Data Analysis using Pandas

What is Pandas?

Pandas is an opensource library that allows to you perform data manipulation in Python. This library is built on top of Numpy, meaning Pandas needs Numpy to operate. Pandas provide an easy way to create, manipulate and wrangle the data. Pandas is also an elegant solution for time series data.

This tool is essentially your data's home. Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it.

For example, say you want to explore a dataset stored in a CSV on your computer. Pandas will extract the data from that CSV into a DataFrame — a table, basically — then let you do things like:

- Calculate statistics and answer questions about the data.
- Clean the data by doing things like removing missing values and filtering rows or columns by some criteria.
- Visualize the data with help from Matplotlib. Plot bars, lines, histograms, bubbles, and more.
- Store the cleaned, transformed data back into a CSV, other file or database.

Why use pandas?

Pandas is a useful library in data analysis. It can be used to perform data manipulation and analysis. Pandas provide powerful and easy-to-use data structures, as well as the means to quickly perform operations on these structures.

- ❖ Easily handles missing data.
- ❖ It uses Series for one-dimensional data structure and DataFrame for multi-dimensional data structure.
- ❖ It provides an efficient way to slice the data.
- ❖ It provides a flexible way to merge, concatenate or reshape the data.

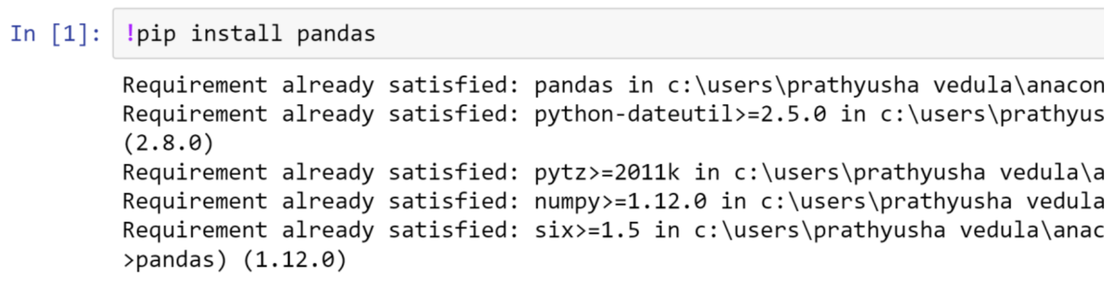
How to install and import Pandas?

Pandas is an easy package to install. Open up your terminal program (for Mac users) or command prompt (for PC users) and install it using either of the following commands:

```
conda install pandas
```

OR

```
pip install pandas
```



```
In [1]: !pip install pandas

        Requirement already satisfied: pandas in c:\users\prathyusha vedula\anacon
        Requirement already satisfied: python-dateutil>=2.5.0 in c:\users\prathyus
        (2.8.0)
        Requirement already satisfied: pytz>=2011k in c:\users\prathyusha vedula\a
        Requirement already satisfied: numpy>=1.12.0 in c:\users\prathyusha vedula
        Requirement already satisfied: six>=1.5 in c:\users\prathyusha vedula\anac
        >pandas) (1.12.0)
```

# The ! at the beginning runs cells as if they were in a terminal.

To import pandas we usually import it with a shorter name since it's used so much:

```
import pandas as pd
```

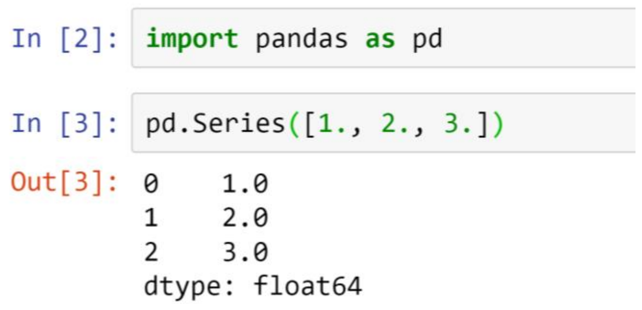
Core Components of Pandas?

The primary two components of pandas are the 1. Series and 2. DataFrame.

1.What is Series?

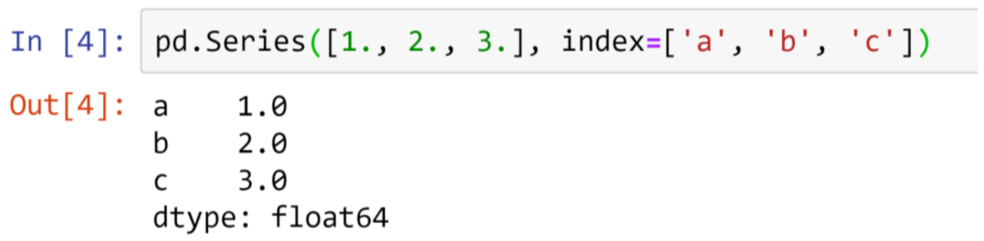
A series is a one-dimensional data structure. It can have any data structure like integer, float, and string. It is useful when you want to perform computation or return a one-dimensional array. A series, by definition, cannot have multiple columns. For the latter case, please use the data frame structure. Series has one parameters:

Data: can be a list, dictionary or scalar value



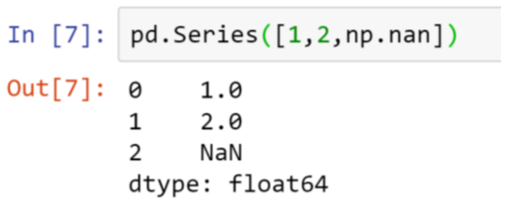
```
In [2]: import pandas as pd

In [3]: pd.Series([1., 2., 3.])

Out[3]: 0    1.0
        1    2.0
        2    3.0
        dtype: float64
```

You can add the index with index. It helps to name the rows. The length should be equal to the size of the column



```
In [4]: pd.Series([1., 2., 3.], index=['a', 'b', 'c'])

Out[4]: a    1.0
        b    2.0
        c    3.0
        dtype: float64
```

You can also create a Pandas series with a missing value for the third rows. Note, missing values in Python are noted "NaN." You can use numpy to create missing value: np.nan artificially



```
In [7]: pd.Series([1,2,np.nan])

Out[7]: 0    1.0
        1    2.0
        2    NaN
        dtype: float64
```

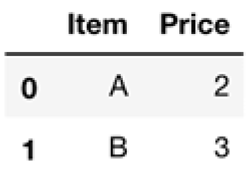
// here np is what we declare as numpy --> import numpy as np.

2.What is DataFrame?

A data frame is a two-dimensional array, with labelled axes (rows and columns). A data frame is a standard way to store data.

Data frame is well-known by statistician and other data practitioners. A data frame is a tabular data, with rows to store the information and columns to name the information. For instance, the price can be the name of a column and 2, 3, 4 the price values.

Below a picture of a Pandas data frame:



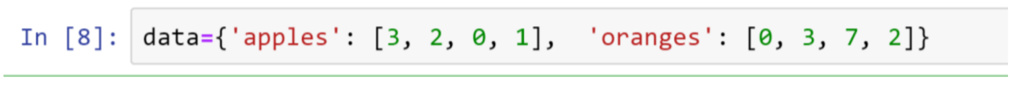
	Item	Price
0	A	2
1	B	3

2.1.How to create a DataFrame?

Creating DataFrames right in Python is good to know and quite useful when testing new methods and functions you find in the pandas docs.

There are many ways to create a DataFrame from scratch, but a great option is to just use a simple dict.

