**PANDAS**

**What is Pandas?**

* **Pandas is a fast, powerful, flexible, and easy to use open-source data analysis and manipulation tool, built on top of the Python programming language.**
* **about pandas:** [**https://pandas.pydata.org/about/index.html**](https://pandas.pydata.org/about/index.html)

**Use of Pandas:**

* **Data Cleaning and Preparation:**

**Pandas provides tools for handling missing data, data alignment, transformation, and data normalization.**

* **Time Series Analysis:**

**Pandas offers specialized functionality for working with time series data.**

**Pandas integrate well with visualization libraries like Matplotlib and Seaborn which are used for Data Visualization.**

**Basic data structure in pandas**

* **Pandas provide two types of classes for handling data:**

**~ Series  
 ~ DataFrame**

**Importing Pandas**

**There are different way to import pandas library:**

**1.Most common way is import statement with alias:**

**~ import pandas as pd**

**2.Importing all the function and class:**

**~ from pandas import \***

**3.Importing the specific function and class library:**

**~ from pandas import DataFrame, read\_csv**

**Pandas Series**

* **A Pandas Series is like a column in a table.**
* **It is a one-dimenational labelled array capable of holding of any datatype.**
* **Integers, string, floating point numbers, python object etc.**
* **Each value in a pandas series is associated with the index.**
* **The default index value of it is from 0 to number – 1, or you can specify your own index values.**

**Note:**

* **‘Series’ is a class provided by the panda’s library.**
* **When you create a ‘Series’ in your code, means your creating an ‘Series’ object of that ‘Series’ class.**
* **Syntax: pd.Series( data, index, name )**

**Parameteres:**

|  |  |
| --- | --- |
| **data:** | **It can be array, list, dictionary, csv file or excel file** |
| **index:** | **Default, also create custom index** |
| **name:** | **Series name** |

**Creating a pandas series**

* **Pandas series can be created from the lists, dictionary and from other scalar values etc.**
* **Series can be created in different ways, here are some ways to create a series.**

**Create an empty series:**

**Code:**

**import pandas as pd**

**empty\_series = pd.Series()**

**print(empty\_series)**

**Output:**

**Series([ ], dtype: float64)**

**Create a series from lists:**

**We first creating a list after that we can create series of list.**

**# String**

**Code:**

**import pandas as pd**

**import numpy as np**

**countries = [ 'India' , 'Nepal' , 'Srilanka' , 'Bhutan' ]**

**print(pd.Series(countries))**

**Output:**

**0 India**

**1 Nepal**

**2 Srilanka**

**3 Bhutan**

**dtype: object**

**# Custom index and name**

**Code:**

**marks = [ 58, 93, 89, 60 ]**

**subjects = [ 'C++' , 'Python', 'R', 'Java' ]**

**print(pd.Series(marks, index=subjects, name='student'))**

**Output:**

**C++ 58**

**Python 93**

**R 89**

**Java 60**

**Name: student, dtype: int64**

**Create a series from dictionary:**

**We must first create a dictionary then we can pefrom series on dictionary.**

**Code:**

**import pandas as pd**

**marks = { 'maths' : 78, 'english' : 70, 'science' : 89 }**

**pd.Series(marks, name='student score')**

**Output:**

**maths 78**

**english 70**

**science 89**

**Name: student score, dtype: int64**

**Pandas Series Attribute:**

**In Pandas, a Series object has several important attributes that is commonly used attributes of a Pandas Series include:**

**size attribute:**

* **Returns the number of elements in a Series, including any elements that might contain missing or NaN (Not-a-Number) values.**

**Code:**

**import pandas as pd**

**data = [ 10, 20, 30, None, 50 ]**

**print(pd.Series(data).size)**

**data1 = [ 10, 20, 30, 50 ]**

**print(pd.Series(data1).size)**

**Output:**

**5**

**4**

**dtype attribute:**

* **Returns the data type of the elements in the Series.**
* **Used to check the data type of the data within the Series.**

**Code:**

**import pandas as pd**

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

**print(pd.Series(data).dtype)**

**Output:**

**int64**

**name attribute:**

* **Assign a name to a eries when creating it or later using the name attribute.**
* **The name is typically used in the context of DataFrames, where a Series can represent a column.**

**Code:**

**import pandas as pd**

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

**print(pd.Series(data**, **name='my\_data').name)**

**Output:**

**my\_data**

**is\_unique attribute:**

* **Returns a boolean value indicating whether all the values in the Series are unique (no duplicates) or not.**

**Code:**

**import pandas as pd**

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

**print(pd.Series(data).is\_unique)**

**data1 = [ 10, 20, 30, 40, 50 ]**

**print(pd.Series(data1).is\_unique)**

**Output:**

**True**

**False**

**values attribute:**

* **Returns the data in the Series as a NumPy array.**

**Code:**

**import pandas as pd**

**marks = { 'maths' : 78, 'english' : 70, 'science' : 89 }**

**print(pd.Series(marks).values)**

**Output:**

**[ 78 70 89 ]**

**Series using read\_csv() function:**

**~ squeeze=True:**

**Attribute specifies that the result should be squeezed into a series if it has only one column.**

**Here we intentionally use this attribute to avoid the dataframe to understand the about series.**

**Dataset link:**

**Code:**

**import pandas as pd**

**subs = pd.read\_csv('subs.csv', squeeze=True )**

**print(subs)**

**print(type(subs))**

**Output:**

**0 48**

**1 57**

**...**

**363 144**

**364 172**

**Name: Subscribers gained, Length: 365, dtype: int64**

**<class 'pandas.core.series.Series'>**

**~ index\_col:**

**used to specify which column in the CSV file should be used as the index for the resulting DataFrame/Series.**

**The index is a way to uniquely identify each row in the DataFrame/Series.**

**By default, if you don't specify the index\_col parameter, Pandas will create a default integer index starting from 0.**

**Dataset link:**

**Code:**

**import pandas as pd**

**vk = pd.read\_csv('kohli\_ipl.csv', index\_col= 'match\_no',**

**squeeze=True )**

**print(vk)**

**print(type(vk))**

**Output:**

**match\_no**

**1 1**

**2 23**

**..**

**214 25**

**215 7**

**Name: runs, Length: 215, dtype: int64**

**<class 'pandas.core.series.Series'>**

**Dataset link:**

**Code:**

**import pandas as pd**

**m = pd.read\_csv('bollywood.csv', index\_col= 'movie',**

**squeeze=True )**

**print(m)**

**print(type(m))**

**Output:**

**movie**

**Uri: The Surgical Strike Vicky Kaushal**

**Battalion 609 Vicky Ahuja**

**...**

**Company (film) Ajay Devgn**

**Awara Paagal Deewana Akshay Kumar**

**Name: lead, Length: 1500, dtype: object**

**<class 'pandas.core.series.Series'>**

**Note: above dataset used to perform series methods**

**Series Methods:**

**These methods are most used in pandas series.**

**head() method:**

* **Used to display the first few rows (default is 5) of a DataFrame or Series.**
* **If we provide negative number as parameter then return all rows.**

**Code:**

**import pandas as pd**

**print(vk.head()) # default is 5  
 print(vk.head(2))**

**Output:**

**match\_no**

**1 1**

**2 23**

**3 13**

**4 12**

**5 1**

**Name: runs, dtype: int64**

**match\_no**

**1 1**

**2 23**

**Name: runs, dtype: int64**

**tail() method:**

**Used to display the last few rows (default is 5) of a DataFrame or Series.**

**Code:**

**import pandas as pd**

**print(vk.tail())**

**print(vk.tail(2))**

**Output:**

**match\_no**

**211 0**

**212 20**

**213 73**

**214 25**

**215 7**

**Name: runs, dtype: int64**

**match\_no**

**214 25**

**215 7**

**Name: runs, dtype: int64**

**sample() method:**

**To randomly select a specified number of rows (default is 1) or elements from a dataframe or series.**

**Useful to obtain a random sample from your data for data exploration, analysis, or testing.**

**Ex. if you have biased in your datasets then we can use this method.**

**Code:**

**import pandas as pd**

**print(vk.sample()) # default 1**

**print(vk.sample(3)) # three random row from dataset**

**Output:**

**match\_no**

**118 33**

**Name: runs, dtype: int64**

**match\_no**

**202 5**

**150 8**

**110 82**

**Name: runs, dtype: int64**

**value\_counts() method:**

* **To count the frequency of values that occur multiple times in a series.**

**Code:**

**import pandas as pd**

**print(m.value\_counts())**

**Output:**

**Akshay Kumar 48**

**Amitabh Bachchan 45**

**..**

**Akanksha Puri 1**

**Edwin Fernandes 1**

**Name: lead, Length: 566, dtype: int64**

**sort\_values() method:**

* **Used to sort the values within a pandas series.**
* **By default, it sorts the values in ascending order, but you can specify the sorting order using the ascending parameter.**
* **Syntax: series.sort\_values( ascending=True, inplace=False )**

**Parameters:**

|  |  |
| --- | --- |
| **inplace:** | **when true then sort and replace sorted data with original data If false (default), it returns a new series with the sorted values while leaving the original series unchanged** |
| **ascending:** | **True (ascending) or False (descending)** |

**Code:**

**import pandas as pd**

**print(vk.sort\_values()) # default is ascending=True**

**Output:**

**match\_no**

**135 0**

**8 0**

**..**

**126 109**

**128 113**

**Name: runs, Length: 215, dtype: int64**

**# with inplace=True for permanent changes in series**

**Code:**

**print(vk.sort\_values(inplace=True)**

**print(vk)**

**Output:**

**match\_no**

**128 113**

**126 109**

**...**

**8 0**

**135 0**

**Name: runs, Length: 215, dtype: int64**

**Method chaining:**

* **It is practice of applying multiple operations or methods to a Series in a single line of code.**
* **This approach is both efficient and readable, making it easier to perform complex data manipulations and transformations.**

**Code:**

**import pandas as pd**

**print(vk.sort\_values(ascending=False).head(1).values[0])**

**Output:**

**113**

**sort\_index() method:**

* **It similar in concept to the sort\_values() method, but in this method it sorts the index (row labels) of the Series.**
* **Both methods allow you to control the sorting order, either ascending or descending, and both can be used with the inplace parameter to modify the original Series.**
* **syntax: series.sort\_index( ascending=True, inplace=False )**

**Code:**

**import pandas as pd**

**print(vk.sort\_index()) # default is ascending=True**

**Output:**

**match\_no**

**1 1**

**2 23**

**...**

**214 25**

**215 7**

**Name: runs, Length: 215, dtype: int64**

**# descending with inplace=True for permanent changes**

**Code:**

**print(vk.sort\_index(ascending=False, inplace=True)**

**print(vk)**

**Output:**

**match\_no**

**215 7**

**214 25**

**..**

**2 23**

**1 1**

**Name: runs, Length: 215, dtype: int64**

**Series Mathematical Methods:**

**Common statistical methods in Pandas Series for analyzing data:**

**count():**

* **Count the non-null elements in the Series.**

**Code:**

**print(vk.count())**

**print(subs.count())**

**Output:**

**215**

**365**

**sum():**

**Used to calculate the sum of all the elements in a Series**

**Code:**

**print(subs.count())**

**Output:**

**49510**

**product() :**

**The product() method in Pandas Series is used to calculate the product of all elements in the Series.**

**It multiplies all the values together and returns the result.**

**Code:**

**print(subs.count())**

**print(vk.subs)**

**Output:**

**215**

**mean():**

* **Calculates the mean (average) of the elements in a Series.**

**Code:**

**import pandas as pd**

**print('avg yt subs of channel:', subs.mean())**

**print('avg ipl runs of virat kohli:', vk.mean())**

**Output:**

**avg yt subs of channel:** **135.64383561643837**

**avg ipl runs of virat kohli:** **30.855813953488372**

**median():**

* **Calculates the median of the elements in a Series, which is the middle value when the data is sorted.**
* **It is a measure of central tendency.**

**Code:**

**import pandas as pd**

**print(subs.median())**

**print(vk.median())**

**Output:**

**123.0**

**24.0**

**mode():**

* **Returns the mode(s) of the elements in a Series, which is the most frequently occurring value(s).**

**Code:**

**import pandas as pd**

**print('Most frequent lead actor :', m.mode())**

**Output:**

**Most frequent lead actor : 0 Akshay Kumar**

**Name: lead, dtype: object**

**std():**

* **Computes the standard deviation of the elements in a Series, which measures the spread or dispersion of the data.**

**Code:**

**import pandas as pd**

**print(vk.std())**

**Output:**

**26.22980132830278**

**var():**

**The var() method calculates the variance of the elements in a Series, which is the average of the squared differences from the mean.**

**Code:**

**import pandas as pd**

**print(subs.var())**

**print(vk.var())**

**Output:**

**3928.1585127201565**

**688.0024777222343**

**min():**

**The min() method returns the minimum value in a Series or DataFrame.**

**Code:**

**import pandas as pd**

**print('minimum subs: ', subs.min())**

**print('minimum runs: ', vk.min())**

**Output:**

**minimum subs: 33**

**minimum runs: 0**

**max():**

**The max() method returns the maximum value in a Series or DataFrame.**

**Code:**

**import pandas as pd**

**print('maximum subs: ', subs.max())**

**print('maximum runs: ', vk.max())**

**Output:**

**maximum subs: 396**

**maximum runs: 113**

**describe():**

* **It is a convenient function to generate descriptive statistics of a numeric Series.**
* **It provides a summary of various statistical measures, giving you insights into the data's distribution and central tendency.**
* **It’s provides the following statistics:**

|  |  |
| --- | --- |
| **count:** | **number of non-null elements in the Series.** |
| **mean:** | **mean (average) of the Series.** |
| **std:** | **standard deviation, which measures the spread of the data.** |
| **min:** | **minimum value in the Series.** |
| **25%:** | **25th percentile (lower quartile).** |
| **50%:** | **median (50th percentile).** |
| **75%:** | **75th percentile (upper quartile).** |
| **max:** | **maximum value in the Series.** |

**Note: the describe() works on numeric data**

**Code:**

**import pandas as pd**

**print(subs.var())**

**print(vk.var())**

**Output:**

**count 215.000000**

**mean 30.855814**

**std 26.229801**

**min 0.000000**

**25% 9.000000**

**50% 24.000000**

**75% 48.000000**

**max 113.000000**

**Name: runs, dtype: float64**

**count 365.000000**

**mean 135.643836**

**std 62.675023**

**min 33.000000**

**25% 88.000000**

**50% 123.000000**

**75% 177.000000**

**max 396.000000**

**Name: Subscribers gained, dtype: float64**

**Some Important Series Methods/Functions:**

**These are some common methods and functions available for working with Pandas Series in Python:**

**astype() method:**

* **This method is used to cast the data type of the elements in a Series to the specified data type (e.g., int, float, str).**
* **Useful to reduce the memory space**
* **Syntax: series.astype(dtype)**

**Code:**

**import pandas as pd**

**import sys**

**print('Original size of dataset:',sys.getsizeof(vk))**

**vk\_size =vk.astype('int32')**

**print('Reduce size of dataset:',sys.getsizeof(vk\_size))**

**Output:**

**Original size of dataset: 3456**

**Reduce size of dataset: 2596**

**between() method:**

* **Checks if each element in the Series falls within the specified range.**
* **Returns a boolean Series.**
* **Syntax: series.between(left, right, inclusive=True)**

