Unit-VI
Pandas & Matplotlib

Contents

- **Overview of Pandas and Matplotlib**
- Introduction to Pandas
- How to install pandas
- Applications of Pandas
- Dataframes and Series
- Reading csv, Excel and text files using pandas
- Getting description of dataset and simple functions of pandas
- Introduction to Matplotlib with its application
- Using label, color and markers, axis
- Scatter plot
- Bar graph
- Histogram
- Line plot.

Introduction

- **What is Python Pandas?**
- Pandas is used for data manipulation, analysis and cleaning. Python pandas is well suited for different kinds of data, such as:

1. Tabular data with heterogeneously-typed columns
2. Ordered and unordered time series data
3. Arbitrary matrix data with row & column labels
4. Unlabelled data
5. Any other form of observational or statistical data sets

Install Pandas

- **How to install Pandas?**
- To install Python Pandas, go to your command line/ terminal and type
- **“pip install pandas”** or
- else, if you have anaconda installed in your system, just type in
- **“conda install pandas”.**
- Once the installation is completed, go to your IDE (Jupyter, PyCharm etc.) and simply import it by typing:
- **“import pandas as pd”**

Introduction to pandas

- Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.
- Pandas is built on top of the **Numpy** package, means **Numpy** is required for operating the Pandas.
- Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis.
- So, Pandas came into the picture and enhanced the capabilities of data analysis.
- It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

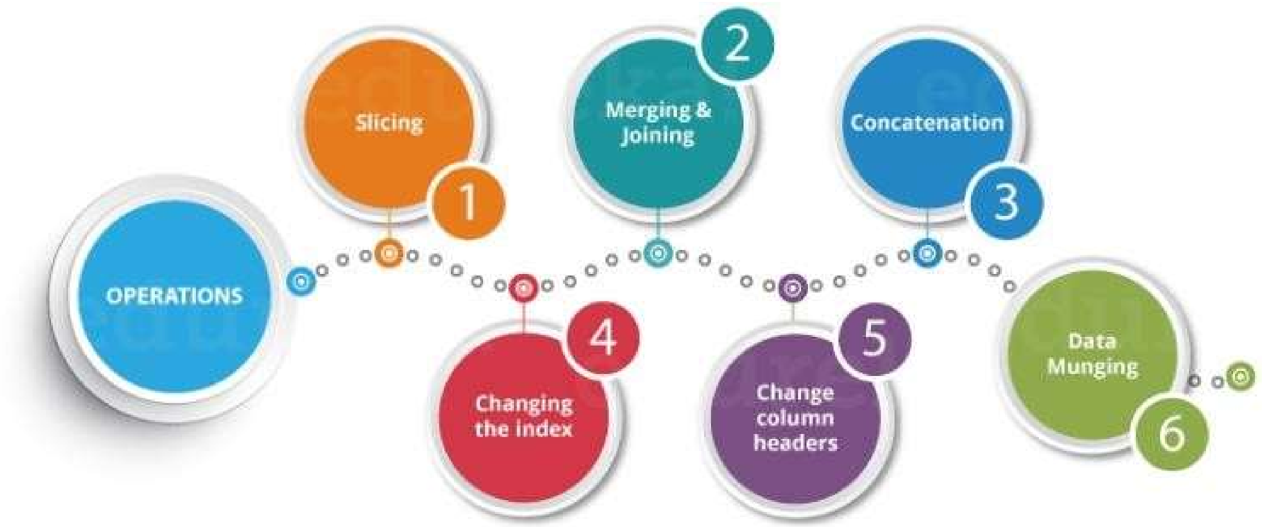
What's Pandas for?

- Pandas has so many uses that it might make sense to list the things it can't do instead of what it can do.
- 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, like:
  - **What's the average, median, max, or min of each column?**
  - **Does column A correlate with column B?**
  - **What does the distribution of data in column C look like?**

- 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 and more.
- Store the cleaned, transformed data back into a CSV, other file or database.

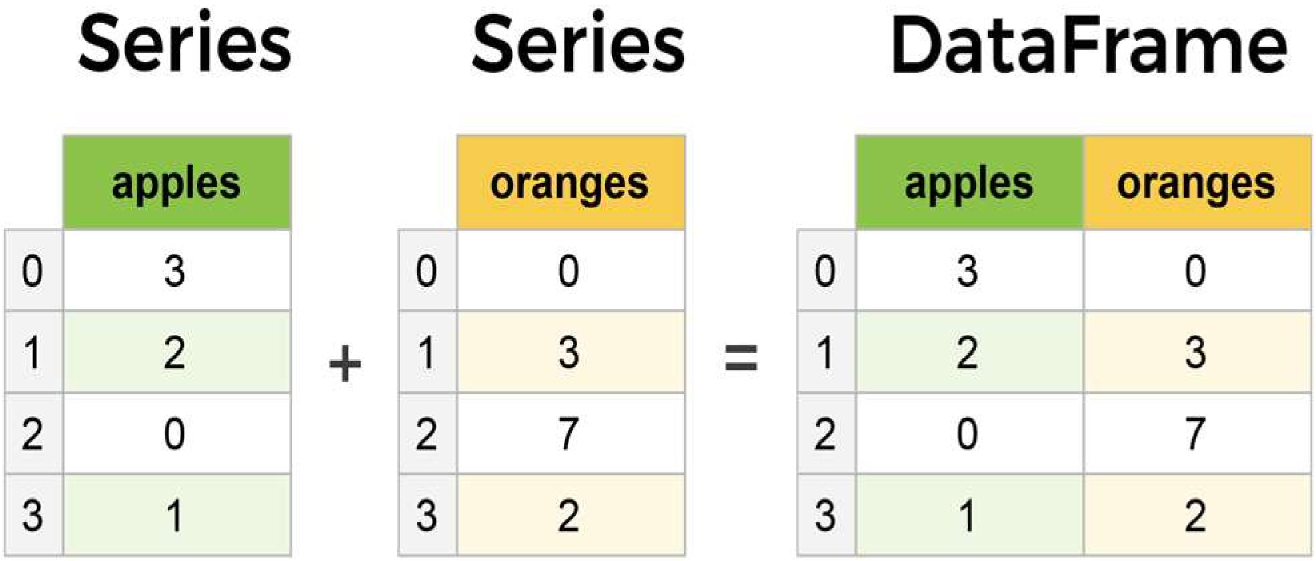
Python Pandas Operations

- Using Python pandas, you can perform a lot of operations with series, data frames, missing data, group by etc. Some of the common operations for data manipulation are listed below:



Core components of pandas: Series and DataFrames

- The primary two components of pandas are the Series and DataFrame.
- A Series is essentially a column, and a DataFrame is a multi-dimensional table made up of a collection of Series.



Series

	apples
0	3
1	2
2	0
3	1

+

Series

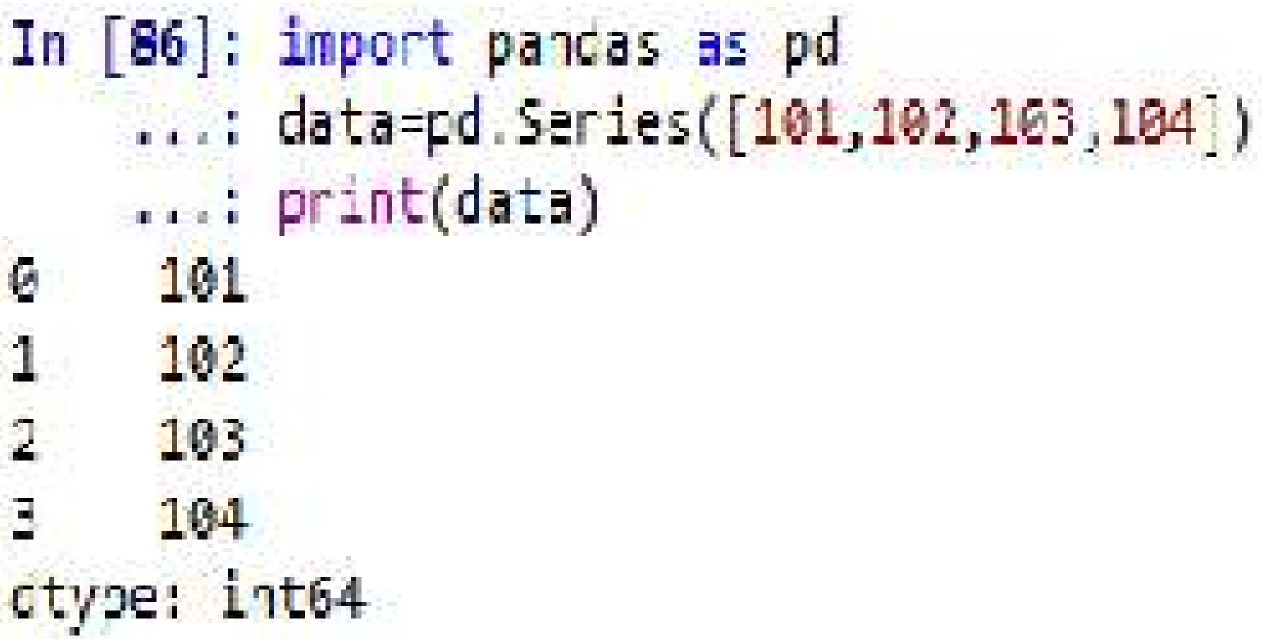
	oranges
0	0
1	3
2	7
3	2

=

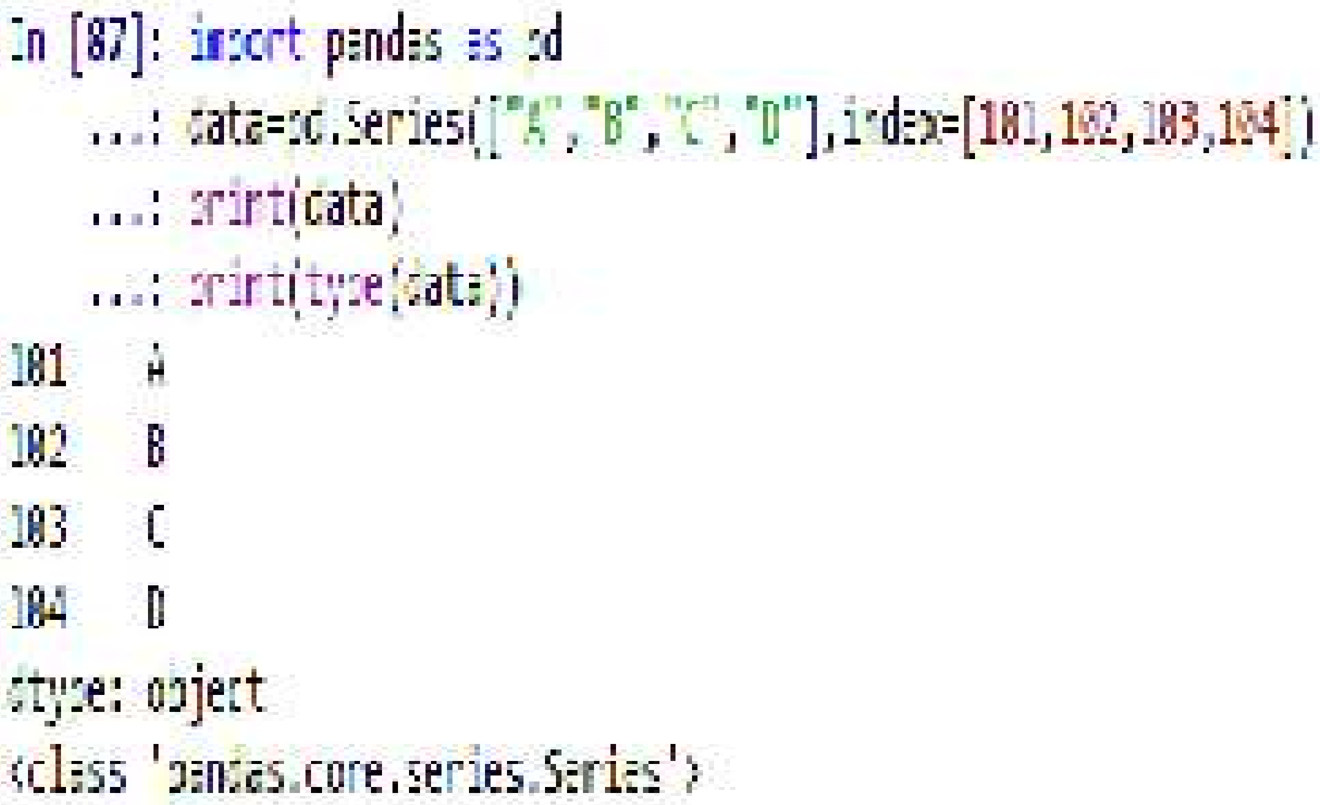
DataFrame

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

- It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the **index**. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:
- **Data:** It can be any list, dictionary, or scalar value.



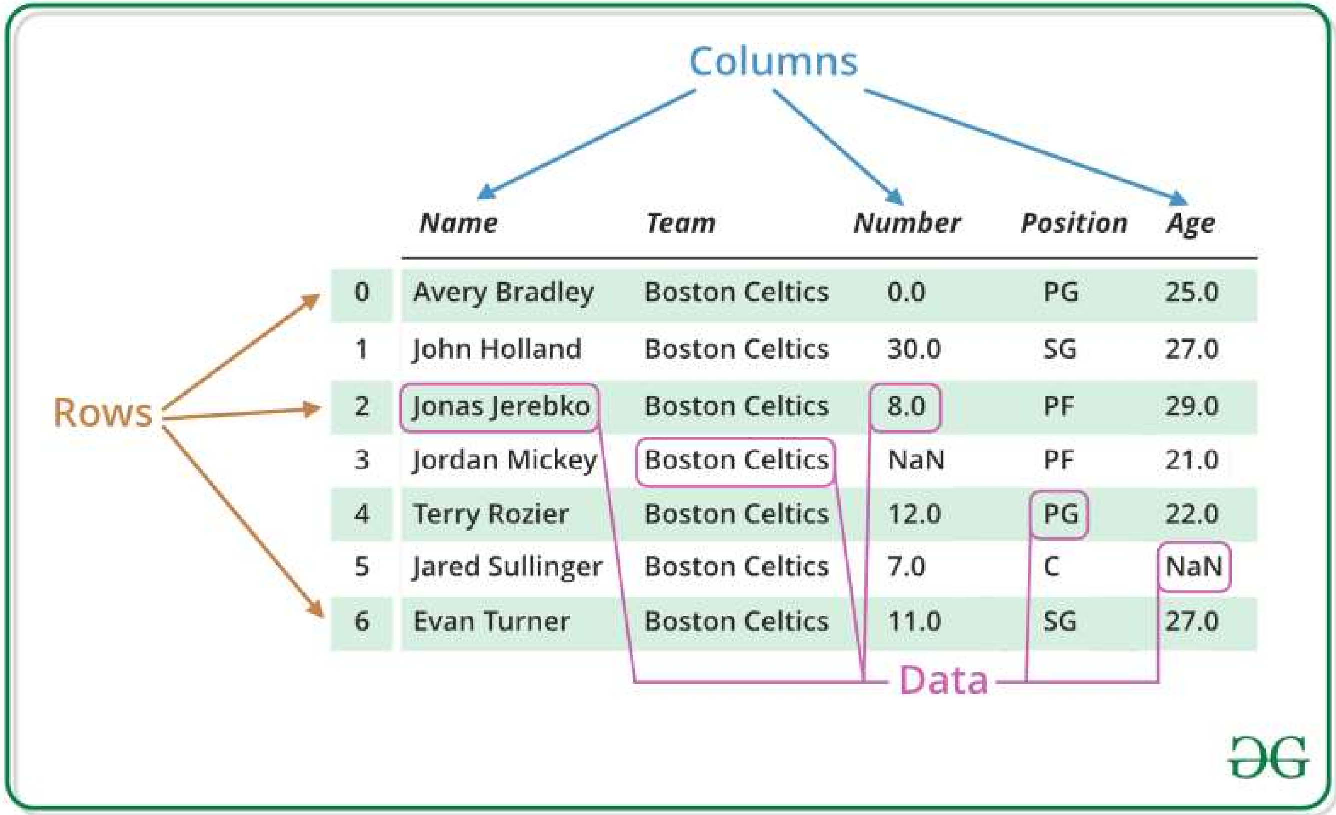
```
In [86]: import pandas as pd
    ...: data=pd.Series([101,102,103,104])
    ...: print(data)
0    101
1    102
2    103
3    104
dtype: int64
```



```
In [87]: import pandas as pd
    ...: data=pd.Series(["A","B","C","D"],index=[101,102,103,104])
    ...: print(data)
    ...: print(type(data))
101    A
102    B
103    C
104    D
dtype: object
<class 'pandas.core.series.Series'>
```

Python Pandas Dataframe

- It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index. It consists of the following properties:
- The columns can be heterogeneous types like int, bool, and so on.
- It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Creating DataFrames from scratch

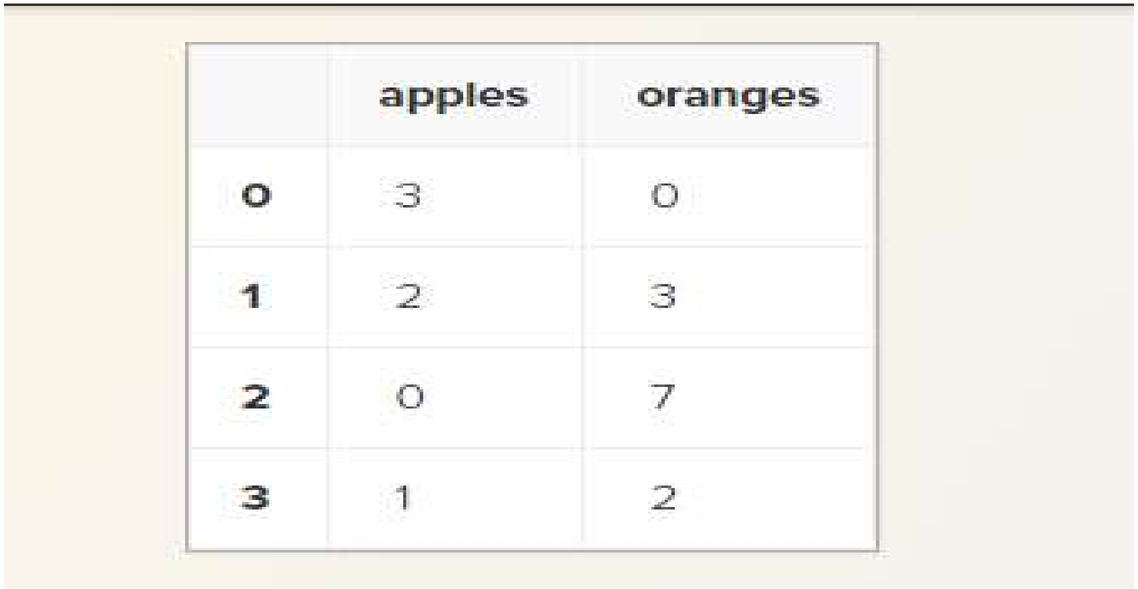
- DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.
- Creating DataFrames right in Python is good to know and quite useful when testing new methods and functions you find in the 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:
- data = { 'apples': [3, 2, 0, 1], 'oranges': [0, 3, 7, 2] }

- We can create a DataFrame using following ways:
- **dict**
- **Lists**
- **Numpy ndarrrays**
- **Series**
- Syntax:
- **pandas.DataFrame(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=None*)**

- **data:** It consists of different forms like ndarray, series, map, constants, lists, array.
- **index:** The Default np.arrange(n) index is used for the row labels if no index is passed.
- **columns:** The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.
- **dtype:** It refers to the data type of each column.
- **copy():** It is used for copying the data.