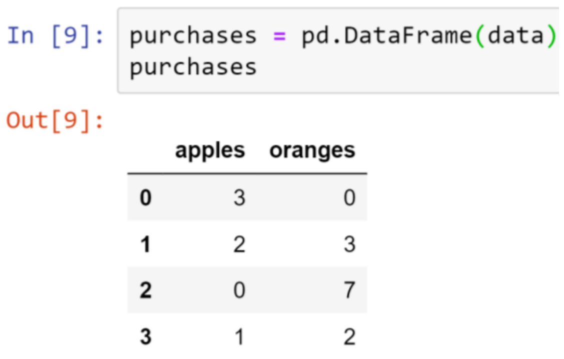
Let's say we have a fruit stand that sells apples and oranges. We want to have a column for each fruit and a row for each customer purchase. To organize this as a dictionary for pandas we could do something like:



```
In [8]: data={'apples': [3, 2, 0, 1],  'oranges': [0, 3, 7, 2]}
```

# Data is an object created to organize dictionary for pandas

Let’s now pass it to the pandas DataFrame constructor:



```
In [9]: purchases = pd.DataFrame(data)
        purchases
Out[9]:
```

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

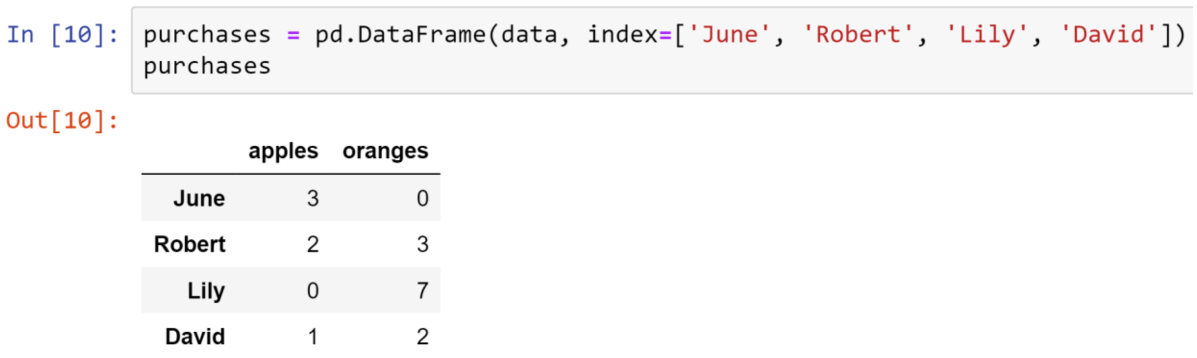
# Purchases is an object that has been created in order to pass pandas to dataframe

2.2.How did that work?

Each (*key, value*) item in data corresponds to a *column* in the resulting DataFrame.

The Index (an inbuilt function in Python, which searches for given element from start of the list and returns the lowest index where the element appears.) of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame.

Let's have customer names as our index:



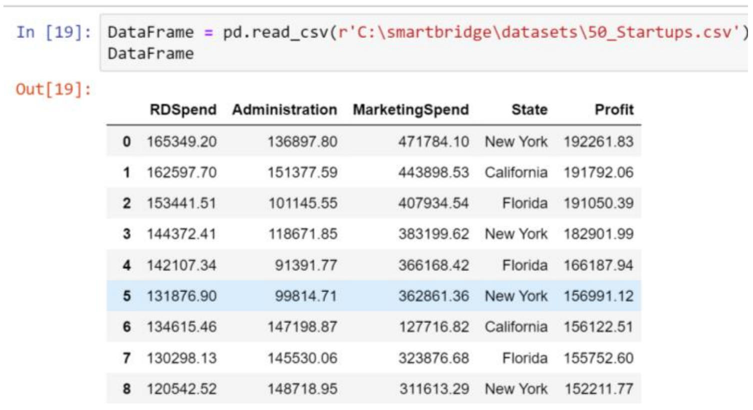
```
In [10]: purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])
         purchases
Out[10]:
```

	apples	oranges
June	3	0
Robert	2	3
Lily	0	7
David	1	2

How to read in data?

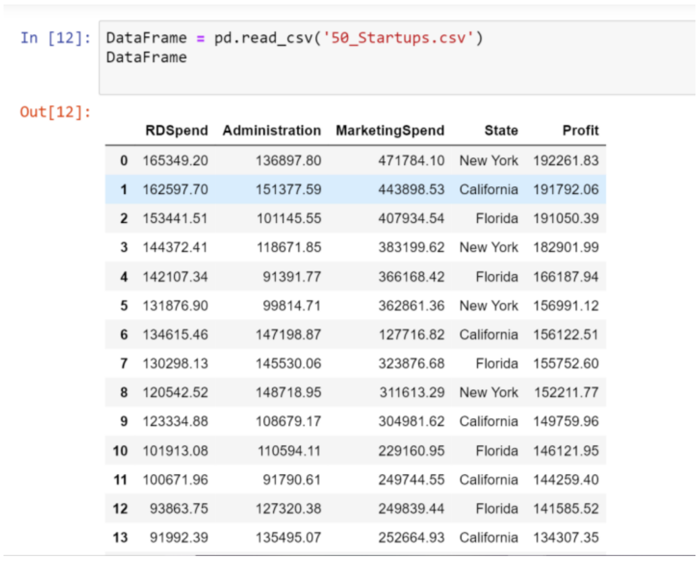
It’s quite simple to load data from various file formats into a DataFrame. In the following examples we'll use 50 start-ups data, but this time it's coming from various files.

With CSV files all you need is a single line to load in the data:



```
In [19]: DataFrame = pd.read_csv(r'C:\smartbridge\datasets\50_Startups.csv')
         DataFrame

Out[19]:
     RDSpend  Administration  MarketingSpend       State     Profit
0  165349.20       136897.80       471784.10    New York  192261.83
1  162597.70       151377.59       443898.53  California  191792.06
2  153441.51       101145.55       407934.54     Florida  191050.39
3  144372.41       118671.85       383199.62    New York  182901.99
4  142107.34        91391.77       366168.42     Florida  166187.94
5  131876.90        99814.71       362861.36    New York  156991.12
6  134615.46       147198.87       127716.82  California  156122.51
7  130298.13       145530.06       323876.68     Florida  155752.60
8  120542.52       148718.95       311613.29    New York  152211.77
```

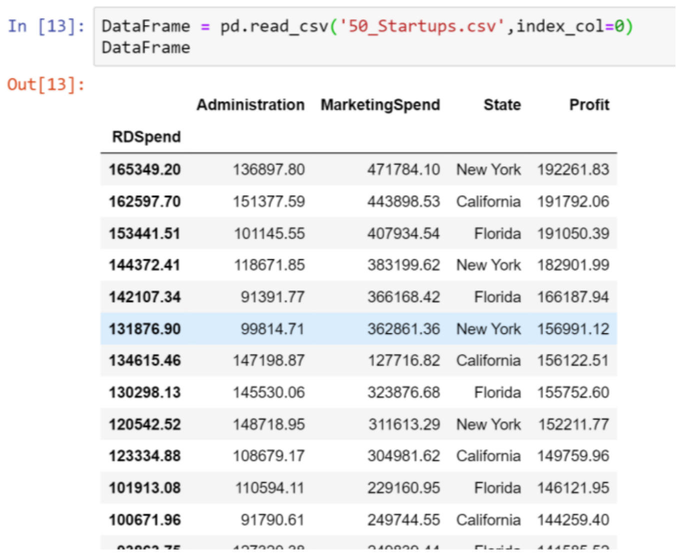
# When the program and the CSV files are not in the same folder we specify the path of the csv file present in that particular folder, as above.



```
In [12]: DataFrame = pd.read_csv('50_Startups.csv')
         DataFrame

Out[12]:
      RDSpend  Administration  MarketingSpend       State     Profit
0   165349.20       136897.80       471784.10    New York  192261.83
1   162597.70       151377.59       443898.53  California  191792.06
2   153441.51       101145.55       407934.54     Florida  191050.39
3   144372.41       118671.85       383199.62    New York  182901.99
4   142107.34        91391.77       366168.42     Florida  166187.94
5   131876.90        99814.71       362861.36    New York  156991.12
6   134615.46       147198.87       127716.82  California  156122.51
7   130298.13       145530.06       323876.68     Florida  155752.60
8   120542.52       148718.95       311613.29    New York  152211.77
9   123334.88       108679.17       304981.62  California  149759.96
10  101913.08       110594.11       229160.95     Florida  146121.95
11  100671.96        91790.61       249744.55  California  144259.40
12   93863.75       127320.38       249839.44     Florida  141585.52
13   91992.39       135495.07       252664.93  California  134307.35
```

# When the program and the CSV file is in the same folder, we simply read the csv file as mentioned in the above picture.