**Code:**

**import pandas as pd**

**print(vk.between(95,110)) # return boolean values**

**print(vk[vk.between(95,110)]) # printing values**

**Output:**

**match\_no**

**1 False**

**2 False**

**…**

**214 False**

**215 False**

**Name: runs, Length: 215, dtype: bool**

**match\_no**

**82 99**

**120 100**

**123 108**

**126 109**

**164 100**

**Name: runs, dtype: int64**

**clip() method:**

* **Clips values in the Series to be within the specified lower and upper bounds.**
* **Syntax: series.clip(lower, upper)**

**Code:**

**import pandas as pd**

**print(subs.clip(100,160))**

**Output:**

**0 100**

**1 100**

**...**

**363 144**

**364 160**

**Name: Subscribers gained, Length: 365, dtype: int64**

**drop\_duplicates() method:**

* **Removes duplicate values from the Series.**
* **Syntax: series.drop\_duplicates(keep='first', inplace=False)**

**Code:**

**import pandas as pd**

**temp = pd.Series([1,1,3,3,3,5,5])**

**# default first**

**print(temp.drop\_duplicates())**

**# deleting first occurance**

**print(temp.drop\_duplicates(keep='last'))**

**Output:**

**0 1**

**2 3**

**5 5**

**dtype: int64**

**1 1**

**4 3**

**6 5**

**dtype: int64**

**duplicated() function:**

* **Used to identify and mark duplicate values in a Series (column) of a DataFrame.**
* **It returns a Boolean Series.**
* **Syntax: Series.duplicated()**

**Code:**

**import pandas as pd**

**temp = pd.Series([1,1,3,3,3,5,5])**

**print(temp.duplicated()) # True means duplicate**

**print('Duplicate value count:',temp.duplicated().sum())**

**Output:**

**0 False**

**1 True**

**2 False**

**3 True**

**4 True**

**5 False**

**6 True**

**dtype: bool**

**Duplicate value count: 4**

**isnull() method:**

* **Returns a boolean Series indicating whether each element is NaN (missing data).**
* **Syntax: series.isnull()**

**Code:**

**import pandas as pd**

**import numpy as np**

**temp = pd.Series([1,3,np.nan,np.nan,5,np.nan,7,np.nan])**

**print(temp.isnull()) # return boolean**

**print('Missing values:',temp.isnull().sum())**

**Output:**

**0 False**

**1 False**

**2 True**

**3 True**

**4 False**

**5 True**

**6 False**

**7 True**

**dtype: bool**

**Missing values: 4**

**dropna() method:**

* **Removes missing (NaN) values from the Series.**
* **Syntax: series.dropna(axis=0, inplace=False)**

**Code:**

**import pandas as pd**

**import numpy as np**

**temp = pd.Series([1,3,np.nan,np.nan,5,np.nan,7,np.nan])**

**print(temp.dropna())**

**Output:**

**0 1.0**

**1 3.0**

**4 5.0**

**6 7.0**

**dtype: float64**

**fillna() method:**

* **Fills missing (NaN) values in the Series with the specified value.**
* **Syntax: series.fillna(value)**

**Code:**

**import pandas as pd**

**import numpy as np**

**temp = pd.Series([1,3, np.nan,5,np.nan,7,np.nan])**

**print(temp.fillna())**

**Output:**

**0 1.0**

**1 3.0**

**2 0.0**

**3 5.0**

**4 0.0**

**5 7.0**

**6 0.0**

**dtype: float64**

**isin() method:**

* **Checks if each element in the Series is in the provided list of values.**
* **Returns a boolean Series.**
* **Syntax: series.isin(values)**

**Code:**

**import pandas as pd**

**print(vk.isin([49,99])) # return Boolean**

**print(vk[vk.isin([49,99])]) # printing values**

**Output:**

**match\_no**

**1 False**

**2 False**

**…**

**214 False**

**215 False**

**Name: runs, Length: 215, dtype: bool**

**match\_no**

**82 99**

**86 49**

**Name: runs, dtype: int64**

**apply() method:**

* **Applies a given function to each element in the Series and returns a new Series with the results.**
* **Syntax: series.apply(func)**

**Code:**

**import pandas as pd**

**print(m.apply(lambda x:x.split()[0].upper()).head(5))**

**Output:**

**movie**

**Uri: The Surgical Strike VICKY**

**Battalion 609 VICKY**

**The Accidental Prime Minister (film) ANUPAM**

**Why Cheat India EMRAAN**

**Evening Shadows MONA**

**Name: lead, dtype: object**

**EXTRA Series method that is used in Exercise:**

**to\_numeric() method:**

* **Used to convert the values in a Series (or DataFrame column) to numeric data types.**
* **Useful when you have a Series containing strings or other non-numeric data, and you want to convert them to numeric types like integers or floating-point numbers.**
* **Syntax: pd.to\_numeric(series, errors='coerce', downcast='integer')**

**Here are some common options for the errors parameter:**

**~ raise(default): Raises an error if any value cannot be converted to**

**a number.**

**~ coerce: Replaces non-convertible values with NaN.**

**~ ignore: Ignores non-convertible values**

**quantile() method:**

**PANDAS DATAFRAMES**

**Pandas DataFrame?**

* **A Pandas DataFrame is a two-dimensional data structure, like a two-dimensional array, or a table with rows and columns.**
* **Used for data manipulation, analysis, and cleaning, featuring labelled rows and columns, support for various data types, and flexibility for adding, removing, and transforming data.**
* **It is widely used in data science and analysis tasks for handling structured data efficiently.**
* **Syntax: pd.DataFrame(data, index, columns, dtype, copy)**

**~ data: ndarray, Iterable, dict, or DataFrame**

**~ index: Index or array-like**

**~ columns: Index or array-like**

**~ dtype: dtype, default None**

**~ copy: bool or None, default None**

**Creating a DataFrame using lists**

**Code:**

**import pandas as pd**

**import numpy as np**

**student\_data = [ [100,95,14], [107,87,16], [89,78,12] ]**

**print(pd.DataFrame(student\_data,**

**columns= [ 'iq', 'marks', 'package' ] ))**

**Output:**

**iq marks package**

**0 100 95 14**

**1 107 87 16**

**2 89 78 12**

**Creating a DataFrame using dictionary**

**Code:**

**import pandas as pd**

**import numpy as np**

**dictionary = {**

**'iq' : [100,95,14],**

**'marks' : [107,87,16],**

**'package' : [89,78,12]**

**}**

**students = pd.DataFrame(dictionary)**

**print(students)**

**Output:**

**iq marks package**

**0 100 95 14**

**1 107 87 16**

**2 89 78 12**

**Creating a DataFarame using read\_csv() function**

**Dataset link:**

**Code:**

**import pandas as pd**

**movies = pd.read\_csv('movies.csv')**

**Dataset link:**

**Code:**

**import pandas as pd**

**movies = pd.read\_csv(‘ipl.csv')**

**Note: above datasets use for performing dataframe attributes and methods.**

**Pandas DataFrame Attribute**

**Accessing a DataFrame through its attributes allows us to get the intrinsic properties of the DataFrame.**

**shape attribute:**

* **Used to display the total number of rows and columns of a particular data frame.**
* **Returns a tuple representing the dimensionality of the DataFrame.**
* **For example, if we have 3 rows and 2 columns in a DataFrame then the shape will be (3,2).**

**Code:**

**import pandas as pd**

**print('Shape of the DataFrame:',movies.shape)**

**print('Shape of the DataFrame:',ipl.shape)**

**Output:**

**Shape of the DataFrame: (1629, 18)**

**Shape of the DataFrame: (950, 20)**

**dtypes attribute:**

**Return datatype of each column present in a dataframe.**

**Code:**

**import pandas as pd**

**print(ipl.dtypes) # or print(movies.dtypes)**

**Output:**

**ID int64**

**City object**

**Date object**

**Season object**

**MatchNumber object**

**Team1 object**

**...**

**Team2Players object**

**Umpire1 object**

**Umpire2 object**

**dtype: object**

**index attribute:**

* **Display the row labels of a the dataframe object.**

**Code:**

**import pandas as pd**

**print(movies.index)**

**print(ipl.index)**

**print(students.indexs)**

**Output:**

**RangeIndex(start=0, stop=1629, step=1)**

**RangeIndex(start=0, stop=950, step=1)**

**RangeIndex(start=0, stop=3, step=1)**

**columns attribute:**

**Fetch the label values for columns present in a dataframe.**

**Code:**

**import pandas as pd**

**print(ipl.columns)**

**print(students.columns)**

**Output:**

**Index(['ID', 'City', 'Date', 'Season', 'MatchNumber', 'Team1', 'Team2',**

**'Venue', 'TossWinner', 'TossDecision', 'SuperOver',**

**'WinningTeam', 'WonBy', 'Margin', 'method',**

**'Player\_of\_Match', 'Team1Players', 'Team2Players',**

**'Umpire1', 'Umpire2'], dtype='object')**

**Index(['iq', 'marks', 'package'], dtype='object')**

**values attribute:**

* **Represent the values/data of dataframe in numpy array from.**

**Code:**

**import pandas as pd**

**# print(movies.values)**

**# print(ipl.values)**

**print(students.values)**

**Output:**

**[[100 107 89]**

**[ 95 87 78]**

**[ 14 16 12]]**

**Pandas DataFrame Methods/Functions:**

**These methods are most used in Pandas DataFrame.**

**head() method:**

* **Used to display the first few rows (default is 5) of a DataFrame.**

**Code:**

**import pandas as pd**

**print(movies.head()) # default is 5  
 print(ipl.head(2))**

**Output:**

**tail() method:**

* **Used to display the last few rows (default is 5) of a DataFrame.**

**Code:**

**import pandas as pd**

**print(movies.tail()) # default is 5**

**print(movies.tail(2))**

**Output:**

**sample() method:**

**To randomly select a specified number of rows or columns (default is 1) or elements from a dataframe.**

**Useful to obtain a random sample from your data for data exploration, analysis, or testing.**

**Ex. if we have biased in your datasets then we can use this method.**

**Code:**

**import pandas as pd**

**print(students.sample()) # default is 1**

**# specify the number for random sample**

**# print(ipl.sample(3))**

**Output:**

**iq marks package**

**1 107 87 16**

**info() method:**

**Used to get a concise summary of the dataframe.**

**This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.**

**To get a quick overview of the dataset we use the info() method.**

**Code:**

**import pandas as pd**

**# print(ipl.info())**

**# print(movies.info())**

**print(students.info())**

**Output:**

**<class 'pandas.core.frame.DataFrame'>**

**RangeIndex: 3 entries, 0 to 2**

**Data columns (total 3 columns):**

**# Column Non-Null Count Dtype**

**--- ----------- --------------------- -------**

**0 iq 3 non-null int64**

**1 marks 3 non-null int64**

**2 package 3 non-null int64**

**dtypes: int64(3)**

**memory usage: 204.0 bytes**

**Pandas DataFrame Mathematical Methods:**

**Common statistical methods in Pandas DataFrame**

**~ index(0) represents axis=0 )**

**~ column(1) represents axis=1**

**sum() function:**

* **Return the sum of the values over the requested axis.**
* **Syntax: DataFrame.sum(axis, numeric\_only ,skipna, \*\*kwargs)**

**~ axis :**

**It represents index or column axis, '0' for index and '1' for the column. When the axis=0, method applied over the index axis and when the axis=1 method applied over the column axis.**

**~ skipna:**

**bool(True or False). The default value is None.**

**~ numeric\_only :**

**bool, default False Include only float, int, boolean columns. Not implemented for Series.**

**~ \*\*kwargs : Additional keyword arguments to be passed to the function.**

**max() function:**

* **Used to get the maximum of the values over the requested axis. It returns Series and if the level is specified, it returns the DataFrame.**
* **Syntax: DataFrame.max(axis=None, skipna=None, numeric\_only=None \*\*kwargs)**

**Code:**

**import pandas as pd**

**Output:**

**min() :**

**Used to get the minimum of the values over the requested axis.**

**It returns Series and if the level is specified, it returns the DataFrame.**

**Syntax: DataFrame.min(axis, skipna, numeric\_only, \*\*kwargs)**

**Code:**

**import pandas as pd**

**Output:**

**mean() function:**

* **Used to get the mean of the values over the requested axis.**
* **It returns Series and if the level is specified, it returns the DataFrame.**
* **Syntax: DataFrame.mean(axis, skipna, numeric\_only, \*\*kwargs)**

**median() function:**

* **Used to get the median of the values over the requested axis.**
* **It returns Series and if the level is specified, it returns the DataFrame.**
* **Syntax: DataFrame.median(axis, skipna, \*\*kwargs)**

**Code:**

**import pandas as pd**

**Output:**

**mode():**

* **we can get each element mode along the specified axis.**
* **When this method applied to the DataFrame, it returns the DataFrame which consists of the modes of each column or row.**
* **Syntax: DataFrame.mode(axis=0, drpna=True)**

**~ dropna: It represents the bool, and the default is True. It does not**

**consider the null values.**

**Code:**

**import pandas as pd**

**Output:**

**std():**

* **Return sample standard deviation over requested axis. By default the standard deviations are normalized by N-1.**
* **It is a measure that is used to quantify the amount of variation or dispersion of a set of data values.**
* **Syntax : DataFrame.std(axis, skipna, level, ddof=1, numeric\_only,**

**\*\*kwargs)**

**~ dropna: It represents the bool, and the default is True. It does not**

**consider the null values.**

**~ ddof : int, default 1 Delta Degrees of Freedom. The divisor used in**

**calculations is N - ddof, where N represents the number of**

**elements.**

**Code:**

**import pandas as pd**

**Output**

**var() function:**

* **Returns the unbiased variance over the specified axis.**
* **Syntax :DataFrame.var(axis, skipna, level, ddof=1)**

**Code:**

**import pandas as pd**

**Output:**

**sort\_values() function: (Sereis or DataFrame)**

* **sort\_values() is used to sort the values within a Pandas Series.**
* **By default, it sorts the values in ascending order, but you can specify the sorting order using the ascending parameter, where ascending=True sorts in ascending order, and ascending=False sorts in descending order**
* **syntax: series.sort\_values(axis=0, ascending=True, inplace=False,**

**kind=’quicksort’, na\_position=’last’)  
 ~ axis : axis to direct sorting.  
 ~ ascending : If True, sort values in ascending order, otherwise**