Example

- And then pass it to the pandas DataFrame constructor:
- purchases = pd.DataFrame(data)
- Purchase

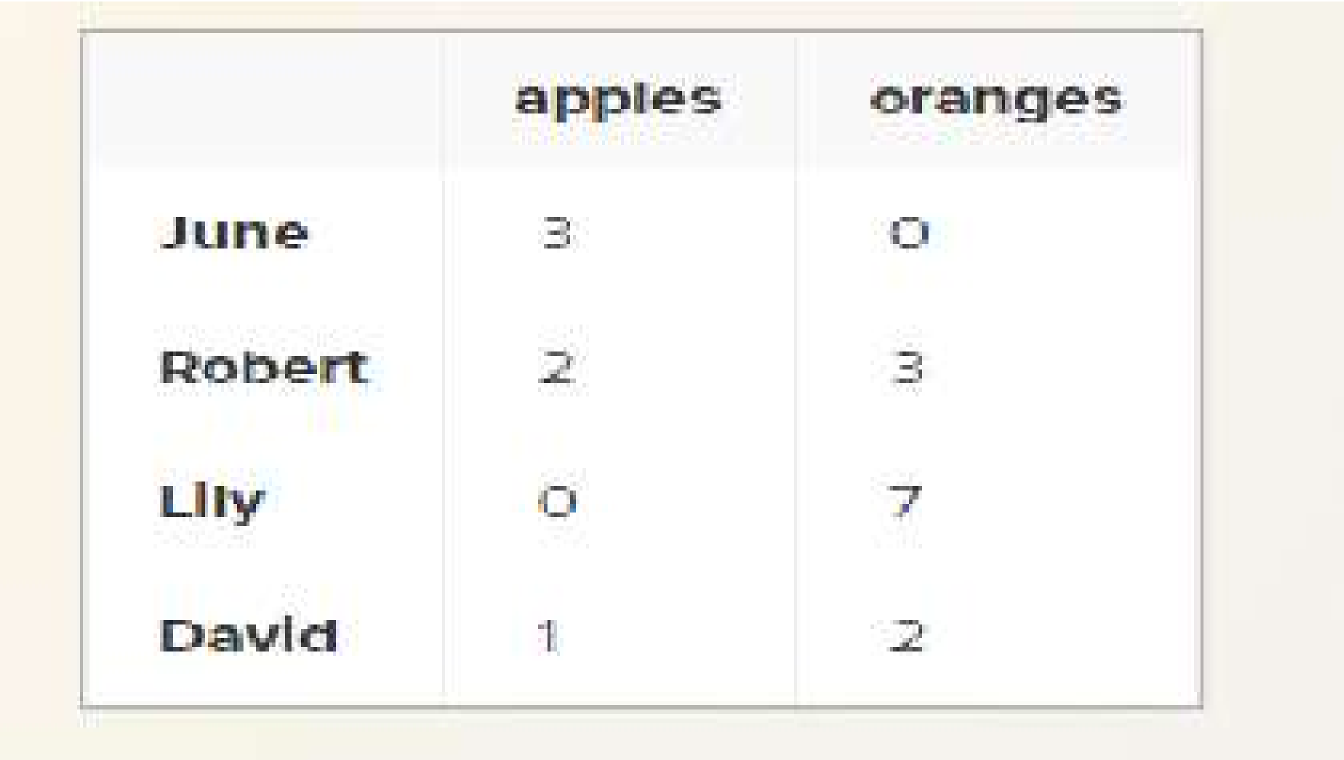


	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

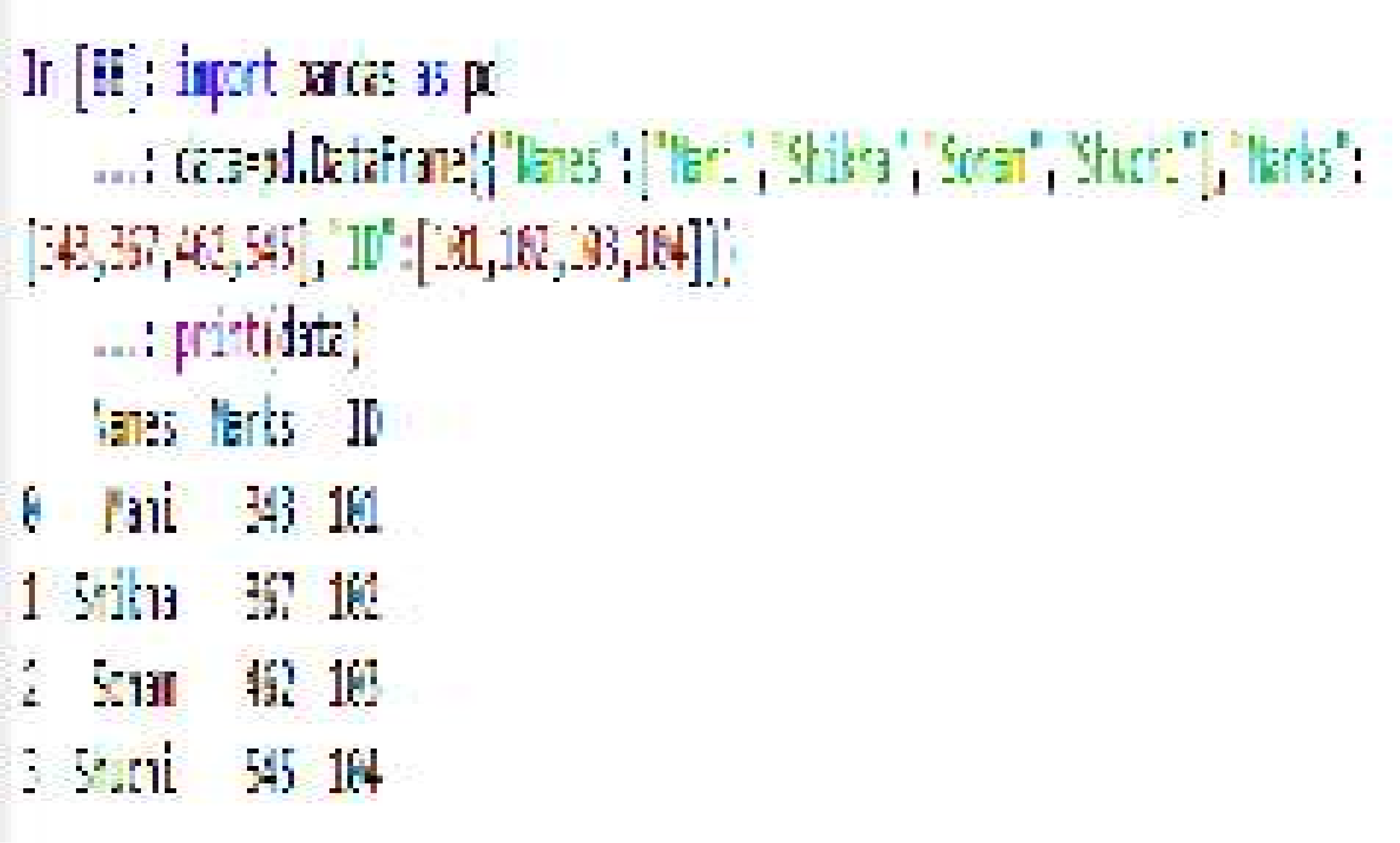
- **How did that work?**

1. Each *(key, value)* item in data corresponds to a *column* in the resulting DataFrame.
2. The **Index** 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.
3. Let's have customer names as our index:
4. purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])
5. purchases

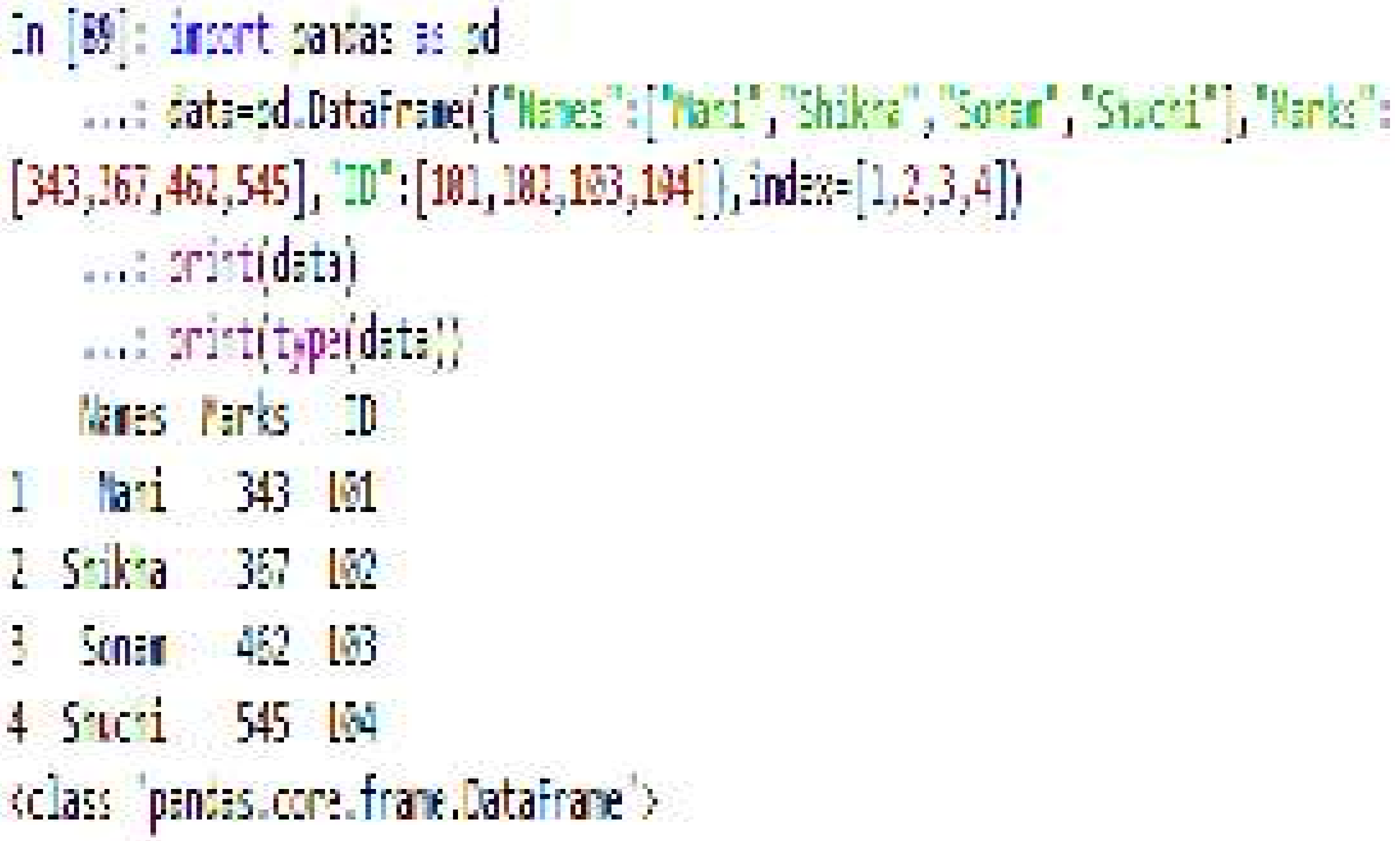
Example



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



```
In [88]: import pandas as pd
    ...: data=pd.DataFrame({"Names":["Mani","Shikha","Sonam","Shuchi"],"Marks":
[343,357,462,545],"ID":[101,102,103,104]})
    ...: print(data)
    Names  Marks   ID
0    Mani    343  101
1  Shikha    357  102
2   Sonam    462  103
3  Shuchi    545  104
```



```
In [89]: import pandas as pd
    ...: data=pd.DataFrame({"Names":["Mani","Shikha","Sonam","Shuchi"],"Marks":
[343,367,462,545],"ID":[101,102,103,104]},index=[1,2,3,4])
    ...: print(data)
    ...: print(type(data))
    Names  Marks   ID
1    Mani    343  101
2  Shikha    367  102
3   Sonam    462  103
4  Shuchi    545  104
<class 'pandas.core.frame.DataFrame'>
```

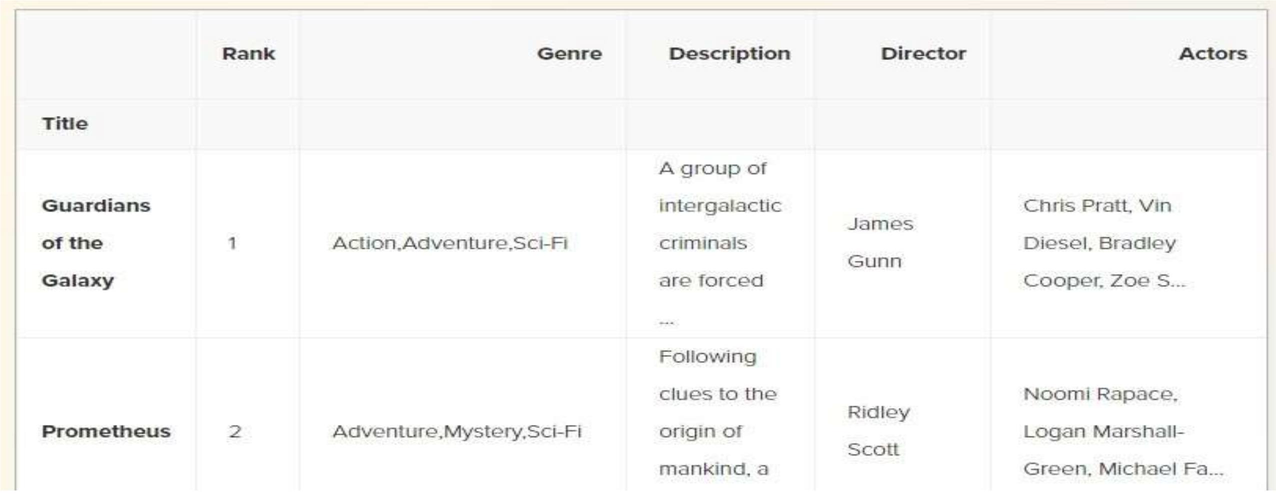
DataFrame Operations

- **Most important DataFrame operations**
- DataFrames possess hundreds of methods and other operations that are crucial to any analysis. As a beginner, you should know the operations that perform simple transformations of your data and those that provide fundamental statistical analysis.
- Let's load in the IMDB movies dataset to begin:
- **movies_df = pd.read_csv("IMDB-Movie-Data.csv", index_col="Title")**
- We're loading this dataset from a CSV and designating the movie titles to be our index.

- **Syntax: pd.read_csv**(filepath_or_buffer, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)
- **filepath_or_buffer**: It is the location of the file which is to be retrieved using this function. It accepts any string path or URL of the file.
- **sep**: It stands for separator, default is ', ' as in CSV(comma separated values).
- **header**: It accepts int, a list of int, row numbers to use as the column names, and the start of the data. If no names are passed, i.e., header=None, then, it will display the first column as 0, the second as 1, and so on.
- **usecols**: It is used to retrieve only selected columns from the CSV file.
- **nrows**: It means a number of rows to be displayed from the dataset.
- **index_col**: If None, there are no index numbers displayed along with records.
- **skiprows**: Skips passed rows in the new data frame.

Head and Tail

- **Viewing your data**
- The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with .head():
- **movies_df.head()**

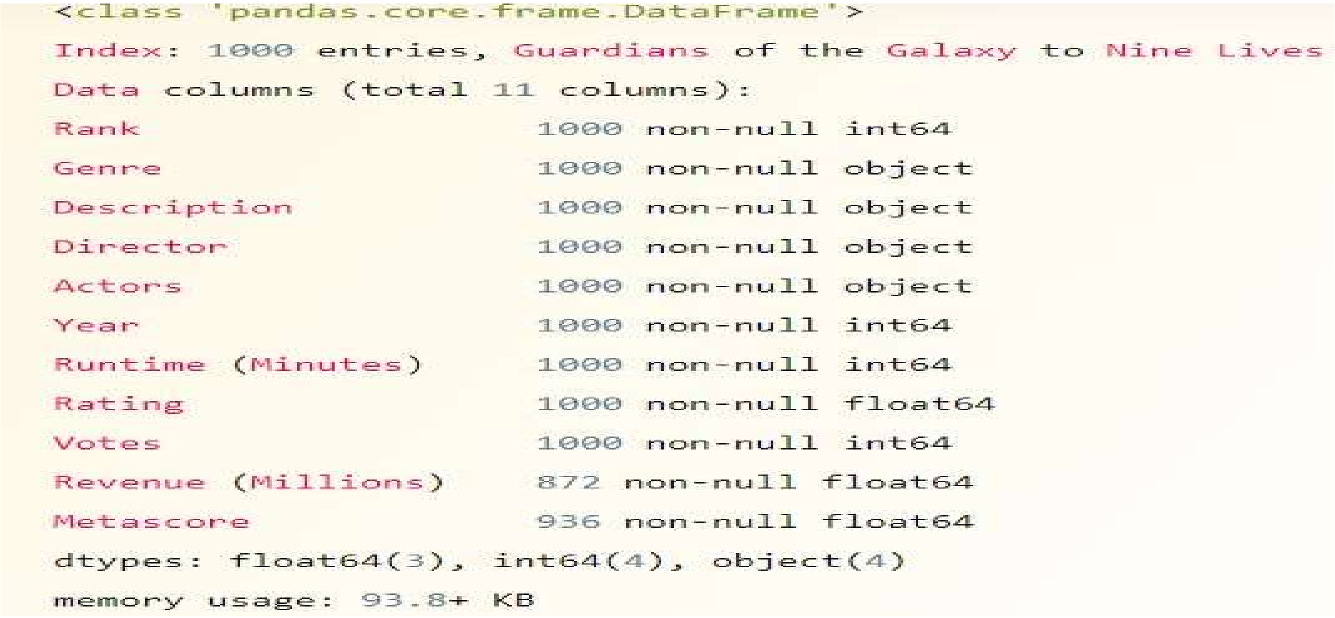


	Rank	Genre	Description	Director	Actors
Title					
Guardians of the Galaxy	1	Action,Adventure,Sci-Fi	A group of intergalactic criminals are forced ...	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...