CSV don't have indexes like our DataFrames, so all we need to do is just designate the *index_col* when reading:



```
In [13]: DataFrame = pd.read_csv('50_Startups.csv',index_col=0)
         DataFrame
Out[13]:
```

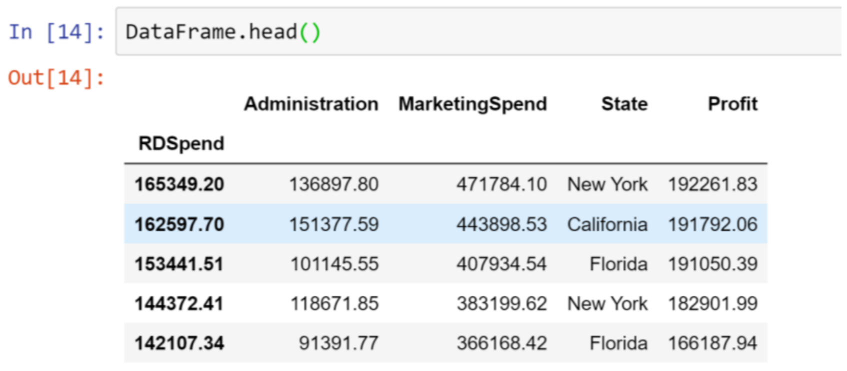
RDSpend	Administration	MarketingSpend	State	Profit
165349.20	136897.80	471784.10	New York	192261.83
162597.70	151377.59	443898.53	California	191792.06
153441.51	101145.55	407934.54	Florida	191050.39
144372.41	118671.85	383199.62	New York	182901.99
142107.34	91391.77	366168.42	Florida	166187.94
131876.90	99814.71	362861.36	New York	156991.12
134615.46	147198.87	127716.82	California	156122.51
130298.13	145530.06	323876.68	Florida	155752.60
120542.52	148718.95	311613.29	New York	152211.77
123334.88	108679.17	304981.62	California	149759.96
101913.08	110594.11	229160.95	Florida	146121.95
100671.96	91790.61	249744.55	California	144259.40

# Here we're setting the index to be column zero.

How do we inspect the data?

You can check the head or tail of the dataset with 1. head(), or 2 tail() preceded by the name of the panda's data frame.

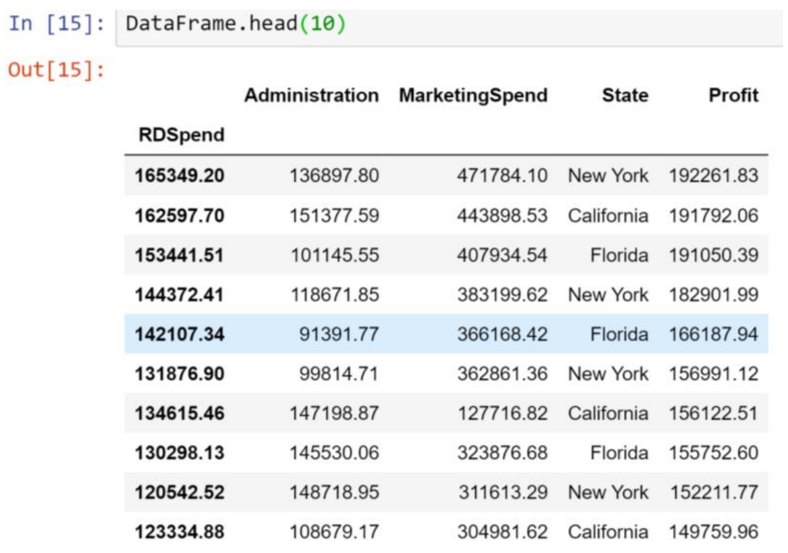
1. Head: head() by default show up the data of first 5 rows, as below:



```
In [14]: DataFrame.head()
Out[14]:
```

RDSpend	Administration	MarketingSpend	State	Profit
165349.20	136897.80	471784.10	New York	192261.83
162597.70	151377.59	443898.53	California	191792.06
153441.51	101145.55	407934.54	Florida	191050.39
144372.41	118671.85	383199.62	New York	182901.99
142107.34	91391.77	366168.42	Florida	166187.94

When we need the data of first 10 rows we can specify the number as parameter as below:

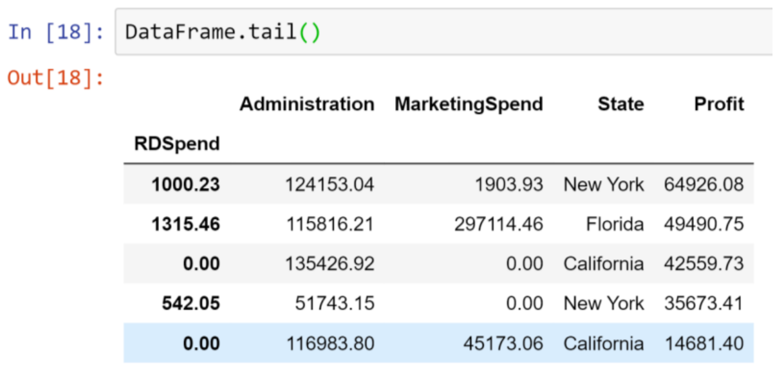


In [15]:
```
DataFrame.head(10)
```

Out[15]:

	Administration	MarketingSpend	State	Profit
RDSpend				
165349.20	136897.80	471784.10	New York	192261.83
162597.70	151377.59	443898.53	California	191792.06
153441.51	101145.55	407934.54	Florida	191050.39
144372.41	118671.85	383199.62	New York	182901.99
142107.34	91391.77	366168.42	Florida	166187.94
131876.90	99814.71	362861.36	New York	156991.12
134615.46	147198.87	127716.82	California	156122.51
130298.13	145530.06	323876.68	Florida	155752.60
120542.52	148718.95	311613.29	New York	152211.77
123334.88	108679.17	304981.62	California	149759.96

2. Tail: tail() by default show up the data of last 5 rows, as mentioned in the below image:

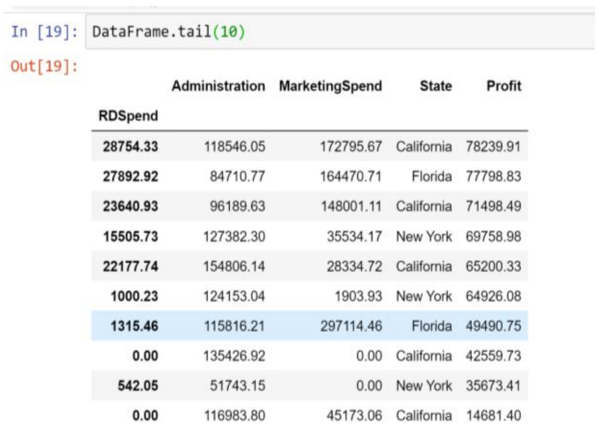


In [18]:
```
DataFrame.tail()
```

Out[18]:

	Administration	MarketingSpend	State	Profit
RDSpend				
1000.23	124153.04	1903.93	New York	64926.08
1315.46	115816.21	297114.46	Florida	49490.75
0.00	135426.92	0.00	California	42559.73
542.05	51743.15	0.00	New York	35673.41
0.00	116983.80	45173.06	California	14681.40

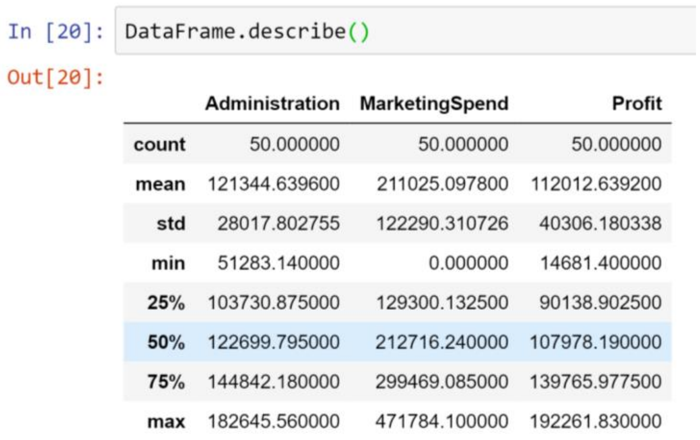
When we need the data of last 10 rows we can specify the number as a parameter as mentioned in the below image:



```
In [19]: DataFrame.tail(10)
Out[19]:
```