**descending.  
 ~ inplace : If True, perform operation in-place.  
 ~ kind : choice of sorting algorithm.**

**~ na\_position : argument ‘first’ puts NaNs at the beginning,**

**‘last’ puts NaNs at the end.**

**Code:**

**import pandas as pd**

**movies = pd.read\_csv('movies.csv')**

**movies.sort\_values()**

**# Sorting based on column**

**movies.sort\_values(by='title\_x', ascending=False)**

**# Sorting with multiple columns movies.columns**

**movies.sort\_values(by=['year\_of\_release', 'title\_x'],**

**ascending=[False, False])**

**code:**

**students = pd.DataFrame( {**

**'name':['nitish','ankit','rupesh',np.nan,'mrityunjay',**

**np.nan,'rishabh',np.nan,'aditya',np.nan],**

**'college':['bit','iit','vit',np.nan,np.nan,'vlsi','ssit',np.nan,**

**np.nan,'git'],**

**'branch':['eee','it','cse',np.nan,'me','ce','civ','cse','bio',**

**np.nan],**

**'cgpa':[6.66,8.25,6.41,np.nan,5.6,9.0,7.4,10,7.4,np.nan],**

**'package':[4,5,6,np.nan,6,7,8,9,np.nan,np.nan] } )**

**print(students)**

**# Sorting with NaN values**

**students.sort\_values(by='name', na\_position='first')**

**Output:**

**Name college branch cgpa package**

**0 nitish bit eee 6.66 4.0**

**1 ankit iit it 8.25 5.0**

**2 rupesh vit cse 6.41 6.0**

**3 NaN NaN NaN NaN NaN**

**4 mrityunjay NaN me 5.60 6.0**

**5 NaN vlsi ce 9.00 7.0**

**6 rishabh ssit civ 7.40 8.0**

**7 NaN NaN cse 10.00 9.0**

**8 aditya NaN bio 7.40 NaN**

**9 NaN git NaN NaN NaN**

**Name college branch cgpa package**

**3 NaN NaN NaN NaN NaN**

**5 NaN vlsi ce 9.00 7.0**

**7 NaN NaN cse 10.00 9.0**

**9 NaN git NaN NaN NaN**

**8 aditya NaN bio 7.40 NaN**

**1 ankit iit it 8.25 5.0**

**4 mrityunjay NaN me 5.60 6.0**

**0 nitish bit eee 6.66 4.0**

**6 rishabh ssit civ 7.40 8.0**

**2 rupesh vit cse 6.41 6.0**

**rank(): (sereis)**

**The rank() function of Pandas is used to compute numerical data ranks (1 through n) along axis.**

**After sorting (by default in ascending order), the position is used to determine the rank that is returned.**

**If data contains equal values, then they are assigned with the average of the ranks of each value by default.**

**Syntax: DataFrame.rank(axis=0, method='average',**

**numeric\_only=NoDefault. no\_default, na\_option='keep',**

**ascending=True, pct=False)**

**~ axis : index to direct ranking**

**~ method : {‘average’, ‘min’, ‘max’, ‘first’, ‘dense’}**

**~ numeric\_only :Include only float, int, boolean data. Valid only**

**for DataFrame or Panel objects**

**~ na\_option : {‘keep’, ‘top’, ‘bottom’}**

**~ ascending : False for ranks by high (1) to low (N)**

**~ pct : Computes percentage rank of data**

**Dataset link:**

**Code:**

**import pandas as pd**

**batsman = pd.read\_csv(‘batsman\_runs\_ipl.csv')**

**# apply rank() function on specific column**

**batsman['ranks'] = batsman['batsman\_run'].rank(ascending=False)**

**batsman.sort\_values('ranks')**

**Output:**

**sort\_index(): (Series or DataFrame)**

* **Pandas Series.sort\_index() function is used to sort the index labels of the given series or DataFrame.**
* **syntax: Series.sort\_index(axis=0, ascending=True, inplace=False,**

**kind=’quicksort’, na\_position=’last’)**

**Code:**

**import pandas as pd**

**marks = { 'maths' : 78, 'english' : 70, 'science' : 89 }**

**marks\_series = pd.Series(marks)**

**print(marks\_series)**

**Output:**

**maths 78**

**english 70**

**science 89**

**dtype: int64**

**# alphabetically sorted index for string index**

**Code:**

**print(marks\_series.sort\_index())**

**# print(marks\_series.sort\_index(ascending=False))**

**Output:**

**english 70**

**maths 78**

**science 89**

**dtype: int64**

**set\_index() :(dataframe)**

* **Used to re-assign a row label using the existing column of the DataFrame.**
* **It can assign one or multiple columns as a row index.**
* **Syntax: DataFrame.set\_index( keys, drop=True, append=False,**

**inplace=False, verify\_integrity=False )**

**~ keys: column name or list of column name.**

**~ drop: boolean value which drops the column used for index if True.**

**~ append: appends the column to existing index column if True.**

**~ inplace: makes the changes in the dataframe if True.**

**~ verify\_integrity: checking new index column for duplicates if True.**

**Code:**

**import pandas as pd**

**# set batter column as index**

**batsman.set\_index('batter').head()**

**# batsman.set\_index('batter',inplace=True)**

**Output:**

**batter batsman\_run batting\_rank**

**A Ashish Reddy 280 166.5**

**A Badoni 161 226.0**

**A Chandila 4 535.0**

**A Chopra 53 329.0**

**A Choudhary 25 402.5**

**reset\_index() : ( series or dataframe )**

* **It is opposite to set\_index()**
* **Convert the series into dataframe.**
* **Function reset the index to the default integer index beginning at 0, We can simply use the reset\_index() function.**
* **Syntax: DataFrame.reset\_index(inplace=False)**

**Code:**

**import pandas as pd**

**# reset the index column**

**batsman.reset\_index().head()**

**# batsman.reset\_index(inplace=True)**

**Output:**

**batter batsman\_run batting\_rank**

**0 A Ashish Reddy 280 166.5**

**1 A Badoni 161 226.0**

**2 A Chandila 4 535.0**

**3 A Chopra 53 329.0**

**4 A Choudhary 25 402.5**

**Code:**

**# how to replace existing index without loosing**

**batsman.reset\_index().set\_index('batting\_rank').head(3)**

**# series to dataframe using reset\_index**

**marks\_series.reset\_index()**

**Output:**

**batting\_rank index batter batsman\_run**

**166.5 0 A Ashish Reddy 280**

**226.0 1 A Badoni 161**

**535.0 2 A Chandila 4**

**Output:**

**index 0**

**0 maths 67**

**1 english 57**

**2 science 89**

**rename() method: (dataframe)**

* **Used to rename any index, column, or row.**
* **Syntax: DataFrame.rename(mapper,index,columns,axis,copy**

**,inplace,level)**

**~ mapper, index and columns: Dictionary value, key refers to the**

**old name and value refers to new name. Only one of these**

**parameters can be used at once.**

**~ axis: int or string value, 0/’row’ for Rows and 1/’columns’ for**

**columns.**

**~ copy: copies underlying data if True.**

**~ inplace: makes changes in original Data Frame if True.**

**~ level: specify level in dataframe if it have multiple level index.**

**Code:**

**import pandas as pd**

**# display specific columns to from movies dataset**

**movie\_list = movies[['title\_x','imdb\_rating','imdb\_votes']].head(3)**

**movie\_list**

**Output:**

**title\_x imdb\_rating imdb\_votes**

**0 Uri: The Surgical Strike 8.4 35112**

**1 Battalion 609 4.1 73**

**2 The Accidental Prime Minister (film) 6.1 5549**

**Code:**

**# set the index**

**movie\_list.set\_index('title\_x',inplace=True)**

**# rename the columns**

**movie\_list.rename(columns={'imdb\_votes':'votes',**

**'imdb\_rating':'rating'}).head(3)**

**Output:**

**rating votes**

**title\_x**

**0 Uri: The Surgical Strike 8.4 35112**

**1 Battalion 609 4.1 73**

**2 The Accidental Prime Minister (film) 6.1 5549**

**Code:**

**# rename the index labels**

**movie\_list.rename(index={'Uri: The Surgical Strike':'Uri',**

**'Battalion 609':'Battalion'}).head(3)**

**Output:**

**rating votes**

**title\_x**

**0 Uri 8.4 35112**

**1 Battalion 4.1 73**

**2 The Accidental Prime Minister (film) 6.1 5549**

**unique() method: (series)**

* **Return unique values based on a hash table.**
* **Multiple missing values consider as one NaN value.**
* **Syntax: Series.unique()**

**Code:**

**import pandas as pd**

**temp = pd.Series([1,1,2,2,3,3,4,np.nan,np.nan])**

**print(temp.unique())**

**# return unique ipl season form ipl dataset**

**print(ipl['Season'].unique())**

**# count the unique ipl season form ipl dataset**

**print(ipl['Season'].unique().shape[0])**

**Output:**

**array([ 1., 2., 3., 4., nan])**

**['2022' '2021' '2020/21' '2019' '2018' '2017' '2016' '2015' '2014' '2**

**013'**

**'2012' '2011' '2009/10' '2009' '2007/08']   
 15**

**nunique() method: (series or dataframe)**

* **The number of unique observations over the requested axis.**
* **It returns Series with a number of distinct observations.**

**Syntax : DataFrame.nunique(axis=0, dropna=True)**

**axis:**

**It represents index or column axis, '0' for index and '1' for the column.**

**When the axis=0, function applied over the index axis and when the axis=1 function applied over the column axis**

**dropna:**

**It represents the bool(True or False), and the default is True. It doesn’t include NaN in the counts.**

**Code:**

**# Count unique values of the dataframe**

**temp = pd.DataFrame([1,1,2,2,3,3,4,np.nan,np.nan])**

**# count with default parameter**

**print('without dropna parameter :',temp.nunique()[0])**

**# count with dropna=False parameter**

**print('with dropna parameter :',temp.nunique(dropna=False)[0])**

**# return unique ipl season form ipl dataset**

**print('ipl season count :',ipl['Season'].nunique())**

**Output:**

**without dropna parameter : 4**

**with dropna parameter : 5**

**ipl season count : 15**

**Functions/Methods for handling missing values**

**isnull() method: (series or dataframe)**

* **Returns the DataFrame of the boolean values.**
* **If the resultant DataFrame consists of the True, it indicates that the element is a null value and if it is False it indicates that the element is not a null value.**
* **Syntax: Series.isnull()**

**Code:**

**# apply on entire dataframe**

**students.isnull()**

**# return boolean values**

**students['name'].isnull()**

**# display nan values**

**students['name'][students['name'].isnull()]**

**Output:**

**3 NaN**

**5 NaN**

**7 NaN**

**9 NaN**

**Name: name, dtype: object**

**notnull() method: (series or dataframe)**

* **Used to detect the existing values.**
* **It returns a DataFrame consisting of bool values for each element in DataFrame that indicates whether an element is not a null value.**
* **While detecting the existing values, this method does not consider the characters such as empty strings '' or numpy.inf as null values.**
* **Syntax: DataFrame.notnull()**

**Code:**

**# apply on entire dataframe**

**students.notnull()**

**# detecting existing values**

**students['name'][students['name'].notnull()]**

**Output:**

1. **nitish**
2. **ankit**
3. **rupesh**
4. **mrityunjay**
5. **rishabh**
6. **aditya**

**Name: name, dtype: object**

**hasnans attribute: (series)**

* **Returns a boolean value.**
* **It return True if the given Series object has missing values in it else it return False.**
* **Syntax:Series.hasnans**

**Code:**

**students['name'].hasnans**

**Output:**

**True**

**drona() method: (series or dataframe)**

* **It removes the missing values and returns the DataFrame with NA**

**entries dropped from it or None if inplace=True.**

* **Syntax : DataFrame.dropna(axis=0, how='any', thresh, subset,**

**inplace=False)**

**Parameters**

**axis: {0 or ‘index’, 1 or ‘columns’}, default 0. Determine if rows or columns which contain missing values are removed.**

**~ 0, or ‘index’: Drop rows that contain missing values.**

**~ 1, or ‘columns’:\*\* Drop columns that contain the missing value.**

**how: {‘any’, ‘all’}, default ‘any’. Determine if row or column is removed from DataFrame when we have at least one NA or all NA.**

**~ ‘any’: If any NA values are present, drop that row or column.**

**~ ‘all’: If all values are NA, drop that row or column.**

**thresh: int, optional. Require that many non-NA values.**

**Subset: array-like, optional labels along other axis**

**inplace: bool, default False. If True, do operation inplace and return None.**

**Code:**

**# apply on sereis**

**students['name'].dropna()**

**# apply on dataframe & remove rows if any one columns have nan**

**value**

**students.dropna()**

**# with how parameter**

**students.dropna(how='all')**

**# remove nan value from specific column**

**students.dropna(subset=['name'])**

**# remove nan values based on mutiple columns**

**students.dropna(subset=['name','college'])**

**Output:**

**Name college branch cgpa package**

**0 nitish bit eee 6.66 4.0**

**1 ankit iit it 8.25 5.0**

**2 rupesh vit cse 6.41 6.0**

**3 NaN NaN NaN NaN NaN**

**4 mrityunjay NaN me 5.60 6.0**

**5 NaN vlsi ce 9.00 7.0**

**6 rishabh ssit civ 7.40 8.0**

**7 NaN NaN cse 10.00 9.0**

**8 aditya NaN bio 7.40 NaN**

**9 NaN git NaN NaN NaN**

**fillna() method: (series or dataframe)**

* **Fills NA/NaN values using the specified method.**
* **It returns the DataFrame object with missing values filled or None if inplace=True.**
* **Syntax: DataFrame.fillna(value, method, axis, inplace=False, limit, downcast)**

**~ value: scalar, dict, Series, or DataFrame.Value to use to fill**

**~ method: {‘backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, default None.**

**~ axis: {0 or ‘index’, 1 or ‘columns’}. Axis along which to fill missing values.**

**Note: most of the time apply on specific columns**

**Code:**

**# Series**

**students['name'].fillna('unknown')**

**# replace students package nan values with their average package**

**students['package'].fillna(students['package'].mean())**

**Output:**

**0 nitish**

1. **ankit**
2. **rupesh**
3. **unknown**
4. **mrityunjay**
5. **unknown**
6. **rishabh**
7. **unknown**
8. **aditya**
9. **unknown**

**Name: name, dtype: object**

1. **4.000000**
2. **5.000000**
3. **6.000000**
4. **6.428571**
5. **6.000000**
6. **7.000000**
7. **8.000000**
8. **9.000000**
9. **6.428571**
10. **6.428571**

**Name: package, dtype: float64**

**drop\_duplicates() method: (series or dataframe)**

* **It returns a DataFrame with duplicate rows removed.**
* **Considering certain columns is optional. Indexes, including time indexes, are ignored.**
* **Syntax : DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False, ignore\_index=False)**

**Code:**

**import pandas as pd**

**import numpy as np**

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

**temp.drop\_duplicates()**

**Output:**

1. **1**
2. **2**
3. **3**

**6 4**

**dtype: int64**

**# drop\_duplicates() function with keep parameter**

**Code:**

**marks = pd.DataFrame( [ [100,80,10], [120,100,14],**

**[80,70,14], [80,70,14] ],**

**columns= [ 'iq', 'marks', 'package' ] )**

**print(marks)**

**print(marks.drop\_duplicates(keep='last'))**

**Output:**

**iq marks package**

**0 100 80 10**

**1 120 100 14**

**2 80 70 14**

**3 80 70 14**

**iq marks package**

**0 100 80 10**

**1 120 100 14**

**3 80 70 14**

**drop() method: (series or dataframe)**

* **It drops specified labels from rows or columns.**
* **It removes rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names.**
* **When using a multi-index, labels on different levels can be removed by specifying the level.**
* **It returns the DataFrame or None. DataFrame without the removed index or column labels or None if inplace=True.**
* **It raises KeyError exception if any of the labels are not found in the selected axis.**

**Syntax: DataFrame.drop(labels=None,axis=0,index=None,columns=None,level=None,inplace=False,errors='raise')**

**Code:**

**temp = pd.Series([10,2,3,8,10])**

**print(temp)**

**# apply on sereis with index parameter**

**print(temp.drop(index=[0,3]))**

**# DataFrame with index & columns parameter**

**students.drop(index=[1,3],columns=['branch','cgpa']).head(4)**

**Output:**

**0 10**

**1 2**

**2 3**

**3 8**

**4 10**

**dtype: int64**

**1 2**

**2 3**

**4 10**

**dtype: int64**

**name college package**

**0 nitish bit 4.0**

**2 rupesh vit 6.0**

**1 mrityunjay NaN 6.0**

**1 NaN vlsi 7.0**

**apply() method: (series or dataframe)**

* **Using this method we can apply different functions on rows and columns of the DataFrame.**
* **The objects passed to the method are Series objects whose index is either the DataFrame’s index (axis=0) or the DataFrame’s columns (axis=1).**
* **Syntax: DataFrame.apply(func, axis=0, raw=False, result\_type=None, args=(),\*\*kwds)**

