Pandas

- Pandas is a built in library using for data analysis. You'll be using Pandas heavily for data manipulation, visualisation, building machine learning models, etc.

- Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.
- There are two main data structures in Pandas - Series and Dataframes. The default way to store data is dataframes, and thus manipulating dataframes quickly is probably the most important skill set for data analysis.

  Source: https://pandas.pydata.org/pandas-docs/stable/overview.html

Pandas Series

- A series is similar to a 1-D numpy array, and contains values of the same type (numeric, character, datetime etc.). A dataframe is simply a table where each column is a pandas series.
- creating series
  - List
  - Tuple
  - Dictionary
  - Numpy
  - Date_Range
- Series Indexing

Creating Pandas Series

In [4]:

```
# by using List
li = [23,45,56,78,89]
se1 = pd.Series(li)
se1
# 0 - 4 indicates that Index values
# index starts from 0  to (n-1)
# n --- rows
```

Out[4]:

```
0    23
1    45
2    56
3    78
4    89
dtype: int64
```

In [6]:

```
type(se1)
```

Out[6]:

```
pandas.core.series.Series
```

In [7]:

```
se1.dtype
```

Out[7]:

```
dtype('int64')
```

In [9]:

```
# by using tuple
tu  = (23,45,5,676,878.67, 67.3)
se2 = pd.Series(tu)
se2
# numpy and series are having same data type
```

Out[9]:

```
0     23.00
1     45.00
2      5.00
3    676.00
4    878.67
5     67.30
dtype: float64
```

In [11]:

```
tu  = (23,45,5,676,878.67, 67.3,"APSSDC")
se3 = pd.Series(tu)
se3
```

Out[11]:

```
0        23
1        45
2         5
3       676
4    878.67
5      67.3
6    APSSDC
dtype: object
```

In [13]:

```
se3.dtype   # "o" --- object
```

Out[13]:

```
dtype('O')
```

In [15]:

```
# explicit indexing
se3.index = np.arange(100,107)
se3
```

Out[15]:

```
100        23
101        45
102         5
103       676
104    878.67
105      67.3
106    APSSDC
dtype: object
```

In [23]:

```
# by using Dict
di = {"a":245, "t":56,"o":567,657:789,67.67:"SDC"}
se4 = pd.Series(di, index = ["a",657])
se4
# every key acts as index value
```

Out[23]:

```
a      245
657    789
dtype: object
```

In [22]:

```
# by using numpy
num = np.array([23,45,56,87])
se5 = pd.Series(num, index = ["a","s",23.45,89])
se5
```

Out[22]:

```
a        23
s        45
23.45    56
89       87
dtype: int32
```

In [47]:

```
# data can be scalar,
se6 = pd.Series("Sai Pavan", index = ["vij","gun","vizag"])
se6
```

Out[47]:

```
vij      Sai Pavan
gun      Sai Pavan
vizag    Sai Pavan
dtype: object
```

Task

- **Create Pandas series object having 10 to 20 index values, data values are cube of index values**

In [26]:

```
index = list(range(10,21))
data = [i**3 for i in index]
s = pd.Series(data, index=index)
s
```

Out[26]:

```
10    1000
11    1331
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
18    5832
19    6859
20    8000
dtype: int64
```

In [39]:

```
se7 = pd.Series(np.arange(10,21)**3 , index = range(10,21))
se7
```

Out[39]:

```
10    1000
11    1331
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
18    5832
19    6859
```

```
19    6859
20    8000
dtype: int32
```

Pandas Series Indexing

In [40]:

```
se7[10]   # accessing single element
```

Out[40]:

```
1000
```

In [35]:

```
se7
```

Out[35]:

```
10    1000
11    1331
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
18    5832
19    6859
20    8000
dtype: int32
```

In [42]:

```
se7[12:]
```

Out[42]:

```
Series([], dtype: int32)
```

In [41]:

```
se7[2:8]   # explict slicing
```

Out[41]:

```
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
dtype: int32
```

In [43]:

```
se7[10 ] # implict slicing
```

Out[43]:

```
1000
```

In [44]:

```
se7[0:10:2]
```

Out[44]:

```
10    1000
12    1728
14    2744
16    4096
```

```
18    5832
dtype: int32
```