	Administration	MarketingSpend	State	Profit
RDSpend				
28754.33	118546.05	172795.67	California	78239.91
27892.92	84710.77	164470.71	Florida	77798.83
23640.93	96189.63	148001.11	California	71498.49
15505.73	127382.30	35534.17	New York	69758.98
22177.74	154806.14	28334.72	California	65200.33
1000.23	124153.04	1903.93	New York	64926.08
1315.46	115816.21	297114.46	Florida	49490.75
0.00	135426.92	0.00	California	42559.73
542.05	51743.15	0.00	New York	35673.41
0.00	116983.80	45173.06	California	14681.40

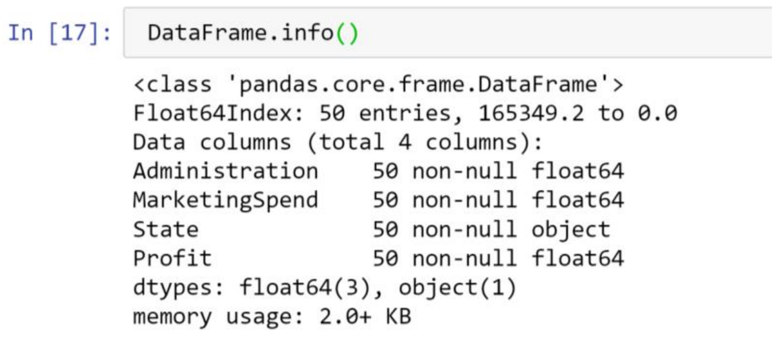
3. Describe: An excellent practice to get a clue about the data is to use describe(). It provides the counts, mean, std, min, max and percentile of the dataset.



```
In [20]: DataFrame.describe()
Out[20]:
```

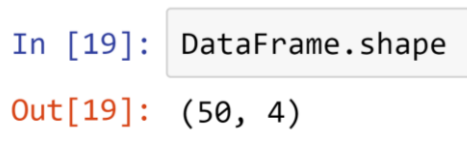
	Administration	MarketingSpend	Profit
count	50.000000	50.000000	50.000000
mean	121344.639600	211025.097800	112012.639200
std	28017.802755	122290.310726	40306.180338
min	51283.140000	0.000000	14681.400000
25%	103730.875000	129300.132500	90138.902500
50%	122699.795000	212716.240000	107978.190000
75%	144842.180000	299469.085000	139765.977500
max	182645.560000	471784.100000	192261.830000

4. Info: info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.



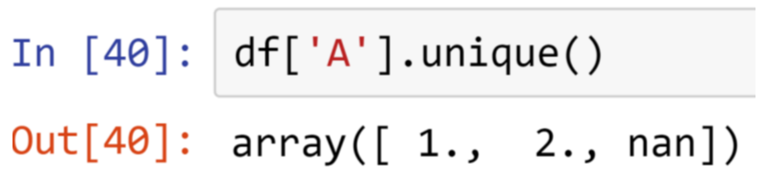
```
In [17]: DataFrame.info()

<class 'pandas.core.frame.DataFrame'>
Float64Index: 50 entries, 165349.2 to 0.0
Data columns (total 4 columns):
Administration     50 non-null float64
MarketingSpend     50 non-null float64
State              50 non-null object
Profit             50 non-null float64
dtypes: float64(3), object(1)
memory usage: 2.0+ KB
```

5. Shape: shape has no parentheses and is a simple tuple of format (rows, columns). So we have 50 rows and 4 columns in our movies DataFrame.



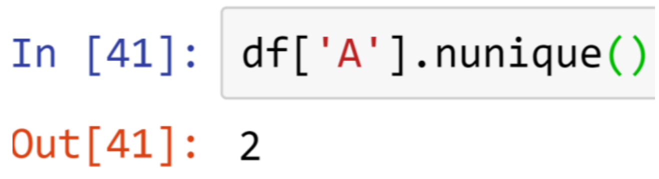
```
In [19]: DataFrame.shape
Out[19]: (50, 4)
```

6. Unique: unique() prints the unique values in the form of array.



```
In [40]: df['A'].unique()
Out[40]: array([ 1.,  2., nan])
```

7. Nunique: nunique() print sthe number of unique values in the data.

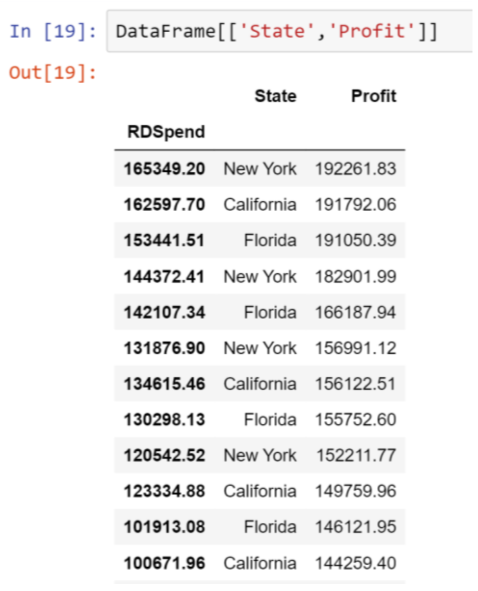


```
In [41]: df['A'].nunique()
Out[41]: 2
```

How do we slice the data?

1. Using name: You can use the column name to extract data in a particular column. To select multiple columns, you need to use two times the bracket, [[...,...]]

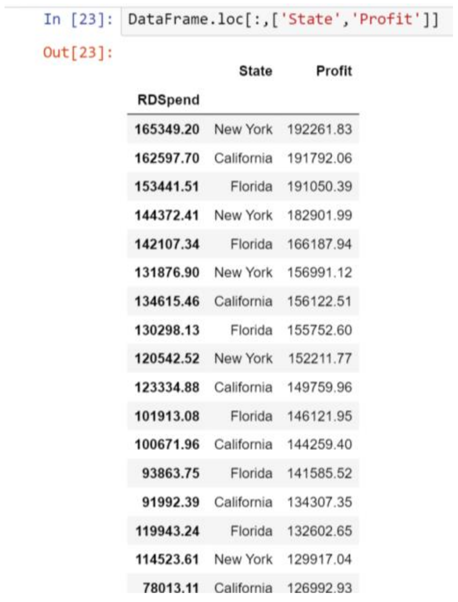
The first pair of bracket means you want to select columns, the second pairs of bracket tells what columns you want to return.



```
In [19]: DataFrame[['State','Profit']]
Out[19]:
                State     Profit
RDSpend
165349.20    New York  192261.83
162597.70  California  191792.06
153441.51     Florida  191050.39
144372.41    New York  182901.99
142107.34     Florida  166187.94
131876.90    New York  156991.12
134615.46  California  156122.51
130298.13     Florida  155752.60
120542.52    New York  152211.77
123334.88  California  149759.96
101913.08     Florida  146121.95
100671.96  California  144259.40
```

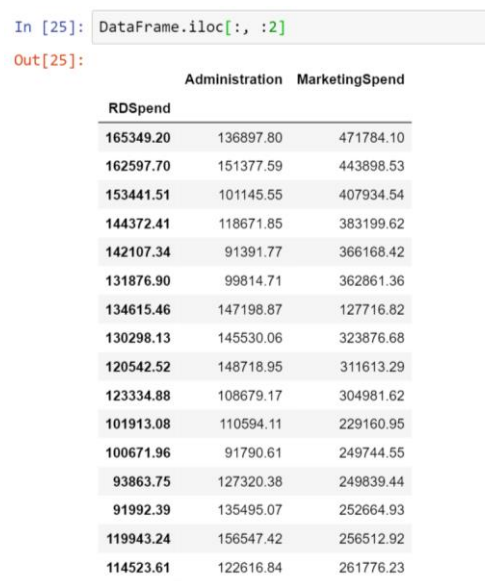
# By giving the specific column names, it will show up the data of those particular coulmns.

