv("mpg.csv") |

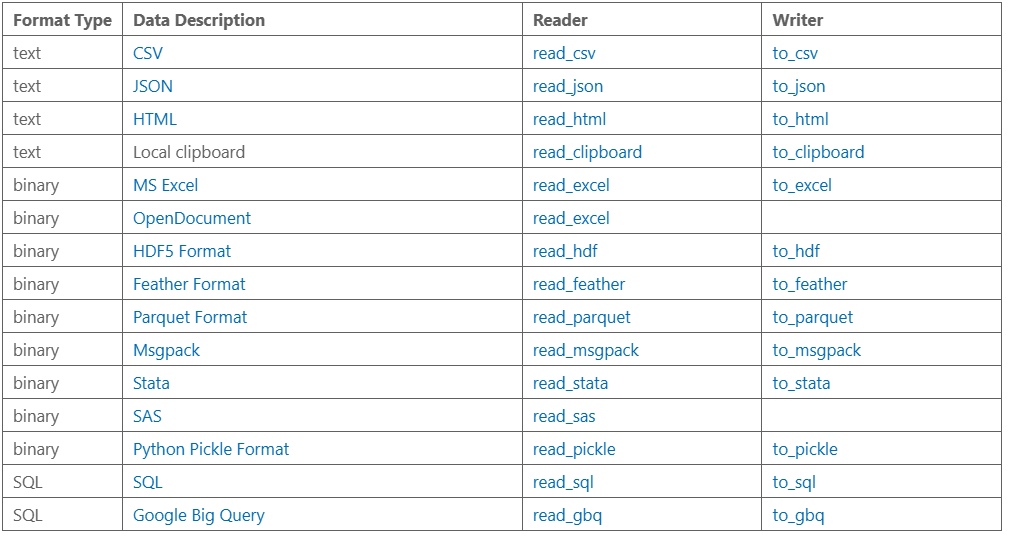


|  |  |  |
| --- | --- | --- |
| * Find average of mpg(miles per gallon) for each year(model\_year)   **avg\_mpg = df.groupby("model\_year")["mpg"].mean()** |  | |
| * **Grouping by Multiple Columns :** Group by “model\_year” 🡪 cylinders 🡪 Average of mpg   **avg\_mpg\_by\_cylinders = df.groupby(["model\_year", "cylinders"])["mpg"].mean()**  This will group the dataframe in following order   * It will first group by model\_year * The resulting data then group by cylinder * Finally in the resulting data average of mpg is calculated. | |  |

# COMBINING DATAFRAMES

# TEXT AND TIME METHODS

# INPUT AND OUTPUTS



Note : We use the pandas object to read data, but for writing, we use the DataFrame object.

## CSV

|  |  |
| --- | --- |
| **READ CSV (IN DATAFRAME)** | **df = pd.read\_csv("example.csv")** |
| **pd.read\_csv("example.csv", header=None)** | **header=None**: Tells Pandas **not to treat any row as column headers**. Instead, it will assign default integer column names: 0, 1, 2, ....  **When to Use header=None**   * When the CSV file **does not contain column headers**. * To **manually assign column names** later   **df1 = pd.read\_csv("example\_csv.csv", header=None)**  **df1.columns = ["a", "b", "c", "d"]** |

### WRITING TO CSV

* The **to\_csv()** method in Pandas is used to **export a DataFrame to a CSV file**

|  |  |
| --- | --- |
| **df.to\_csv("filename.csv")** | This saves the DataFrame df to a file named filename.csv in the current working directory. |
| **df.to\_csv("data.csv", columns=['Name', 'Score'])** | Save only selected columns |
| **df.to\_csv("data.tsv", sep='\t')** | Custom separator (e.g., tab-delimited) |
| **df.tocsv("data.csv", narep='NA')** | Save with missing values as 'NA' |
| **df.to\_csv("data.csv", encoding='utf-8', header=False)** | Save with UTF-8 encoding and no header |

#### INDEX WHILE SAVING

* By default, Pandas writes the index as the first column in the CSV. For instance, when saving the dataframe as a CSV,

|  |  |
| --- | --- |
| df = pd.DataFrame({"Name": ["Alice", "Bob"], "Score": [85, 90]})  print(df)  df.to\_csv("students.csv") | CSV will be saved with the index ,Name,Score  0,Alice,85  1,Bob,90 |
| SAVING FILE WITHOUT INDEX | df.to\_csv("students.csv", index=False) |

## HTML

## EXCEL

## SQL DATABASE