In [45]:

```
# 10, 11, 13, 17
se7[[10,11,13,17]]   # fancy slicing
```

Out[45]:

```
10    1000
11    1331
13    2197
17    4913
dtype: int32
```

In [46]:

```
# Series Masking

se7
```

Out[46]:

```
10    1000
11    1331
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
18    5832
19    6859
20    8000
dtype: int32
```

In [48]:

```
se6
```

Out[48]:

```
vij      Sai Pavan
gun      Sai Pavan
vizag    Sai Pavan
dtype: object
```

In [49]:

```
se6["vij"]
```

Out[49]:

```
'Sai Pavan'
```

In [52]:

```
#data > 1111 and data < 6000
se7[(se7 > 1111) & (se7 < 6000)]
```

Out[52]:

```
11    1331
12    1728
13    2197
14    2744
15    3375
16    4096
17    4913
18    5832
dtype: int32
```

Note : Series object having equal legth of index values and specified data values

In [53]:

```
# date range
dates = pd.date_range(start = "2020-11-16", end = "2020-11-24" )
dates
```

Out[53]:

```
DatetimeIndex(['2020-11-16', '2020-11-17', '2020-11-18', '2020-11-19',
               '2020-11-20', '2020-11-21', '2020-11-22', '2020-11-23',
               '2020-11-24'],
              dtype='datetime64[ns]', freq='D')
```

In [54]:

```
help(pd.date_range)
```

```
Help on function date_range in module pandas.core.indexes.datetimes:

date_range(start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=
None, closed=None, **kwargs) -> pandas.core.indexes.datetimes.DatetimeIndex
    Return a fixed frequency DatetimeIndex.

    Parameters
    ----------
    start : str or datetime-like, optional
        Left bound for generating dates.
    end : str or datetime-like, optional
        Right bound for generating dates.
    periods : int, optional
        Number of periods to generate.
    freq : str or DateOffset, default 'D'
        Frequency strings can have multiples, e.g. '5H'. See
        :ref:`here <timeseries.offset_aliases>` for a list of
        frequency aliases.
    tz : str or tzinfo, optional
        Time zone name for returning localized DatetimeIndex, for example
        'Asia/Hong_Kong'. By default, the resulting DatetimeIndex is
        timezone-naive.
    normalize : bool, default False
        Normalize start/end dates to midnight before generating date range.
    name : str, default None
        Name of the resulting DatetimeIndex.
    closed : {None, 'left', 'right'}, optional
        Make the interval closed with respect to the given frequency to
        the 'left', 'right', or both sides (None, the default).
    **kwargs
        For compatibility. Has no effect on the result.

    Returns
    -------
    rng : DatetimeIndex

    See Also
    --------
    DatetimeIndex : An immutable container for datetimes.
    timedelta_range : Return a fixed frequency TimedeltaIndex.
    period_range : Return a fixed frequency PeriodIndex.
    interval_range : Return a fixed frequency IntervalIndex.

    Notes
    -----
    Of the four parameters ``start``, ``end``, ``periods``, and ``freq``,
    exactly three must be specified. If ``freq`` is omitted, the resulting
    ``DatetimeIndex`` will have ``periods`` linearly spaced elements between
    ``start`` and ``end`` (closed on both sides).

    To learn more about the frequency strings, please see `this link
    <https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-alias
es>`
```

```
es>`__.

    Examples
    --------
    **Specifying the values**

    The next four examples generate the same `DatetimeIndex`, but vary
    the combination of `start`, `end` and `periods`.

    Specify `start` and `end`, with the default daily frequency.

    >>> pd.date_range(start='1/1/2018', end='1/08/2018')
    DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
                   '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
                  dtype='datetime64[ns]', freq='D')

    Specify `start` and `periods`, the number of periods (days).

    >>> pd.date_range(start='1/1/2018', periods=8)
    DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
                   '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
                  dtype='datetime64[ns]', freq='D')

    Specify `end` and `periods`, the number of periods (days).

    >>> pd.date_range(end='1/1/2018', periods=8)
    DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
                   '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
                  dtype='datetime64[ns]', freq='D')

    Specify `start`, `end`, and `periods`; the frequency is generated
    automatically (linearly spaced).

    >>> pd.date_range(start='2018-04-24', end='2018-04-27', periods=3)
    DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
                   '2018-04-27 00:00:00'],
                  dtype='datetime64[ns]', freq=None)

    **Other Parameters**

    Changed the `freq` (frequency) to ``'M'`` (month end frequency).

    >>> pd.date_range(start='1/1/2018', periods=5, freq='M')
    DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
                   '2018-05-31'],
                  dtype='datetime64[ns]', freq='M')

    Multiples are allowed

    >>> pd.date_range(start='1/1/2018', periods=5, freq='3M')
    DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
                   '2019-01-31'],
                  dtype='datetime64[ns]', freq='3M')

    `freq` can also be specified as an Offset object.

    >>> pd.date_range(start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3))
    DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
                   '2019-01-31'],
                  dtype='datetime64[ns]', freq='3M')

    Specify `tz` to set the timezone.

    >>> pd.date_range(start='1/1/2018', periods=5, tz='Asia/Tokyo')
    DatetimeIndex(['2018-01-01 00:00:00+09:00', '2018-01-02 00:00:00+09:00',
                   '2018-01-03 00:00:00+09:00', '2018-01-04 00:00:00+09:00',
                   '2018-01-05 00:00:00+09:00'],
                  dtype='datetime64[ns, Asia/Tokyo]', freq='D')

    `closed` controls whether to include `start` and `end` that are on the
    boundary. The default includes boundary points on either end.

    >>> pd.date_range(start='2017-01-01', end='2017-01-04', closed=None)
```