2. loc: The loc function is used to select columns by names. As usual, the values before the coma stand for the rows and after refer to the column. You need to use the brackets to select more than one column.



```
In [23]: DataFrame.loc[:,['State','Profit']]
Out[23]:
                State     Profit
RDSpend
165349.20    New York  192261.83
162597.70  California  191792.06
153441.51     Florida  191050.39
144372.41    New York  182901.99
142107.34     Florida  166187.94
131876.90    New York  156991.12
134615.46  California  156122.51
130298.13     Florida  155752.60
120542.52    New York  152211.77
123334.88  California  149759.96
101913.08     Florida  146121.95
100671.96  California  144259.40
 93863.75     Florida  141585.52
 91992.39  California  134307.35
119943.24     Florida  132602.65
114523.61    New York  129917.04
 78013.11  California  126992.93
```

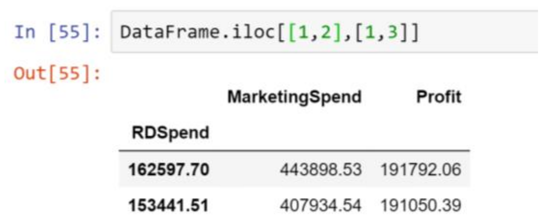
3. iloc: There is another method to select multiple rows and columns in Pandas. You can use iloc[]. This method uses the index instead of the columns name. The code below returns the same data frame as above



```
In [25]: DataFrame.iloc[:, :2]
Out[25]:
```

	Administration	MarketingSpend
RDSpend		
165349.20	136897.80	471784.10
162597.70	151377.59	443898.53
153441.51	101145.55	407934.54
144372.41	118671.85	383199.62
142107.34	91391.77	366168.42
131876.90	99814.71	362861.36
134615.46	147198.87	127716.82
130298.13	145530.06	323876.68
120542.52	148718.95	311613.29
123334.88	108679.17	304981.62
101913.08	110594.11	229160.95
100671.96	91790.61	249744.55
93863.75	127320.38	249839.44
91992.39	135495.07	252664.93
119943.24	156547.42	256512.92
114523.61	122616.84	261776.23

When we want some specific columns and rows we do it in the following manner:

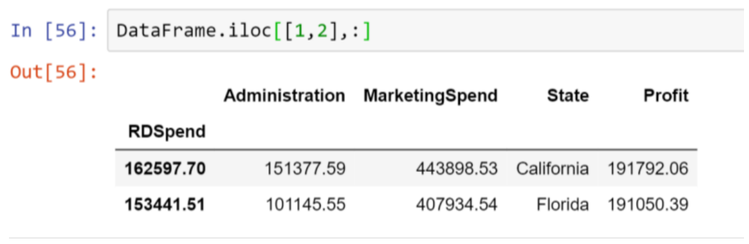


```
In [55]: DataFrame.iloc[[1,2],[1,3]]
Out[55]:
```

	MarketingSpend	Profit
RDSpend		
162597.70	443898.53	191792.06
153441.51	407934.54	191050.39

# here we have given a condition to print (1 and 2) rows and (1 and 3) columns.

When we want all the columns and specific rows to be printed we do it like:

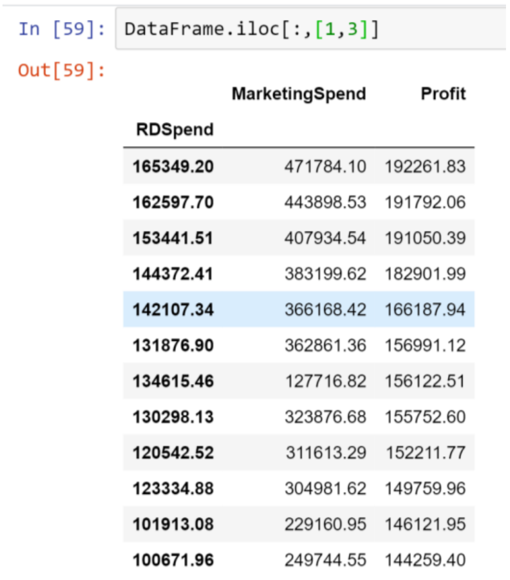


```
In [56]: DataFrame.iloc[[1,2],:]
Out[56]:
```

	Administration	MarketingSpend	State	Profit
RDSpend				
162597.70	151377.59	443898.53	California	191792.06
153441.51	101145.55	407934.54	Florida	191050.39

# here we have given a condition to print all the columns and (1 and 2) rows.

When we want all the rows and specific columns to be printed we do it like:

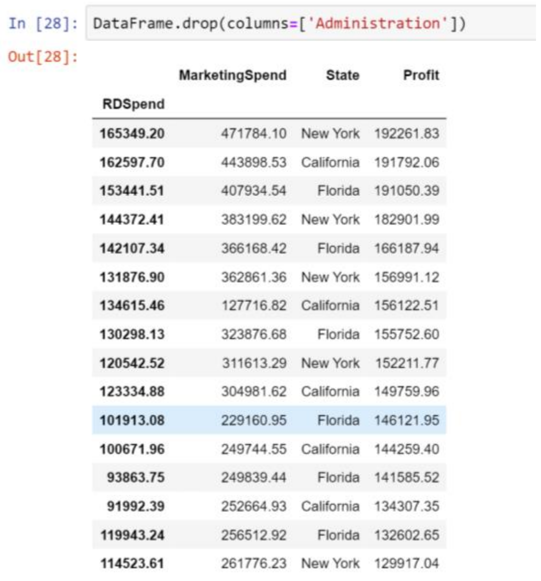


```
In [59]: DataFrame.iloc[:,[1,3]]
Out[59]:
```

RDSpend	MarketingSpend	Profit
165349.20	471784.10	192261.83
162597.70	443898.53	191792.06
153441.51	407934.54	191050.39
144372.41	383199.62	182901.99
142107.34	366168.42	166187.94
131876.90	362861.36	156991.12
134615.46	127716.82	156122.51
130298.13	323876.68	155752.60
120542.52	311613.29	152211.77
123334.88	304981.62	149759.96
101913.08	229160.95	146121.95
100671.96	249744.55	144259.40

# here we have given a condition to print all the rows and (1 and 3) columns.

4. Drop: Removing the columns is called as dropping. When there is any unwanted or any data that is not needed, using drop we can remove that unwanted data. We can drop a column using pd.drop().



```
In [28]: DataFrame.drop(columns=['Administration'])
Out[28]:
```

RDSpend	MarketingSpend	State	Profit
165349.20	471784.10	New York	192261.83
162597.70	443898.53	California	191792.06
153441.51	407934.54	Florida	191050.39
144372.41	383199.62	New York	182901.99
142107.34	366168.42	Florida	166187.94
131876.90	362861.36	New York	156991.12
134615.46	127716.82	California	156122.51
130298.13	323876.68	Florida	155752.60
120542.52	311613.29	New York	152211.77
123334.88	304981.62	California	149759.96
101913.08	229160.95	Florida	146121.95
100671.96	249744.55	California	144259.40
93863.75	249839.44	Florida	141585.52
91992.39	252664.93	California	134307.35
119943.24	256512.92	Florida	132602.65
114523.61	261776.23	New York	129917.04

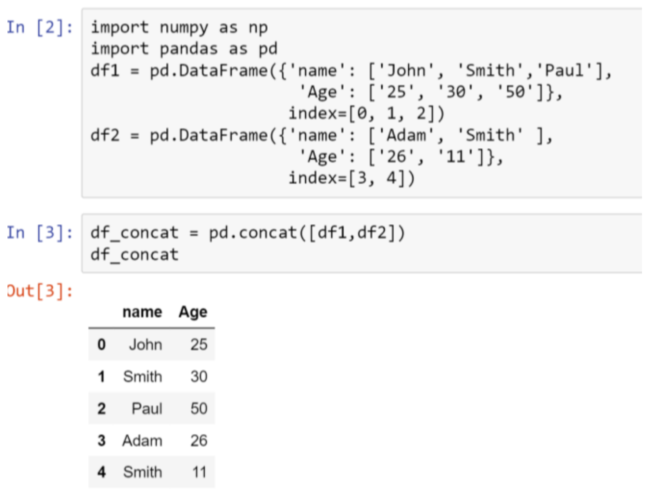
# We have dropped Administration column from our dataset.

