**Python-Pandas**

**Outline**

- **Installation of pandas**

- Importing pandas

- Importing the dataset

- Dataframe/Series

- **Creating series and Dataframes from scratch**

**-** Series.to\_frame()

- Indexing in series

- Slicing in series

- **Basic ops on a DataFrame**

- df.info()

- df.head()

- df.tail()

- df.shape()

- df.describe()

- **Basic ops on columns**

- Different ways of accessing cols

- Check for Unique values

- Rename column

- Deleting col

- Creating new cols

- **Basic ops on rows**

- df.index[]

- Adding a row

- Check for duplicates

- Deleting a row

**-****Data Modification**

**-** df.apply()

- isin()

-melt()

-pivot()

- **Feature Exploration**

**-** Extract days, year, week, month from datetime data

- loc/iloc

- **More in-built ops in pandas**

- sum()

- count()

- mean()

**-** **Sorting**

**-** value\_counts()

- sort\_values()

- **Index - Dataframe**

-set\_index()

-reset\_index()

- **Concatenation**

- pd.concat() - axis for concat - Inner join - Outer join

**- Merging dataframes**

- Concat v/s Merge - Outer join, Left join, Inner join

**-Group by**

-Aggregative functions

- **Data Save as csv**

**INTRODUCTION**

* Pandas is a powerful open-source data manipulation and analysis library for Python. It provides data structures and functions needed to manipulate structured data seamlessly.
* The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

**Key Features**

1. **Data Structures**
2. **Data Cleaning**
3. **Data Manipulation-** Data manipulation involves merging, concatenating, and reshaping data.
4. **File I/O**- Reading from and writing to various file formats
5. **Indexing and Selecting Data**:
6. **Time Series Analysis**:

## **Why Use Pandas?**

* Pandas allows us to analyze big data and make conclusions based on statistical theories.
* Pandas can clean messy data sets, and make them readable and relevant.
* Relevant data is very important in data science.

**Pandas Installation:**

**Pip install pandas**

**Importing pandas**:

Import pandas as pd

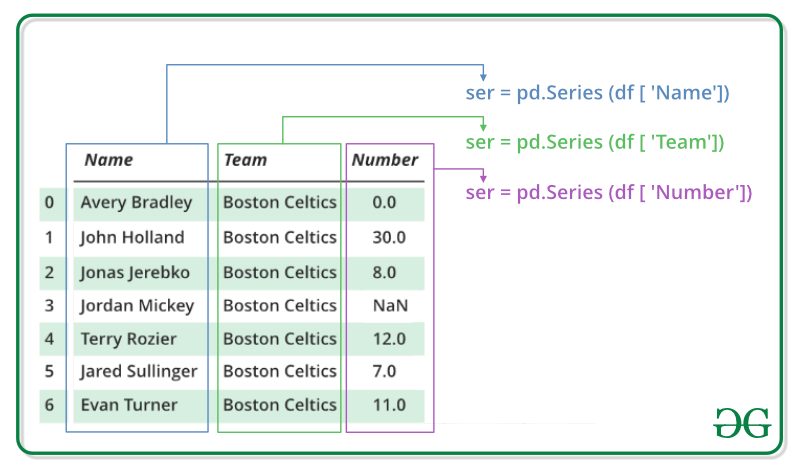
**Data Structures**:

Pandas provides two primary data structures: Series and DataFrame.

* **Series**: A one-dimensional labeled array capable of holding any data type.
* **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types.

**Series**

* A Series in pandas is a one-dimensional array-like object that can hold data of any type (integers, strings, floats, Python objects, etc.).
* A Pandas Series is like a column in a table.
* It is labeled, meaning each element has an associated label, which is often referred to as an index.
* **Integer and Label-Based Indexing:** Pandas supports both integer-based (positional) and label-based indexing. This flexibility allows for versatile data manipulation and access.
* **Labels Need Not Be Unique**: The index labels (like row labels) in a pandas DataFrame or Series do not need to be unique. This means you can have duplicate labels, although unique labels are often preferable for clarity and avoiding unexpected results.



**How to create a series**

1. **From a list:**

import pandas as pd

data = [1, 2, 3, 4, 5]

series = pd.Series(data)

print(series)

1. From array:

import numpy as np

data=np.array([1,2,3,4,5])

series=pd.Series(data)

print(series)

1. **From a dictionary**:

data = { 'a': 'Apple', 'b': 'Orenge', 'c': 'Strawberry', 'd': 'Kiwi'}

series = pd.Series(data)

print(series)

1. **From a scalar value:**

series = pd.Series(5, index=['a', 'b', 'c', 'd'])

print(series)

**Manually indexing**

* Indexing could mean selecting all the data, some of the data from particular columns.
* Indexing can also be known as Subset Selection.
* Indexing a Series using indexing operator []: Indexing operator is used to refer to the square brackets following an object.

# Original Series

data = [10, 20, 30, 40, 50]

series = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])

print("Original Series:") print(series)

**Reindexing to a new index**

new\_index = ['a', 'c', 'e', 'f', 'g']

reindexed\_series = series.reindex(new\_index)

print(reindexed\_series)

**Accessing Data in a Series**

There are two ways through which we can access element of series, they are:

* **Accessing Element from Series with Position**: In order to access the series element refers to the index number. Use the index operator [ ] to access an element in a series. The index must be an integer. In order to access multiple elements from a series, we use Slice operation.
* **Accessing Element Using Label (index):** In order to access an element from series, we have to set values by index label. A Series is like a fixed-size dictionary in that you can get and set values by index label.

By index:

print(series [0])

By label:

print(series ['a'])

Using slicing:

print(series [0:3])

print(series ['a' :'c' ])

**Operations on Series**

**Vectorized operations**:

series = pd.Series([1, 2, 3, 4])

print(series + 2) # Add 2 to each element

print(series \* 2) # Multiply each element by 2

series = pd.Series([2, 4, 6, 4])

print(series // 2) # Division on each element by 2

**Element-wise operations**:

series1 = pd.Series([1, 2, 3])

series2 = pd.Series([4, 5, 6])

print(series1 + series2) # Add corresponding elements

print(series1 - series2) # Substract corresponding elements

print(series1 \* series2) # Multiplication corresponding elements

**Applying functions**:

# series = pd.Series([1, 2, 3, 4])

# print(series.apply(lambda x: x \*\* 2)) # Square each element

**Binary Operation on Series**

* We can perform binary operation on series like addition, subtraction and many other operation. In order to perform binary operation on series we have to use some function like.add(),.sub() etc..