- .head() outputs the **first** five rows of your DataFrame by default, but we could also pass a number as well:
  - **movies_df.head(10)**
- It would output the top ten rows, for example.
- To see the **last** five rows use .tail(). tail() also accepts a number, and in this case we printing the bottom two rows.:
  - **movies_df.tail(2)**
- Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.
- You'll notice that the index in our DataFrame is the *Title* column, which you can tell by how the word *Title* is slightly lower than the rest of the columns.

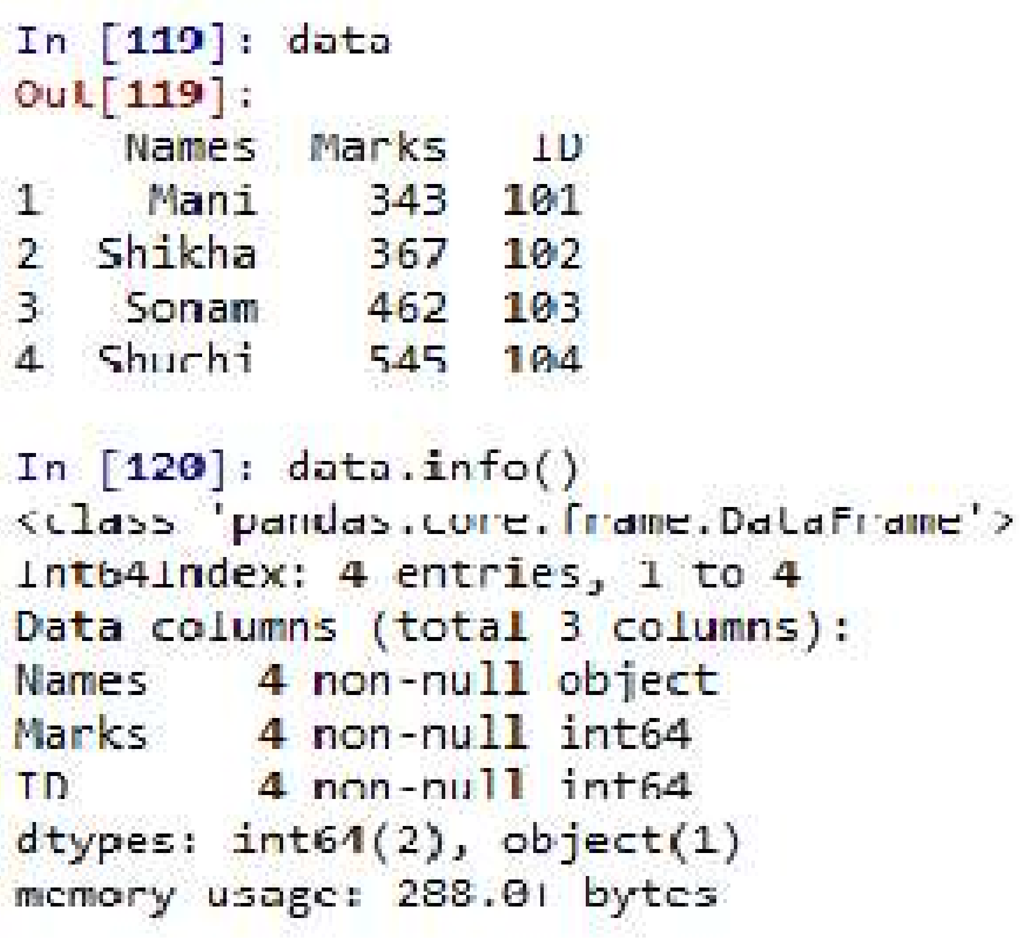
Info()

- **Getting info about your data**
- .info() should be one of the very first commands you run after loading your data:
- **movies_df.info()**

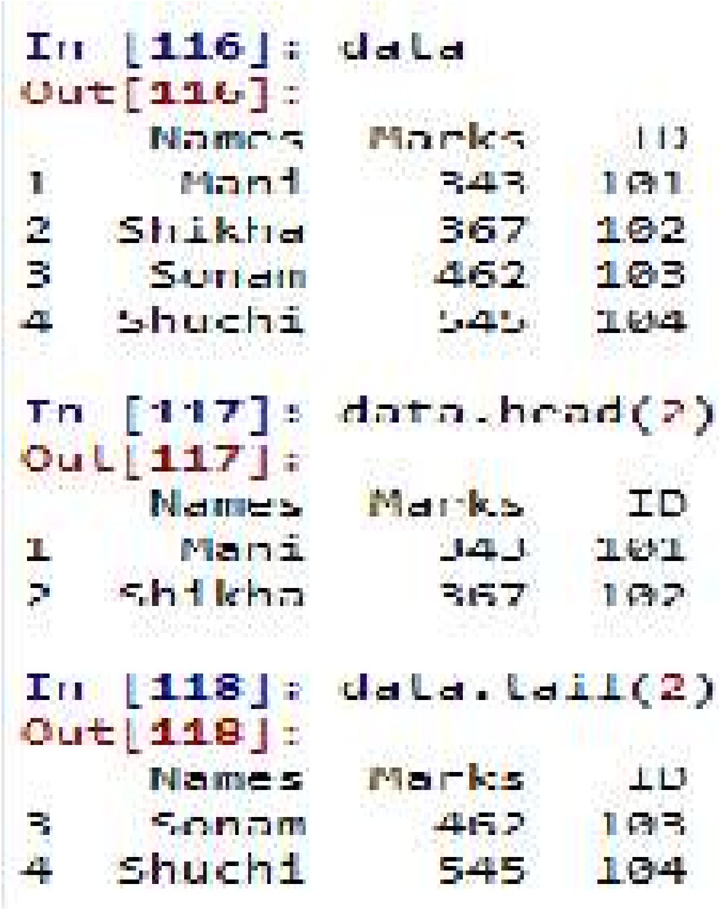


```
<class 'pandas.core.frame.DataFrame'>
Index: 1000 entries, Guardians of the Galaxy to Nine Lives
Data columns (total 11 columns):
Rank                    1000 non-null int64
Genre                   1000 non-null object
Description             1000 non-null object
Director                1000 non-null object
Actors                  1000 non-null object
Year                    1000 non-null int64
Runtime (Minutes)       1000 non-null int64
Rating                  1000 non-null float64
Votes                   1000 non-null int64
Revenue (Millions)      872 non-null float64
Metascore               936 non-null float64
dtypes: float64(3), int64(4), object(4)
memory usage: 93.8+ KB
```

- .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.
- Notice in our movies dataset we have some obvious missing values in the Revenue and Metascore columns.



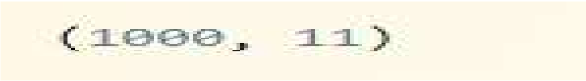
```
In [119]: data
Out[119]:
     Names  Marks   ID
1     Mani    343  101
2   Shikha    367  102
3    Sonam    462  103
4   Shuchi    545  104

In [120]: data.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 1 to 4
Data columns (total 3 columns):
Names     4 non-null object
Marks     4 non-null int64
ID        4 non-null int64
dtypes: int64(2), object(1)
memory usage: 288.0+ bytes
```



```
In [116]: data
Out[116]:
     Names  Marks   ID
1     Mani    343  101
2   Shikha    367  102
3    Sonam    462  103
4   Shuchi    545  104

In [117]: data.head(2)
Out[117]:
     Names  Marks   ID
1     Mani    343  101
2   Shikha    367  102

In [118]: data.tail(2)
Out[118]:
     Names  Marks   ID
3    Sonam    462  103
4   Shuchi    545  104
```

Shape

- Another fast and useful attribute is .shape, which outputs just a tuple of (rows, columns):
- **movies_df.shape**

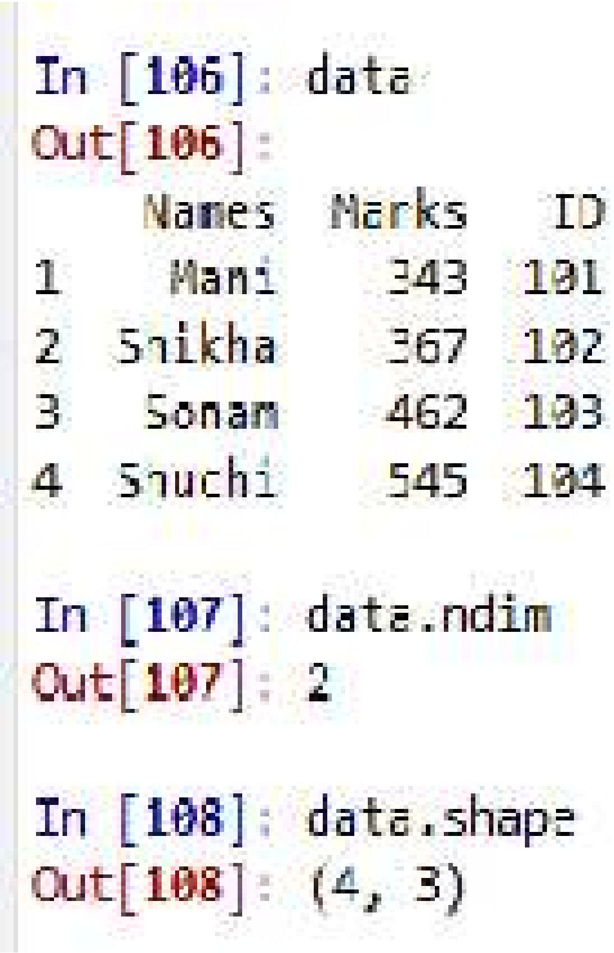


```
(1000, 11)
```

- Note that .shape has no parentheses and is a simple tuple of format (rows, columns). So we have **1000 rows** and **11 columns** in our movies DataFrame.
- You'll be going to .shape a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

Size

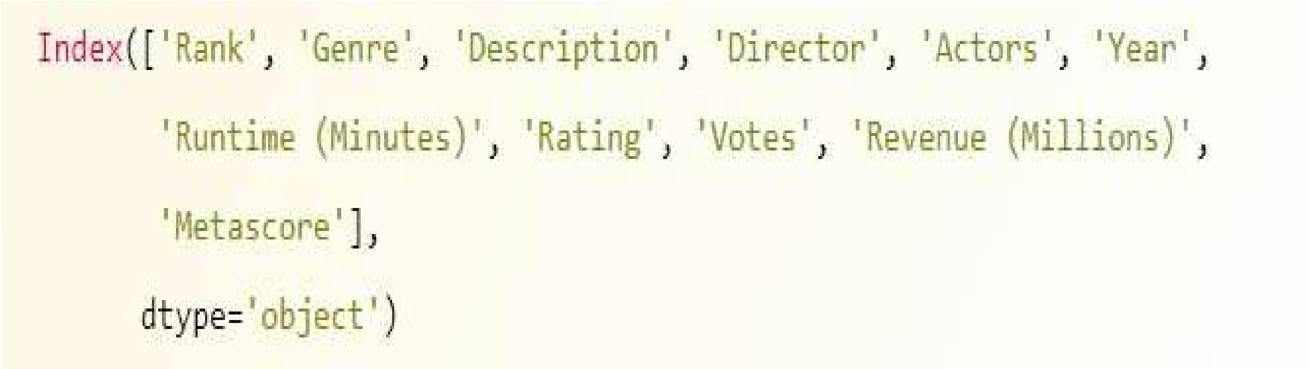
- ***Syntax**: dataframe.size*
  ***Return** : Returns size of dataframe/series which is equivalent to total number of elements. That is rows x columns.*
- ***Syntax**: dataframe.ndim*
  ***Return** : Returns dimension of dataframe/series. 1 for one dimension (series), 2 for two dimension (dataframe)*



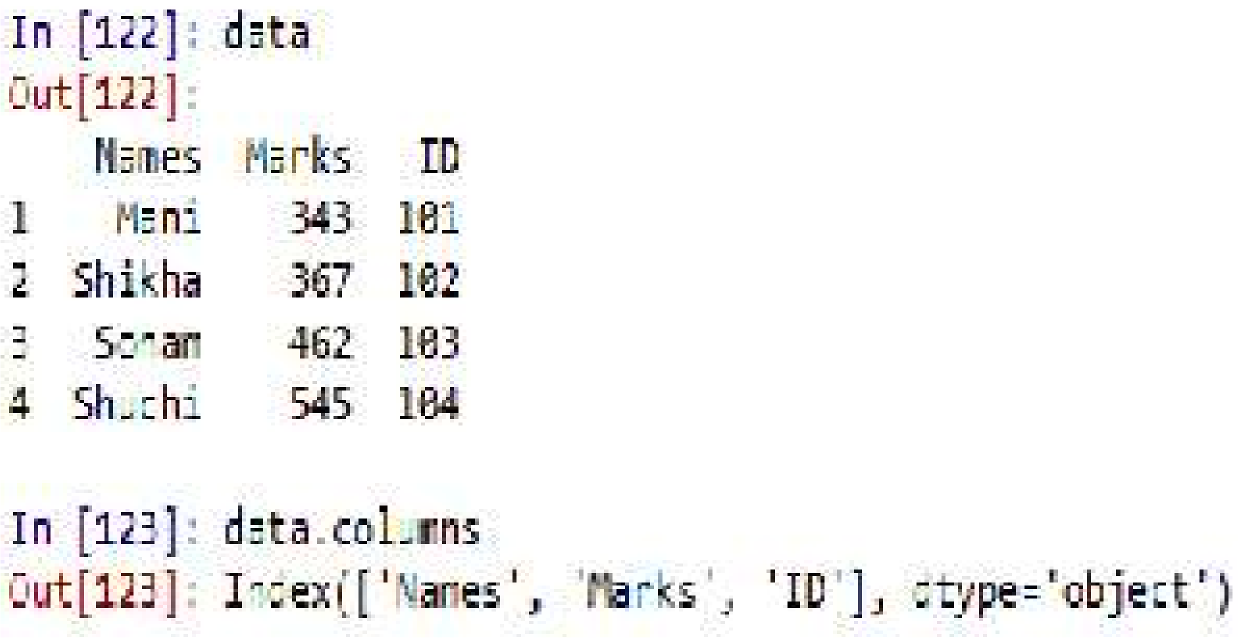
```
In [106]: data
Out[106]:
    Names  Marks   ID
1    Mani    343  101
2  Shikha    367  102
3   Sonam    462  103
4  Shuchi    545  104

In [107]: data.ndim
Out[107]: 2

In [108]: data.shape
Out[108]: (4, 3)
```

Column

- Many times datasets will have verbose column names with symbols, upper and lowercase words, spaces, and types. To make selecting data by column name easier we can spend a little time cleaning up their names.
- Here's how to print the column names of our dataset:
- **movies_df.columns**



```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year',
       'Runtime (Minutes)', 'Rating', 'Votes', 'Revenue (Millions)',
       'Metascore'],
      dtype='object')
```



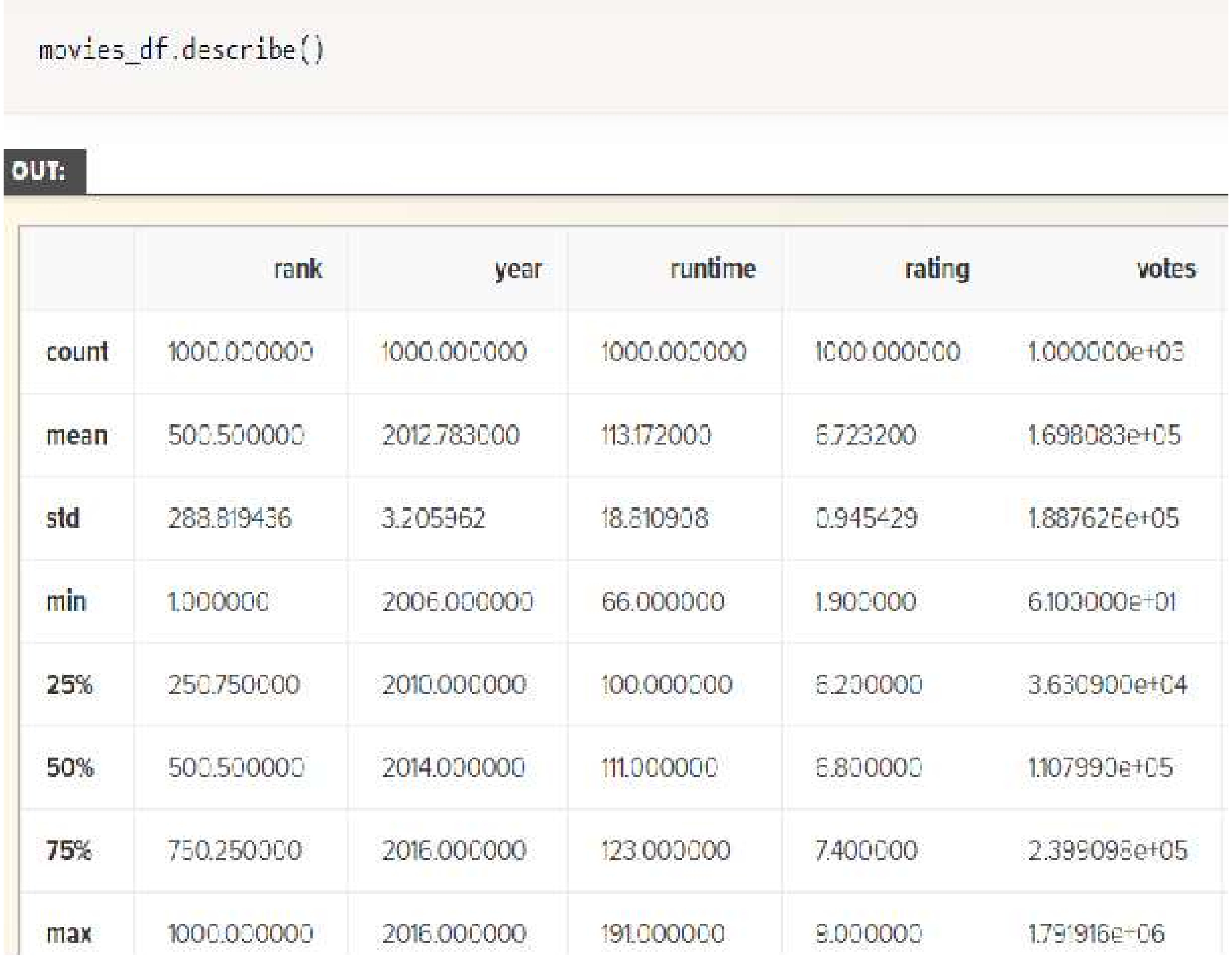
```
In [122]: data
Out[122]:
    Names  Marks   ID
1    Mani    343  101
2  Shikha    367  102
3   Sonam    462  103
4  Shuchi    545  104

In [123]: data.columns
Out[123]: Index(['Names', 'Marks', 'ID'], dtype='object')
```

Describe

- Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.
- .describe() can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

The describe() method returns description of the data in the DataFrame.
If the DataFrame contains numerical data, the description contains these information for each column:

count - The number of not-empty values.
mean - The average (mean) value.
std - The standard deviation.
min - the minimum value.
25% - The 25% percentile*.
50% - The 50% percentile*.
75% - The 75% percentile*.
max - the maximum value.