**Code:**

**points\_df = pd.DataFrame({**

**'1st point':[(3,4),(-6,5),(0,0),(-10,1),(4,5)],**

**'2nd point':[(-3,4),(0,0),(2,2),(10,10),(1,1)]})**

**# print(points\_df)**

**# creating function for apply() function**

**def euclidean(row):**

**point\_A = row['1st point']**

**point\_B = row['2nd point']**

**return ((point\_A[0] - point\_B[0])\*\*2 +**

**(point\_A[1] - point\_B[1])\*\*2)\*\*0.5**

**# apply on DataFrame and asign new column**

**points\_df['distance'] = points\_df.apply(euclidean, axis=1)**

**print(points\_df)**

**Output:**

**1st point 2nd point distance**

**0 (3, 4) (-3, 4) 6.000000**

**1 (-6, 5) (0, 0) 7.810250**

**2 (0, 0) (2, 2) 2.828427**

**3 (-10, 1) (10, 10) 21.931712**

**4 (4, 5) (1, 1) 5.000000**

**nlargest() method: (series or dataframe)**

**nsmallest() method: (series or dataframe)**

**insert() method: (series or dataframe)**

**copy() method: (series or dataframe)**

**GROUPBY OBJECT IN PANDAS**

**WHAT IS PANDAS GROUPBY?**

* **Pandas groupby splits all the records from your data set into different categories or groups so that you can analyze the data by these groups.**
* **When you use the groupby() function on any categorical column of DataFrame, it returns a GroupBy object, which you can use other methods on to group the data.**
* **Generally we two types of columns in datasets numerical and categorical**
* **Numerical Columns:**
* **Numerical columns contain data that consists of numbers. These numbers can be integers or floating-point numbers (decimals).**
* **Examples of numerical columns include columns like "Age," "Salary," "Temperature," "Number of Items Sold," and "Height.**
* **Categorical Columns:**
* **Categorical columns contain data that represents categories or discrete values. These values are often labels or strings.**
* **Categorical columns include columns like "Gender" (with values like "Male" and "Female"), "Product Category" (with values like "Electronics," "Clothing," and "Furniture"), and "Country" (with values like "USA," "Canada," and "UK").**

**Note : groupby() always apply on categorical columns**

**Syntax: DataFrame.groupby(by, axis=0, level, as\_index=True, sort=True, group\_keys=True, squeeze=False, \*\*kwargs)**

**~ by : mapping, function, str, or iterable**

**~ axis : int, default 0**

**~ level : If the axis is a MultiIndex (hierarchical), group by a**

**particular level or levels**

**~ as\_index : For aggregated output, return object with group**

**labels as the index. Only relevant for DataFrame**

**input. as\_index=False is effectively “SQL-style”**

**grouped output**

**~ sort : Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.**

**~ group\_keys : When calling apply, add group keys to index to**

**identify pieces**

**~ squeeze : Reduce the dimensionality of the return type if**

**possible, otherwise return a consistent type**

**Code:**

**import pandas as pd**

**movies = pd.read\_csv('imdb-top-1000.csv')**

**movies.head(20)**

**Output:**

**Code:**

**import pandas as pd**

**# creating group by object**

**genres = movies.groupby('Genre')**

**Output:**

**Code:**

**import pandas as pd**

**# Applying builtIn aggregation function on groupby objects**

**genres.mean() # sum() min() mode() median() std() etc**

**genres.mean()[['Runtime' , 'IMDB\_Rating']].head()**

**Output:**

**Runtime IMDB\_Rating**

**Genre**

**Action 129.046512 7.949419**

**Adventure 134.111111 7.937500**

**Animation 99.585366 7.930488**

**Biography 136.022727 7.938636**

**Comedy 112.129032 7.901290**

**Code:**

**import pandas as pd**

**# find the top 3 genres by total earning**

**movies.groupby('Genre').sum()['Gross'].sort\_values(**

**ascending=False).head(3)**

**Output:**

**Genre**

**Drama 3.540997e+10**

**Action 3.263226e+10**

**Comedy 1.566387e+10**

**Name: Gross, dtype: float64**

**Code:**

**import pandas as pd**

**# efficienct way**

**movies.groupby('Genre')['Gross'].sum().sort\_values(ascending=**

**False).head(3)**

**Output:**

**Genre**

**Drama 3.540997e+10**

**Action 3.263226e+10**

**Comedy 1.566387e+10**

**Name: Gross, dtype: float64**

**Code:**

**import pandas as pd**

**# find the genre with highest avgrage IMDB rating**

**movies.groupby('Genre')['IMDB\_Rating'].mean().sort\_values(asc**

**ending=False).head(1)**

**Output:**

**Genre**

**Western 8.35**

**Name: IMDB\_Rating, dtype: float64**

**Code:**

**import pandas as pd**

**# find director with most popularity**

**movies.groupby('Director')['No\_of\_Votes'].sum().sort\_values(asc**

**ending=False).head(1)**

**Output:**

**Director**

**Christopher Nolan 11578345**

**Name: No\_of\_Votes, dtype: int64**

**Code:**

**import pandas as pd**

**# find number of movies done by each actor**

**# movies['Star1'].value\_counts()**

**movies.groupby('Star1')['Series\_Title'].count().sort\_values**

**(ascending=False).head(3)**

**Output:**

**Star1**

**Tom Hanks 12**

**Robert De Niro 11**

**Clint Eastwood 10**

**Name: Series\_Title, dtype: int64**

**Groupby Attributes and Methods/function:**

**len() function :**

* **To find the total number of groups created by the groupby operation.**

**Code:**

**import pandas as pd**

**# find total number of groups**

**len(movies.groupby('Genre')) # or movies['Genre'].nunique()**

**Output:**

**14**

**size() method:**

* **To find the number of items in each group.**

**Code:**

**import pandas as pd**

**# find items in each group size & sort based on index**

**movies.groupby('Genre').size().head()**

**Output:**

**Genre**

**Action 172**

**Adventure 72**

**Animation 82**

**Biography 88**

**Comedy 155**

**dtype: int64**

**Code:**

**import pandas as pd**

**# sort based on values**

**movies['Genre'].value\_counts().head()**

**Output:**

**Drama 289**

**Action 172**

**Comedy 155**

**Crime 107**

**Biography 88**

**Name: Genre, dtype: int64**

**first() & last():**

* **To retrieve the first or last item within each group.**

**Code:**

**import pandas as pd**

**genres = movies.groupby('Genre')**

**# retrive the first item of each group using groupby**

**genres.first().head()**

**# retrive the last item of each group using groupby**

**genres.last().head()**

**Output:**

**nth() method:**

* **To retrieve the nth item from each group within a DataFrame after performing a groupby operation.**
* **Syntax: DataFrameGroupBy.nth(n, dropna='all')**

**Code:**

**import pandas as pd**

**# nth()**

**genres.nth(7)**

**Output:**

**get\_group() method:**

* **To retrieve a specific group by its name, which is useful for selective group access as opposed to filtering.**

**Code:**

**import pandas as pd**

**genres.get\_group('Family')**

**Output:**

**groups attribute:**

* **To access the groups as a dictionary where keys are unique group labels and values are group indices.**

**Code:**

**import pandas as pd**

**# index position of movies based on genres**

**# genres.groups**

**Output:**

**describe() method:**

* **To generate descriptive statistics for each group, providing information like mean, std deviation, min, max, and more.**

**Code:**

**import pandas as pd**

**# describe each column or specified columns**

**# genres.describe()**

**genres.describe()['Runtime'].head()**

**Output:**

**Count mean … 75% max**

**Genre**

**Action 172.0 129.046512 … 143.25 321.0**

**Adventure 72.0 134.111111 … 149.00 228.0**

**Animation 82.0 99.585366 … 106.75 137.0**

**Biography 88.0 136.022727 … 146.25 209.0**

**Comedy 155.0 112.129032 … 124.50 188.0**

**sample() method:**

* **To obtain a random sample from each group.**
* **Syntax: pd.DataFrame.groupby().sample(n, replace=False)**

**~ n : Sample size to return for each group.**

**~ replace : default false and allow or disallow sampling of the same**

**row more than once.**

**Code:**

**import pandas as pd**

**#genres.sample()**

**# genres.sample(2, replace=True)**

**genres.sample(2, replace=True)[['Series\_Title', 'Released\_Year',**

**'Genre']].head(4)**

**Output:**

**Series\_Title Released\_Year Genre**

**900 Serbuan maut 2011 Action**

**223 Mad Max: Fury Road 2015 Action**

**675 Back to the Future Part II 1989 Adventure**

**406 The Princess Bride 1987 Adventure**

**nunique() method:**

**To count the number of unique values within each group.**

**Code:**

**import pandas as pd**

**# genres.nunique()**

**genres.nunique()[['Series\_Title','Released\_Year']].head()**

**Output:**

**Series\_Title Released\_Year**

**Genre**

**Action 172 61**

**Adventure 72 49**

**Animation 82 35**

**Biography 88 44**

**Comedy 155 72**

**agg() method :**

* **Apply multiple aggrigation function at same time.**
* **Syntax : DataFrameGroupBy.agg(arg, *args, \*\**kwargs)**

**Code:**

**import pandas as pd**

**# passing dictionary**

**genres.agg(**

**{**

**'Runtime' : 'mean',**

**'IMDB\_Rating' : 'mean',**

**'No\_of\_Votes' : 'sum',**

**'Gross' : 'sum',**

**'Metascore' : 'mean'**

**}**

**)**

**Output:**

**# passing list**

**Code:**

**import pandas as pd**

**genres.agg([ 'min' , 'max' , 'mean' ]).[** **'Runtime'].head()**

**Output:**

**Runtime**

**min max mean**

**Genre**

**Action 45 321 129.046512**

**Adventure 88 228 134.111111**

**Animation 71 137 99.585366**

**Biography 93 209 136.022727**

**Comedy 68 188 112.129032**

**# adding both dictionary and list**

**Code:**

**import pandas as pd**

**genres.agg( { 'Runtime' : [ 'max' , 'min' , 'mean' ],**

**'IMDB\_Rating' : [ 'max' , 'min' ], }).head()**

**Output:**

**Runtime IMDB\_Rating**

**min max mean min max**

**Genre**

**Action 45 321 129.046512 9.0 7.6**

**Adventure 88 228 134.111111 8.6 7.6**

**Animation 71 137 99.585366 8.6 7.6**

**Biography 93 209 136.022727 8.9 7.6**

**Comedy 68 188 112.129032 8.6 7.6**

**Looping On Groups:**

* **Groupby is a method in Pandas that allows you to group a DataFrame or Series by one or more columns.**
* **Once you have your data grouped, you can perform various operations on each group, such as aggregation, transformation, or filtering.**

**loop over groupby:**

* **Once you have your data grouped using the groupby method, you can loop over each group using a for loop.**
* **Syntax:**

**for group\_name, group\_data in df.groupby('column\_name'):**

**#perform some analysis or visualization**

**Code:**

**import pandas as pd**

**# Find the higest rated movies of each genre**

**df = pd.DataFrame(columns=movies.columns)**

**for group, data in genres:**

**df = df.append(data[data['IMDB\_Rating'] == data['IMDB\_Rating'].**

**max()])**

**print(df)**

**Output:**

**Split-Apply-Combine Strategy**

**By “group by” we are referring to a process involving one or more of the following steps:**

* **Splitting the data into groups based on some criteria.**
* **Applying a function to each group independently.**
* **Combining the results into a new DataFrame.**

**Code:**

**import pandas as pd**

**# find number of movies starting with A for each group**

**def alphabet(group):**

**return group['Series\_Title'].str.startswith('A').sum()**

**genres.apply(alphabet)**

**Output:**

**Genre**

**Action 10**

**Adventure 2**

**Animation 2**

**Biography 9**

**Comedy 14**

**dtype: int64**

**# find ranking of each movie in the group according to IMDB score**

**def ranking(group):**

**# assign the new column for ranking**

**group['genre\_rank'] = group['IMDB\_Rating'].rank(ascending=False)**

**return group**

**genres.apply(ranking)**

**# find normalized IMDB rating group wise**

**# normalized formula: (X - Xmin) / (Xmax - Xmin)**

**Code:**

**import pandas as pd**

**def normalized(group):**

**X = group['IMDB\_Rating']**

**Xmin = group['IMDB\_Rating'].min()**

**Xmax = group['IMDB\_Rating'].max()**

**group['Normalized\_Rating'] = (X - Xmin) / (Xmax - Xmin)**

**return group**

**genres.apply(normalized)**

**Output:**

**Code:**

**import pandas as pd**

**# groupby on multiple cols**

**duo = movies.groupby(['Director','Star1'])**

**# size**

**duo.size()**

**# get\_group**

**duo.get\_group(( 'Aamir Khan' , 'Amole Gupte' ))**

**Output:**

**Code:**

**import pandas as pd**

**# find the most earning actor->director combo**

**duo['Gross'].sum().sort\_values(ascending=False).head(1)**

**Output:**

**Director Star1**

**Akira Kurosawa Toshirô Mifune 2.999877e+09**

**Name: Gross, dtype: float64**

**Code:**

**import pandas as pd**

**# find the best(in-terms of metascore(avg)) actor->genre combo**

**movies.groupby(['Star1','Genre'])['Metascore'].mean()**

**.reset\_index().sort\_values('Metascore',ascending=False).head()**

**Output:**

**Star1 Genre Metascore**

**230 Ellar Coltrane Drama 100.0**

**329 Humphrey Bogart Drama 100.0**

**360 James Stewart Mystery 100.0**

**77 Bertil Guve Drama 100.0**

**590 Orson Welles Drama 100.0**

**Code:**

**import pandas as pd**

**# agg on multiple groupby**

**duo.agg(['min','max','mean'])[['Runtime','IMDB\_Rating']].head(2)**

**Output:**

**Runtime IMDB\_Rating**

**min max min max**

**Director Star1**

**Aamir Khan Amole Gupte 165 165 8.4 8.4**

**Aaron Sorkin Eddie Redmayne 129 129 7.8 7.8**

**S20: MERGING, JOINING AND CONCATENATION**

**pd.concat() function:**

* **Pandas concat() method is used to concatenate pandas objects such as DataFrames and Series.**
* **We can pass various parameters to change the behavior of the concatenation operation.**
* **Syntax: pd.concat(objs, axis, join, ignore\_index, keys, levels, names,**

**verify\_integrity, sort, copy)**

**Parameters:**

|  |  |
| --- | --- |
| **objs:** | **Series or DataFrame objects** |
| **axis:** | **axis to concatenate along; default = 0** |
| **join:** | **way to handle indexes on other axis; default = ‘outer’** |
| **ignore\_index:** | **if True, do not use the index values along the**  **concatenation axis; default = False** |
| **keys:** | **sequence to add an identifier to the result indexes; default = None** |
| **levels:** | **specific levels (unique values) to use for constructing a MultiIndex; default = None** |
| **names:** | **names for the levels in the resulting hierarchical index; default = None** |
| **verify\_integrity:** | **check whether the new concatenated axis**  **contains duplicates; default = False** |
| **sort:** | **sort non-concatenation axis if it is not already aligned when join is ‘outer’; default = False** |
| **copy:** | **if False, do not copy data unnecessarily; default = True** |