# creating a seriesdata = pd.Series([5, 2, 3,7], index=['a', 'b', 'c', 'd']) # creating a seriesdata1 = pd.Series([1, 6, 4, 9], index=['a', 'b', 'd', 'e'])

data.add(data1, fill\_value=0) #Add, fill\_value :Value to be replaced by 0 in series

data.sub(data1, fill\_value=0) #Subtract

**Converting Series to frame**

* The to\_frame() method in pandas is used to convert a Series into a DataFrame. This can be particularly useful when you want to perform operations that require a DataFrame or when you want to ensure that your data structure is a DataFrame for consistency or compatibility reasons.

# Sample Series

s = pd.Series([1, 2, 3, 4, 5], name='Numbers')

# Convert Series to DataFrame

df = s.to\_frame()

* **Joining/Merging with Another DataFrame**

# Sample Series

s = pd.Series([1, 2, 3, 4, 5], name='Numbers')

# Sample DataFrame data = { 'Letters': ['A', 'B', 'C', 'D', 'E'] }

df2 = pd.DataFrame(data)

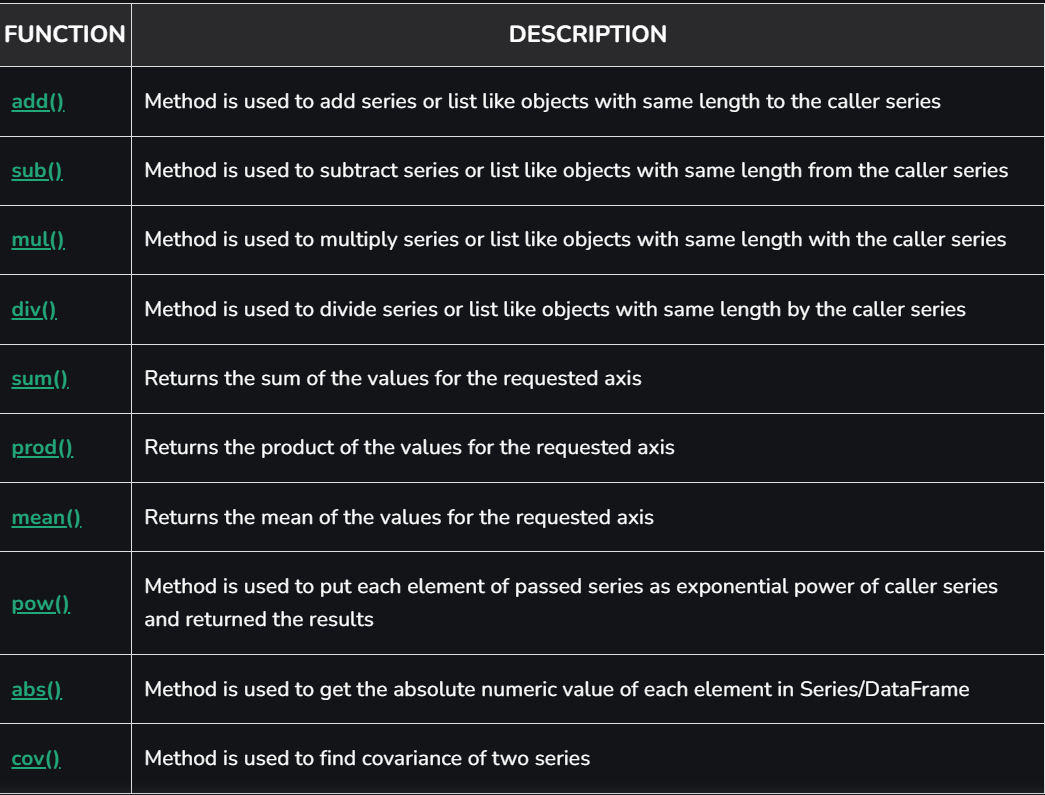
# Convert Series to DataFrame

df1 = s.to\_frame()