What are the methods used?

1. Concatenation:

You can concatenate two DataFrame in Pandas. You can use pd.concat()

First of all, you need to create two DataFrames. So far so good, you are already familiar with dataframe creation.



```
In [2]: import numpy as np
        import pandas as pd
        df1 = pd.DataFrame({'name': ['John', 'Smith','Paul'],
                             'Age': ['25', '30', '50']},
                            index=[0, 1, 2])
        df2 = pd.DataFrame({'name': ['Adam', 'Smith' ],
                             'Age': ['26', '11']},
                            index=[3, 4])
```

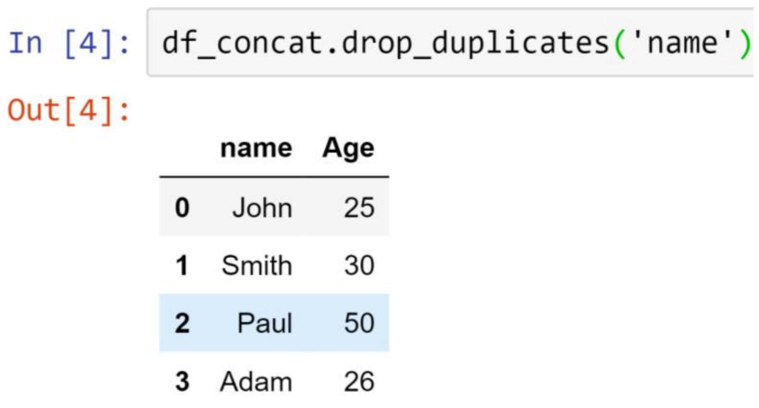
```
In [3]: df_concat = pd.concat([df1,df2])
        df_concat
```

Out[3]:

	name	Age
0	John	25
1	Smith	30
2	Paul	50
3	Adam	26
4	Smith	11

2. Drop_duplicates:

If a dataset can contain duplicates information use, `drop_duplicates` is an easy to exclude duplicate rows. You can see that `df_concat` has a duplicate observation, `Smith` appears twice in the column `name.`

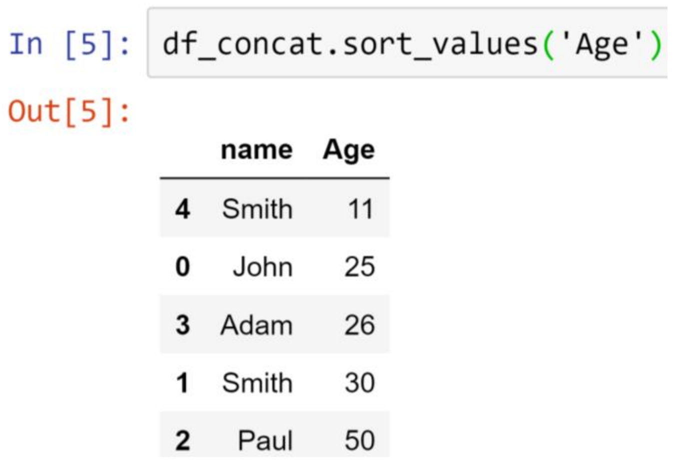


```
In [4]: df_concat.drop_duplicates('name')
```

Out[4]:

	name	Age
0	John	25
1	Smith	30
2	Paul	50
3	Adam	26

3. Sort Values: We can sort the values using sort_values.



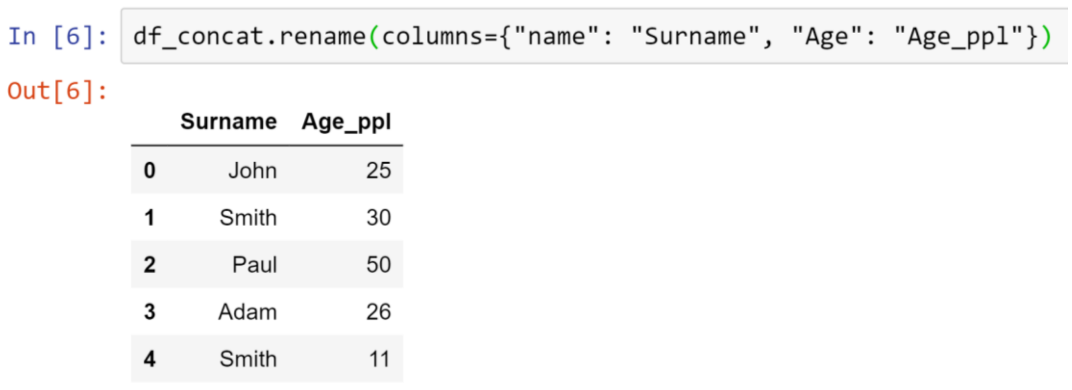
```
In [5]: df_concat.sort_values('Age')
```

Out[5]:

	name	Age
4	Smith	11
0	John	25
3	Adam	26
1	Smith	30
2	Paul	50

# Here we have given a condition to sort the age column so, the age column has been sorted in an ascending order.

4. Rename – change of index: You can use rename to rename a column in Pandas. The first value is the current column name and the second value is the new column name.

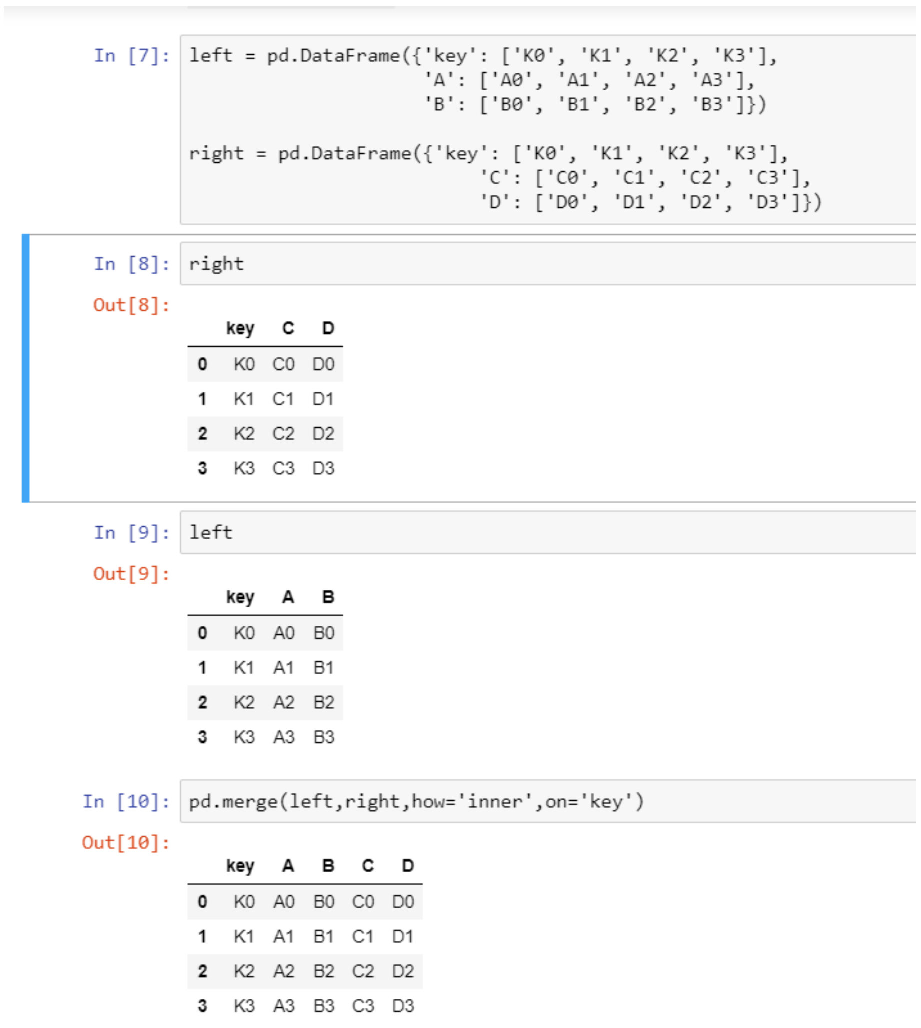


```
In [6]: df_concat.rename(columns={"name": "Surname", "Age": "Age_ppl"})
Out[6]:
```