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed=None)
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')

Use ``closed='left'`` to exclude `end` if it falls on the boundary.

>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='left')
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'],
              dtype='datetime64[ns]', freq='D')

Use ``closed='right'`` to exclude `start` if it falls on the boundary.

>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='right')
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

In [55]:

```
import calendar
import time
import datetime
```

In [ ]:

```

```

In [ ]:

```

```

Data Analysis with Pandas

Dataframe is the most widely used data-structure in data analysis. It is a table with rows and columns, with rows having an index and columns having meaningful names.

- **Creating Pandas DataFrame**
- **File I/O (Importing CSV data files as pandas dataframes)**
- **Merging and Concatenating Dataframes**
  - **Merge multiple dataframes using common columns/keys using pd.merge()**
  - **Concatenate dataframes using pd.concat()**
- **Indexing and Selecting Data**
  - **Select rows from a dataframe**
  - **Select columns from a dataframe**
  - **Select subsets of dataframes**
  - **Position and Label Based Indexing: df.iloc and df.loc**
    - **You have seen some ways of selecting rows and columns from dataframes. Let's now see some other ways of indexing dataframes, which pandas recommends, since they are more explicit (and less ambiguous).**
    - **There are two main ways of indexing dataframes:**

```
* Position based indexing using df.iloc
* Label based indexing using df.loc
```

- **Grouping and Summarising Dataframes**
  - **Grouping and aggregation are some of the most frequently used operations in data analysis, especially while doing exploratory data analysis (EDA), where comparing summary statistics across groups of data is common.**
  - **Grouping analysis can be thought of as having three parts:**

    1. **Splitting the data into groups (e.g. groups of customer segments, product categories, etc.)**
    2. **Applying a function to each group (e.g. mean or total sales of each customer segment)**
    3. **Combining the results into a data structure showing the summary statistics**
- **Features**
- **Filtering**
- **Sorting**

- Sorting
- **Statistical**
- **Plotting**
- **Saving**

id	col1	col2
1	678	xyz
2	123	sdf
3	454	jhg

In [1]:

```
#
pip install pandas
```

```
Requirement already satisfied: pandas in c:\users\lavan\anaconda3\lib\site-packages (1.0.
5)Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: numpy>=1.13.3 in c:\users\lavan\anaconda3\lib\site-package
s (from pandas) (1.18.5)
Requirement already satisfied: python-dateutil>=2.6.1 in c:\users\lavan\anaconda3\lib\sit
e-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.2 in c:\users\lavan\anaconda3\lib\site-packages
(from pandas) (2020.1)
Requirement already satisfied: six>=1.5 in c:\users\lavan\anaconda3\lib\site-packages (fr
om python-dateutil>=2.6.1->pandas) (1.15.0)
```

In [3]:

```
import pandas as pd
import numpy as np
```

1. Creating Pandas DataFrame

In [57]:

```
# by using list
li = [[12,34],[34,56],[56,89],[100,109]]
df1 = pd.DataFrame(li)
df1
```

Out[57]:

	0	1
0	12	34
1	34	56
2	56	89
3	100	109

In [58]:

```
df1.shape   # (rows, columns)
```

Out[58]:

```
(4, 2)
```

In [60]:

```
tu = [("a",34),("b",56),("t",89),("y",109)]
df2 = pd.DataFrame(tu)
df2
```