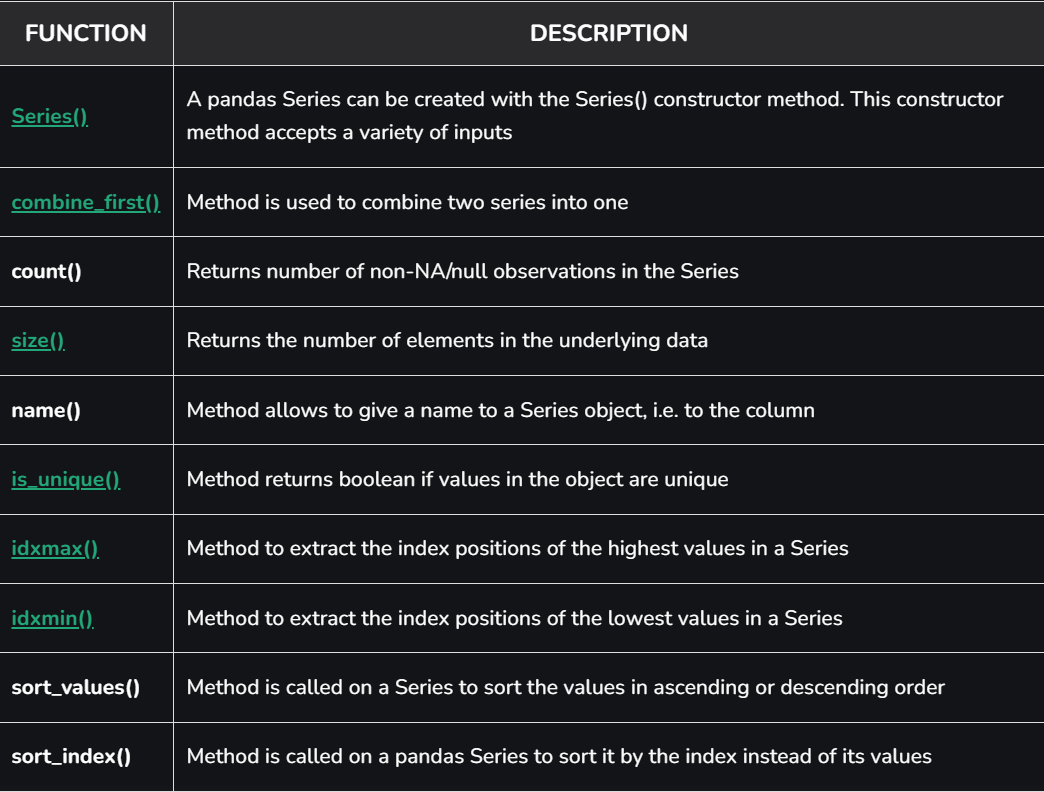
# Join DataFrame

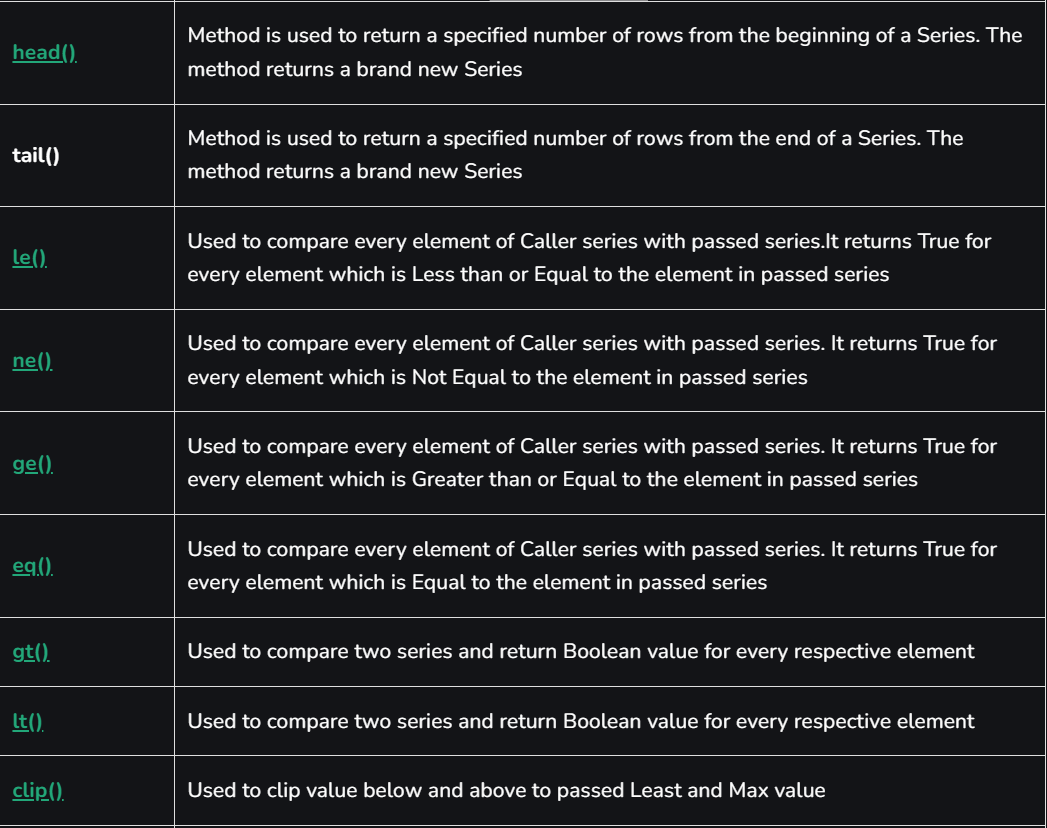
joined\_df = df1.join(df2)

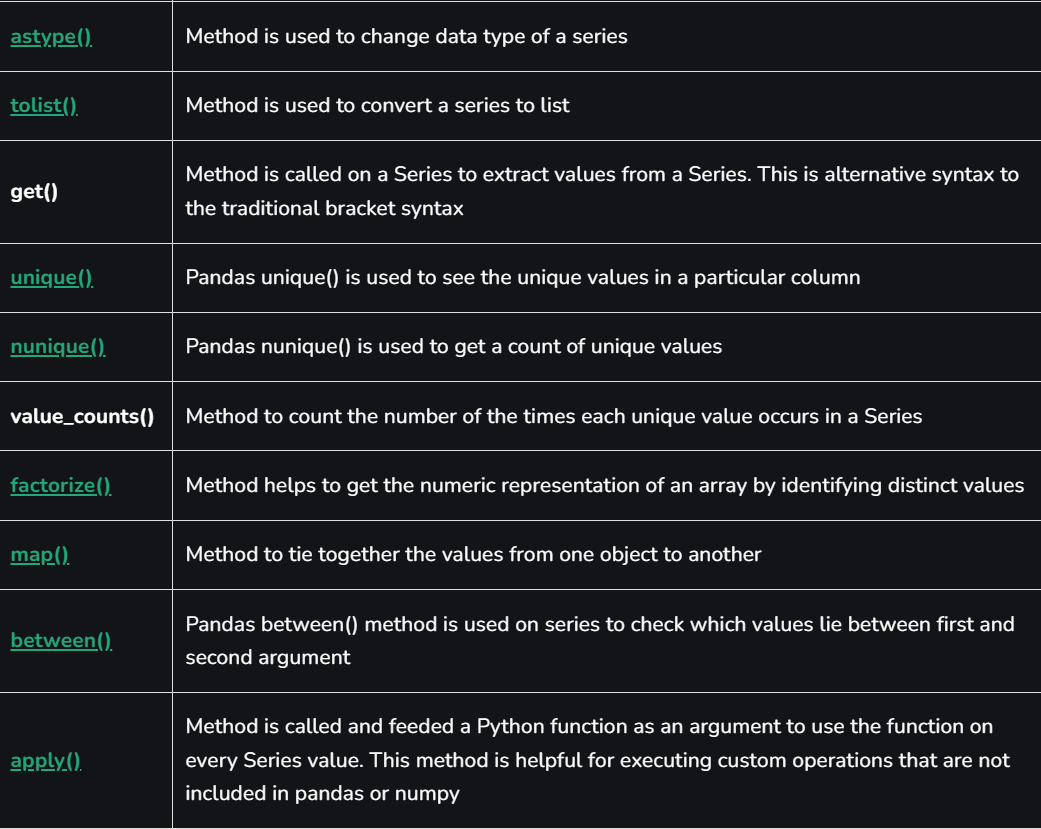
joined\_df



**Pandas Series methods**

****

****

****

**Handling Missing Values**

Detecting null values :

* series = pd.Series([1, 2, None, 4])

print(series.isna())

* series = pd.Series([1, 2, None, 4])

print(series.isnull())

**Detect missing values**

**Filling missing values**:

series = pd.Series([1, 2, None, 4])

print(series.fillna(0)) # Fill missing values with 0

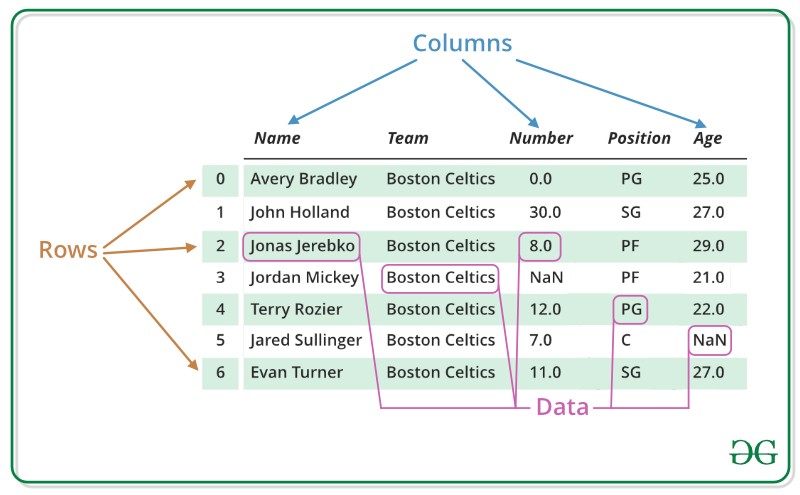
**Dropping missing values**:

series = pd.Series([1, 2, None, 4])

print(series.dropna())

**DataFrame**

* A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.
* Pandas DataFrame consists of three principal components, the data, rows, and columns.
* In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, json and Excel file.
* Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc.