**# concat the columns vertically(default)**

**registered = pd.concat([nov,dec], ignore\_index=True)**

**registered**

**import pandas as pd**

**d1 = {"Name": ["Pankaj", "Lisa"], "ID": [1, 2]}**

**d2 = {"Name": "David", "ID": 3}**

**df1 = pd.DataFrame(d1, index=[1, 2])**

**df2 = pd.DataFrame(d2, index=[3])**

**df3 = pd.concat([df1, df2])**

**print(df3)**

**pd.append() function:**

* **Used to append rows of other data frames to the end of the given data frame, returning a new data frame object.**
* **Columns not in the original data frames are added as new columns and the new cells are populated with NaN value.**
* **Syntax: DataFrame.append(other, ignore\_index=False,**

**verify\_integrity=False, sort=None)**

**~other: DataFrame or Series/dict-like object, or list of these The**

**data to append.**

**~ ignore\_index: If True, do not use the index labels.**

**~ verify\_integrity: If True, raise ValueError on creating an index**

**with duplicates.**

**~ sortPandas: default False, Sort columns if the columns of self**

**and other are not aligned.**

**Note: this function is deprecated and will be removed from pandas in a future version.**

**Code:**

**# example**

**nov.append(dec, ignore\_index=True)**

**# Multiindex dataframe (keep original index as it is)**

**multi = pd.concat([nov, dec], keys=['Nov', 'Dec'])**

**# accessing each months**

**multi.loc['Nov']**

**multi.loc['Dec']**

**# accessing the items**

**multi.loc[('Nov',0)]**

**# concat dataframe horizontally**

**pd.concat([nov,dec], axis=1)**

**merge() function:**

* **Used to merge two DataFrame objects with a database-style join operation.**
* **The joining is performed on columns or indexes.**
* **If the joining is done on columns, indexes are ignored.**
* **This function returns a new DataFrame and the source DataFrame objects are unchanged.**
* **Syntax : DataFrame.merge(self, right, how='inner', on, left\_on,**

**right\_on, left\_index=False, right\_index=False, sort=False,**

**suffixes=('\_x', '\_y'))**

**different joins with merge() function:**

**Inner Join:**

* **An inner join returns only the rows where there is a match in both DataFrames' specified columns.**
* **It retains only the common elements from both DataFrames.**
* **Use the pd.merge() function with the how='inner' parameter or the .merge() method with the how='inner' argument**

**Code:**

**# inner join**

**students.merge(registered, how='inner', on='student\_id')**

**left join:**

* **A left join returns all the rows from the left DataFrame and the matching rows from the right DataFrame.**
* **If there's no match in the right DataFrame, NaN values are filled in for columns from the right DataFrame.**
* **Use the pd.merge() function with the how='left' parameter or the .merge() method with the how='left' argument.**

**right join:**

* **A right join is the opposite of a left join.**
* **It returns all the rows from the right DataFrame and the matching rows from the left DataFrame.**
* **If there's no match in the left DataFrame, NaN values are filled in for columns from the left DataFrame.**
* **Use the pd.merge() function with the how='right' parameter or the .merge() method with the how='right' argument.**

**Outer Join (Full Outer Join):**

* **An outer join returns all the rows when there is a match in either the left or the right DataFrame.**
* **If there's no match in one of the DataFrames, NaN values are filled in for the corresponding columns.**
* **Use the pd.merge() function with the how='outer' parameter or the .merge() method with the how='outer' argument.**

**# 1. find total revenue generated**

**registered.merge(courses, how='inner', on='course\_id')['price'].sum()**

**# 2. find month by month revenue**

**temp\_df = pd.concat([nov,dec], keys=['Nov','Dec']).reset\_index()**

**temp\_df.merge(courses, on='course\_id').groupby('level\_0')['price'].sum()**

**# 3. Print the registration table**

**# cols -> name -> course -> price**

**stu = registered.merge(students, on='student\_id')**

**stu.merge(courses, on='course\_id')[['name','course\_name','price']]**

**# 4. Plot bar chart for revenue/course**

**registered.merge(courses, on='course\_id').groupby('course\_name')['price'].sum().plot(kind='bar')**

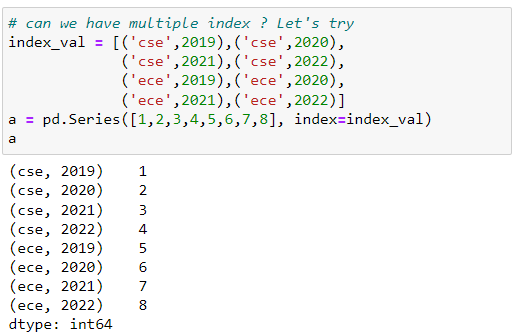
**# 5. find students who enrolled in both the monthsnov**

**common\_student\_id = np.intersect1d(nov['student\_id'],dec['student\_id'])**

**common\_student\_id**

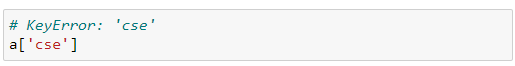
**S21: MULTIINDEX SERIES AND DATAFRAMES**

**MultiIndex Series**

****

## **Problem in MultiIndex Series**

* **What if we want to fetch values based on specific branch.**
* **Here branch and year is not independent then we can't access specific branch, so it will raise KeyError.**

****

## **MultiIndex Series (also known as Hierarchical Indexing)**

* **In pandas, a MultiIndex, also known as hierarchical indexing, allows you to have multiple index levels within single axis.**
* **This is particularly useful when you're working with higher-dimensional data that can be naturally represented as a hierarchical structure.**
* **MultiIndex can be applied to both rows (index) and columns, but we'll focus on row-based MultiIndexing here pd.MultiIndex is a class in the pandas library in Python that represents a multi-level or hierarchical index.**
* **It allows you to have multiple levels of indices on one axis, either for rows or columns in a DataFrame.**
* **syntax: pd.MultiIndex(levels=[level\_values1, level\_values2, ...],**

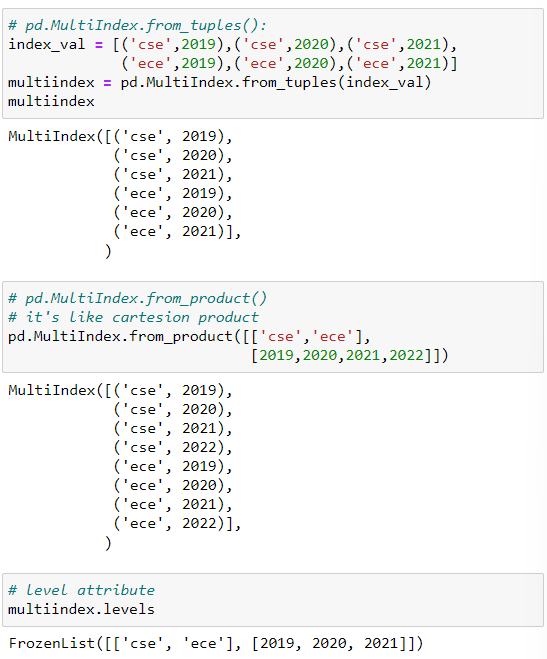
**codes=[code\_values1, code\_values2, ...],names=[name1, name2, ...])**

**parameters:**

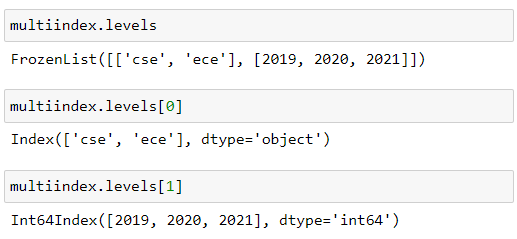
|  |  |
| --- | --- |
| **levels:** | **A list of arrays containing the unique values for each level of the MultiIndex.** |
| **codes:** | **A list of arrays containing the integer codes that represent the labels for each level. The codes are indices into the corresponding levels.** |
| **names:** | **A list of names for each level. Names are optional and are used to provide meaningful labels to the levels.** |

**Creating a MultiIndex (hierarchical index) object**

* **We can create a MultiIndex in several ways. One common way is to pass a list of arrays or tuples as the index when creating a DataFrame**
* **Also you can create a MultiIndex by using the pd.MultiIndex constructor or A MultiIndex can be created from a list of arrays (using MultiIndex.from\_arrays()), an array of tuples (using MultiIndex.from\_tuples()), a crossed set of iterables (using MultiIndex.from\_product()), or a DataFrame (using MultiIndex.from\_frame()).**
* **The Index constructor will attempt to return a MultiIndex when it is passed a list of tuples.**

****

**levels attribute:**

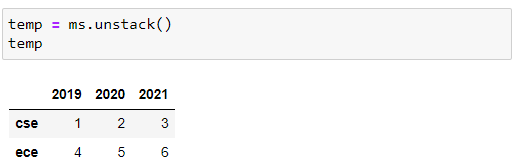
****

**Creating a series with MultiIndex objects**

****

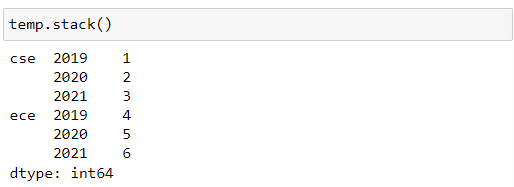
**unstack() function :**

**Convert the multiindex series into Dataframe**

****

**stack() function :**

**Convert the dataframe into multiindex series**

****

#### **Main purpose of MultiIndexing Objects:**

**To representation of high dimensional data into lower dimensions like 1d(Series) and 2D(DataFrame)**

**MultiIndex DataFrame:**

* **A multi-index DataFrame in pandas refers to a DataFrame that has multiple levels of indexing for both rows and columns.**
* **It is a way of handling higher-dimensional data by creating a hierarchical index structure.**

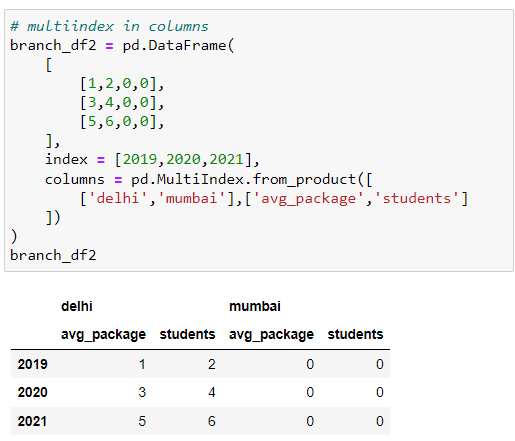
****

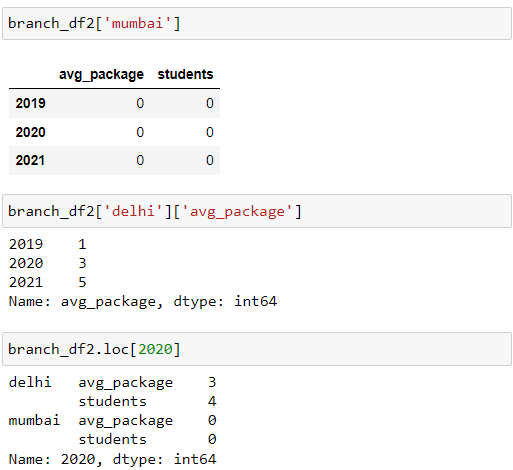
**Are columns really different from index?**

* **In pandas, if you transpose a DataFrame using the T attribute or the transpose() method, the columns effectively become the index and vice versa.**
* **After transposing, what were originally columns become the index, and what were originally the index becomes the columns.**

**MultiIndex DataFrame based on columns**

**The MultiIndex has two levels: the first level consists of city names ('delhi' and 'mumbai'), and the second level consists of attributes ('avg\_package' and 'students').**

****

****

**unstack() method:**

* **In simple words, unstack() method specified index labels becomes new columns and return new dataframe.**
* **It's used to pivot specified levels of the index labels into new columns, returning a new DataFrame.**
* **syntax: DataFrame.unstack(level=1, fill\_value=None)**
* **parameters:**

|  |  |
| --- | --- |
| **level:** | **(default is 1)**  **Specifies the level(s) of the index to unstack. If you have a MultiIndex, you can choose which level(s) you want to move to columns.** |
| **fill\_value:** | **If there are missing values after unstacking, you can specify a value to replace those missing values. By default, missing values are filled with NaN.** |

****

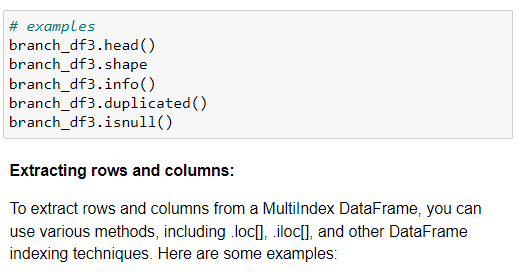
**stack() function :**

* **In simple words, Convert the dataframe into multiindex series**
* **Used to reshape a DataFrame by moving or pivoting specified levels of columns to become inner-most levels of the index.**
* **This operation results in a new DataFrame or Series with a multi-level index.**
* **Here's a simpler explanation:**
* **Imagine you have a table (DataFrame) where some information is stored in both rows and columns.**
* **The stack() method allows you to take information from the columns and move it to the rows, creating a new structure.**
* **If your columns have only one level (like a regular DataFrame), using stack() will give you a Series.**
* **If your columns have multiple levels, you can choose which levels to move to the index, and the result will be a new DataFrame with a multi-level index.**
* **In essence, stack() helps you transform data by rearranging it from a wide format (with information in columns) to a long format (with information in rows).**
* **This can be useful for certain types of analyses or when you need the data in a different structure.**

****

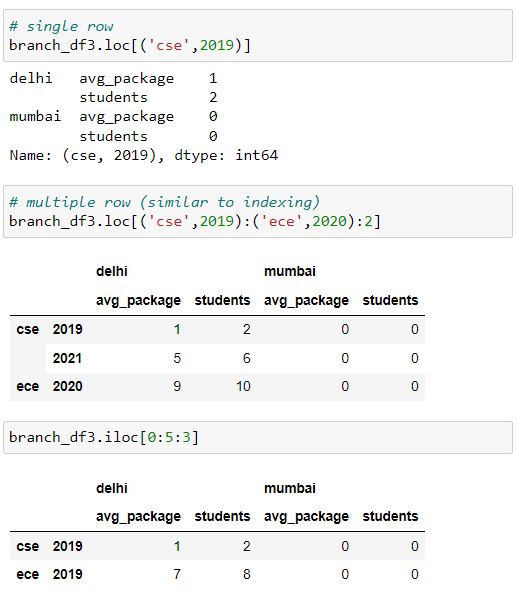
**Working with MultiIndex dataframe:**

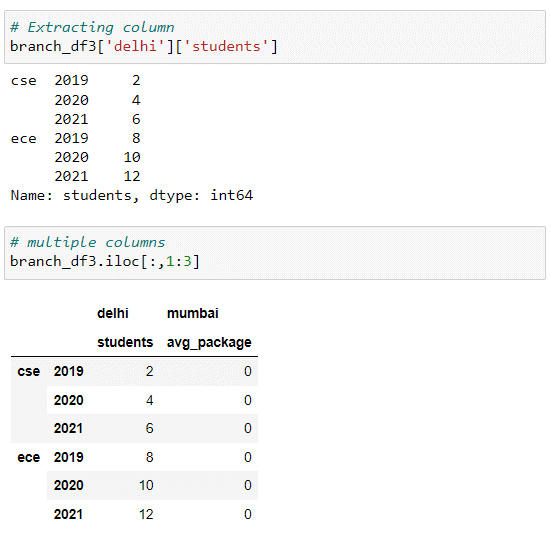
**We can use pandas DataFrame methods, functions, and attributes on a MultiIndex DataFrame just like you would on a regular DataFrame.**