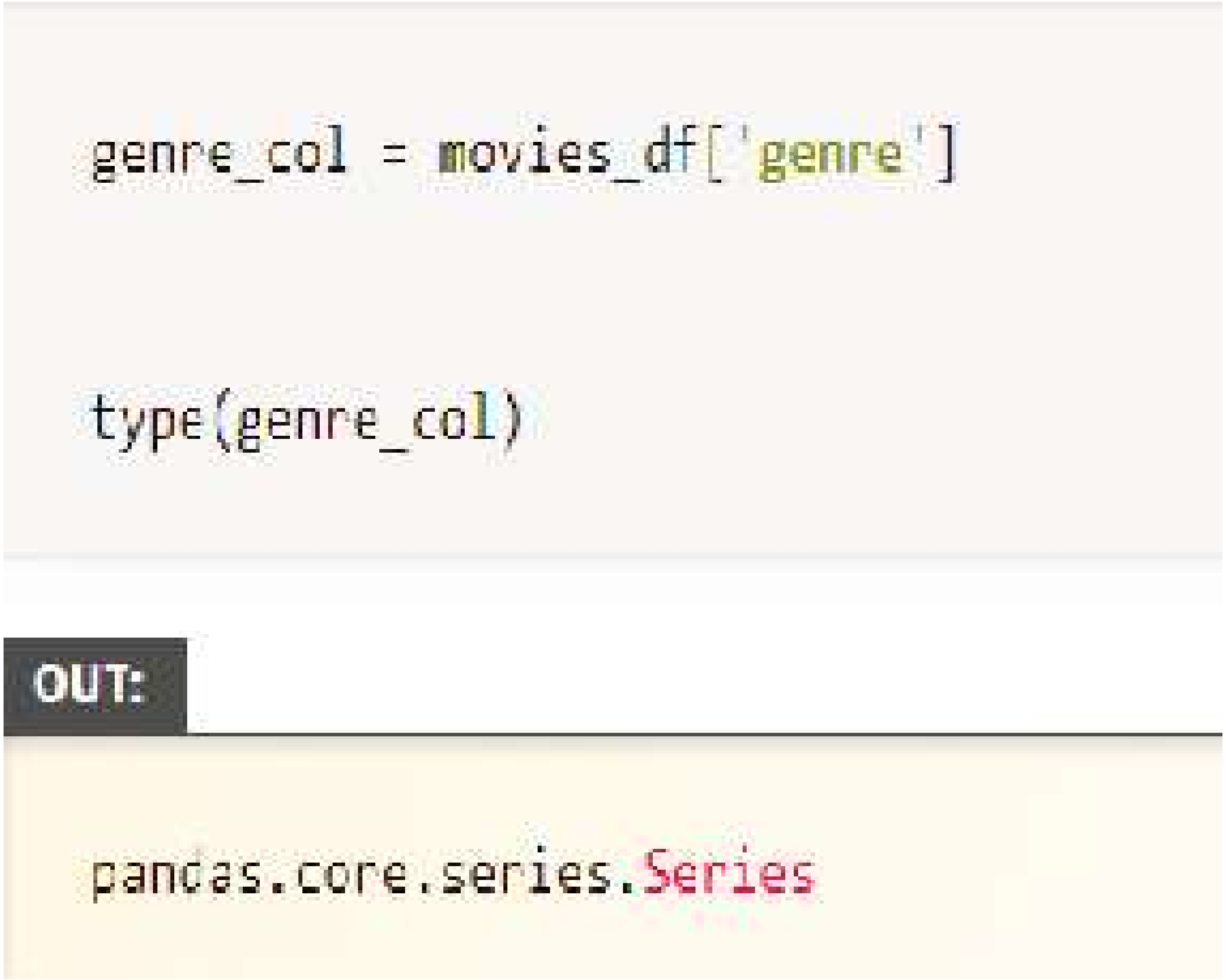
```
movies_df.describe()
```

OUT:

	rank	year	runtime	rating	votes
count	1000.000000	1000.000000	1000.000000	1000.000000	1.000000e+03
mean	500.500000	2012.783000	113.172000	6.723200	1.698083e+05
std	288.819436	3.205962	18.810908	0.945429	1.887626e+05
min	1.000000	2006.000000	66.000000	1.900000	6.100000e+01
25%	250.750000	2010.000000	100.000000	6.200000	3.630900e+04
50%	500.500000	2014.000000	111.000000	6.800000	1.107990e+05
75%	750.250000	2016.000000	123.000000	7.400000	2.399098e+05
max	1000.000000	2016.000000	191.000000	9.000000	1.791916e+06

DataFrame slicing, selecting, extracting

- Below are the other methods of slicing, selecting, and extracting you'll need to use constantly.
- It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.
- Let's look at working with columns first.
- **By column**
- You already saw how to extract a column using square brackets like this:

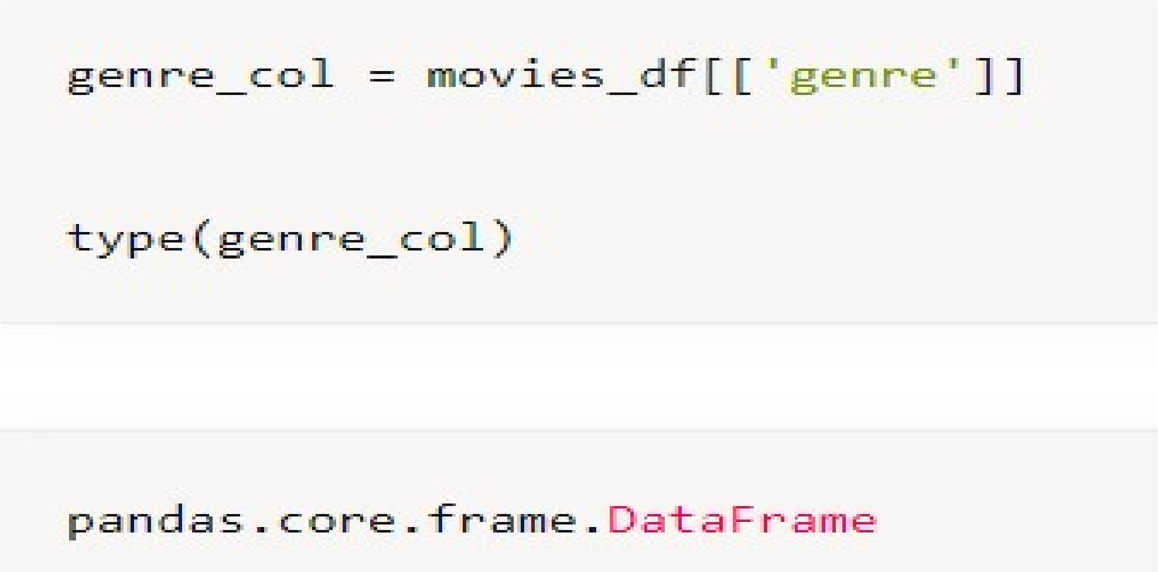


```
genre_col = movies_df['genre']

type(genre_col)
```

OUT:

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

- This will return a *Series*. To extract a column as a *DataFrame*, you need to pass a list of column names. In our case that's just a single column:



```
genre_col = movies_df[['genre']]


type(genre_col)
```

```
pandas.core.frame.DataFrame
```

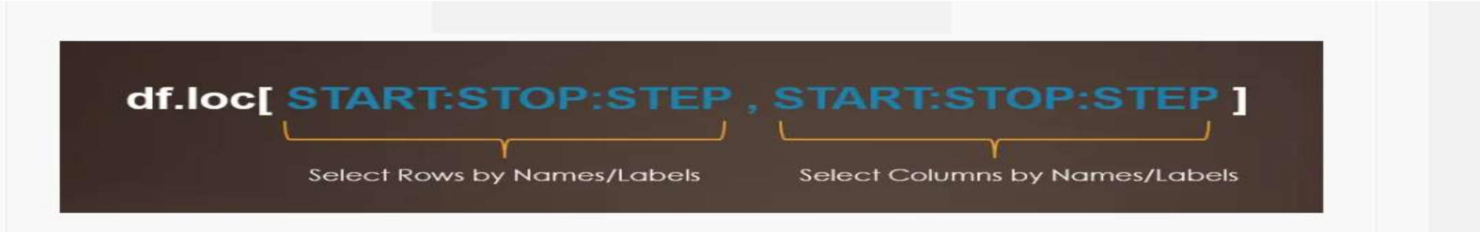
Loc and iloc

- Now we'll look at getting data by rows.
- **By rows**
- For rows, we have two options:

1. .loc - **loc**ates by name
2. .iloc- **loc**ates by numerical **i**ndex

- Remember that we are still indexed by movie Title, so to use .loc we give it the Title of a movie:

- DataFrame.loc[] is label-based to select rows and/or columns in pandas. It accepts single labels, multiple labels from the list, indexes by a range (between two indexes labels), and many more



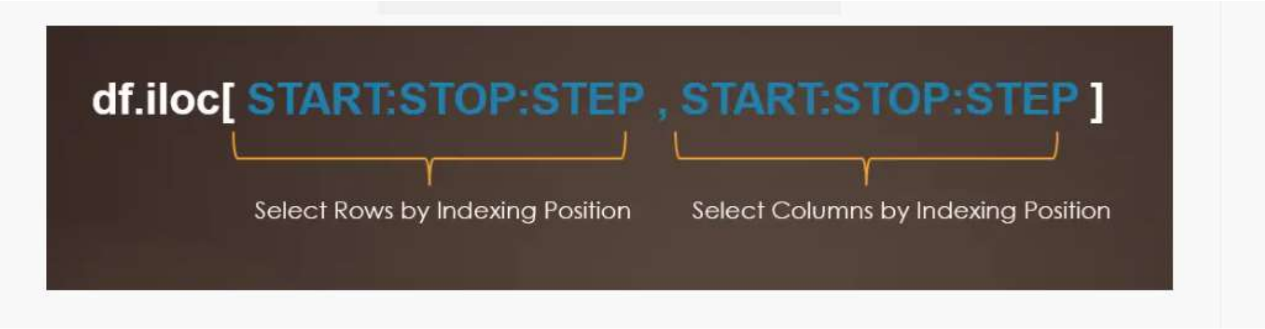
```
df.loc[ START:STOP:STEP , START:STOP:STEP ]
```

- START is the name of the row/column label
- STOP is the name of the last row/column label to take, and
- STEP as the number of indices to advance after each extraction

Some point to note about loc[].

- By not providing a start row/column, loc[] selects from the beginning.
- By not providing stop, loc[] selects all rows/columns from the start label.
- Providing both start and stop, selects all rows/columns in between

- DataFrame.iloc[] is a **index-based to select rows and/or columns** in pandas. It accepts a single index, multiple indexes from the list, indexes by a range, and many more.



```
df.iloc[ START:STOP:STEP , START:STOP:STEP ]
```

Select Rows by Indexing Position
Select Columns by Indexing Position

•START is the integer index of the row/column.

•STOP is the integer index of the last row/column where you wanted to stop the selection, and

•STEP as the number of indices to advance after each extraction.

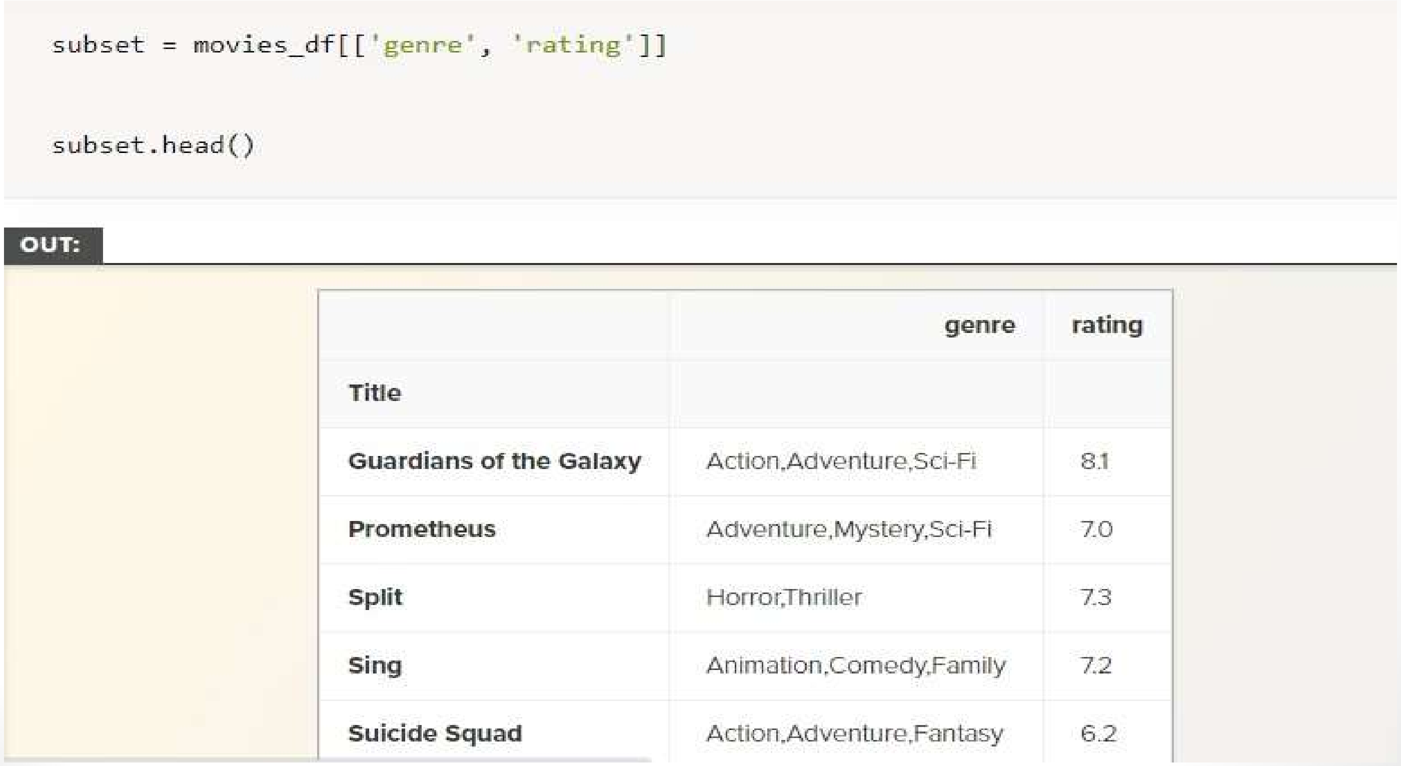
Some point to note about iloc[].

•By not providing a start index, iloc[] selects from the first row/column.

•By not providing stop, iloc[] selects all rows/columns from the start index.

•Providing both start and stop, selects all rows/columns in between.