Out[60]:

	0	1
0	a	34
1	b	56
2	t	89
3	y	109

In [61]:

```
df2.T # swaps rows and columns
```

Out[61]:

	0	1	2	3
0	a	b	t	y
1	34	56	89	109

In [62]:

```
df2.T.shape
```

Out[62]:

```
(2, 4)
```

In [63]:

```
df2
```

Out[63]:

	0	1
0	a	34
1	b	56
2	t	89
3	y	109

In [64]:

```
df2.columns = ["Murali","Raghava"]
df2
# columns and index starts from 0
```

Out[64]:

	Murali	Raghava
0	a	34
1	b	56
2	t	89
3	y	109

In [68]:

```
df2.index = ["a","b","c","d"]
df2
```

Out[68]:

	Murali	Raghava
a	a	34

	Murali	Raghava
b	b	56
c	t	89
d	y	109

In [75]:

```
tu = [("a",34),("b",56),("t",89),("y",109)]
df2 = pd.DataFrame(tu)
df2.index = list("stuw")
df2
```

Out[75]:

	Murali	Raghava
s	a	34
t	b	56
u	t	89
w	y	109

Task2

- **DF object having index 1 to 30 and data values squares, cubes**

In [70]:

```
index = list(range(1,31))
data = {'square':[i**2 for i in index],'cube':[i**3 for i in index]}
df = pd.DataFrame(data,index)
df
```

Out[70]:

	square	cube
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000
11	121	1331
12	144	1728
13	169	2197
14	196	2744
15	225	3375
16	256	4096
17	289	4913
18	324	5832
19	361	6859

	square	cube
20	400	8000
21	441	9261
22	484	10648
23	529	12167
24	576	13824
25	625	15625
26	676	17576
27	729	19683
28	784	21952
29	841	24389
30	900	27000

In [71]:

```
df3 = pd.DataFrame([{"squares" : i**2, "Cubes":i**3} for i in range(1,31)])
df3
```

Out[71]:

	squares	Cubes
0	1	1
1	4	8
2	9	27
3	16	64
4	25	125
5	36	216
6	49	343
7	64	512
8	81	729
9	100	1000
10	121	1331
11	144	1728
12	169	2197
13	196	2744
14	225	3375
15	256	4096
16	289	4913
17	324	5832
18	361	6859
19	400	8000
20	441	9261
21	484	10648
22	529	12167
23	576	13824
24	625	15625
25	676	17576
26	729	19683
27	784	21952

	squares	Cubes
28	841	24389
29	900	27000

In [76]:

```
t =[(23,5),(4,2),(78,"anu")]
df2=pd.DataFrame(t)
df2.index = list("ABD")
df2
```

Out[76]:

	0	1
A	23	5
B	4	2
D	78	anu

In [78]:

```
# by using Dict
di = { "Name"  : ["Anooja","Teja","Kiran","Himabindu"],
       "Color"  : ["Black","Green","Blue","White"],
        "Number" : [8,9,18,2]
}
df4 = pd.DataFrame(di)
df4
```

Out[78]:

	Name	Color	Number
0	Anooja	Black	8
1	Teja	Green	9
2	Kiran	Blue	18
3	Himabindu	White	2

In [ ]:

```
# columns / labels / features
# rows / records / observations
```

In [79]:

```
df4.columns
```

Out[79]:

```
Index(['Name', 'Color', 'Number'], dtype='object')
```

In [80]:

```
df4.index
```

Out[80]:

```
RangeIndex(start=0, stop=4, step=1)
```

In [81]:

```
di2 = [{"a":45,"b":657},{"c":456,"b":645}]
df5 = pd.DataFrame(di2)
df5
# missing value replaced by NaN(not a number)
```

Out[81]:

	a	b	c
0	45.0	657	NaN
1	NaN	645	456.0

2. File I/O

Reading

In [95]:

```
# Csv file to Dataframe
data_market = pd.read_csv("market_fact.csv")
data_market
```

Out[95]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.8100	0.01	23	-30.51	3.60	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.2700	0.01	13	4.56	0.93	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.90	2.50	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	14.30	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	26.30	
...	...	...	...	...	...	...	...	...	...	
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63	7.69	
8395	Ord_5411	Prod_6	SHP_7555	Cust_1798	127.1600	0.10	20	-74.03	6.92	
8396	Ord_5388	Prod_6	SHP_7524	Cust_1798	243.0500	0.02	39	-70.85	5.35	
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34	30.00	
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.39	4.86	