**Creating DataFrame:**

**From a dictionary of lists**:

import pandas as pd

data = {

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

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

'City': ['New York', 'Los Angeles', 'Chicago']

}

df = pd.DataFrame(data)

print(df)

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **City** |
| **1** | Alice | **25** | New York |
| 2 | Bob | 30 | Los Angeles |
| **3** | Charlie | **35** | Chicago |

**From a list of dictionary**

data = [

{'Name': 'Alice', 'Age': 25, 'City': 'New York'},

{'Name': 'Bob', 'Age': 30, 'City': 'Los Angeles'},

{'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}

]

df = pd.DataFrame(data)

print(df)

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **City** |
| **1** | Alice | **25** | New York |
| **2** | Bob | **30** | Los Angeles |
| **3** | Charlie | **35** | Chicago |

**From a list of list**

data = [

['Alice', 25, 'New York'],

['Bob', 30, 'Los Angeles'],

['Charlie', 35, 'Chicago']

]

df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])

print(df)

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **City** |
| **1** | Alice | **25** | New York |
| **2** | Bob | **30** | Los Angeles |
| **3** | Charlie | **35** | Chicago |

**DataFrame common functions and keywords related to datatypes**

* 1. Df.index: The index (row labels) of the DataFrame.
  2. Df.columns: The column labels of the DataFrame.
  3. Df.dtypes: Return the dtypes in the DataFrame.
  4. Df.info() : Give information’s about a DataFrame including the index dtype and columns, non-null values and memory usage.
  5. Df.select\_dtypes: Return a subset of the DataFrame’s columns based on the column dtypes.
  6. Df.values or df.to\_numpy() : Return a Numpy representation of the DataFrame.
  7. Df.ndim: Return an int representing the number of axes / array dimensions.
  8. Df.size : Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

* 1. Df.shape : Return a tuple representing the dimensionality of the DataFrame.
  2. Df.head(): This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.
  3. Df.pop():Return item and drop from frame. Raise KeyError if not found.
  4. Df.tail(): Returns last n rows from the object based on position
  5. Df.astype() : Cast a pandas object to a specified dtype .
  6. Df.copy(): Make a copy of this object’s indices and data.
* When deep=True (default), a new object will be created with a copy of the calling object’s data and indices.
* Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).
* When deep=False, a new object will be created without copying the calling object’s data or index (only references to the data and index are copied).
* Any changes to the data of the original will be reflected in the shallow copy (and vice versa).
  1. Df.isin() : Used to filter a DataFrame based on whether each element is contained in a specified list of values.
* **DataFrame.isin(values)**
* **Checking if elements are in a list:** If the values are iterable (e.g., list, set, tuple), each element in the DataFrame is compared against the values in the iterable.
* **# Sample DataFrame**

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],

'Age': [24, 27, 22, 32, 29],

'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago']

}

# Filter rows where City is either 'New York' or 'Chicago'

filtered\_df = df[df['City'].isin(['New York', 'Chicago'])]

# Suppose you want to filter rows where the City is either 'New York' or 'Chicago' and the Age is greater than 25

filtered\_df = df[(df['City'].isin(['New York', 'Chicago'])) & (df['Age'] > 25)]

* **# Sample DataFrame**

data = {

'A': [1, 2, 3, 4],

'B': ['a', 'b', 'c', 'd']

}

df = pd.DataFrame(data)

# Check if elements are in the list [1, 3, 'a', 'd']

result = df.isin([1, 3, 'a', 'd'])

print(result)

O/P: A B

0 True True

1 False False

2 True False

3 False True

* **Using a dictionary to specify values for specific columns:** If values is a dict, the keys are column names, and the values are iterables, Series, or DataFrames to compare with the DataFrame's columns.

# Check if elements in column 'A' are in [1, 2] and elements in column 'B' are in ['a', 'c']

result = df.isin({'A': [1, 2], 'B': ['a', 'c']})

print(result)

O/P:

A B

0 True True

1 True False

2 False True

3 False False

* **Using a DataFrame to specify values:** If values is a DataFrame, it's aligned with the DataFrame both on the index and columns.

Df\_2 = pd.DataFrame({

'A': [1, 2, 3, 4],

'B': ['a', 'b', 'c', 'd']

})

# Check if elements are in the DataFrame

result = df.isin(Df\_2)

print(result)

O/P

A B

0 True True

1 True True

2 True True

3 True True

**Practical Scenario: Filtering Sales Data**

Imagine you have a sales dataset for an e-commerce company, and you want to filter out rows where the sales category is either 'Electronics' or 'Clothing' and where the sales are made by specific sales representatives.

import pandas as pd

# Sample DataFrame

data = {

'OrderID': [1001, 1002, 1003, 1004, 1005, 1006],

'SalesRep': ['John', 'Alice', 'John', 'Bob', 'Alice', 'Bob'],

'Category': ['Electronics', 'Clothing', 'Furniture', 'Electronics', 'Clothing', 'Furniture'],

'Amount': [1500, 1200, 800, 900, 1300, 750]

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

**Task: Filter Sales Data**

1. Filter the rows where the Category is either 'Electronics' or 'Clothing'.
2. Further filter the rows where the SalesRep is either 'John' or 'Alice'.

# Define the categories and sales representatives of interest

categories = ['Electronics', 'Clothing']

sales\_reps = ['John', 'Alice']

# Apply the filtering using isin

filtered\_df = df[df['Category'].isin(categories) & df['SalesRep'].isin(sales\_reps)]

print("\n Filtered DataFrame:")

print(filtered\_df)

**O/P**

OrderID SalesRep Category Amount

0 1001 John Electronics 1500

1 1002 Alice Clothing 1200

4 1005 Alice Clothing 1300

* 1. Data.apply()
* The apply() function in pandas is a powerful tool that allows you to apply a function along an axis of the DataFrame or Series.
* It is commonly used for data transformation and aggregation.