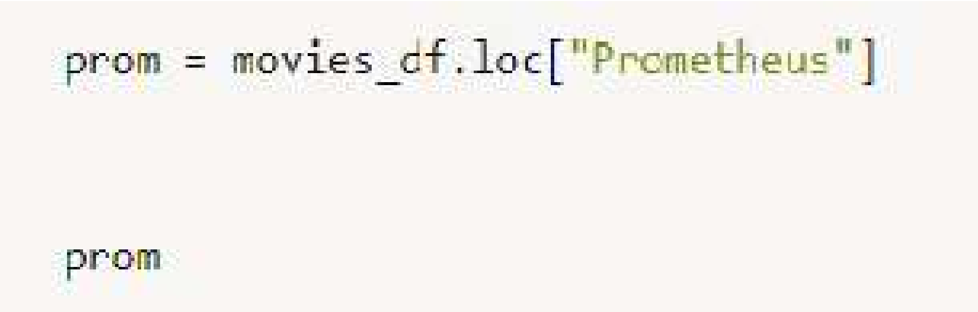
- Since it's just a list, adding another column name is easy:



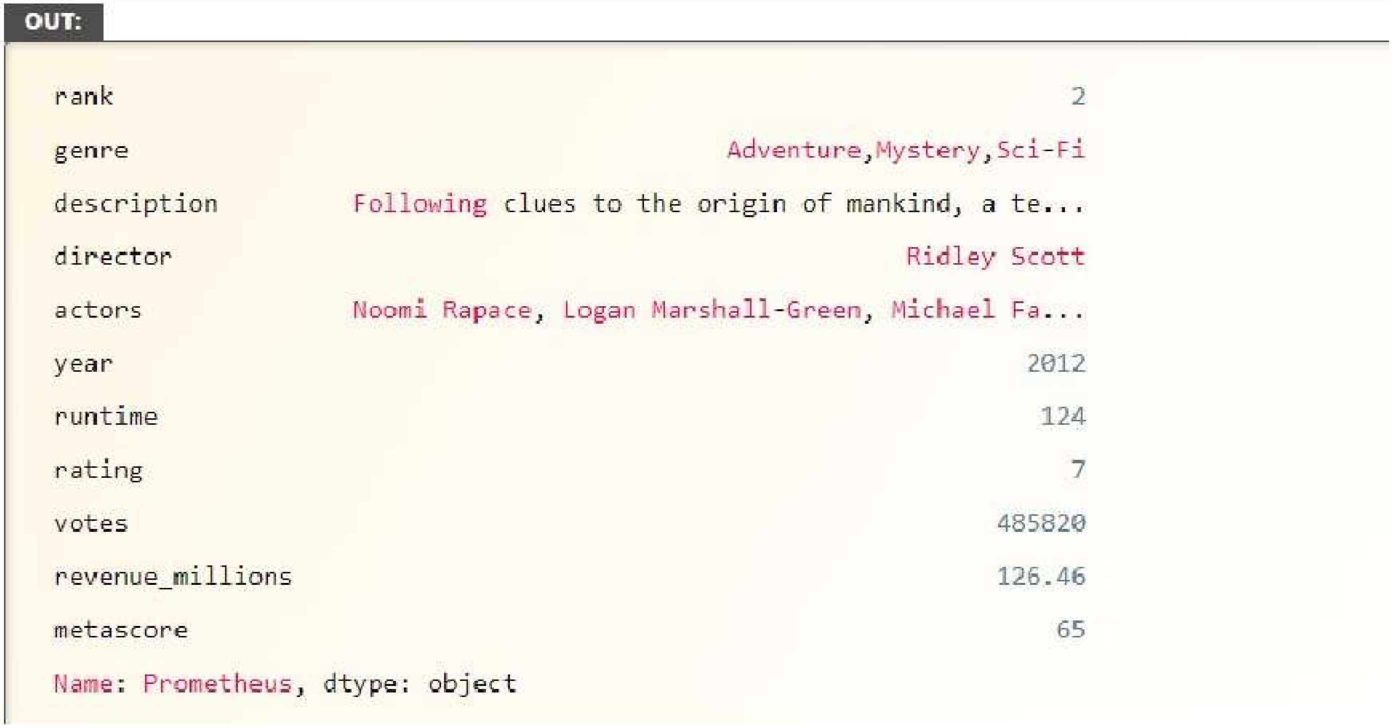
```
subset = movies_df[['genre', 'rating']]


subset.head()
```

OUT:

	genre	rating
Title		
Guardians of the Galaxy	Action,Adventure,Sci-Fi	8.1
Prometheus	Adventure,Mystery,Sci-Fi	7.0
Split	Horror,Thriller	7.3
Sing	Animation,Comedy,Family	7.2
Suicide Squad	Action,Adventure,Fantasy	6.2

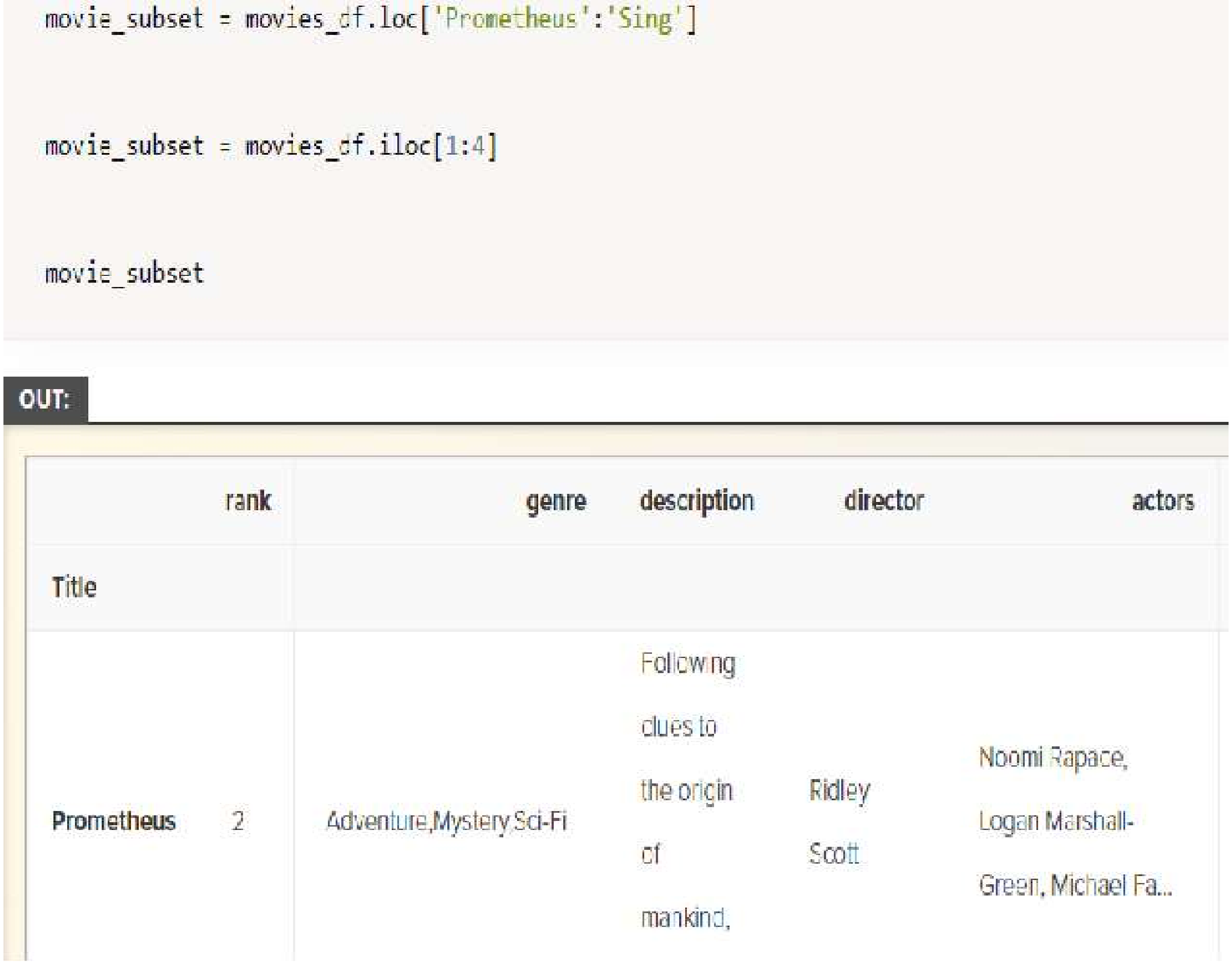


```
prom = movies_df.loc["Prometheus"]


prom
```



```
rank                                                        2
genre                                  Adventure,Mystery,Sci-Fi
description       Following clues to the origin of mankind, a te...
director                                           Ridley Scott
actors            Noomi Rapace, Logan Marshall-Green, Michael Fa...
year                                                       2012
runtime                                                     124
rating                                                        7
votes                                                    485820
revenue_millions                                         126.46
metascore                                                    65
Name: Prometheus, dtype: object
```

- On the other hand, with iloc we give it the numerical index of Prometheus:
- prom = movies_df.iloc[1]
- loc and iloc can be thought of as similar to Python list slicing. To show this even further, let's select multiple rows.
- How would you do it with a list? In Python, just slice with brackets like example_list[1:4]. It's works the same way in pandas:

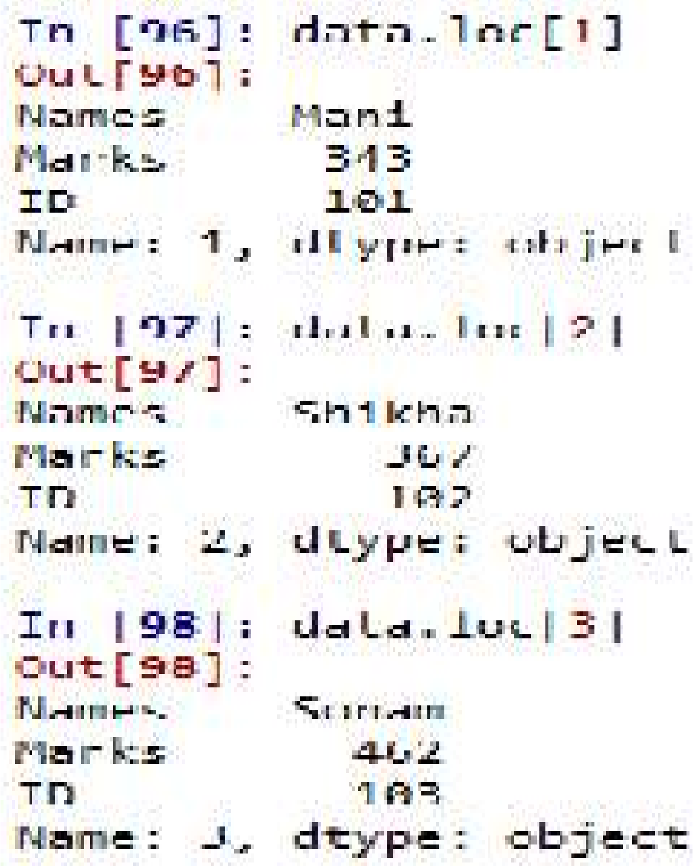


```
movie_subset = movies_df.loc['Prometheus':'Sing']


movie_subset = movies_df.iloc[1:4]


movie_subset
```

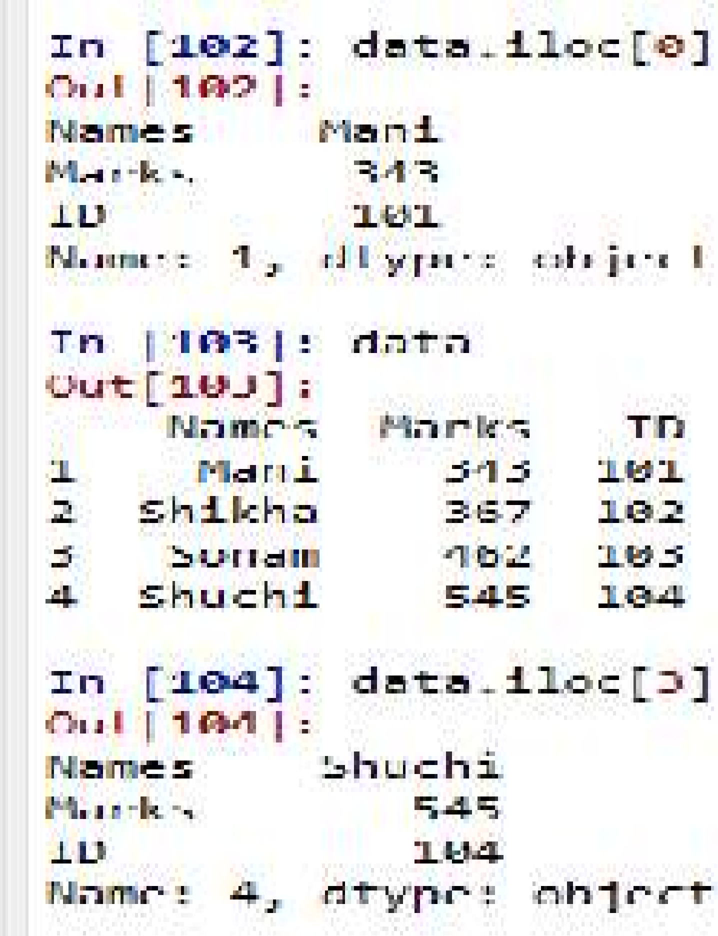
OUT:

	rank	genre	description	director	actors
Title					
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind,	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...

- One important distinction between using .loc and .iloc to select multiple rows is that .loc includes the movie *Sing* in the result, but when using .iloc we're getting rows 1:4 but the movie at index 4 (*Suicide Squad*) is not included.
- Slicing with .iloc follows the same rules as slicing with lists, the object at the index at the end is not included.



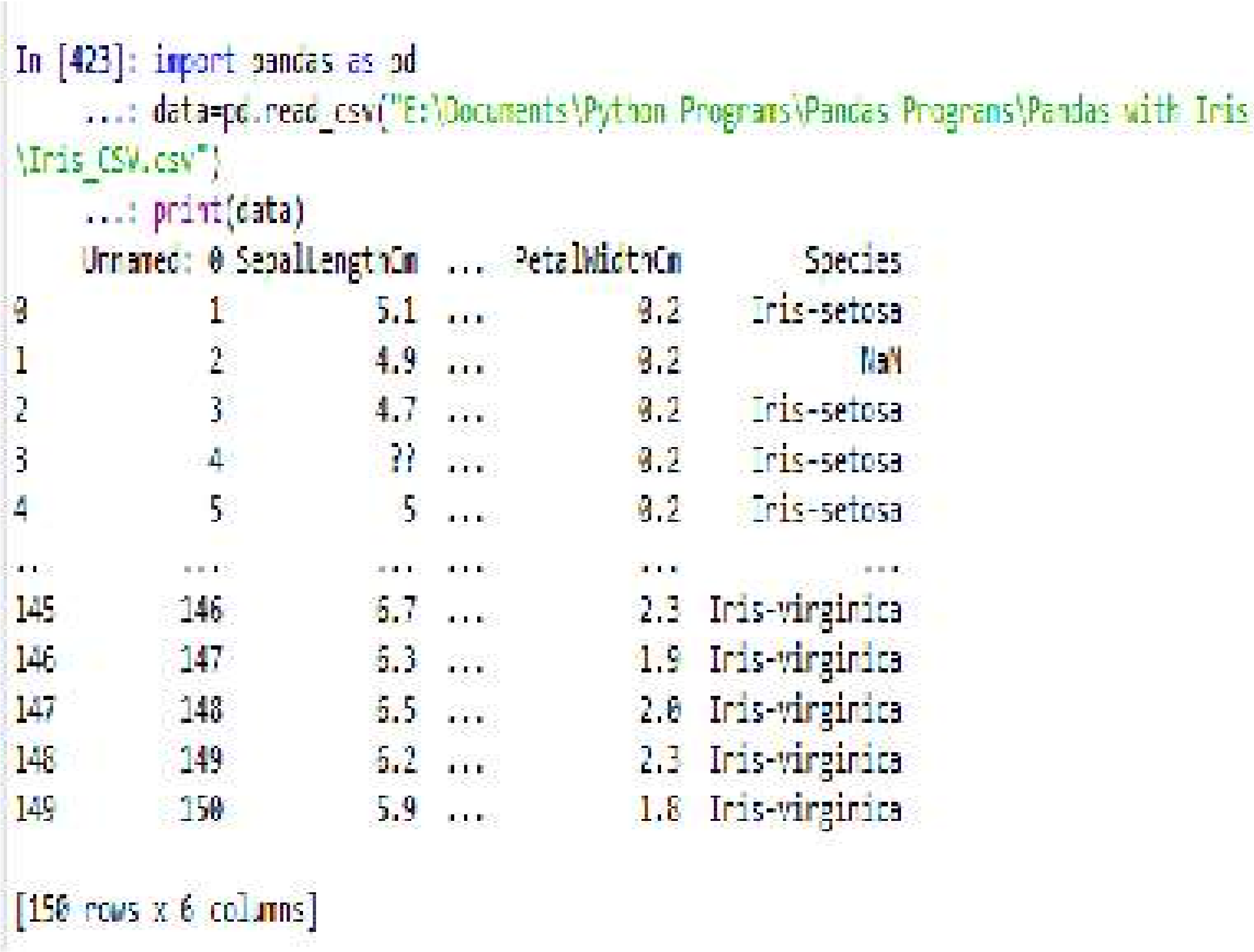
```
In [96]: data.loc[1]
Out[96]:
Names    Mani
Marks     343
ID        101
Name: 1, dtype: object

In [97]: data.loc[2]
Out[97]:
Names    Shikha
Marks       367
ID          102
Name: 2, dtype: object

In [98]: data.loc[3]
Out[98]:
Names    Sonam
Marks      402
ID         103
Name: 3, dtype: object
```



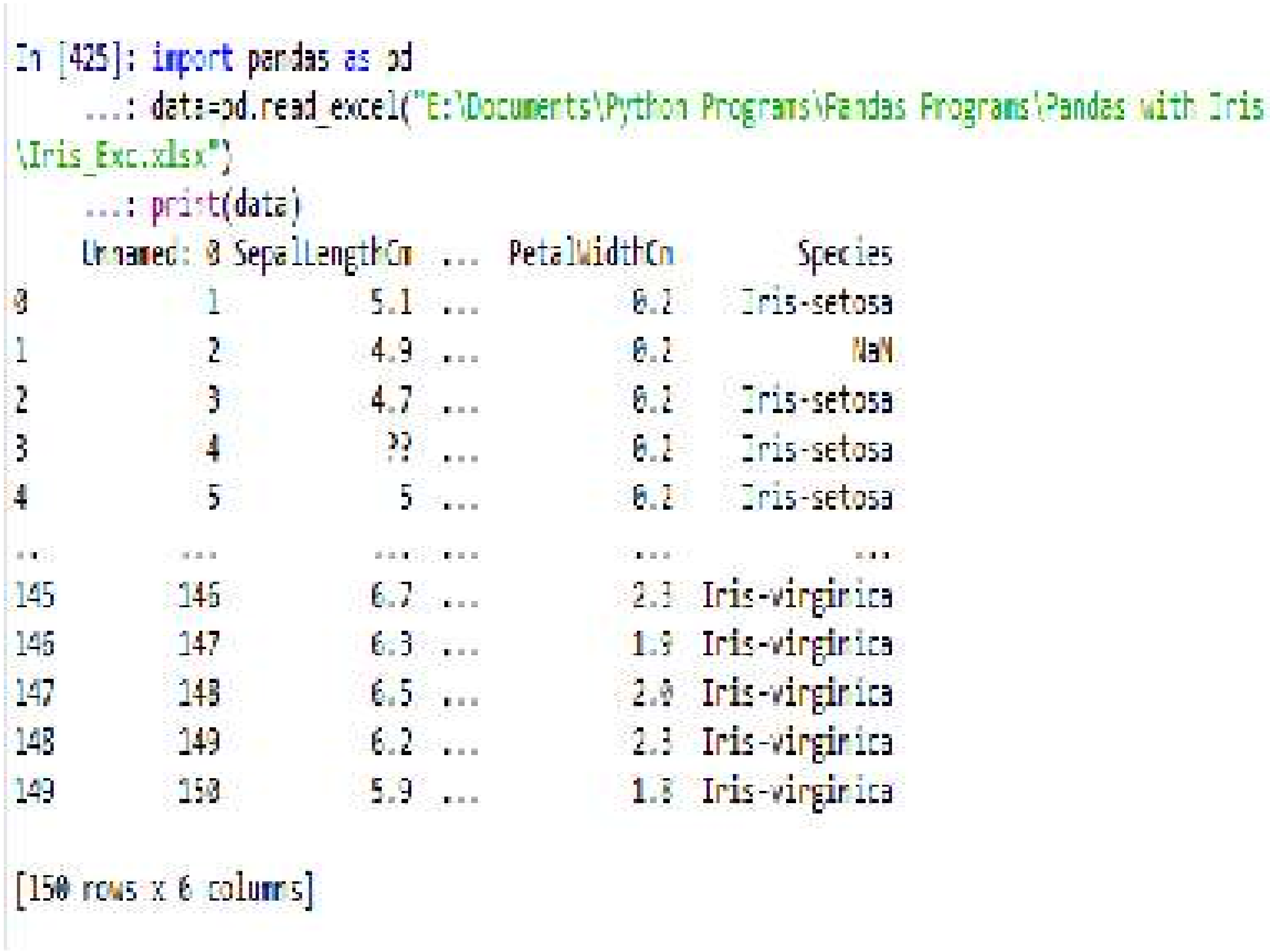
```
In [102]: data.iloc[0]
Out[102]:
Names     Mani
Marks      343
ID         101
Name: 1, dtype: object

In [103]: data
Out[103]:
    Names  Marks   ID
1    Mani    343  101
2  Shikha    367  102
3   Sonam    402  103
4  Shuchi    545  104

In [104]: data.iloc[3]
Out[104]:
Names     Shuchi
Marks        545
ID           104
Name: 4, dtype: object
```