8399 rows × 10 columns

In [101]:

```
data_market.head(3)   # accessing default 5 recods
```

Out[101]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margi
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.5
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.5
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.5

In [97]:

```
data_market.tail() # last 5
```

Out[97]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_M
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63	7.69	
8395	Ord_5411	Prod_6	SHP_7555	Cust_1798	127.1600	0.10	20	-74.03	6.92	
8396	Ord_5388	Prod_6	SHP_7524	Cust_1798	243.0500	0.02	39	-70.85	5.35	
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34	30.00	

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_M
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.39	4.86	

In [99]:

```
data_market.sample()
```

Out[99]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Mar
2772	Ord_3239	Prod_9	SHP_4491	Cust_1205	2300.45	0.02	36	624.64	19.99	0

In [102]:

```
data_market.shape
```

Out[102]:

```
(8399, 10)
```

In [103]:

```
data_market.columns
```

Out[103]:

```
Index(['Ord_id', 'Prod_id', 'Ship_id', 'Cust_id', 'Sales', 'Discount',
       'Order_Quantity', 'Profit', 'Shipping_Cost', 'Product_Base_Margin'],
      dtype='object')
```

In [93]:

```
data = pd.read_excel("OCT  2020 GM and WATER.xlsx")
data
```

Out[93]:

	SAIRAM SRINIDHI GARDENS RESIDENTS WELFARE ASSOCIATION,SANGEETHA NAGAR, HYDERABAD	Unnamed: 1	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6	Unnamed: 7	Unnamed: 8	U
0	MONTH OF OCTOBER 2020 WATER MAINTENANCE...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
1	Flat No,	Bore water	NaN	NaN	Manjeera water	NaN	NaN	Total	Unit cost	
2	NaN	2020-01-10 00:00:00	2020-01-11 00:00:00	NET RE	2020-01-10 00:00:00	1/11/20	NET RE	NaN	NaN	
3	101	851.69	863.4	11.71	214.24	218.5	4.26	15.97	33.5	
4	102	545.56	556.2	10.64	61.48	62.5	1.02	11.66	33.5	
...	...	...	...	...	...	...	...	...	...	
89	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
90	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
91	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
92	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
93	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

94 rows × 15 columns

In [104]:

In [104]:

```
data_market.index
```

Out[104]:

```
RangeIndex(start=0, stop=8399, step=1)
```

In [105]:

```
len(data_market)
```

Out[105]:

```
8399
```

3. Merging and Concatenating Dataframes

In [112]:

```
# 2020 IPL Team

# 2019 IPL Team

IPL_2020 = { "IPL Team" : ["RCB","CSK","MI","DC","RR"],
             "Matches Played" : [20, 19,12,10,15],
             "Matches Win"   : [15, 14, 5,9,10]

}
df8 = pd.DataFrame(IPL_2020)
df8.set_index('IPL Team', inplace = True)
df8
```

Out[112]:

IPL Team	Matches Played	Matches Win
RCB	20	15
CSK	19	14
MI	12	5
DC	10	9
RR	15	10

In [113]:

```
IPL_2019 = { "IPL Team" : ["SRH", "RCB","CSK","DC","RR", "kkr"],
             "Matches Played" : [19,20, 19,12,10,15],
             "Matches Win"   : [18,15, 14, 5,9,10]

}
df9 = pd.DataFrame(IPL_2019)
df9.set_index("IPL Team", inplace = True)
df9
```

Out[113]:

IPL Team	Matches Played	Matches Win
SRH	19	18
RCB	20	15
CSK	19	14
DC	12	5
RR	10	9

kkr	Matches Played 15	Matches Win 10

Concatenating Dataframes Having the Same columns

In [114]:

```
# Simply add the two DFs using the add opearator
IPL = df8+df9
IPL
```

Out[114]:

	Matches Played	Matches Win
IPL Team		
CSK	38.0	28.0
DC	22.0	14.0
MI	NaN	NaN
RCB	40.0	30.0
RR	25.0	19.0
SRH	NaN	NaN
kkr	NaN	NaN

In [119]:

```
# The fill_value argument inside the df.add() function replaces all the NaN values
# in the two dataframes w.r.t. each other with zero.
IPL = df8.add(df9, fill_value = 0)
IPL
```