# Sample DataFrame

data = {

'A': [1, 2, 3, 4],

'B': [10, 20, 30, 40],

'C': [100, 200, 300, 400]

}

df = pd.DataFrame(data)

# Function to apply

def multiply\_by\_two(x):

return x \* 2

# Apply function to each column

df\_applied = df.apply(multiply\_by\_two)

print(df)

print(df\_applied)

OR

# Apply lambda function to each column

df\_applied = df.apply(lambda x: x \* 2)

**Dataset for melt and pivot:**

**import pandas as pd**

**data = {**

**'Region': ['North', 'North', 'North', 'North', 'North',**

**'South', 'South', 'South', 'South', 'South',**

**'East', 'East', 'East', 'East', 'East',**

**'West', 'West', 'West', 'West', 'West',**

**'North', 'North', 'North', 'North', 'North',**

**'South', 'South', 'South', 'South', 'South',**

**'North', 'North', 'North', 'North', 'North',**

**'South', 'South', 'South', 'South', 'South',**

**'East', 'East', 'East', 'East', 'East',**

**'West', 'West', 'West', 'West', 'West'],**

**'Product': ['A', 'B', 'C', 'D', 'E', 'A', 'B', 'C', 'D', 'E',**

**'A', 'B', 'C', 'D', 'E',**

**'F', 'G', 'H', 'I', 'J',**

**'F', 'G', 'H', 'I', 'J',**

**'F', 'G', 'H', 'I', 'J',**

**'K', 'L', 'M', 'N', 'O', 'K', 'L', 'M', 'N', 'O', 'K', 'L', 'M', 'N', 'O',**

**'K', 'L', 'M', 'N', 'O'],**

**'Q1\_Sales': [150, 120, 130, 140, 160, 180, 140, 150, 160, 170,**

**170, 130, 140, 150, 160, 160, 110, 120, 130, 140,**

**110, 120, 130, 140, 150, 120, 130, 140, 150, 160,**

**110, 120, 130, 140, 150, 120, 130, 140, 150, 160,**

**170, 180, 190, 200, 210, 170, 180, 190, 200, 210,**

**170, 180, 190, 200, 210, 170, 180, 190, 200, 210],**

**'Q2\_Sales': [200, 160, 180, 190, 210, 230, 190, 200, 210, 220,**

**220, 180, 190, 200, 210, 210, 150, 160, 180, 190,**

**160, 170, 180, 190, 200, 170, 180, 190, 200, 210,**

**160, 170, 180, 190, 200, 170, 180, 190, 200, 210,**

**220, 230, 240, 250, 260, 220, 230, 240, 250, 260,**

**220, 230, 240, 250, 260, 220, 230, 240, 250, 260],**

**'Q3\_Sales': [250, 210, 230, 240, 260, 280, 240, 250, 260, 270,**

**270, 230, 240, 250, 260, 260, 200, 210, 230, 240,**

**210, 220, 230, 240, 250, 220, 230, 240, 250, 260,**

**210, 220, 230, 240, 250, 220, 230, 240, 250, 260,**

**270, 280, 290, 300, 310, 270, 280, 290, 300, 310,**

**270, 280, 290, 300, 310, 270, 280, 290, 300, 310],**

**'Q4\_Sales': [300, 260, 280, 290, 310, 330, 290, 300, 310, 320,**

**320, 280, 290, 300, 310, 310, 250, 260, 280, 290,**

**260, 270, 280, 290, 300, 270, 280, 290, 300, 310,**

**260, 270, 280, 290, 300, 270, 280, 290, 300, 310,**

**320, 330, 340, 350, 360, 320, 330, 340, 350, 360,**

**320, 330, 340, 350, 360, 320, 330, 340, 350, 360]**

**}**

**df = pd.DataFrame(data)**

**print(df)**

* 1. Pd.melt()
* The melt function is used to unpivot the DataFrame from wide format to long format.
* The melt function is used to unpivot a DataFrame from wide format to long format.
* This is especially useful when you have multiple columns representing different measurements or variables, and you want to transform them into a single column with corresponding values.

Syntax:

(data, id\_vars,var\_name,value\_name) – [weather.csv]

* Pass in the DataFrame
* Pass in the -column names that we DON'T want to change
* Pass in the -new column name that will store labels of columns to melt
* From `day1`, `day2`... through to `day31` into a single column`day`
* Finally pass in -new column name that will have values from columns we are melting.

• weather\_melt = pd.melt(weather, id\_vars=['year', 'month', 'element'], var\_name="day", value\_name="temp")

 **frame**: The DataFrame to unpivot.

 **id\_vars**: Columns to use as identifier variables.

 **value\_vars**: Columns to unpivot. If not specified, all columns not set as id\_vars will be used.

 **var\_name**: Name to use for the 'variable' column.

 **value\_name**: Name to use for the 'value' column.

 **col\_level**: If columns are a MultiIndex, the level to melt.

 **ignore\_index**: If True, the original index is ignored.

* + 1. **Pivot ()**
* \*   The pivot function in Pandas is used to reshape or transform data by pivoting the values of one column into multiple columns.
* \*   pivot is used to transform long-form data to wide-form
* \* It's particularly useful when you have long-form data (e.g., data in a tidy (a standard way of organizing data ) format with rows for each observation) and want to convert it into wide-form data with a column for each unique value in another column.

Syntax: DataFrame.pivot(index=None, columns=None, values=None)

* + - We want to split one single column into multiple columns
    - Pass in the column names that we DON'T want to change
    - Pass in the column name that we want to split -Labels of new columns will come from the column we want to split
    - Pass in the column from which we will get our values
    - The pivot function is used to pivot a DataFrame from long format to wide format
    - The pivot function is used to reshape data (produce a “pivot” table) based on column values. It uses unique values from specified columns to form axes of the resulting DataFrame

 **index**: Column to use to make new frame’s index. If None, uses existing index.

 **columns**: Column to use to make new frame’s columns.

 **values**: Column(s) to use for populating new frame’s values. If not specified, all remaining columns will be used.

weather\_tidy = weather\_melt.pivot(index=['year', 'month', 'day'], columns = 'element', values='temp') weather\_tidy

* When you use the pivot function and reset the index, the column headers that were created by the pivot function (e.g., Q1\_Sales, Q2\_Sales, etc.) become part of a multi-level (hierarchical) index.
* To address this and ensure that the Quarter levels are included as regular column headers, you can use reset\_index and then properly handle the resulting DataFrame.
  + 1. Data.set\_index()

The set\_index function in pandas is used to set one or more columns of a DataFrame as its index. This allows for more intuitive data handling and access, especially when dealing with hierarchical or multi-level indices.