Reading Files with Pandas

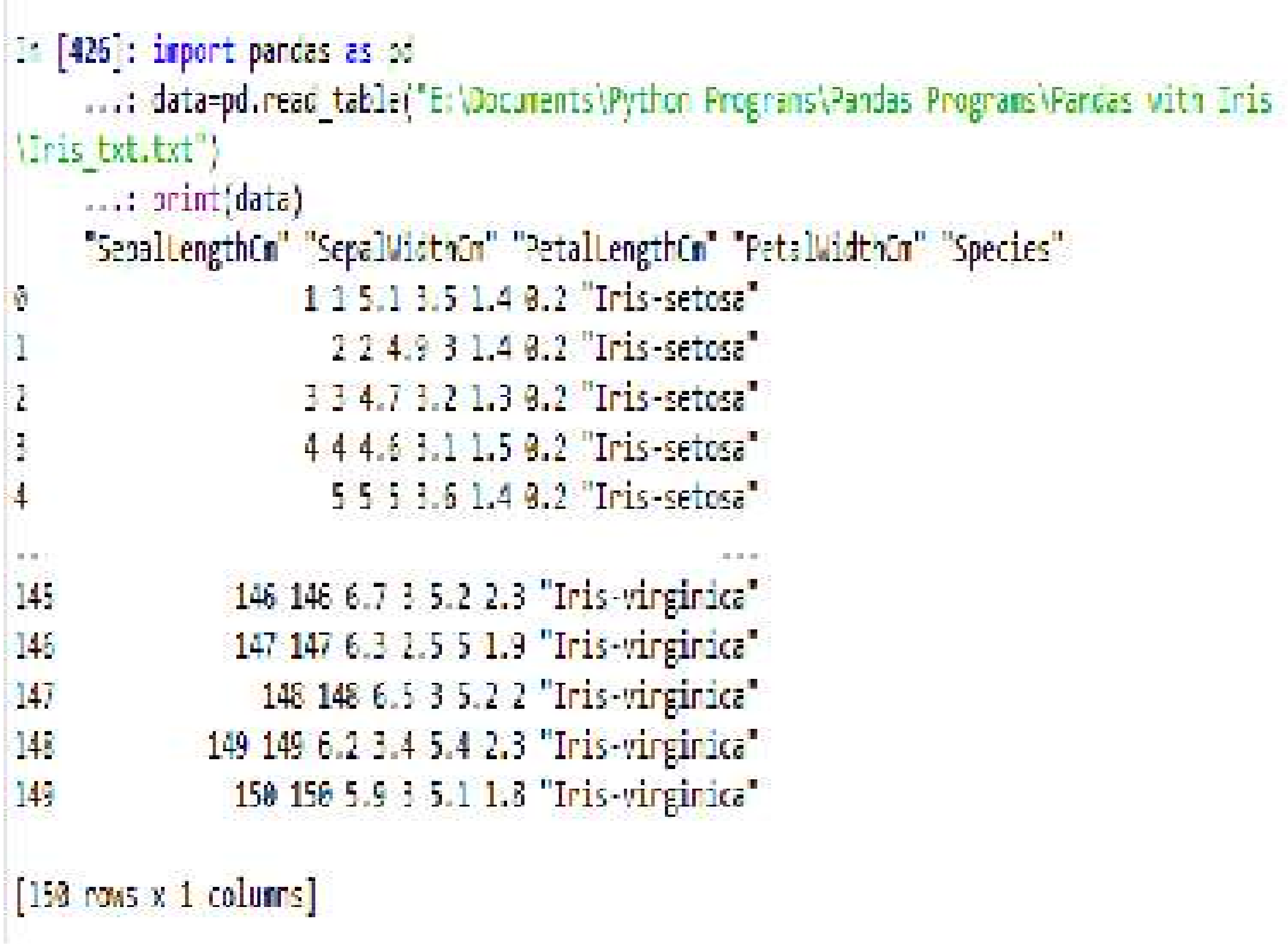
- Reading CSV Files
  - read_csv()
- Reading Excel Files
  - read_excel()
- Reading Text Files
  - read_table()



```
In [423]: import pandas as pd
     ...: data=pd.read_csv("E:\Documents\Python Programs\Pandas Programs\Pandas with Iris
\Iris_CSV.csv")
     ...: print(data)
     Unnamed: 0 SepalLengthCm  ...  PetalWidthCm         Species
0             1           5.1  ...           0.2     Iris-setosa
1             2           4.9  ...           0.2             NaN
2             3           4.7  ...           0.2     Iris-setosa
3             4            ??  ...           0.2     Iris-setosa
4             5             5  ...           0.2     Iris-setosa
..          ...           ...  ...           ...             ...
145         146           6.7  ...           2.3  Iris-virginica
146         147           6.3  ...           1.9  Iris-virginica
147         148           6.5  ...           2.0  Iris-virginica
148         149           6.2  ...           2.3  Iris-virginica
149         150           5.9  ...           1.8  Iris-virginica

[150 rows x 6 columns]
```



```
In [425]: import pandas as pd
     ...: data=pd.read_excel("E:\Documents\Python Programs\Pandas Programs\Pandas with Iris
\Iris_Exc.xlsx")
     ...: print(data)
     Unnamed: 0 SepalLengthCm  ...  PetalWidthCm         Species
0             1           5.1  ...           0.2     Iris-setosa
1             2           4.9  ...           0.2             NaN
2             3           4.7  ...           0.2     Iris-setosa
3             4            ??  ...           0.2     Iris-setosa
4             5             5  ...           0.2     Iris-setosa
..          ...           ...  ...           ...             ...
145         146           6.7  ...           2.3  Iris-virginica
146         147           6.3  ...           1.9  Iris-virginica
147         148           6.5  ...           2.0  Iris-virginica
148         149           6.2  ...           2.3  Iris-virginica
149         150           5.9  ...           1.8  Iris-virginica

[150 rows x 6 columns]
```



```
In [426]: import pandas as pd
     ...: data=pd.read_table("E:\Documents\Python Programs\Pandas Programs\Pandas with Iris
\Iris_txt.txt")
     ...: print(data)
     "SepalLengthCm" "SepalWidthCm" "PetalLengthCm" "PetalWidthCm" "Species"
0                    1 1 5.1 3.5 1.4 0.2 "Iris-setosa"
1                      2 2 4.9 3 1.4 0.2 "Iris-setosa"
2                    3 3 4.7 3.2 1.3 0.2 "Iris-setosa"
3                    4 4 4.6 3.1 1.5 0.2 "Iris-setosa"
4                      5 5 5 3.6 1.4 0.2 "Iris-setosa"
..                                                 ...
145             146 146 6.7 3 5.2 2.3 "Iris-virginica"
146             147 147 6.3 2.5 5 1.9 "Iris-virginica"
147               148 148 6.5 3 5.2 2 "Iris-virginica"
148           149 149 6.2 3.4 5.4 2.3 "Iris-virginica"
149             150 150 5.9 3 5.1 1.8 "Iris-virginica"

[150 rows x 1 columns]
```

Matplotlib Tutorial

What is Matplotlib

- To make necessary statistical inferences, it becomes necessary to visualize your data and Matplotlib is one such solution for the Python users.
- It is a very powerful plotting library useful for those working with Python and NumPy.
- The most used module of Matplotib is Pyplot which provides an interface like MATLAB but instead, it uses Python and it is open source.

What Is Python Matplotlib?

- **matplotlib.pyplot** is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.
- **What is Matplotlib used for?**
- Matploitlib is a Python Library used for plotting, this python library provides and objected-oriented APIs for integrating plots into applications.

Installing Matplotlib

- To install Matplotlib on your local machine, open Python command prompt and type following commands:

- **python -m pip install -U pip**
  python -m pip install -U matplotlib

General Concepts

- A Matplotlib figure can be categorized into several parts as below:

1.Figure: It is a whole figure which may contain one or more than one axes (plots). You can think of a **Figure** as a canvas which contains plots.

2.Axes: It is what we generally think of as a plot. A **Figure** can contain many Axes. It contains two or three (in the case of 3D) **Axis** objects. Each Axes has a title, an x-label and a y-label.

3. Axis: They are the number line like objects and take care of generating the graph limits.

4.Artist: Everything which one can see on the figure is an artist like Text objects, Line2D objects, collection objects. Most Artists are tied to Axes.

Getting Started with Pyplot

- Pyplot is a module of Matplotlib which provides simple functions to add plot elements like lines, images, text, etc. to the current axes in the current figure.
- **Make a simple plot**

  -
    ```
    import matplotlib.pyplot as plt
    import numpy as np
    ```

- Here we import Matplotlib’s Pyplot module and Numpy library as most of the data that we will be working with will be in the form of arrays only.

- ***Syntax:*** matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)

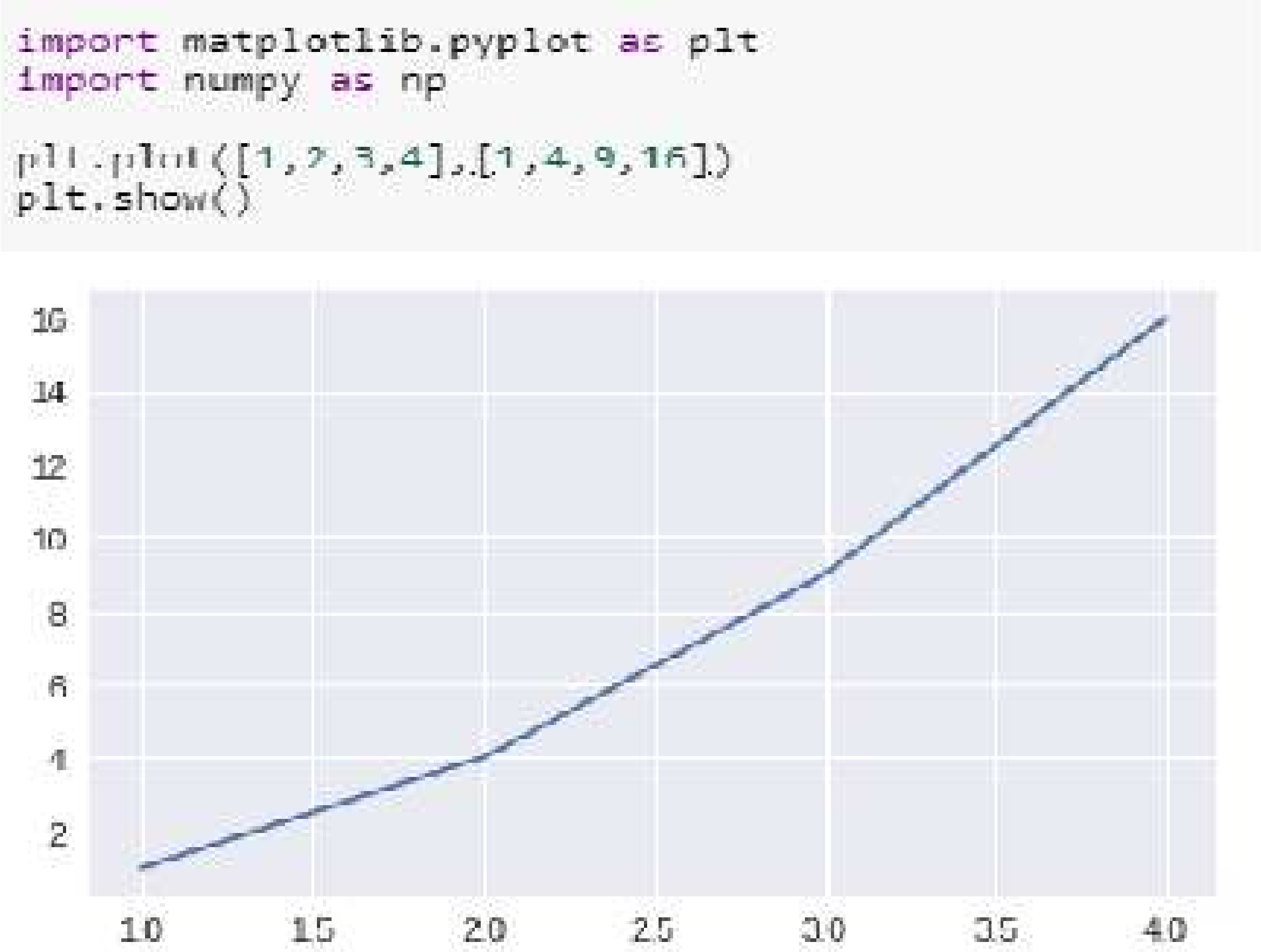
•***x, y:*** *These parameter are the horizontal and vertical coordinates of the data points. x values are optional.*

•***fmt:*** *This parameter is an optional parameter and it contains the string value.*

•***data:*** *This parameter is an optional parameter and it is an object with labelled data.*

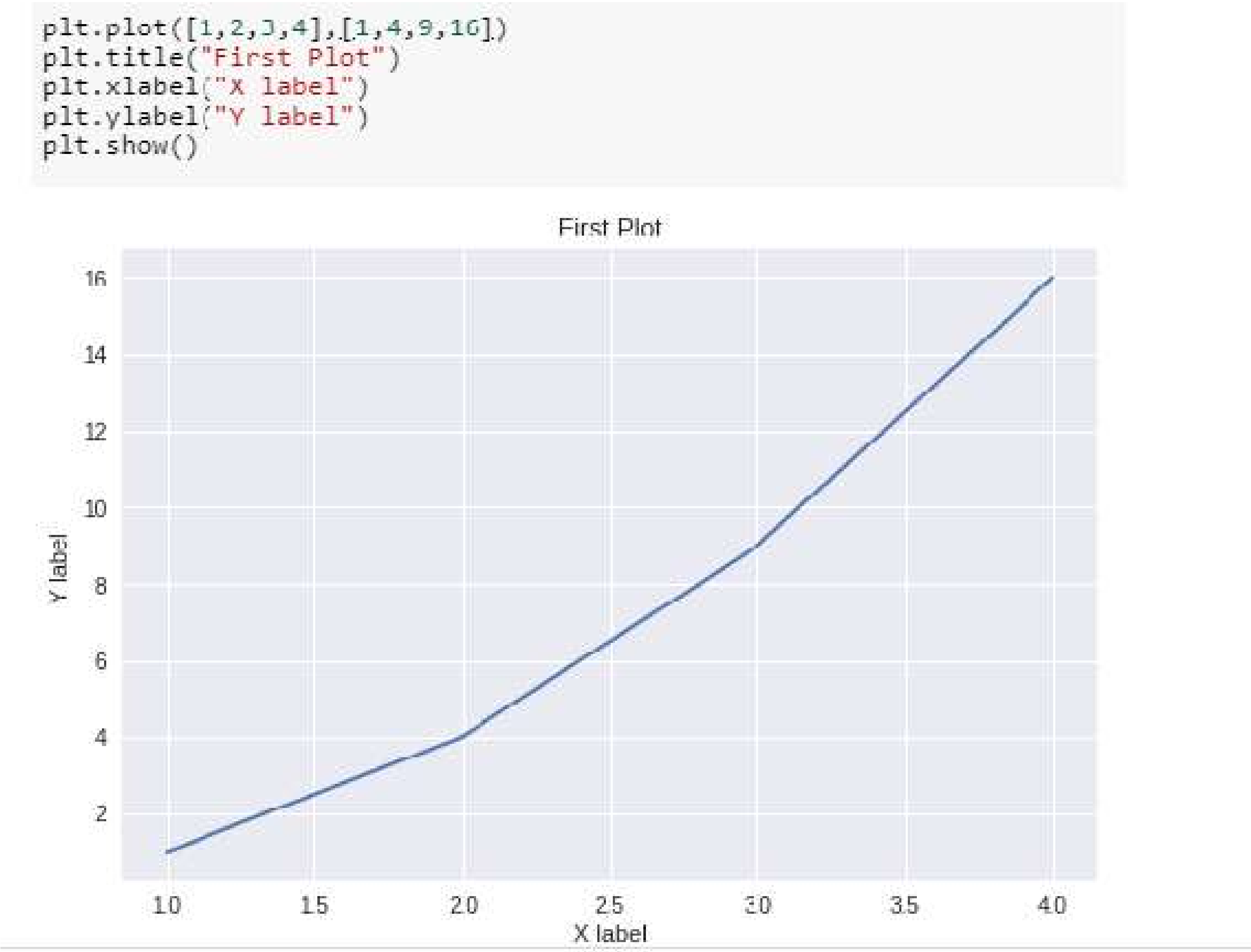
Returns: *This returns the following:*

•***lines :*** *This returns the list of Line2D objects representing the plotted data.*



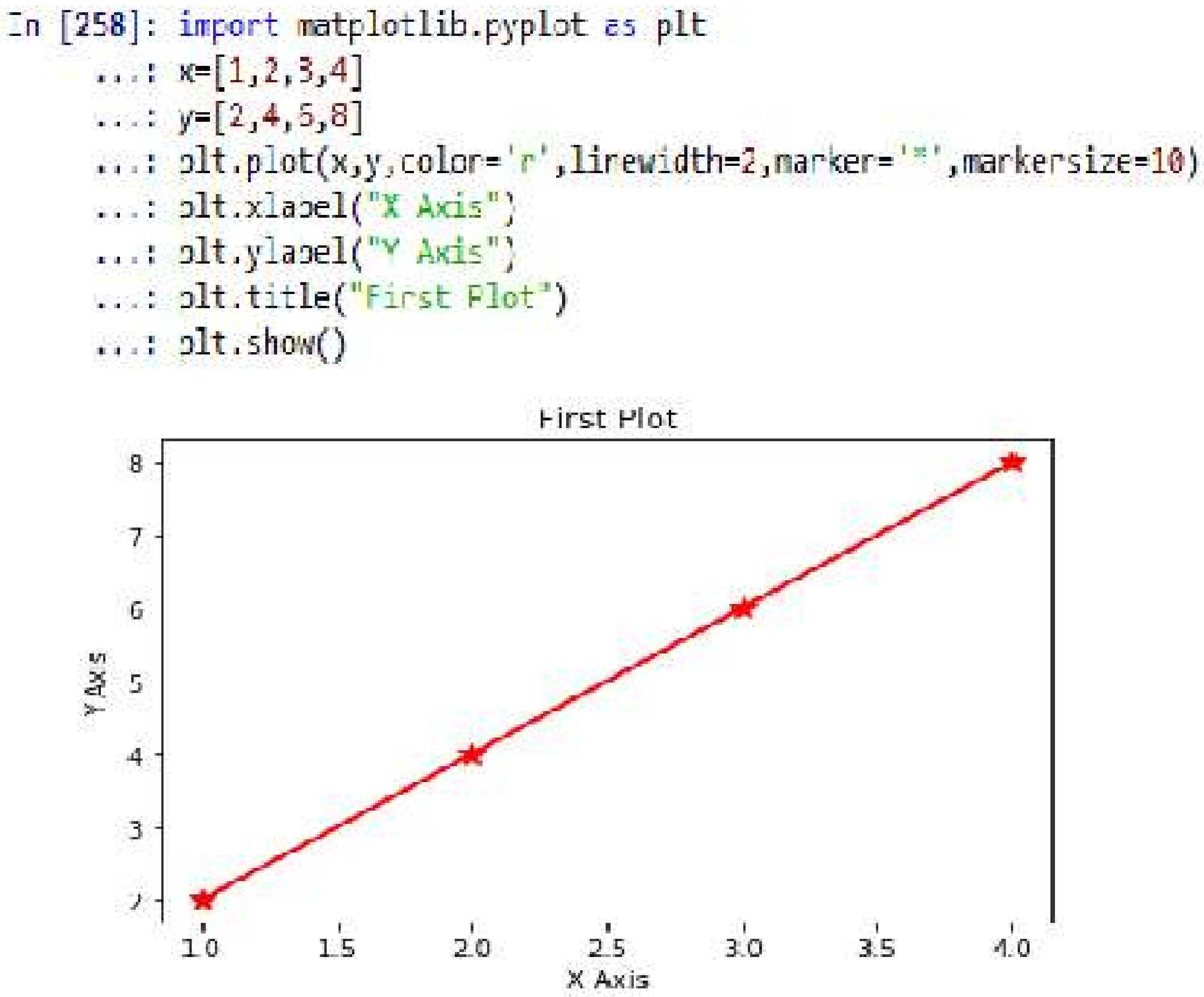
```
import matplotlib.pyplot as plt
import numpy as np