	Surname	Age_ppl
0	John	25
1	Smith	30
2	Paul	50
3	Adam	26
4	Smith	11

# Here the column name age have been renamed by age_ppl.

5. Merge: The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:



```
In [7]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                             'A': ['A0', 'A1', 'A2', 'A3'],
                             'B': ['B0', 'B1', 'B2', 'B3']})

        right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                              'C': ['C0', 'C1', 'C2', 'C3'],
                              'D': ['D0', 'D1', 'D2', 'D3']})
```

```
In [8]: right
Out[8]:
```

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

```
In [9]: left
Out[9]:
```

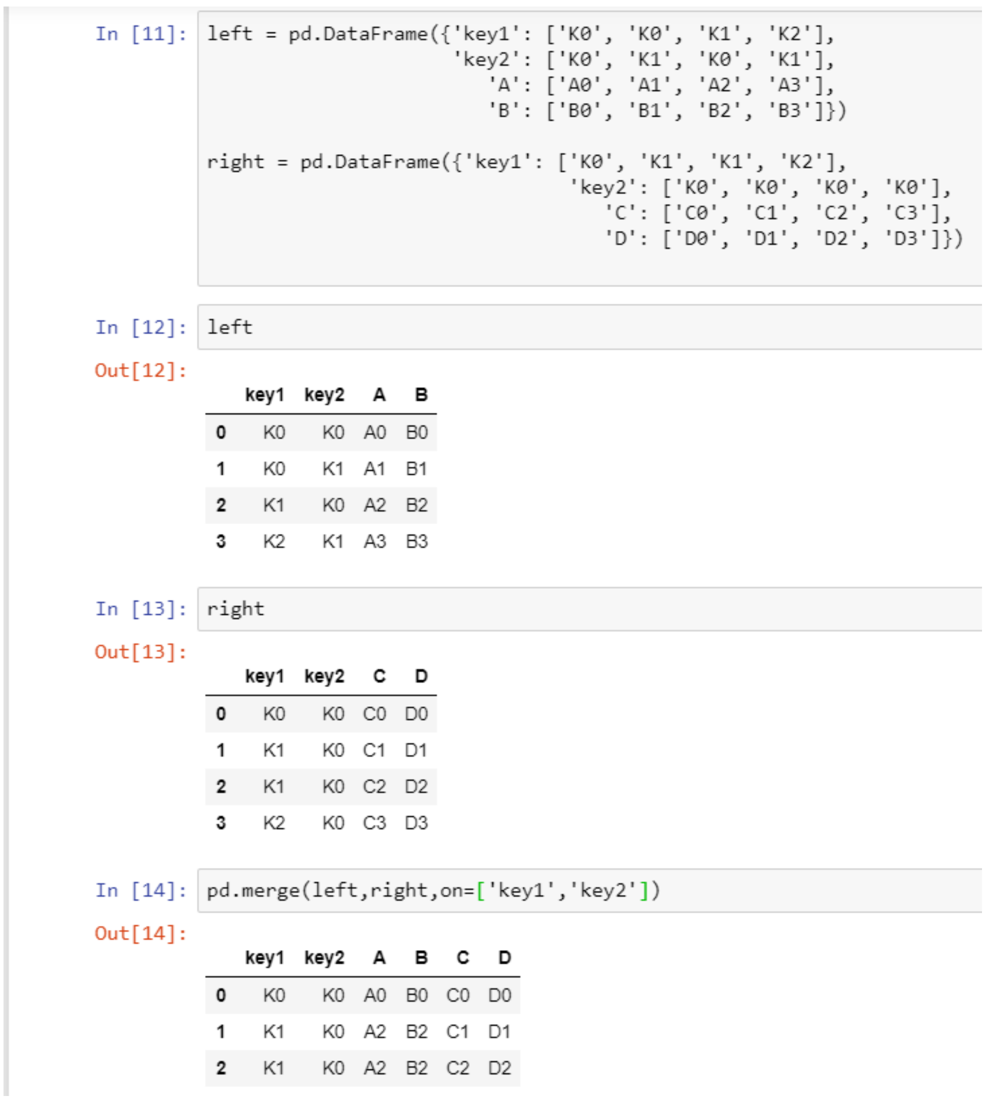
	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

```
In [10]: pd.merge(left,right,how='inner',on='key')
Out[10]:
```

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

In the above example, we have created two dataframes i.e.; left and right and then we have applied merge function to the data by using sql tables. We have given *inner* merge on the *key* column. This will check for common elements in key column and print the values accordingly.

Let us consider another example which is little more complex. Example:



```
In [11]: left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                              'key2': ['K0', 'K1', 'K0', 'K1'],
                                 'A': ['A0', 'A1', 'A2', 'A3'],
                                 'B': ['B0', 'B1', 'B2', 'B3']})

         right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                                      'key2': ['K0', 'K0', 'K0', 'K0'],
                                         'C': ['C0', 'C1', 'C2', 'C3'],
                                         'D': ['D0', 'D1', 'D2', 'D3']})
```

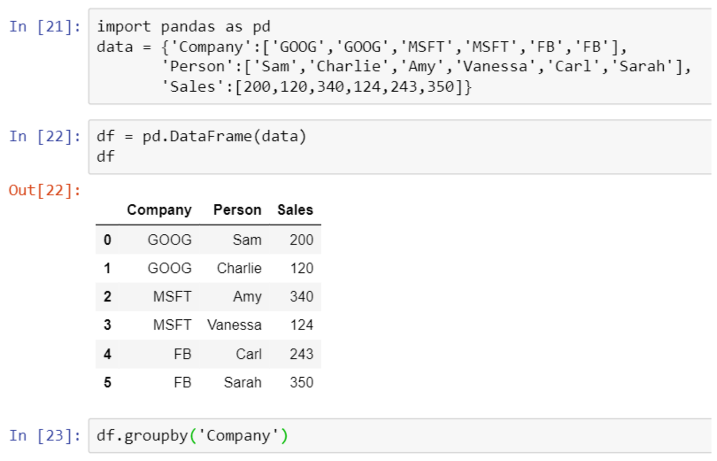
```
In [12]: left
Out[12]:
   key1 key2  A   B
0  K0   K0    A0  B0
1  K0   K1    A1  B1
2  K1   K0    A2  B2
3  K2   K1    A3  B3
```

```
In [13]: right
Out[13]:
   key1 key2  C   D
0  K0   K0    C0  D0
1  K1   K0    C1  D1
2  K1   K0    C2  D2
3  K2   K0    C3  D3
```

```
In [14]: pd.merge(left,right,on=['key1','key2'])
Out[14]:
   key1 key2  A   B   C   D
0  K0   K0    A0  B0  C0  D0
1  K1   K0    A2  B2  C1  D1
2  K1   K0    A2  B2  C2  D2
```

Here, in this example we have again created two dataframes with two different keys (key1 and key2). We are applying merge on *key1 and key2,* which will check for sm=imilar values in both key1 and key 2 and print the values accordingly.