DataFrame.set\_index(keys, drop=True, append=False, inplace=False, verify\_integrity=False)

 **keys**: Column label or list of column labels to set as the index.

 **drop**: Boolean, default True. Delete columns to be used as the new index.

 **append**: Boolean, default False. Whether to append columns to existing index.

 **inplace**: Boolean, default False. Whether to modify the DataFrame in place (without creating a new DataFrame).

 **verify\_integrity**: Boolean, default False. Check the new index for duplicates.

* + 1. Data.reset\_index()

The reset\_index function in pandas is used to reset the index of a DataFrame. This function can convert the index into a column, and it can also remove the current index and replace it with the default integer index.

Syntax:

DataFrame.reset\_index(level=None, drop=False, inplace=False, col\_level=0, col\_fill='')

 **level**: Specific levels to reset (for MultiIndex).

 **drop**: Whether to drop the index or reset it as a column.

 **inplace**: Whether to modify the DataFrame in place.

 **col\_level**: If columns are a MultiIndex, determine which level to place the labels in.

 **col\_fill**: If columns are a MultiIndex, determine how to fill other levels.

21.sort\_values()

The sort\_values function in pandas is used to sort a DataFrame or a Series by one or more columns or index labels. This function can sort data in ascending or descending order.

Syntax:

**DataFrame.sort\_values(by,axis=0, ascending=True, inplace=False, kind='quicksort', na\_position='last', ignore\_index=False, key=None)**

 **by**: Column name(s) to sort by.

 **axis**: Axis to sort along (0 for index, 1 for columns).

 **ascending**: Whether to sort in ascending order (default is True).

 **inplace**: If True, perform operation in-place.

 **kind**: Sorting algorithm ('quicksort', 'mergesort', 'heapsort', 'stable').

 **na\_position**: Position of NaNs ('first' or 'last').

 **ignore\_index**: If True, the resulting index will be labeled 0, 1, 2, etc.

 **key**: Function to apply to each element before sorting.

22.Date\_range()

The pd.date\_range() function in pandas is used to generate a sequence of dates. It creates a DatetimeIndex with a fixed frequency, which is very useful for creating time series data.

**Parameters**

* **start**: The starting date of the sequence.
* **end**: The ending date of the sequence.
* **periods**: The number of periods (timestamps) to generate.
* **freq**: The frequency of the date range (e.g., daily, monthly, etc.). Default is 'D' for daily frequency.
* **tz**: The time zone for the resulting DatetimeIndex.
* **normalize**: If True, normalizes the start and end dates to midnight.
* **name**: Name for the resulting DatetimeIndex.
  1. Generate a Daily Date Range

# Generate a range of dates from 2024-01-01 to 2024-01-10

date\_range = pd.date\_range(start='2024-01-01', end='2024-01-10')

print(date\_range)

* 1. Generate a Date Range with a Specific Number of Periods

# Generate 10 daily dates starting from 2024-01-01

date\_range = pd.date\_range(start='2024-01-01', periods=10)

print(date\_range)

* 1. Generate a Monthly Date Range

# Generate a range of dates from January 2024 to December 2024 with monthly frequency

date\_range = pd.date\_range(start='2024-01-01', end='2024-12-31', freq='M')

print(date\_range)

* 1. Generate a Date Range with Hourly Frequency

# Generate hourly dates from 2024-01-01 00:00 to 2024-01-02 00:00

date\_range = pd.date\_range(start='2024-01-01', end='2024-01-02', freq='H')

print(date\_range)

**Accessing Data in a DataFrame**

**1. Selecting a column**:

#1\_st\_column : print(df['Name'])

* 1. **Selecting Multiple columns:**

print(df[['Name', 'City']])

* 1. **Selecting row by index :**

print(df.loc[0]) # Select row by label (feature)

print(df.iloc[0]) # Select row by position

* loc is label-based, meaning that you have to specify the name of the rows and columns that you want to filter out.
* iloc is integer position-based, so you have to specify rows and columns by their integer index.

import pandas as pd

data = {

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

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

'City': ['New York', 'Los Angeles', 'Chicago']

}

df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Select row 'a' and all columns

print(df.loc['a'])

# Select rows 'a' and 'b', and columns 'Name' and 'City'

print(df.loc[['a', 'b'], ['Name', 'City']])

# Select all rows and column 'Age'

print(df.loc[:, 'Age'])

* Selecting rows and columns by integer position

import pandas as pd

data = {

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

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

'City': ['New York', 'Los Angeles', 'Chicago']

}

df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Select the first row (index 0) and all columns

print(df.iloc[0])

# Select the first and second rows (indices 0 and 1), and the first and third columns (indices 0 and 2)

print(df.iloc[0:2, [0, 2]])

# Select all rows and the second column (index 1)

print(df.iloc[:, 1])

* use loc when you want to select data based on labels, and use iloc when you want to select data based on integer positions.

**Create a feature that calculates the price of the product after applying the discount.**

1. **Adding a new column**: Create a new column in an existing dataframe

df ['Country'] = ['USA', 'USA', 'USA']

print(df)

1. **Adding a row**: The append method adds a new row to the DataFrame and returns a new DataFrame.

DataFrame.append(other, ignore\_index=False, verify\_integrity=False, sort=False)

 **other**: DataFrame or Series/dict-like object to append.

 **ignore\_index**: Boolean, default False. If True, do not use the index labels.

 **verify\_integrity**: Boolean, default False. If True, raise ValueError on creating index with duplicates.

 **sort**: Boolean, default False. Sort columns if the columns of self and other are not aligned.

new\_row= {‘Name’:’Arathi’,’Age’:23,’ City’:’New york’}

# Add the new row using append

df = df.append(new\_row, ignore\_index=True)

# Add the new row using loc

df.loc[len(df)] = new\_row

1. **Checking Duplicate**:

DataFrame.duplicated(subset=None, keep='first')

 **subset**: Column label or list of column labels to consider for identifying duplicates. Default is None, which considers all columns.

 **keep**: {‘first’, ‘last’, False}, default ‘first’. Determines which duplicates to mark as True:

* ‘first’: Mark duplicates as True except for the first occurrence.
* ‘last’: Mark duplicates as True except for the last occurrence.
* False: Mark all duplicates as True.
* # Sample DataFrame

data = { 'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'Eve', 'Alice'],

'Age': [24, 27, 22, 24, 29, 24],

'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago', 'New York'] }

df = pd.DataFrame(data)

# Identify duplicate rows

duplicates = df.duplicated()

# Identify duplicate rows based on specific columns

duplicates\_subset = df.duplicated(subset=['Name', 'City'])

* **Dropping Duplicate Rows with drop\_duplicates**

**DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False)**

 **subset**: Column label or list of column labels to consider for identifying duplicates. Default is None, which considers all columns.

 **keep**: {‘first’, ‘last’, False}, default ‘first’. Determines which duplicates to keep:

* ‘first’: Keep the first occurrence and remove other duplicates.
* ‘last’: Keep the last occurrence and remove other duplicates.
* False: Remove all duplicates.