plt.plot([1,2,3,4],[1,4,9,16])
plt.show()
```

- We pass two arrays as our input arguments to Pyplot’s plot() method and use show() method to invoke the required plot.
- Here note that the first array appears on the x-axis and second array appears on the y-axis of the plot.
- Now that our first plot is ready, let us add the title, and name x-axis and y-axis using
  methods title(), xlabel() and ylabel() respectively.

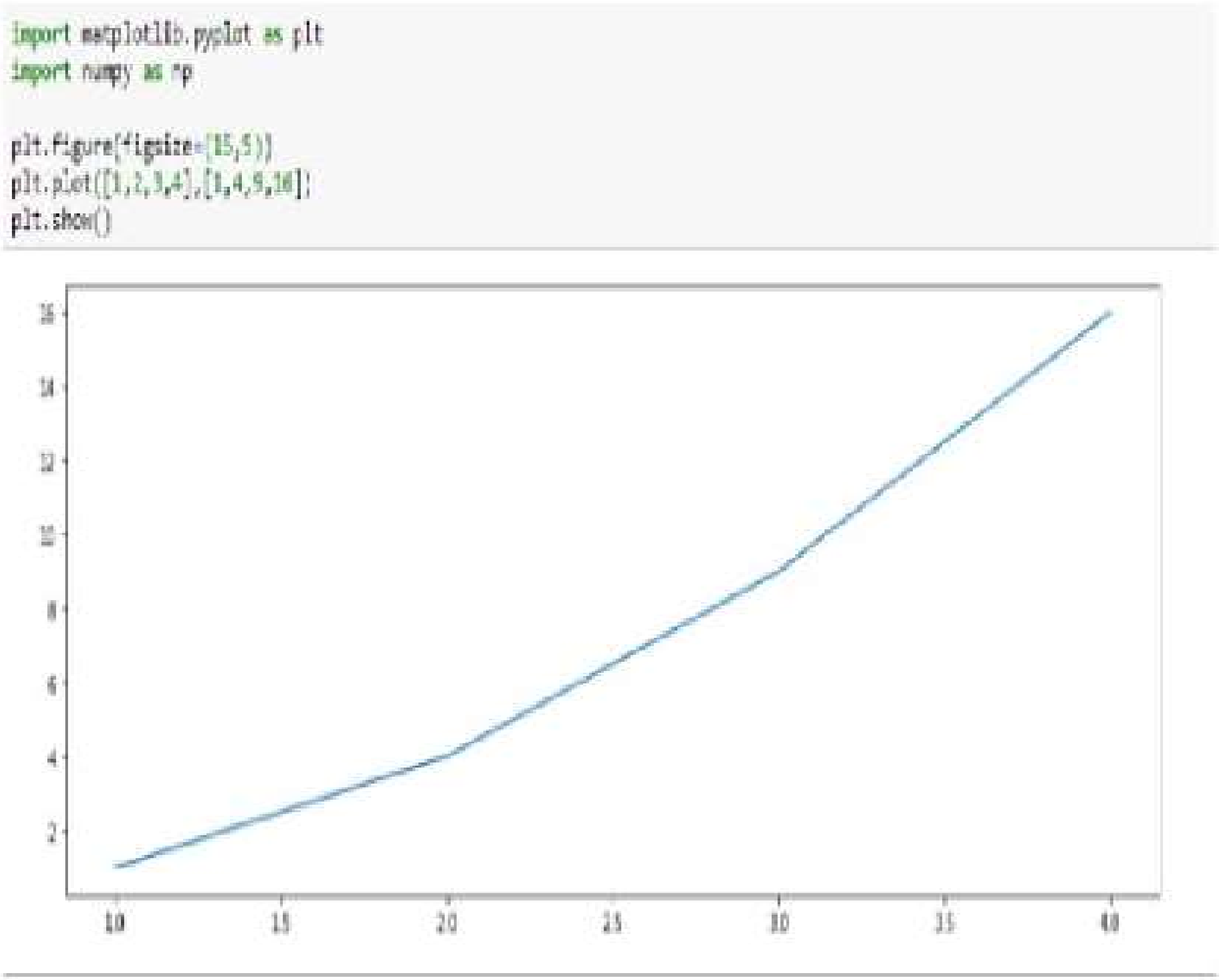


```
plt.plot([1,2,3,4],[1,4,9,16])
plt.title("First Plot")
plt.xlabel("X label")
plt.ylabel("Y label")
plt.show()
```

- We can also specify the size of the figure using method figure() and passing the values as a tuple of the length of rows and columns to the argument figsize

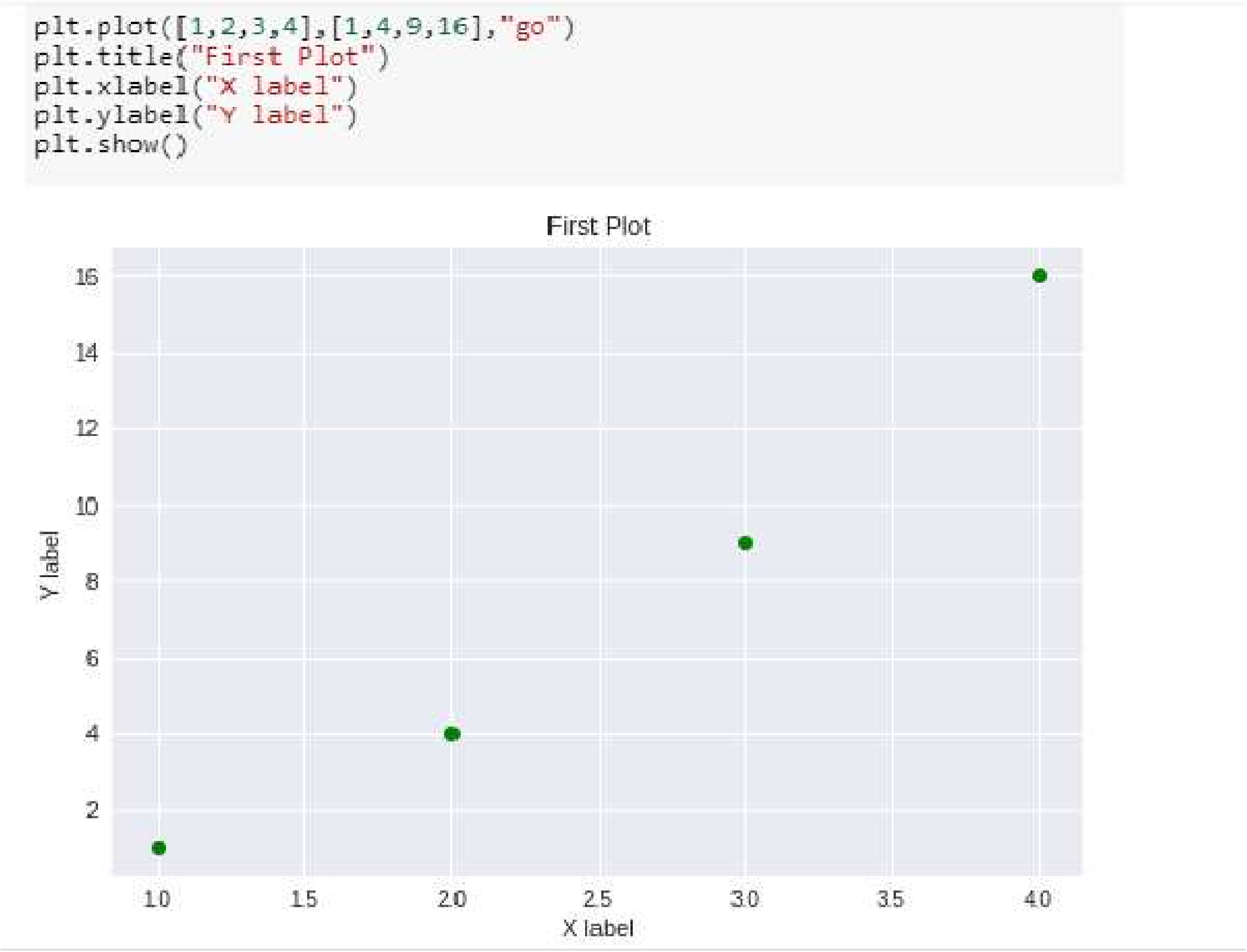


```
In [258]: import matplotlib.pyplot as plt
     ...: x=[1,2,3,4]
     ...: y=[2,4,6,8]
     ...: plt.plot(x,y,color='r',linewidth=2,marker='*',markersize=10)
     ...: plt.xlabel("X Axis")
     ...: plt.ylabel("Y Axis")
     ...: plt.title("First Plot")
     ...: plt.show()
```



```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(15,5))
plt.plot([1,2,3,4],[1,4,9,16])
plt.show()
```

- With every X and Y argument, you can also pass an optional third argument in the form of a string which indicates the colour and line type of the plot.
- The default format is **b-** which means a solid blue line. In the figure below we use **go** which means green circles. Likewise, we can make many such combinations to format our plot.

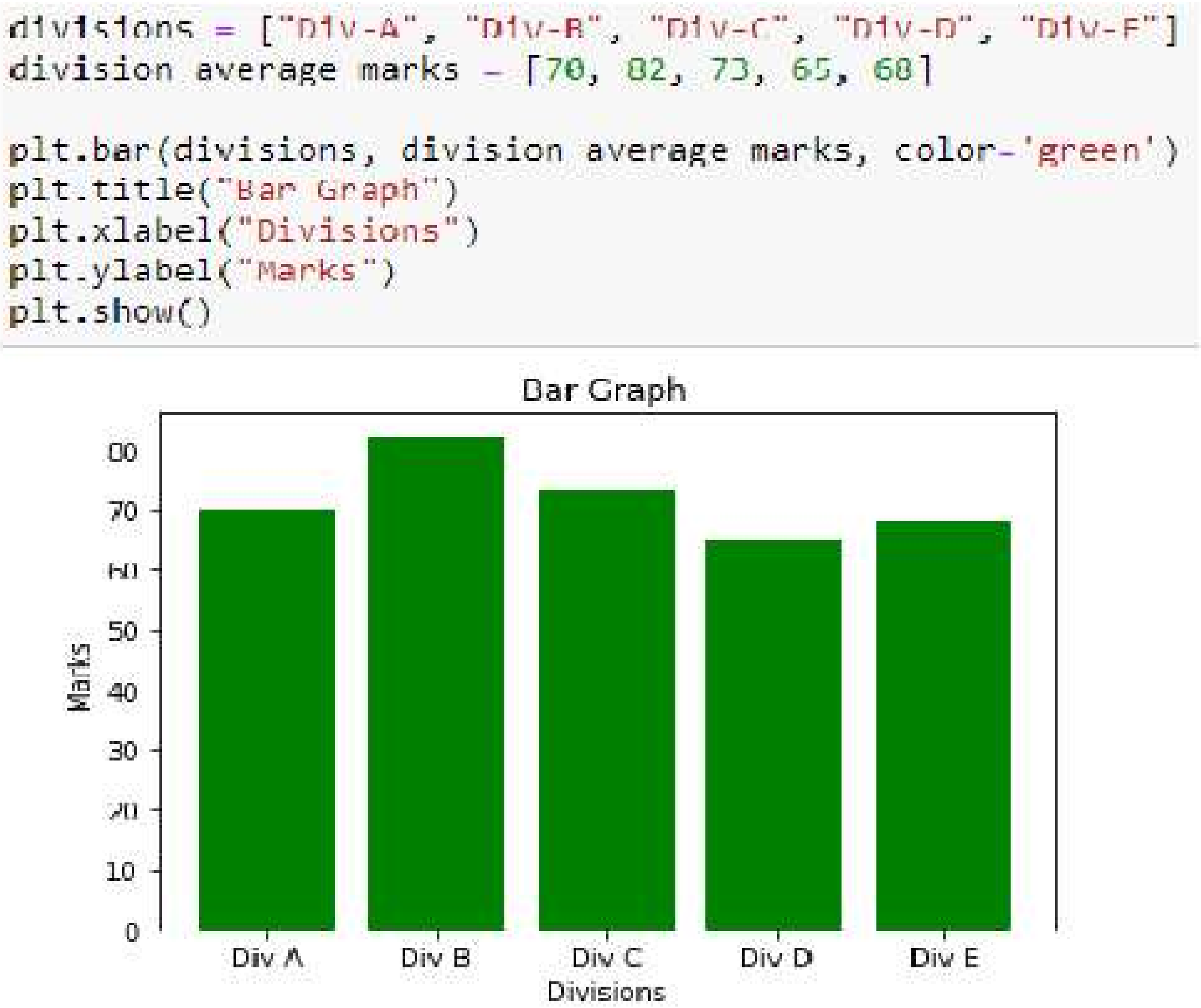