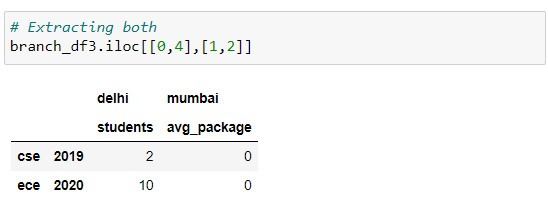
****

**Extracting rows and columns:**

**To extract rows and columns from a MultiIndex DataFrame, you can use various methods, including .loc[], .iloc[], and other DataFrame indexing techniques. Here are some examples:**

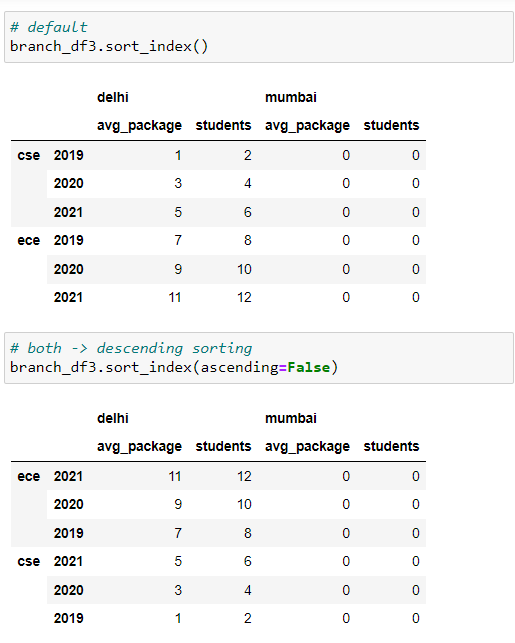
****

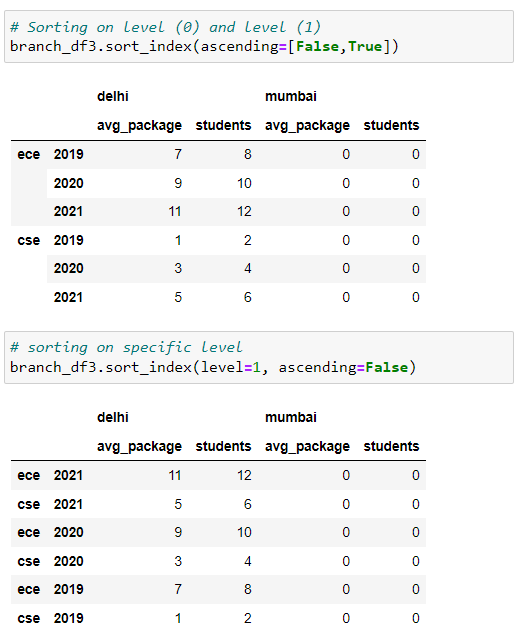


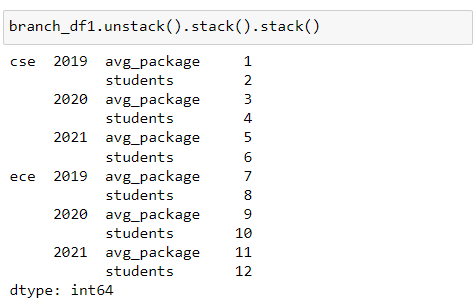
****

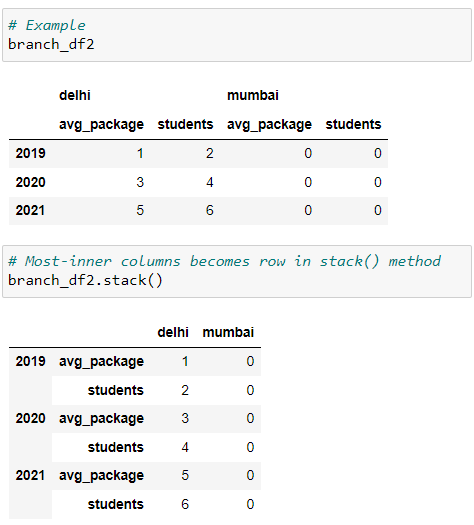
#### **Sorting Index in MultiIndex:**

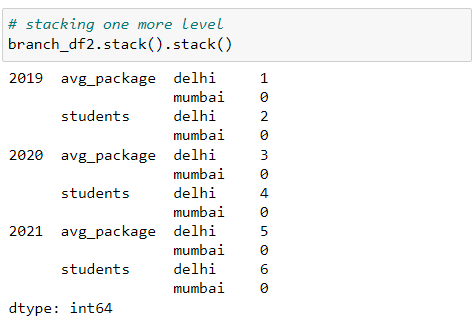
#### **Sorting the index in a MultiIndex DataFrame can be done using the sort\_index() method.**



****

****

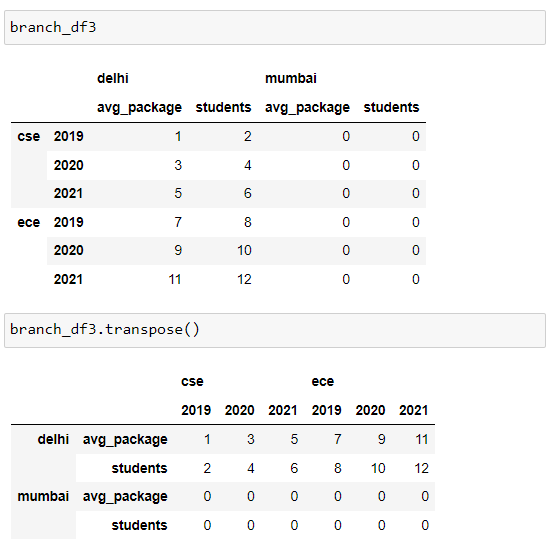
****

****

**Transpose MultiIndex DataFrame:**

**transpose() method:**

* **Used to transpose the rows and columns of a DataFrame.**
* **It switches the rows and columns, effectively converting the rows into columns and vice versa.**
* **Keep in mind that after transposing, the index becomes the columns, and the columns become the index.**
* **If your original DataFrame had column names, they would become the MultiIndex after transposing.**



**Wide format:**

* **Wide format is where we have a single row for every data point with multiple columns to hold the values of various attributes.**
* **A wide format contains values that do not repeat in the first column.**
* **Wide format data common sources of obtain data is:**

**Government Databases**

**CSV Files from Statistical Agencies**

**Machine Learning Datasets**

|  |  |  |
| --- | --- | --- |
| Name | Height | Weight |
| John | **160** | **67** |
| wick | **182** | **78** |

**Long format: (tidy data):**

* **Long format is where, for each data point we have as many rows as the number of attributes and each row contains the value of a particular attribute for a given data point.**
* **A long format contains values that do repeat in the first column.**
* **Long format data common sources of obtain data is:**

**Surveys and Questionnaires**

**Time seires data**

**Sensor data**

**Clinical Trials and Medical Studies**

|  |  |  |
| --- | --- | --- |
| Name | Attribute | Value |
| John | **Height** | **160** |
| John | **Weight** | **67** |
| wick | **Height** | **182** |
| wick | **Weight** | **78** |

**What difference between them?**

**Both are similar, but he choice between long and wide formats in data storage and analysis is often dependent on the problem statement, the nature of the data, and the specific analysis or tasks you plan to perform.**

|  |  |
| --- | --- |
| Wide format: | Long format: |
| |  |  |  | | --- | --- | --- | | Name | Height | Weight | | John | **160** | **67** | | wick | **182** | **78** | | |  |  |  | | --- | --- | --- | | Name | Attribute | Value | | John | **Height** | **160** | | John | **Weight** | **67** | | wick | **Height** | **182** | | wick | **Weight** | **78** | |

**melt() function:**

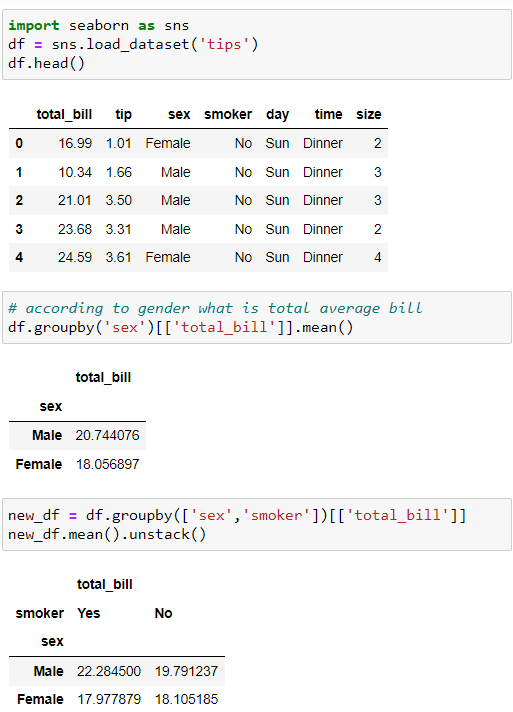
* **In pandas, melt() function was used to transform the dataset from a Wide format into a Long format.**
* **Syntax: pd.melt(frame, id\_vars, value\_vars, var\_name, value\_name='value', col\_level)**

**Parameters:**

|  |  |
| --- | --- |
| Frame: | DataFrame |
| value\_vars: | **[tuple, list, or ndarray, optional]**  **Column(s) to unpivot. If not specified, uses all columns that are not set as id\_vars.** |
| id\_vars: | **[tuple, list, or ndarray, optional]**  **Column(s) to use as identifier variables.** |
| var\_name: | **[scalar]**  **Name to use for the ‘variable’ column.**  **If None it uses frame.columns.name or ‘variable’.** |
| value\_name: | **[scalar, default ‘value’]**  **Name to use for the ‘value’ column.** |
| col\_level: | **[int or string, optional]**  **If columns are a MultiIndex then use this level to melt.** |

**Pivot Table:**

**The pivot table takes simple column-wise data as input and group the entries into a two-dimenational table that provide a multidimensional summarization of the data.**

****

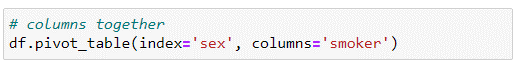
**pivot\_table() function:**

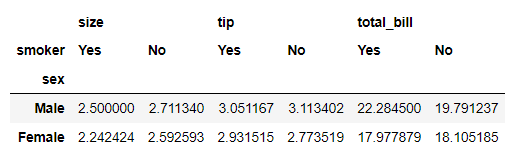
* **Used to create pivot tables from a DataFrame.**
* **It provides a flexible and powerful way to reshape and summarize data.**
* **syntax: df.pivot\_table(values=None, index=None, columns=None, aggfunc='mean', fill\_value=None, margins=False, dropna=True, margins\_name='All')**

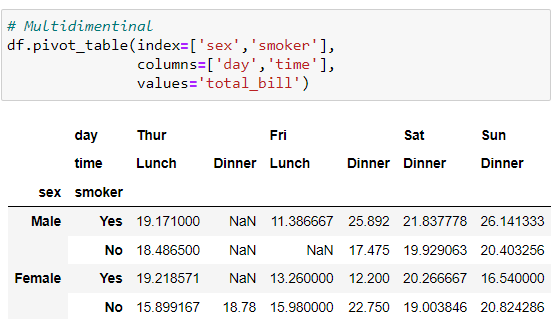
**parameters:**

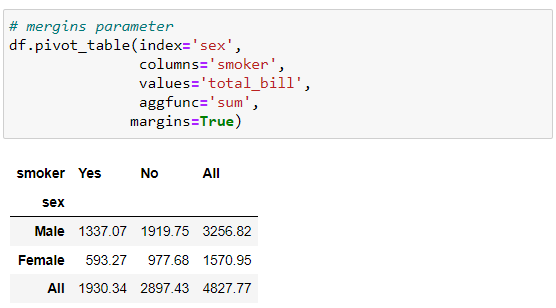
|  |  |
| --- | --- |
| values | This is the column that you want to aggregate. You can specify multiple columns if needed |
| index | **The column whose unique values will become the index of the pivot table** |
| columns | **The column whose unique values will become the columns of the pivot table** |
| aggfunc | **The aggregation function to apply. It could be 'mean', 'sum', 'count', 'min', 'max', etc. You can also pass a dictionary to apply different aggregation functions to different columns** |
| fill\_value | **A scalar value to replace missing values.** |
| margins | **If True, it adds all row/column margins (subtotals)** |
| dropna | **If True, it excludes NA/null values** |
| margins\_name | **Name of the row/column that will contain the totals when margins is True** |

****

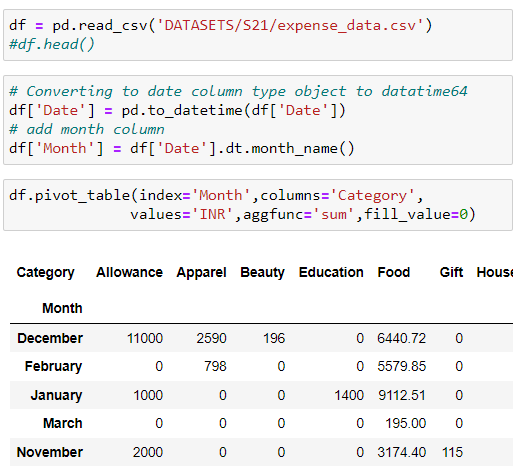


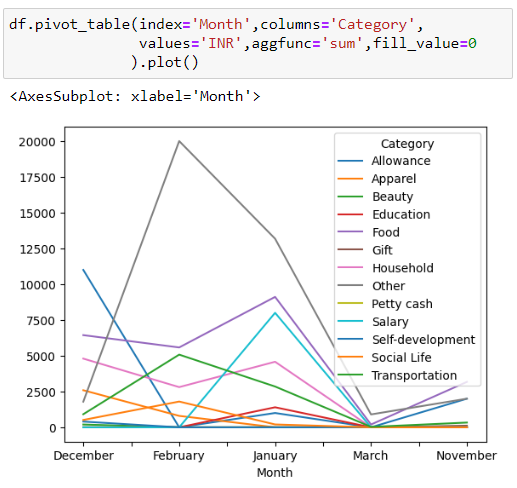
****

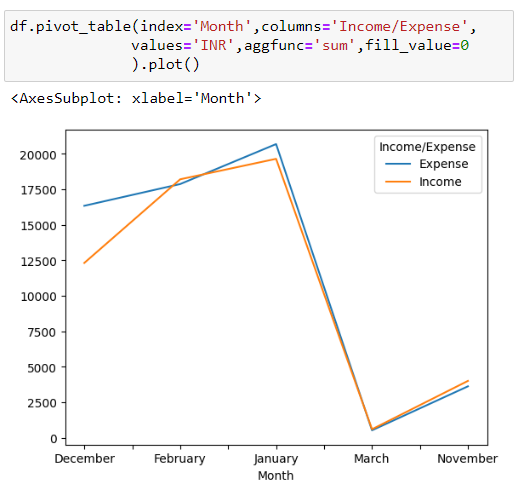
****

****

**Plotting graphs using pivot table:**

****



****

**TASK 21 EXTRA FUNCTION:**

**df.get\_level\_values()**

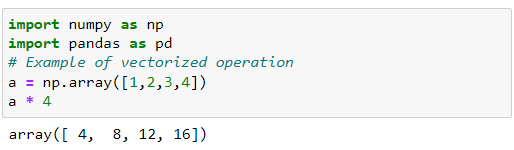
### S22: VECTORIZED STRING OPERATIONS & DATETIME

**Where we can apply this topic?**

**Apply on textual dataset like movies description datasets, customer’s reviews datasets, WhatsApp chat analysis review etc.**