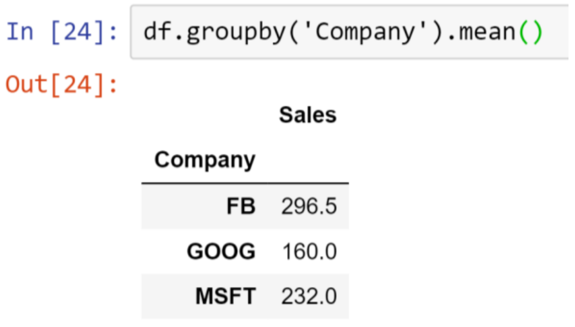
6. Group by: The groupby method allows you to group rows of data together and call aggregate functions.



```
In [21]: import pandas as pd
         data = {'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
                'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
                'Sales':[200,120,340,124,243,350]}

In [22]: df = pd.DataFrame(data)
         df

Out[22]:
            Company   Person  Sales
         0     GOOG      Sam    200
         1     GOOG  Charlie    120
         2     MSFT      Amy    340
         3     MSFT  Vanessa    124
         4       FB     Carl    243
         5       FB    Sarah    350

In [23]: df.groupby('Company')
```

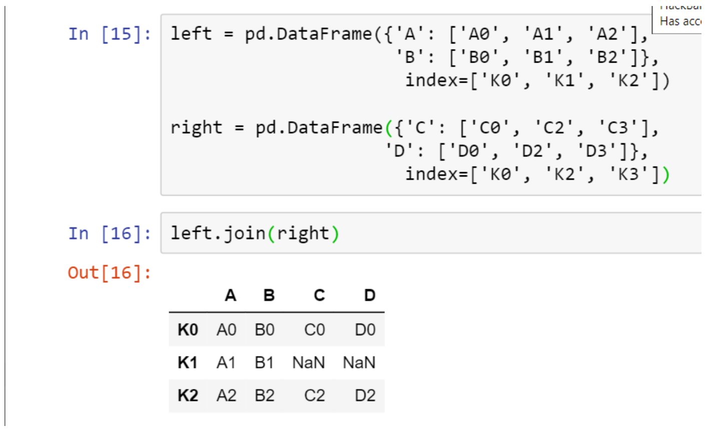
# Once the dataframe is created we can use the .groupby() method to group rows together based off of a column name. For instance let's group based off of Company. This will create a DataFrameGroupBy object(like in the above example)

And then we can call the aggregate(like mean(), standard deviation etc) methods off the object, sown in the figure below:



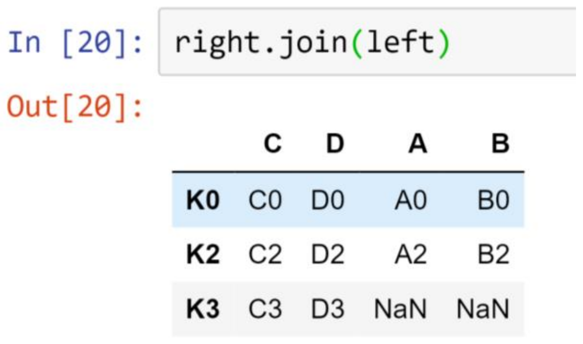
```
In [24]: df.groupby('Company').mean()

Out[24]:
                  Sales
         Company
              FB  296.5
            GOOG  160.0
            MSFT  232.0
```

7. Join: Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Let us see an example:



```
In [15]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                              'B': ['B0', 'B1', 'B2']},
                               index=['K0', 'K1', 'K2'])

         right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                             'D': ['D0', 'D2', 'D3']},
                               index=['K0', 'K2', 'K3'])

In [16]: left.join(right)

Out[16]:
              A   B    C    D
         K0  A0  B0   C0   D0
         K1  A1  B1  NaN  NaN
         K2  A2  B2   C2   D2
```

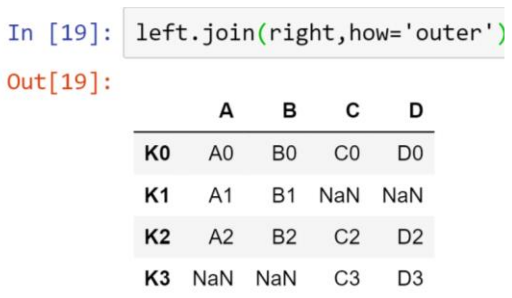
In this example left and right dataframes are created and join is applied to the left, which considers the index of only left dataframe and prints the data accordingly.



```
In [20]: right.join(left)
Out[20]:
```

	C	D	A	B
K0	C0	D0	A0	B0
K2	C2	D2	A2	B2
K3	C3	D3	NaN	NaN

# Here we have applied join to the right, which considers the index of right daframe and joins both the dataframes accordingly.



```
In [19]: left.join(right,how='outer')
Out[19]:
```

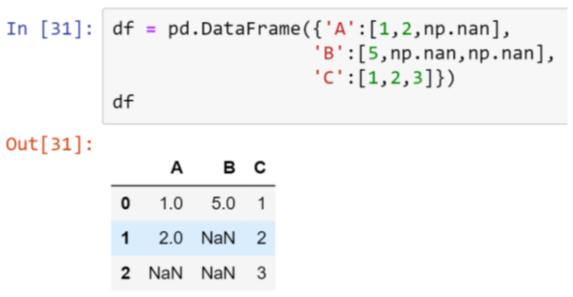
	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

# Here we have applied outer join which consider all the indexes given in the dataframe.

2.7.How to deal with missing values?

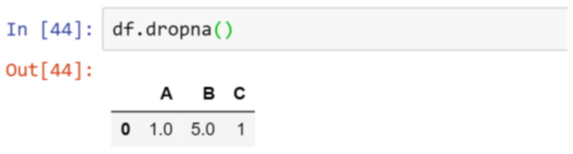
Let's show a few convenient methods to deal with Missing Data in pandas:

1. drop na: dropna() deals with null values. It drops the row or column that contains null values. Let us see some examples:



```
In [31]: df = pd.DataFrame({'A':[1,2,np.nan],
                            'B':[5,np.nan,np.nan],
                            'C':[1,2,3]})
         df
Out[31]:
```

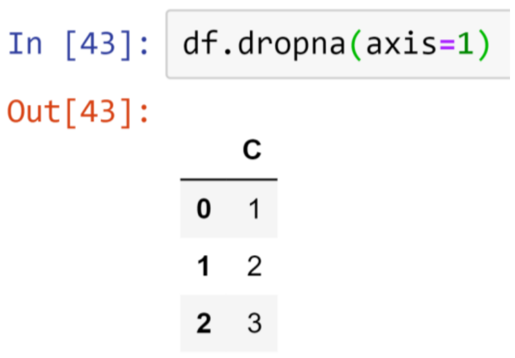
	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3



```
In [44]: df.dropna()
Out[44]:
```

	A	B	C
0	1.0	5.0	1

# here we have created a dataframe with null values and applied dropna which dropped the null values row and column. If the condition is dropna() or
dropna(axis-0) this drops the rows with null values and prints the row which has no null values.

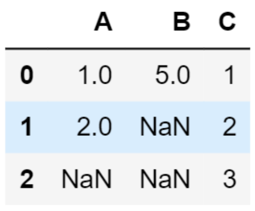


```
In [43]: df.dropna(axis=1)
Out[43]:
```

	C
0	1
1	2
2	3

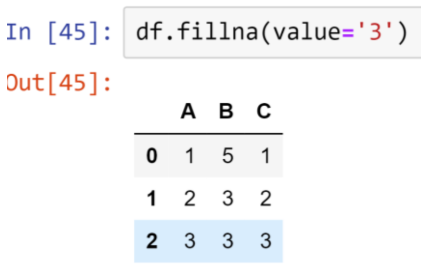
# here we have given axis=1, this drops the columns with null values and prints the column which has no null values

2. fillna: fillna() also deals with null values. It fills the row or column that contains null values with the given value. Let us see an example:



	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

When we consider this data we fill the Nan values in 1-B,2-A,2-B with some value using fillna, as shown below:



```
In [45]: df.fillna(value='3')
Out[45]:
```

	A	B	C
0	1	5	1
1	2	3	2
2	3	3	3

# here we have given the value as 3 and applied fillna() which replaced NaN values with 3.