```
plt.plot([1,2,3,4],[1,4,9,16],"go")
plt.title("First Plot")
plt.xlabel("X label")
plt.ylabel("Y label")
plt.show()
```

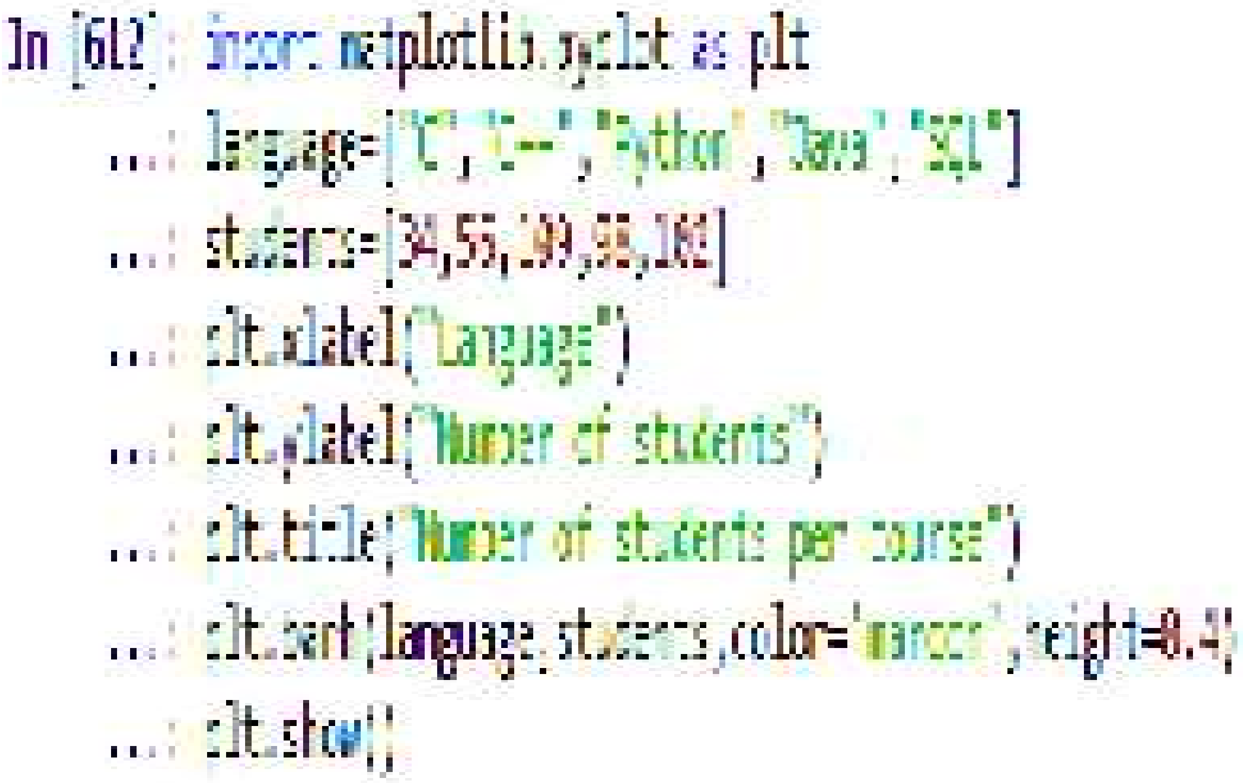
Bar Graph in matplotlib

- **1) Bar Graphs**
- Bar graphs are one of the most common types of graphs and are used to show data associated with the categorical variables. Pyplot provides a method bar() to make bar graphs which take arguments: categorical variables, their values and color (if you want to specify any).
-
```
plt.bar(x, height, width, bottom, align)
```

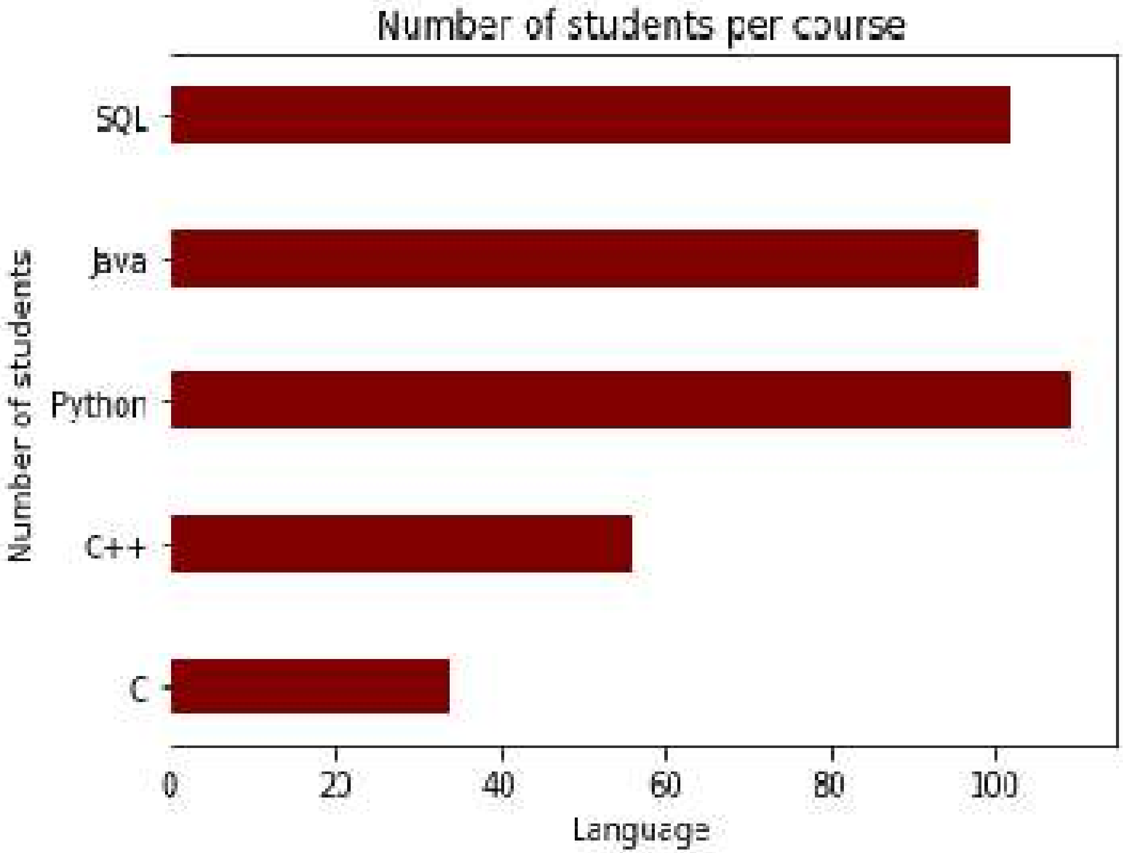
- **Parameters:**
- **x :**
  It is the sequence of scalars representing the x coordinates of the bars. align controls if x is the bar **center**(default) or **left** edge.
- **height :**
  It is a scalar or sequence of scalars representing the height(s) of the bars.
- **width:**
  It is a scalar or array-like value, although optional. However, the width(s) of the bars is by default 0.8.
- **bottom:**
  It is a scalar or array-like value, although optional. Nevertheless, the coordinate(s) of the bars is by default None.
- **align:**
  It aligns the plot in either center or edge. Also, this is an optional parameter; hence by default, it is always the center.



```
divisions = ["Div-A", "Div-B", "Div-C", "Div-D", "Div-E"]
division_average_marks = [70, 82, 73, 65, 68]

plt.bar(divisions, division_average_marks, color='green')
plt.title("Bar Graph")
plt.xlabel("Divisions")
plt.ylabel("Marks")
plt.show()
```



```
In [612]: import matplotlib.pyplot as plt
     ...: language=["C","C++","Python","Java","SQL"]
     ...: students=[34,55,109,98,102]
     ...: plt.xlabel("Language")
     ...: plt.ylabel("Number of students")
     ...: plt.title("Number of students per course")
     ...: plt.barh(language,students,color='maroon',height=0.4)
     ...: plt.show()
```


Number of students per course
SQL
Java
Python
C++
C
Number of students
0
20
40
60
80
100
Language

3) Histogram

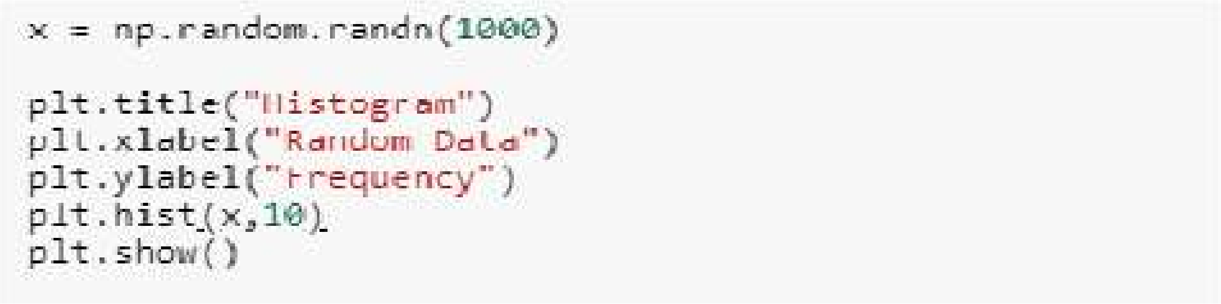
- Histograms are a very common type of plots when we are looking at data like height and weight, stock prices, waiting time for a customer, etc which are continuous in nature. Histogram's data is plotted within a range against its frequency.
- Histograms are very commonly occurring graphs in probability and statistics and form the basis for various distributions like the normal -distribution, t-distribution, etc.
- In the following example, we generate a random continuous data of 1000 entries and plot it against its frequency with the data divided into 10 equal strata. We have used

  NumPy's random.randn() method which generates data with the properties of a standard normal distribution i.e. mean = 0 and standard deviation = 1, and hence the histogram looks like a normal distribution curve.

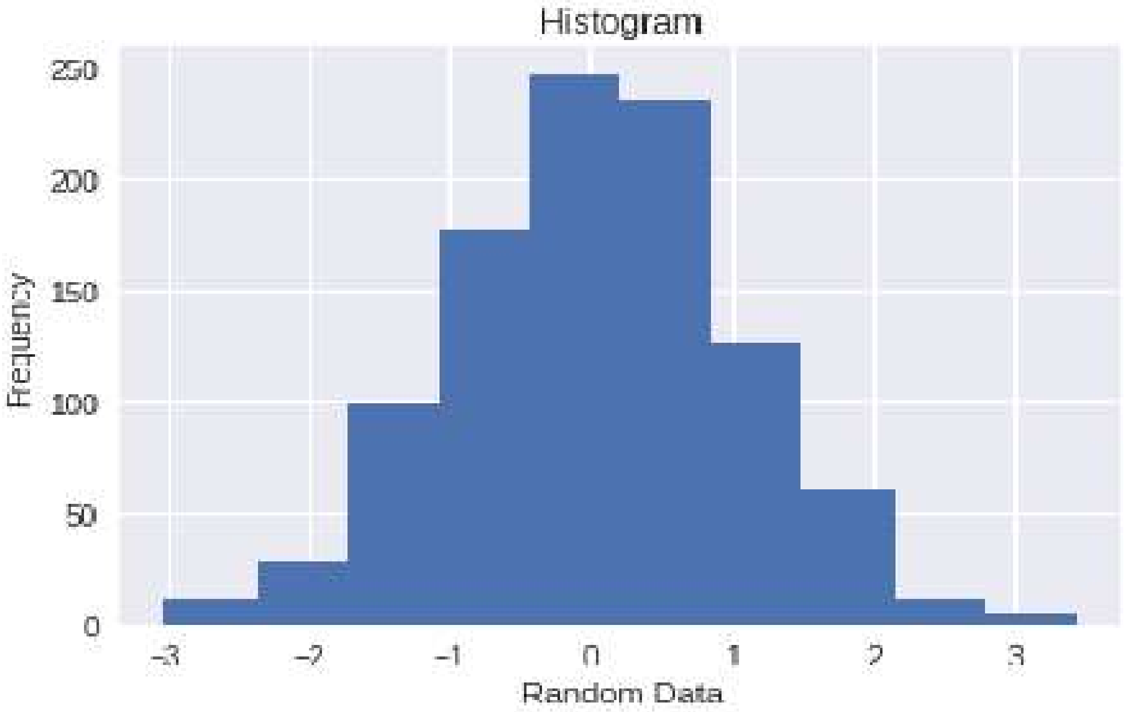
- *The matplotlib.pyplot.hist() function is used to compute and create histogram of x.*

Attribute	parameter
x	array or sequence of array
bins	optional parameter contains integer or sequence or strings
density	optional parameter contains boolean values
range	optional parameter represents upper and lower range of bins
histtype	optional parameter used to create type of histogram [bar, barstacked, step, stepfilled], default is “bar”

align	optional parameter controls the plotting of histogram [left, right, mid]
weights	optional parameter contains array of weights having same dimensions as x
bottom	location of the baseline of each bin
rwidth	optional parameter which is relative width of the bars with respect to bin width
color	optional parameter used to set color or sequence of color specs
label	optional parameter string or sequence of string to match with multiple datasets
log	optional parameter used to set histogram axis on log scale



```
x = np.random.randn(1000)

plt.title("Histogram")
plt.xlabel("Random Data")
plt.ylabel("Frequency")
plt.hist(x,10)
plt.show()
```



4)Scatter Plots and 3-D plotting

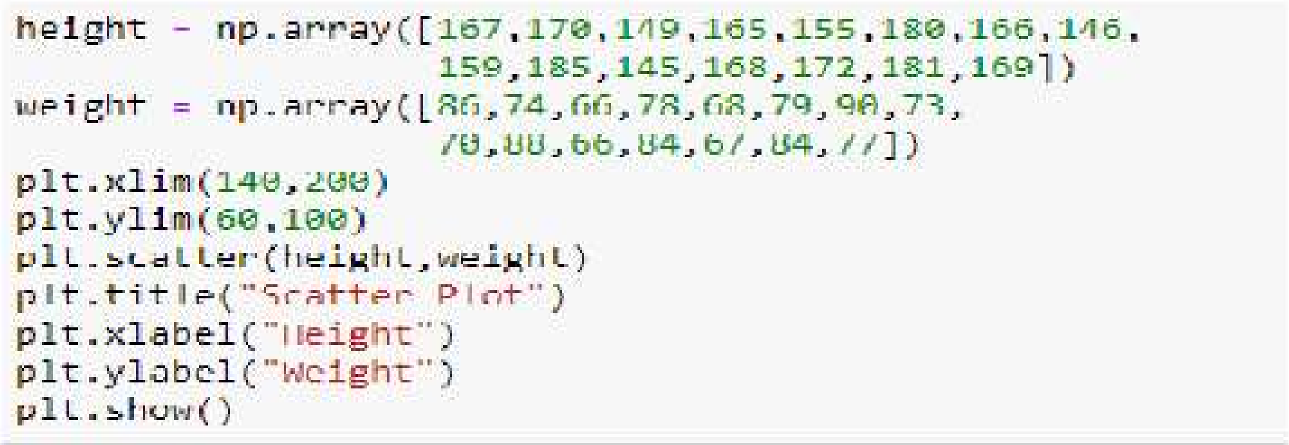
- Scatter plots are widely used graphs, especially they come in handy in visualizing a problem of regression. In the following example, we feed in arbitrarily created data of height and weight and plot them against each other. We used xlim() and ylim() methods to set the limits of X-axis and Y-axis respectively.
- Scatter plots are used to plot data points on a horizontal and a vertical axis in the attempt to show how much one variable is affected by another.

Scatter plots are utilized to see how different variables are related to each other. The dots on the plot shows how the variables are related. A scatter plot is made with the matplotlib library's **scatter() method.**

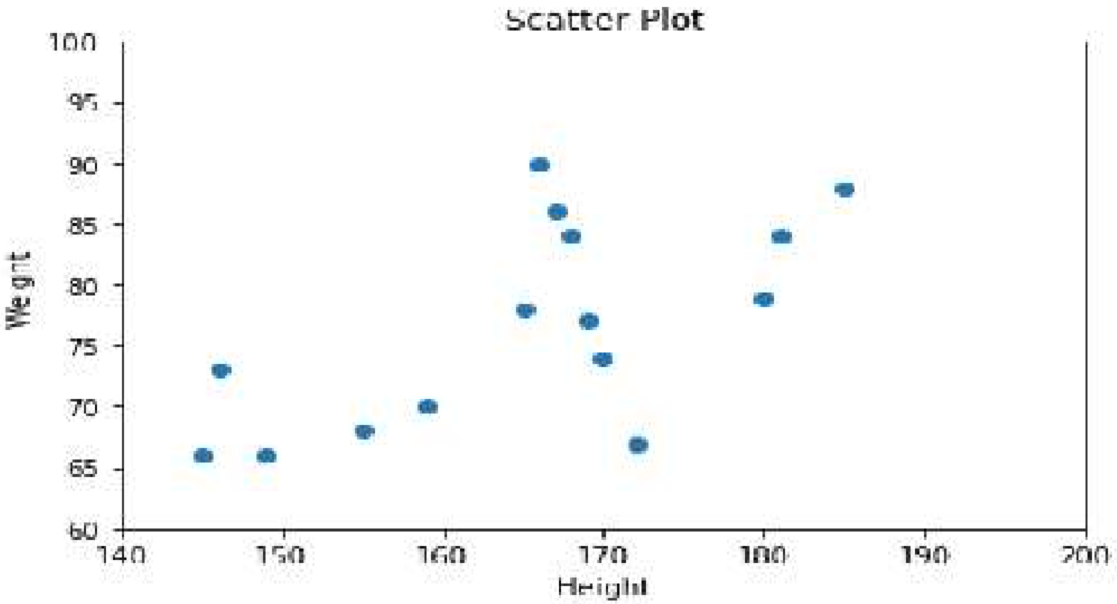
People often use scatter plots to show the relationship between two or more variables and how a change in one affects the other.

- matplotlib.pyplot.scatter (x_axis_value, y_axis_value, s = None, c = None, vmin = None, vmax = None, marker = None, cmap = None, alpha = None, linewidths = None, edgecolors = None)
- The following are the syntax parameters for the scatter() method:
- **x_axis_value** - An array containing x-axis data for scatter in the plot.
- **y_axis_value** - an array with y-axis data.
- **s** - it is the size of the marker (can be scalar or array of size equal to the size of the x-axis or y-axis)
- **c**- the order of the colors of the markers
- **marker**- marker style for scatter plot in the python.
- **cmap** - cmap name for scatter plot in the python.

- **Linewidths** - these are the size of the marker border for scatter in the plot.
- **edgecolor:** the border color of a marker for scatter in the plot.
- **Alpha** - blending value for scatter fucntion, between 0 and 1 (transparent to opaque)
- All of the parameters except x-axis data and y-axis data are optional, and their default value is None.



```
height = np.array([167,170,149,165,155,180,166,146,
                   159,185,145,168,172,181,169])
weight = np.array([86,74,66,78,68,79,90,73,
                   70,88,66,84,67,84,77])
plt.xlim(140,200)
plt.ylim(60,100)
plt.scatter(height,weight)
plt.title("Scatter Plot")
plt.xlabel("Height")
plt.ylabel("Weight")
plt.show()
```



Pie chart in matplotlib

- A pie chart is a specific type of chart that uses pie slices (or wedges) to represent the relative sizes of the data. Therefore, it generates pie slices (wedges) representing data sets and a complete circle representing the entire space.
- It can also be defined as - A circular statistical diagram that only uses one data series to present.
- The entire pie chart represents the overall proportion of the provided data.
- The percentage of the data sections is represented in the pie chart by the area of the wedges.
- The length of the wedge's arc can be used to compute the wedge's area, which primarily depicts the percentage of that portion relative to the entire data set.
- Because they offer a *fast summary*, corporate presentations frequently use pie charts to show data on sales, operations, survey results, resources, etc.

Attribute	Value
data	The array of data values to be depicted is represented by this parameter. Data/sum indicates the fractional area of each slice (data). The resulting pie will have an empty wedge with size = 1-sum if the sum(data) is less than 1, as the data values return the fractional area directly (data).
labels	This argument lists the string sequences that comprise each wedge's label.
autopct	This option, which takes the form of a string, assigns a numerical value to the wedge.
colors	The wedges are given colors using this option.
shadow	The wedges' shadow is produced using this option.

Methods

- **plot(x-axis values, y-axis values)** — plots a simple line graph with x-axis values against y-axis values
- **show()** — displays the graph
- **title("string")** — set the title of the plot as specified by the string
- **xlabel("string")** — set the label for x-axis as specified by the string
- **ylabel("string")** — set the label for y-axis as specified by the string
- **figure()** — used to control a figure level attributes

- **bar(categorical variables, values, color)** — used to create vertical bar graphs
- **hist(values, number of bins)** — used to create a histogram
- **scatter(x-axis values, y-axis values)** — plots a scatter plot with x-axis values against y-axis values