**What are vectorized operations?**

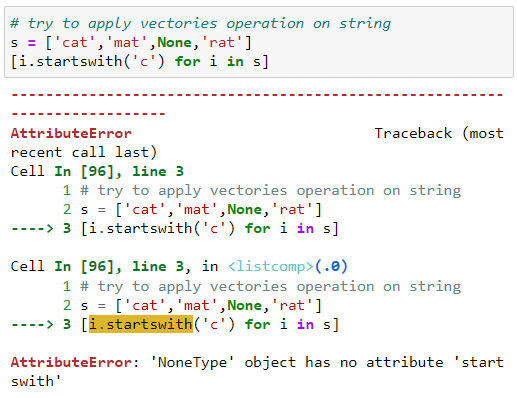
* **Here, a is a NumPy array with the elements [1, 2, 3, 4].**
* **The operation a \* 4 is a vectorized operation. It multiplies each element of the array a by 4.**
* **The result is a new NumPy array where each element is the product of the corresponding element in the original array a and the scalar value is 4.**



**Problems in vectorized operation in vanilla python**

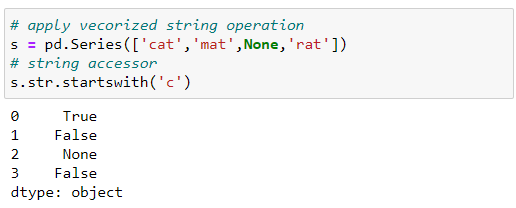
**vanilla python:**

* **It refers to the core, basic, or standard implementation of the Python programming language without any additional libraries or frameworks.**
* **It is the pure, unmodified form of Python, as defined by the Python Software Foundation.**
* **If we have None, missing values, and null values in datasets, in this scenario, Python functionality cannot handle.**
* **Vanilla Python may not be optimized for handling large datasets efficiently.**

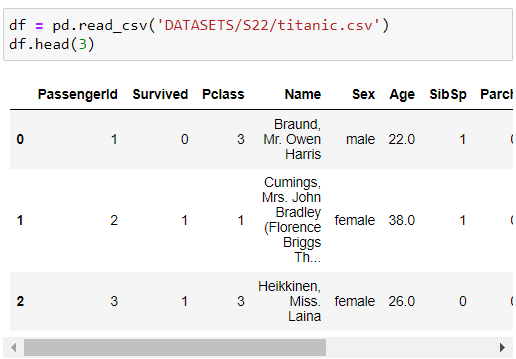
****

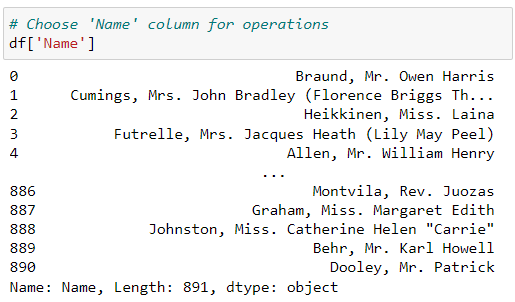
**How pandas solve this issue?**

* **To solve this above issue is use .str accessor**
* **In pandas, the .str accessor is used to perform vectorized string operations on a Pandas Series containing strings.**
* **This accessor provides a collection of methods that allow you to manipulate strings efficiently without using explicit loops.**
* **The .str accessor simplifies and speeds up the process of working with string data in Pandas DataFrames, making it a powerful tool for data cleaning and manipulation.**

****

**Apply vectorized string operations in titanic dataset**

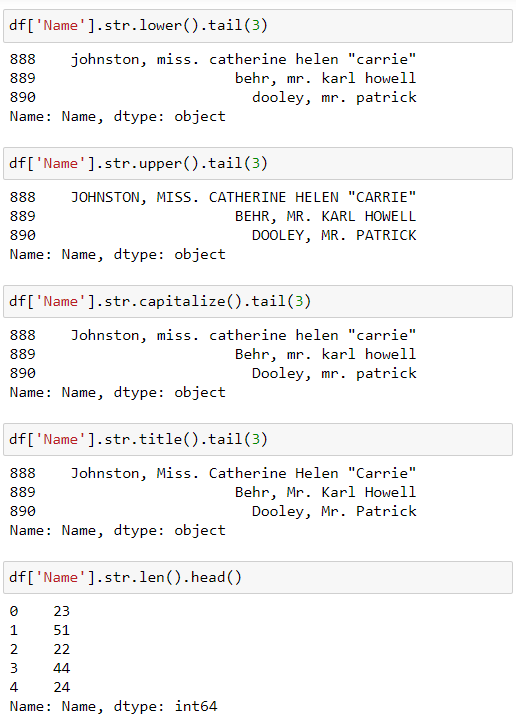
****

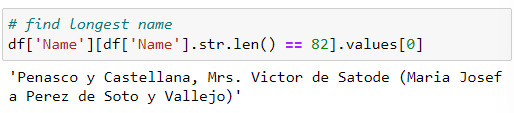
****

**Common .str methods/function:**

**Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method.**

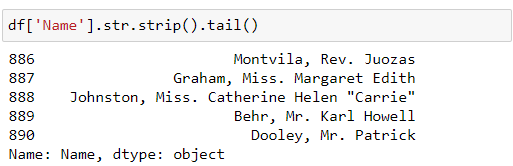
**.str.lower() / .str.upper() / .str.capitalize() / .str.title() / .str.len()**

****

****

**.str.strip() method:**

* **The .str.strip() method removes leading and trailing whitespace from each string in the Series.**
* **For the element with None and empty string, it becomes NaN after stripping.**
* **Useful in nlp projects.**

****

**.str.split() method**

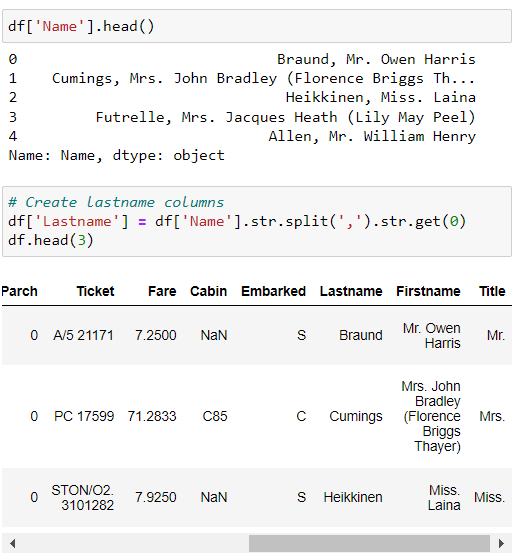
* **The .str.split() method splits each string in the Series into a list of substrings.**
* **If you want to split the string based on a specific delimiter, you can provide that delimiter as an argument to .str.split(). For example, to split based on a comma, you can use s.str.split(',').**
* **Syntax: .str.split(pat=None, n=-1, expand=False)**

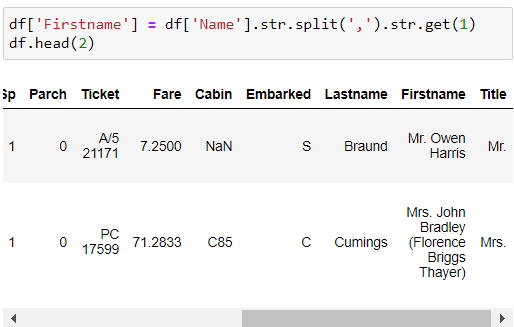
**Parameters:**

|  |  |
| --- | --- |
| n: | Specifies the maximum number of splits to perform.  By default, it is set to -1, which means there is no limit to the number of splits. |
| Expand: | **Controls whether to expand the result into a DataFrame.**  **If expand is set to True, the result will be a DataFrame with one column per split.**  **If False (the default), the result is returned as a Series of lists.** |

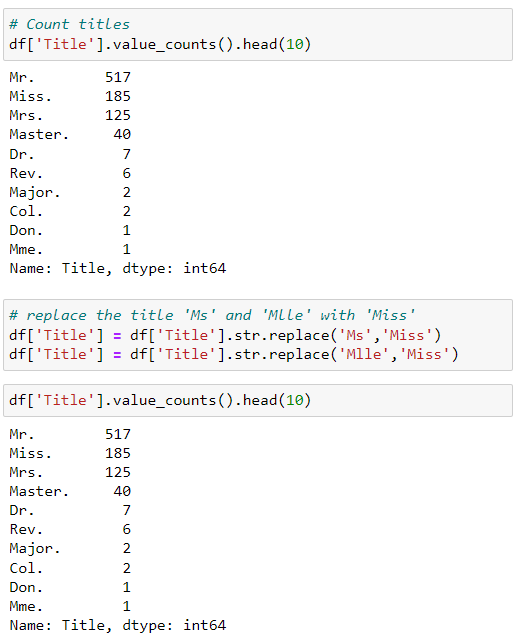
**.str.get() method:**

* **The .str.get() method in pandas is used to get the element at a specified position for each string in a Pandas Series of strings.**
* **It is a vectorized string method, meaning it operates on each element of the Series without requiring explicit loops.**

****

****

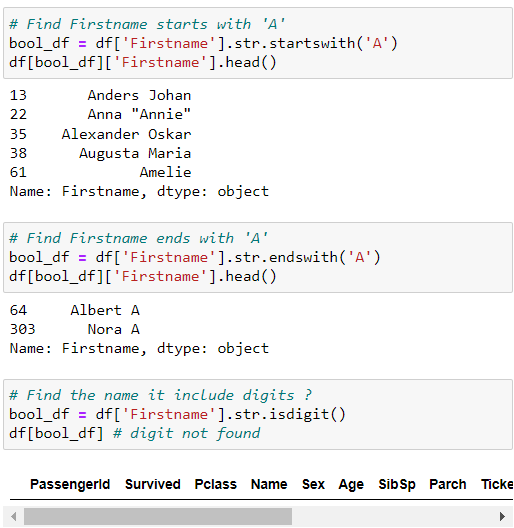
**.str.replace() method:**

****

**Filtering:**

**In pandas, vectorized string operations can be combined with boolean indexing to filter data based on string conditions efficiently.**

**.str.statswith() / .str.endswith() / .str.isalpha() / .str.isdigit**

****

**Advanced Filtering with regex and .str.contains() method:**

**.str.contains() method:**

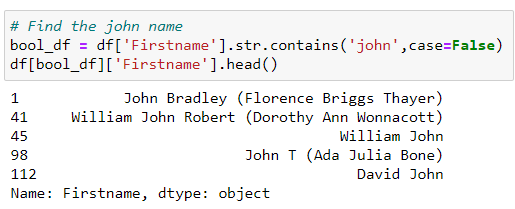
**The .str.contains() method in pandas is used to check if each element in a Pandas Series of strings contains a specified substring or matches a regular expression pattern.**

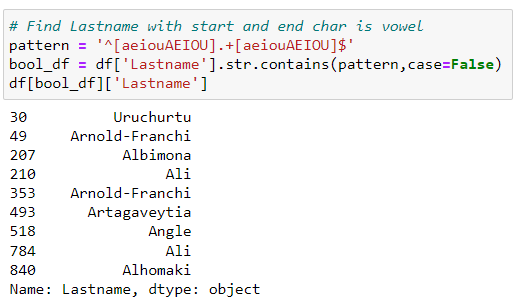
**This method returns a boolean mask indicating whether the condition is met for each element in the Series.**

**Syntax: .str.contains(pat, case=True, flags=0, na=nan, regex=True)**

**Parameters:**

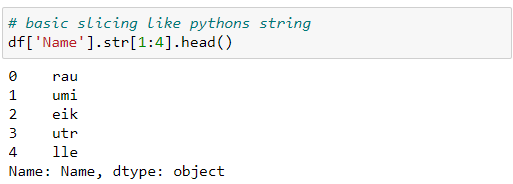
|  |  |
| --- | --- |
| **pat:** | **The substring or regular expression pattern to search for.** |
| **case:** | **If True, the matching is case-sensitive (default is True).** |
| **flags:** | **Additional flags for controlling the behavior of the regex (see the re module for options).** |
| **na:** | **The value to use for missing values (default is nan).** |
| **regex:** | **If True, treats the pat parameter as a regular expression (default is True).** |

****

****

**Slicing:**

* **In pandas, vectorized string operations allow you to perform slicing on each element of a Pandas Series of strings efficiently.**
* **You can use the .str accessor along with indexing or slicing to extract substrings based on position or conditions.**

****

**DATETIME**

**TIMESTAMP OBJECTS**

**What is Timestamp?**

* **Timestamp refers as particular moments in time (e.g., Oct 24th. 2022 at 7:00pm)**
* **Pandas library provide different datatype to store that datatype**

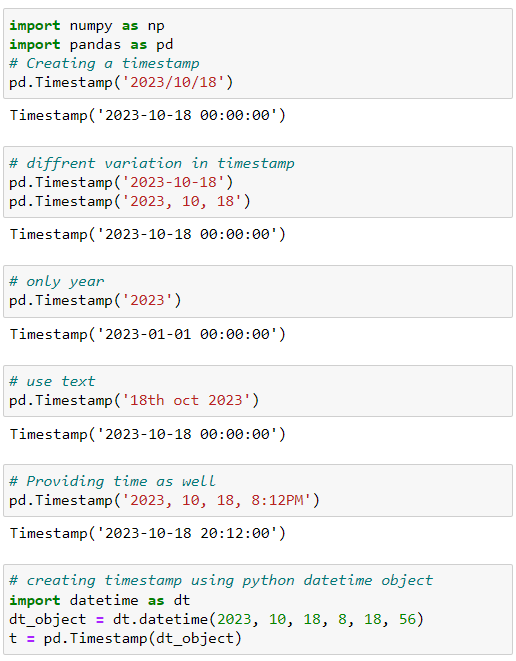
**What is Timestamp object?**

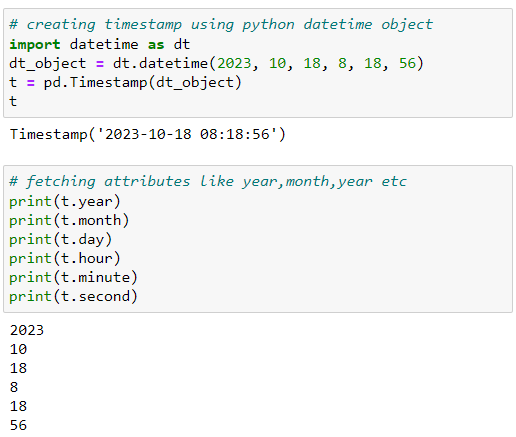
* **Timestamp object is part of the pandas library and is particularly useful when working with time series data.**
* **Patient health metrics, stock price changes, weather records, economic indicators, servers, networks, sensors, and applications performance monitoring are examples of time-series data.**
* **We can also called vectorized datetime operations or vectorized datetime functions.**

**Creating Timestamp object**

* **Create a Timestamp object using the pd.Timestamp constructor.**
* **It can handle various input formats, including strings, datetime objects, or even numeric values representing timestamps.**

**Note: Always follow this format: YYYY/MM/DD**

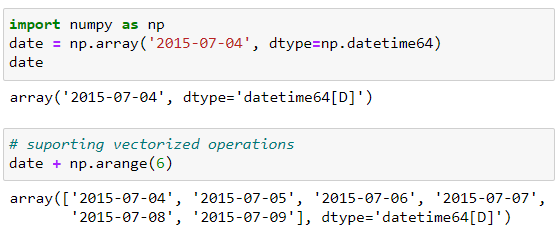
****

****

**(INTERVIEW QUESTION)**

**why separate objects to handle data and time when python already has datetime functionality?**

* **syntax wise datetime is very convenient**
* **But the performance takes a hit while working with huge data. List vs Numpy Array**
* **The weaknesses of Python's datetime format inspired the NumPy team to add a set of native time series data type to NumPy.**
* **The datetime64 dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly.**