 **inplace**: Boolean, default False. If True, do operation in place and return None.

# Remove duplicate rows

df\_no\_duplicates = df.drop\_duplicates()

* **Check Column-Specific Duplicates**: Apply duplicated() to a specific column to identify duplicates within that column.

# Check for duplicate values in the 'Name'column

duplicates\_Name = df['Name'].duplicated()

# Remove duplicate rows based on specific columns

df\_no\_duplicates\_subset = df.drop\_duplicates(subset=['Name', 'City'])

# Remove all duplicate rows (keep=False)

df\_no\_duplicates\_all = df.drop\_duplicates(keep=False)

**Summary**

* **Check for Duplicates**: Use DataFrame.duplicated() to identify duplicate rows.
* **Drop Duplicates**: Use DataFrame.drop\_duplicates() to remove duplicate rows.
* **Check Column-Specific Duplicates**: Apply duplicated() to a specific column to identify duplicates within that column.

1. **Modifying an existing column**

df['Age'] = df['Age'] + 1

print(df)

1. **Deleting a column**:

df.drop('Country', axis=1, inplace=True)

print(df)

1. **Renaming columns**:

df.rename(columns={'Name': 'Full Name', 'Age': 'Age (years)'}, inplace=True)

print(df)

1. **Extracting Date and Time Components**

* Extracting date and time components from datetime objects in pandas is a common task in data analysis.
* you can extract different components such as year, month, day, hour, minute, second, etc.

# Sample DataFrame with datetime column

data = { 'Timestamp': ['2024-07-24 14:45:00', '2024-07-25 16:30:00',

'2024-07-26 08:20:00', '2024-07-27 12:15:00'] }

df = pd.DataFrame(data)

# Convert the Timestamp column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

* Extracting Year, Month, and Day

# Extract year

df['Year'] = df['Timestamp'].dt.year

# Extract month

df['Month'] = df['Timestamp'].dt.month

# Extract day

df['Day'] = df['Timestamp'].dt.day

* Extracting Hour, Minute, and Second

# Extract hour

df['Hour'] = df['Timestamp'].dt.hour

# Extract minute

df['Minute'] = df['Timestamp'].dt.minute

# Extract second

df['Second'] = df['Timestamp'].dt.second

* Extracting Day of the Week and Day of the Year

# Extract day of the week (Monday=0, Sunday=6)

df['DayOfWeek'] = df['Timestamp'].dt.dayofweek

# Extract day of the year

df['DayOfYear'] = df['Timestamp'].dt.dayofyear

* Extracting Week and Quarter:

# Extract week number of the year:

df['Week'] = df['Timestamp'].dt.isocalendar().week

The isocalendar() method returns the ISO year, week number, and weekday.

# Extract quarter

df['Quarter'] = df['Timestamp'].dt.quarter

print(df)

**Data Cleaning**

Data cleaning involves handling missing data, correcting inconsistencies, and transforming data into the right format.

### Handling Missing Data

1. **Detecting missing values**:

print(df.isna())

print(df.isnull())

1. **Filling missing values**:

df.fillna(0, inplace=True)

print(df)

* Pandas provides various methods to fill missing values in a DataFrame

### Filling with a Specific Value: You can fill missing values with a specific constant value.

### Forward Fill (ffill): Forward fill propagates the last valid observation forward to the next valid.

### df\_filled = df.fillna(method=’ffill’)

### Backward Fill (bfill): Backward fill uses the next valid observation to fill the gap.

### df\_filled = df.fillna(method=’bfill’)

### Filling with Mean, Median, or Mode: You can fill missing values with the mean, median, or mode of the column.

df\_filled = df.copy()

df\_filled['A'] = df['A'].fillna(df['A'].mean())

df\_filled['B'] = df['B'].fillna(df['B'].median())

print(df\_filled)

1. **Dropping rows or columns with missing values**:

df.dropna(inplace=True)

print(df)

### Data Aggregation and Grouping [Movies &Director]

### Pandas Dataframe .groupby Method - Naukri Code 360

### **Grouping data**:

The groupby function in pandas is used to split the data into groups based on some criteria, apply a function to each group independently, and then combine the results. This is often summarized by the "split-apply-combine" pattern.

 **by**: Mapping, function, label, or list of labels to group by.

 **axis**: {0 or ‘index’, 1 or ‘columns’}, default 0. The axis to group by.

 **level**: If the axis is a MultiIndex (hierarchical), group by a particular level or levels.

 **as\_index**: Boolean, default True. Return group labels as the index.

 **sort**: Boolean, default True. Sort group keys.

 **group\_keys**: Boolean, default True. Add group keys to group axis.

 **squeeze**: Reduce the dimensionality of the return type if possible.

 **observed**: Boolean, default False. Apply only to categorical data.

 **dropna**: Boolean, default True. Do not include the “NA” group.

grouped = df.groupby('City')

print(grouped.mean())

1. **Aggregating data**:

Aggregation involves applying one or more operations to each group. Common aggregation functions include mean, sum, count, min, and max.

print(df.groupby('City').agg({'Age': ['mean', 'max']}))

**Applying Multiple Aggregations to a Single Column**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward', 'Frank', 'Grace', 'Helen'],

'Age': [24, 27, 22, 32, 29, 24, 27, 22],

'City': ['New York', 'Los Angeles', 'New York', 'Chicago', 'Chicago', 'New York', 'Chicago', 'Los Angeles'],

'Salary': [70000, 80000, 120000, 50000, 70000, 90000, 60000, 110000]

}

df = pd.DataFrame(data)

# Group by 'City' and apply multiple aggregations to 'Salary'

grouped = df.groupby('City')['Salary'].agg(['mean', 'max', 'min', 'sum'])

print(grouped)

**Applying Different Aggregations to Different Columns**

You can apply different aggregation functions to different columns by passing a dictionary to the agg method. The keys of the dictionary specify the columns, and the values specify the list of functions to apply.

# Group by 'City' and apply different aggregations to 'Age' and 'Salary'

aggregation\_functions = {

'Age': ['mean', 'std'],

'Salary': ['min', 'max', 'mean', 'sum']

}

grouped = df.groupby('City').agg(aggregation\_functions)

print(grouped)

### Common GroupBy Operations

# Sample data creation

data = {

'Region': ['North', 'South', 'East', 'West', 'North'],

'Product': ['A', 'B', 'C', 'D', 'E'],

'Month': ['January', 'February', 'March', 'April', 'May'],

'Sales': [100, 170, 240, 310, 380]

}

df = pd.DataFrame(data)

#### 1. **Sum**

To get the sum of values for each group.

grouped\_sum = df.groupby('Region')['Sales'].sum()

#### 2. **Mean**

To get the average value for each group.

grouped\_mean = df.groupby('Region')['Sales'].mean()

#### 3. **Count**

To get the count of values for each group.

grouped\_count = df.groupby('Region')['Sales'].count()

#### 4. **Max**

To get the maximum value for each group.

grouped\_max = df.groupby('Region')['Sales'].max()

#### 5. **Min**

To get the minimum value for each group.

grouped\_min = df.groupby('Region')['Sales'].min()

**Concatenation**

Concatenation in pandas is the process of combining two or more DataFrames along a particular axis (either rows or columns). The pd.concat() function is typically used for this purpose.