Out[119]:

	Matches Played	Matches Win
IPL Team		
CSK	38.0	28.0
DC	22.0	14.0
MI	12.0	5.0
RCB	40.0	30.0
RR	25.0	19.0
SRH	19.0	18.0
kkr	15.0	10.0

In [122]:

```
pd.concat([df8,df9]) # gives all records of both files
```

Out[122]:

	Matches Played	Matches Win
IPL Team		
RCB	20	15
CSK	19	14
MI	12	5
DC	10	9
RR	15	10

IPL Team	Matches Played	Matches Win
SRH	19	18
RCB	20	15
CSK	19	14
DC	12	5
RR	10	9
kkr	15	10

In [126]:

```python
pd.concat([df8,df9] , axis = 1)   # axis = 1 -- adding data at columns
```

Out[126]:

	Matches Played	Matches Win	Matches Played	Matches Win
RCB	20.0	15.0	20.0	15.0
CSK	19.0	14.0	19.0	14.0
MI	12.0	5.0	NaN	NaN
DC	10.0	9.0	12.0	5.0
RR	15.0	10.0	10.0	9.0
SRH	NaN	NaN	19.0	18.0
kkr	NaN	NaN	15.0	10.0

In [124]:

```python
pd.merge(df8,df9)   # common data of both files
```

Out[124]:

	Matches Played	Matches Win
0	20	15
1	19	14
2	12	5
3	10	9
4	15	10

In [135]:

```python
left_merged_file = pd.merge(df8,df9, how = "left")
# left  ---> common data of both files and also it gives left df entire  data
# right --- > common data of both files and also it gives right df entire  data
# inner ---> intersection
# outer --- > union
left_merged_file
#  use only keys from left frame
# left_merged_file.shape
```

Out[135]:

	Matches Played	Matches Win
0	20	15
1	19	14
2	12	5
3	10	9
4	15	10

In [128]:

In [120]:

```
help(pd.merge)
```

```
Help on function merge in module pandas.core.reshape.merge:

merge(left, right, how: str = 'inner', on=None, left_on=None, right_on=None, left_index: bool = False, right_index: bool = False, sort: bool = False, suffixes=('_x', '_y'), copy: bool = True, indicator: bool = False, validate=None) -> 'DataFrame'
    Merge DataFrame or named Series objects with a database-style join.

    The join is done on columns or indexes. If joining columns on
    columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes
    on indexes or indexes on a column or columns, the index will be passed on.

    Parameters
    ----------
    left : DataFrame
    right : DataFrame or named Series
        Object to merge with.
    how : {'left', 'right', 'outer', 'inner'}, default 'inner'
        Type of merge to be performed.

        * left: use only keys from left frame, similar to a SQL left outer join;
          preserve key order.
        * right: use only keys from right frame, similar to a SQL right outer join;
          preserve key order.
        * outer: use union of keys from both frames, similar to a SQL full outer
          join; sort keys lexicographically.
        * inner: use intersection of keys from both frames, similar to a SQL inner
          join; preserve the order of the left keys.
    on : label or list
        Column or index level names to join on. These must be found in both
        DataFrames. If `on` is None and not merging on indexes then this defaults
        to the intersection of the columns in both DataFrames.
    left_on : label or list, or array-like
        Column or index level names to join on in the left DataFrame. Can also
        be an array or list of arrays of the length of the left DataFrame.
        These arrays are treated as if they are columns.
    right_on : label or list, or array-like
        Column or index level names to join on in the right DataFrame. Can also
        be an array or list of arrays of the length of the right DataFrame.
        These arrays are treated as if they are columns.
    left_index : bool, default False
        Use the index from the left DataFrame as the join key(s). If it is a
        MultiIndex, the number of keys in the other DataFrame (either the index
        or a number of columns) must match the number of levels.
    right_index : bool, default False
        Use the index from the right DataFrame as the join key. Same caveats as
        left_index.
    sort : bool, default False
        Sort the join keys lexicographically in the result DataFrame. If False,
        the order of the join keys depends on the join type (how keyword).
    suffixes : tuple of (str, str), default ('_x', '_y')
        Suffix to apply to overlapping column names in the left and right
        side, respectively. To raise an exception on overlapping columns use
        (False, False).
    copy : bool, default True
        If False, avoid copy if possible.
    indicator : bool or str, default False
        If True, adds a column to output DataFrame called "_merge" with
        information on the source of each row.
        If string, column with information on source of each row will be added to
        output DataFrame, and column will be named value of string.
        Information column is Categorical-type and takes on a value of "left_only"
        for observations whose merge key only appears in 'left' DataFrame,
        "right_only" for observations whose merge key only appears in 'right'
        DataFrame, and "both" if the observation's merge key is found in both.

    validate : str, optional
        If specified, checks if merge is of specified type.

        * "one_to_one" or "1:1": check if merge keys are unique in both
```

```
        left and right datasets.
      * "one_to_many" or "1:m": check if merge keys are unique in left
        dataset.
      * "many_to_one" or "m:1": check if merge keys are unique in right
        dataset.
      * "many_to_many" or "m:m": allowed, but does not result in checks.

      .. versionadded:: 0.21.0

  Returns
  -------
  DataFrame
      A DataFrame of the two merged objects.

  See Also
  --------
  merge_ordered : Merge with optional filling/interpolation.
  merge_asof : Merge on nearest keys.
  DataFrame.join : Similar method using indices.

  Notes
  -----
  Support for specifying index levels as the `on`, `left_on`, and
  `right_on` parameters was added in version 0.23.0
  Support for merging named Series objects was added in version 0.24.0

  Examples
  --------

  >>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
  ...                     'value': [1, 2, 3, 5]})
  >>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
  ...                     'value': [5, 6, 7, 8]})
  >>> df1
      lkey value
  0   foo      1
  1   bar      2
  2   baz      3
  3   foo      5
  >>> df2
      rkey value
  0   foo      5
  1   bar      6
  2   baz      7
  3   foo      8

  Merge df1 and df2 on the lkey and rkey columns. The value columns have
  the default suffixes, _x and _y, appended.

  >>> df1.merge(df2, left_on='lkey', right_on='rkey')
    lkey  value_x rkey  value_y
  0  foo        1  foo        5
  1  foo        1  foo        8
  2  foo        5  foo        5
  3  foo        5  foo        8
  4  bar        2  bar        6
  5  baz        3  baz        7

  Merge DataFrames df1 and df2 with specified left and right suffixes
  appended to any overlapping columns.

  >>> df1.merge(df2, left_on='lkey', right_on='rkey',
  ...           suffixes=('_left', '_right'))
    lkey  value_left rkey  value_right
  0  foo           1  foo            5
  1  foo           1  foo            8
  2  foo           5  foo            5
  3  foo           5  foo            8
  4  bar           2  bar            6
  5  baz           3  baz            7

  Merge DataFrames df1 and df2, but raise an exception if the DataFrames have
```

```
any overlapping columns.

>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

In [131]:

```
# use only keys from right frame
right_merged_file = pd.merge(df8,df9, how = "right")
right_merged_file
```

Out[131]:

	Matches Played	Matches Win
0	20	15
1	19	14
2	12	5
3	10	9
4	15	10
5	19	18

In [132]:

```
# use intersection of keys from both frames
inner_merged_file = pd.merge(df8,df9, how = "inner")
inner_merged_file
```

Out[132]:

	Matches Played	Matches Win
0	20	15
1	19	14
2	12	5
3	10	9
4	15	10

In [133]:

```
# # use union of keys from both frames
outer_merged_file = pd.merge(df8,df9, how = "outer")
outer_merged_file
```

Out[133]:

	Matches Played	Matches Win
0	20	15
1	19	14
2	12	5
3	10	9
4	15	10
5	19	18

In [142]:

```
# Notice that
```

```python
print("IPL_2020 shape",df8.shape)
print("IPL_2019 shape",df9.shape)
print("left_merged_file shape ",left_merged_file.shape)
print("right_merged_file shape",right_merged_file.shape)
print("inner_merged_file shape",inner_merged_file.shape) # intersection
print("outer_merged_file shape",outer_merged_file.shape) # Union
```

```
IPL_2020 shape (5, 2)
IPL_2019 shape (6, 2)
left_merged_file shape  (5, 2)
right_merged_file shape (6, 2)
inner_merged_file shape (5, 2)
outer_merged_file shape (6, 2)
```

Task3:

- **Read all 5 market datasets using read_csv**
- **merge all files using pd.merge() Method and merge each file using common key name (use "on" attribute inside merge)**

In [ ]:

```python

```