* **Because of the uniform type in NumPy datetime64 arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's datetime objects, especially as arrays get large**
* **Pandas Timestamp object combines the ease-of-use of python datetime with the efficient storage and vectorized interface of numpy.datetime64**
* **From a group of these Timestamp objects, Pandas can construct a DatetimeIndex that can be used to index data in a Series or DataFrame**

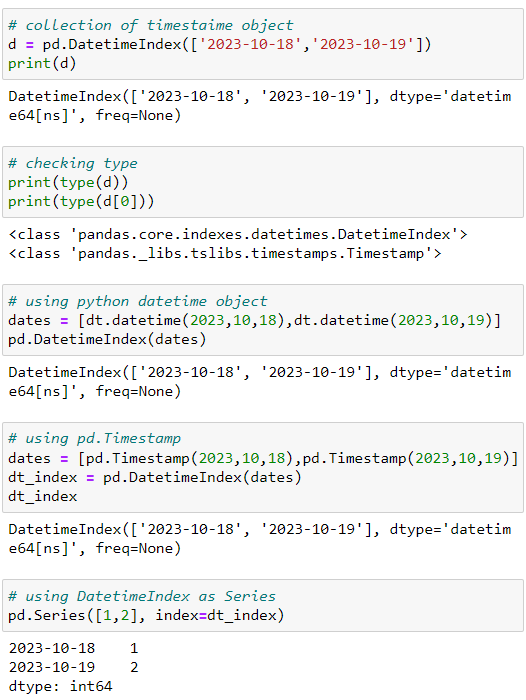
**DatetimeIndex Objects**

* **In simple words, A collection of pandas timestamp**
* **Pandas DatetimeIndex makes it easier to work with Date and Time data in our DataFrame.**
* **DatetimeIndex() can contain metadata related to date and timestamp and is a great way to deal with DateTime related data and do the calculations on data and time.**
* **Syntax:**

**pd.DatetimeIndex(data=None, freq=\_NoDefault.no\_default, tz=None, normalize=False, closed=None, ambiguous='raise', dayfirst=False, yearfirst=False, dtype=None, copy=False, name=None)**

**Parameters:**

|  |  |
| --- | --- |
| **data:** | **The data to be converted to datetime. It can be a list, array, or Series containing datetime-like objects, or a single datetime-like object.** |
| **freq:** | **The frequency of the datetime values if the data is not already a time series. It can be a string (e.g., 'D' for daily, 'H' for hourly) or a Timedelta object.** |
| **tz:** | **Timezone for the datetime values.** |
| **normalize:** | **If True, normalize the datetime values (set time to midnight).** |
| **closed:** | **Specify whether the interval is left-closed ('left'), right-closed ('right'), both closed ('both'), or neither closed ('neither').** |
| **ambiguoius:** | **How to handle daylight savings time ambiguities. Default is 'raise', but it can also be set to 'infer' or 'NaT'.** |
| **dayfirst:** | **If True, parse dates with the day first (e.g., '10/12/2023' is October 12, 2023).** |
| **yearfirst:** | **If True, parse dates with the year first (e.g., '2023-10-12' is October 12, 2023).** |
| **dtype:** | **The dtype of the datetime values.** |
| **copy:** | **If True, ensure that the input data is copied. Default is False.** |
| **name:** | **Name to be assigned to the resulting DatetimeIndex.** |

****

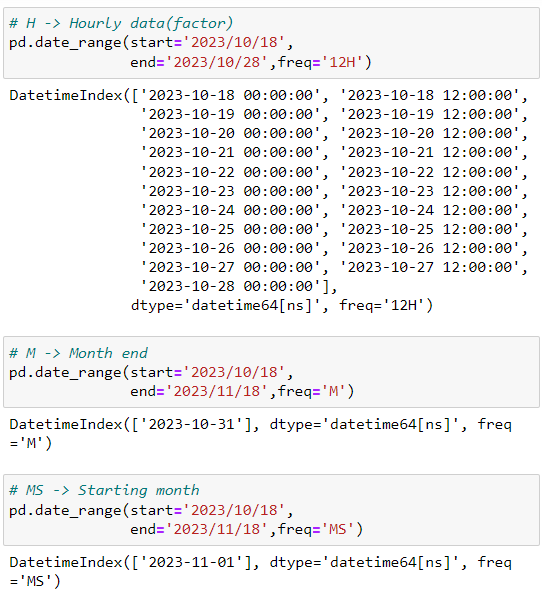
**date\_range() function**

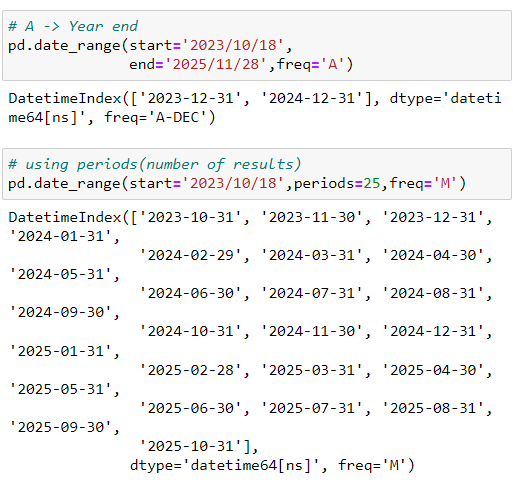
* **The pd.date\_range() function in Pandas is used to generate a fixed-frequency DatetimeIndex.**
* **It's a convenient method for creating date sequences for time-based data, such as time series data.**
* **The function allows you to specify the start date, end date, and frequency of the date range.**
* **Syntax: pd.date\_range(start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=None, \*\*kwargs)**

**Parameters:**

|  |  |
| --- | --- |
| **start:** | **The start date of the range.** |
| **end:** | **The end date of the range.** |
| **periods:** | **The number of periods (int) to generate.** |
| **freq:** | **Frequency of the resulting date sequence. This can be a string representing a frequency alias (e.g., 'D' for day, 'H' for hour), a Timedelta object, or a custom frequency string.** |
| **tz:** | **Timezone for the datetime values.** |
| **normalize:** | **If True, normalize the datetime values (set time to midnight).** |
| **name:** | **Name to be assigned to the resulting DatetimeIndex.** |
| **closed:** | **Specify whether the interval is left-closed ('left'), right-closed ('right'), both closed ('both'), or neither closed ('neither').** |
| **\*\*kwargs:** | **Additional keyword arguments that are passed to the underlying DatetimeIndex constructor.** |

****

****

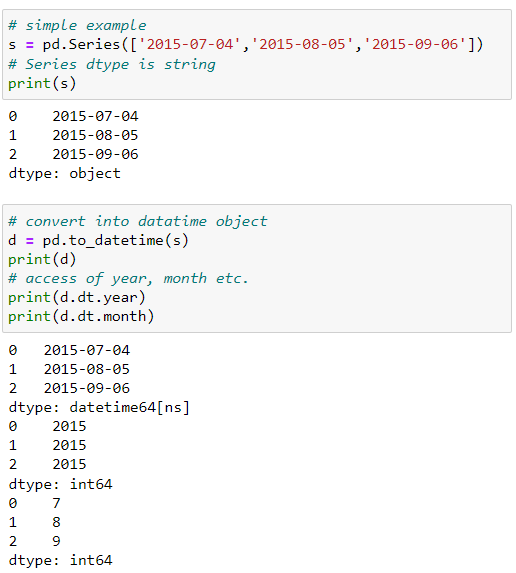
****

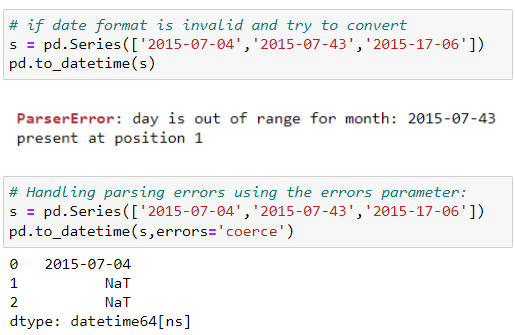
**to\_datetme() function:**

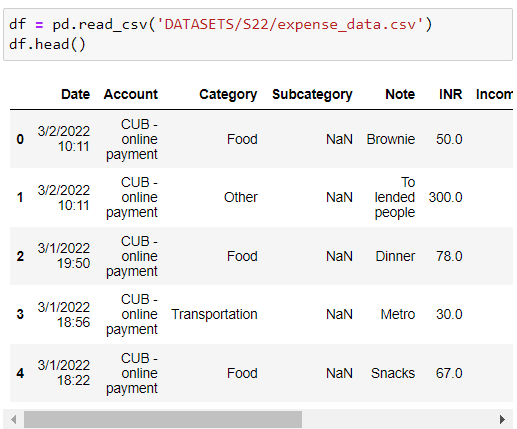
* **The pd.to\_datetime() function in Pandas is used to convert an object to a datetime.**
* **It can be used to convert a wide variety of input types, including strings, integers, floats, and other datetime-like objects, into Pandas datetime objects, such as Timestamp or DatetimeIndex.**
* **In simple words, Converts an existing objects to pandas timestamp/datetimeindex object**
* **Syntax: pd.to\_datetime(arg, errors='raise', format=None, dayfirst=False, yearfirst=False, utc=None, format2=None, exact=True, unit=None, infer\_datetime\_format=False, origin='unix', cache=False)**

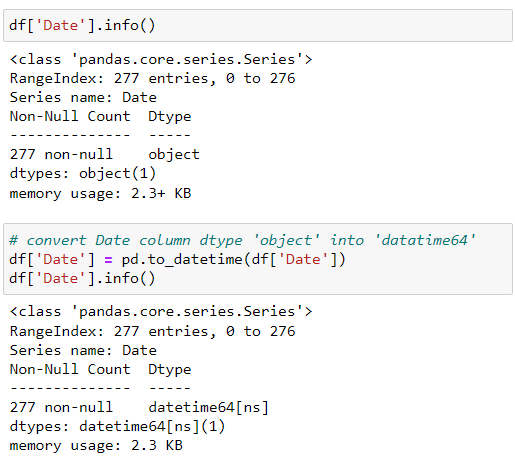
**Parameters:**

|  |  |
| --- | --- |
| **arg:** | **The object to be converted to a datetime. It can be a single value or an iterables (e.g., list, array, or Series).** |
| **errors:** | **How to handle parsing errors. It can be set to 'raise' (default), 'coerce' (to force errors to NaT), or 'ignore' (to skip errors).** |
| **format:** | **A format string to specify the exact format of the input data.** |
| **dayfirst:** | **If True, parse dates with the day first (e.g., '10/12/2023' is October 12, 2023).** |
| **yearfirst:** | **If True, parse dates with the year first (e.g., '2023-10-12' is October 12, 2023).** |
| **utc:** | **If True, return UTC datetime objects.** |
| **unit:** | **The unit of the input data if the input is numeric (e.g., 's' for seconds or 'ns' for nanoseconds).** |
| **Infer\_datetime\_format:** | **If True, infer the datetime format of the input data (can improve parsing performance).** |
| **origin:** | **A reference date for numeric time data (default is 'unix' for Unix timestamps).** |
| **cache:** | **Whether to cache the datetime conversion results for performance improvement.** |

****

****

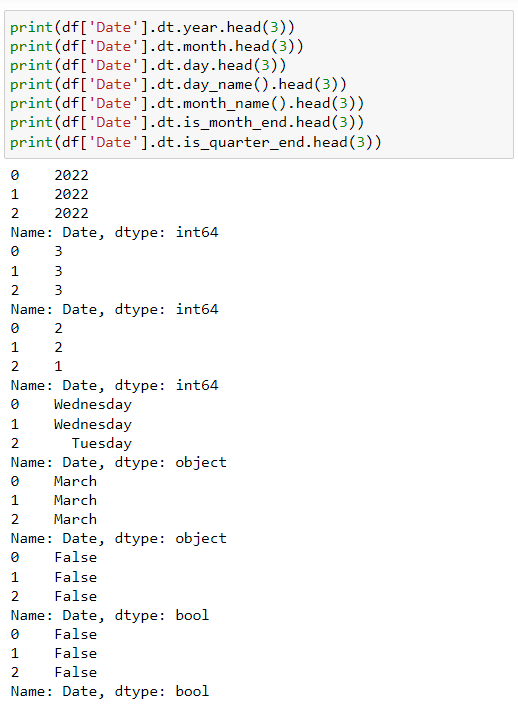
****

****

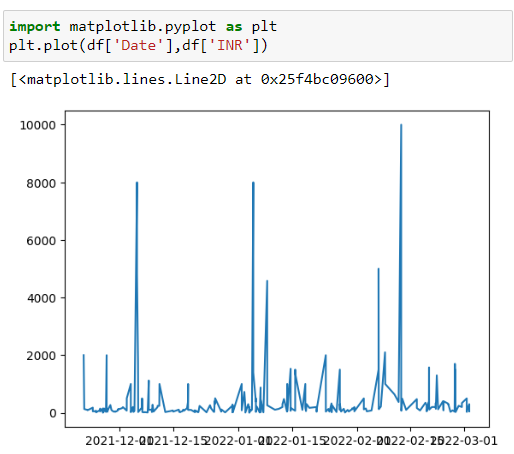
**dt accsessor:**

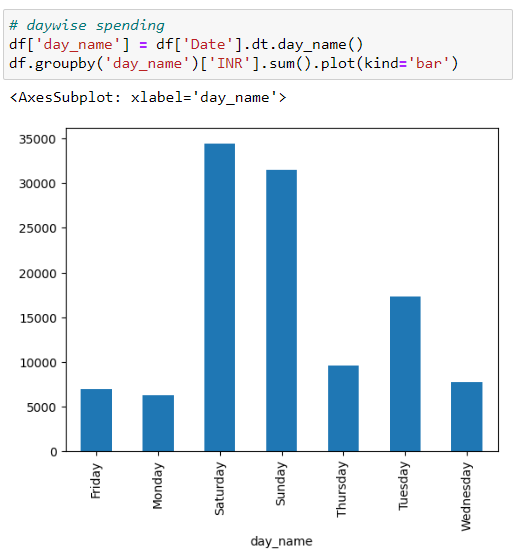
**In Pandas, the .dt accessor is used to access the datetime components of a Series or DataFrame.**

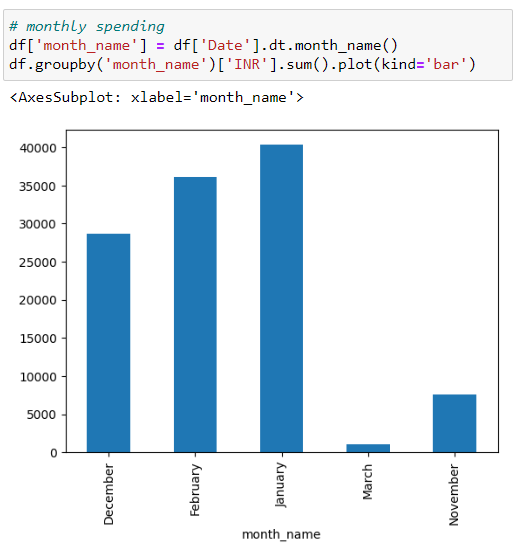
**The .dt accessor provides a convenient way to work with date and time components like year, month, day, hour, minute, second, etc.**



**Plotting Graphs:**





****