#### Key Features

* **Axis**: You can specify the axis along which to concatenate (axis=0 for rows, axis=1 for columns).
* **Keys**: You can create a hierarchical index using the keys parameter.
* **Join Type**: You can specify the type of join (inner or outer) for concatenation.
* **Ignore Index**: You can ignore the existing index and create a new one using the ignore\_index parameter.

**Inner Join with concat()**

An inner join concatenates only the data that is present in both DataFrames, resulting in the intersection of the DataFrames.

# Sample DataFrames

df1 = pd.DataFrame({

'A': ['A0', 'A1', 'A2'],

'B': ['B0', 'B1', 'B2'] })

df2 = pd.DataFrame({

'B': ['B1', 'B2', 'B3'],

'C': ['C1', 'C2', 'C3']

})

# Concatenate DataFrames with inner join

df\_inner = pd.concat([df1, df2], join='inner')

### Outer Join with concat()

An outer join concatenates all data from both DataFrames, filling in missing values with NaN for indices or columns that do not match.

# Sample DataFrames

df1 = pd.DataFrame({

'A': ['A0', 'A1', 'A2'],

'B': ['B0', 'B1', 'B2']

})

df2 = pd.DataFrame({

'B': ['B1', 'B2', 'B3'],

'C': ['C1', 'C2', 'C3']

})

# Concatenate DataFrames with outer join

df\_outer = pd.concat([df1, df2], join='outer')

**Summary**

* **Inner Join with concat()**: Use pd.concat([df1, df2], join='inner') to concatenate only the common parts of the DataFrames.
* **Outer Join with concat()**: Use pd.concat([df1, df2], join='outer') to concatenate all parts of the DataFrames, filling missing values with NaN.

### 1. Concatenating DataFrames Vertically (Along Rows)

When concatenating along rows, the DataFrames are stacked on top of each other. The number of columns should ideally be the same, but if they are not, missing values will be filled with NaN.

# Sample DataFrames

df1 = pd.DataFrame({

'A': ['A0', 'A1', 'A2', 'A3'],

'B': ['B0', 'B1', 'B2', 'B3'] })

df2 = pd.DataFrame({

'A': ['A4', 'A5', 'A6', 'A7'],

'B': ['B4', 'B5', 'B6', 'B7']})

# Concatenate DataFrames vertically

df\_vertical = pd.concat([df1, df2])

### 2. Concatenating DataFrames Horizontally (Along Columns)

When concatenating along columns, the DataFrames are merged side by side. The number of rows should ideally be the same, but if they are not, missing values will be filled with NaN.

import pandas as pd

# Sample DataFrames

df1 = pd.DataFrame({

'A': ['A0', 'A1', 'A2', 'A3'],

'B': ['B0', 'B1', 'B2', 'B3']

})

df2 = pd.DataFrame({

'C': ['C0', 'C1', 'C2', 'C3'],

'D': ['D0', 'D1', 'D2', 'D3']

})

# Concatenate DataFrames horizontally

df\_horizontal = pd.concat([df1, df2], axis=1)

**Summary**

* **Vertical Concatenation**: Use pd.concat([df1, df2]) to stack DataFrames on top of each other.
* **Horizontal Concatenation**: Use pd.concat([df1, df2], axis=1) to merge DataFrames side by side.

### merge()

The merge() function is used to join DataFrames based on common columns or indices. It is similar to SQL joins.

#### Key Features

* **Join Type**: You can specify the type of join (inner, outer, left, right).
* **Keys**: You specify the columns or indices to join on using the on, left\_on, and right\_on parameters.
* **Suffixes**: You can specify suffixes for overlapping columns using the suffixes parameter.
* **Validation**: You can validate the merge with the validate parameter (e.g., 'one\_to\_one', 'one\_to\_many').

**Inner join:**

# Sample DataFrames

df1 = pd.DataFrame({

'Key': ['A', 'B', 'C'],

'Value1': [1, 2, 3]

})

df2 = pd.DataFrame({

'Key': ['B', 'C', 'D'],

'Value2': [4, 5, 6]

})

# Perform inner join

df\_inner = pd.merge(df1, df2, on='Key', how='inner')

**Outer join :**

**# Sample DataFrames**

df1 = pd.DataFrame({

'Key': ['A', 'B', 'C'],

'Value1': [1, 2, 3]

})

df2 = pd.DataFrame({

'Key': ['B', 'C', 'D'],

'Value2': [4, 5, 6]

})

# Perform outer join

df\_outer = pd.merge(df1, df2, on='Key', how='outer')

**Difference B/w Merge and Concat**

* The concat() and merge() functions in pandas are used for combining DataFrames, but they serve different purposes and have different functionalities.

 **Functionality**:

* concat(): Focuses on appending or concatenating DataFrames. You can concatenate vertically (stacking rows) or horizontally (stacking columns).
* merge(): Focuses on combining DataFrames based on key columns or indices, with more complex joining logic (inner, outer, left, right joins).

 **Parameters**:

* concat(): Parameters include axis, keys, join, ignore\_index, etc.
* merge(): Parameters include on, left\_on, right\_on, how, suffixes, validate, etc.

Use concat() when you want to stack DataFrames, and use merge() when you want to perform database-like join operations on DataFrames.

**Saving a DataFrame to a CSV file**

* Save DataFrame to a CSV file

df.to\_csv('output.csv', index=False)

* The index=False parameter is used to prevent pandas from writing row indices to the file.

### Specifying Columns to Save

### You can specify which columns to include in the CSV file using the columns parameter:

### df.to\_csv('output\_selected\_columns.csv', columns=['Name', 'City'], index=False)

**Summary**

* **Basic Saving**: df.to\_csv('filename.csv', index=False)
* **Include Index**: df.to\_csv('filename.csv')
* **Different Separator**: df.to\_csv('filename.csv', sep=';')
* **Specify Columns**: df.to\_csv('filename.csv', columns=['col1', 'col2'])
* **Handle Missing Values**: df.to\_csv('filename.csv', na\_rep='NA')
* **No Header**: df.to\_csv('filename.csv', header=False)

**- Nayana J Nair**