PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

for

CS3361-DATA SCIENCE LABORATORY
(Regulation 2021, III Semester)

(Odd Semester)

ACADEMIC YEAR: 2023 – 2024

PREPARED BY

N.SRIPRIYA,

Assistant Professor / CSE

Subject Code &Name: CS3362 DATA SCIENCE LABORATORY

Branch : CSE/IT

Year/Semester : II/III

LIST OF EXPERIMENTS

S.No	Description of Experiment
1	Download, Install And Explore The Features Of Numpy, Scipy, Jupyter, Statsmodels And Pandas Packages
2	Working With Numpy Arrays
3	Working With Pandas Data Frames
4	Reading Data From Text Files, Excel And The Web And Exploring Various Commands For Doing Descriptive Analytics On The Iris Data Set.
5	Use The Diabetes Data Set From Uci And Pima Indians Diabetes Data Set For Performing The Following: A. Univariate Analysis: Frequency, Mean, Median, Mode, Variance, Standard Deviation, Skewness And Kurtosis. B. Bivariate Analysis: Linear And Logistic Regression Modeling C. Multiple Regression Analysis D. Also Compare The Results Of The Above Analysis For The Two Data Sets
6	Apply And Explore Various Plotting Functions On Uci Data Sets. A. Normal Curves B. Density And Contour Plots C. Correlation And Scatter Plots D. Histograms E. Three Dimensional Plotting
7	Visualizing Geographic Data With Basemap

INDEX

Sl. No.	Date	Program Name	Mark	Signature

Ex.No.1	DOWNLOAD, INSTALL AND EXPLORE THE FEATURES OF NUMPY, SCIPY, JUPYTER, STATSMODELS AND PANDAS PACKAGES

1a. Aim:

To download, install and explore the features of NumPy package.

Problem Description

Python is an open-source object-oriented language. It has many features of which one is the wide range of external packages. There are a lot of packages for installation and use for expanding functionalities. These packages are a repository of functions in python script. NumPy is one such package to ease array computations. To install all these python packages we use the pip- package installer. Pip is automatically installed along with Python. We can then use pip in the command line to install packages from PyPI.

NumPy

NumPy (Numerical Python) is an open-source library for the Python programming language. It is used for scientific computing and working with arrays.

Apart from its multidimensional array object, it also provides high-level functioning tools for working with arrays.

Prerequisites

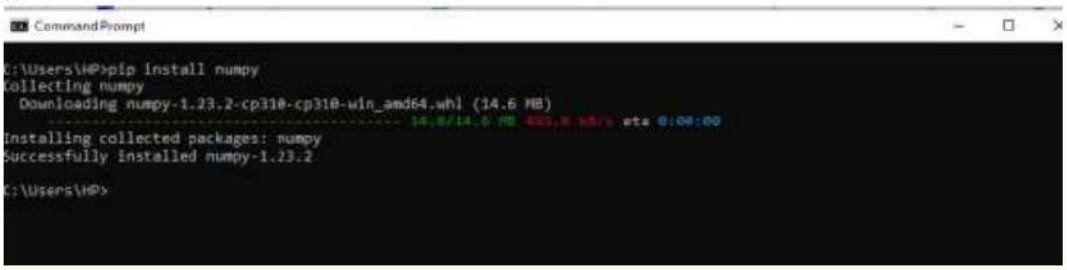
- Access to a terminal window/command line
- A user account with sudo privileges
- Python installed on your system

Downloading and installing Numpy:

Python NumPy is a general-purpose array processing package that provides tools for handling n-dimensional arrays. It provides various computing tools such as comprehensive mathematical functions, linear algebra routines.Use the below command to install NumPy:

pip install numpy

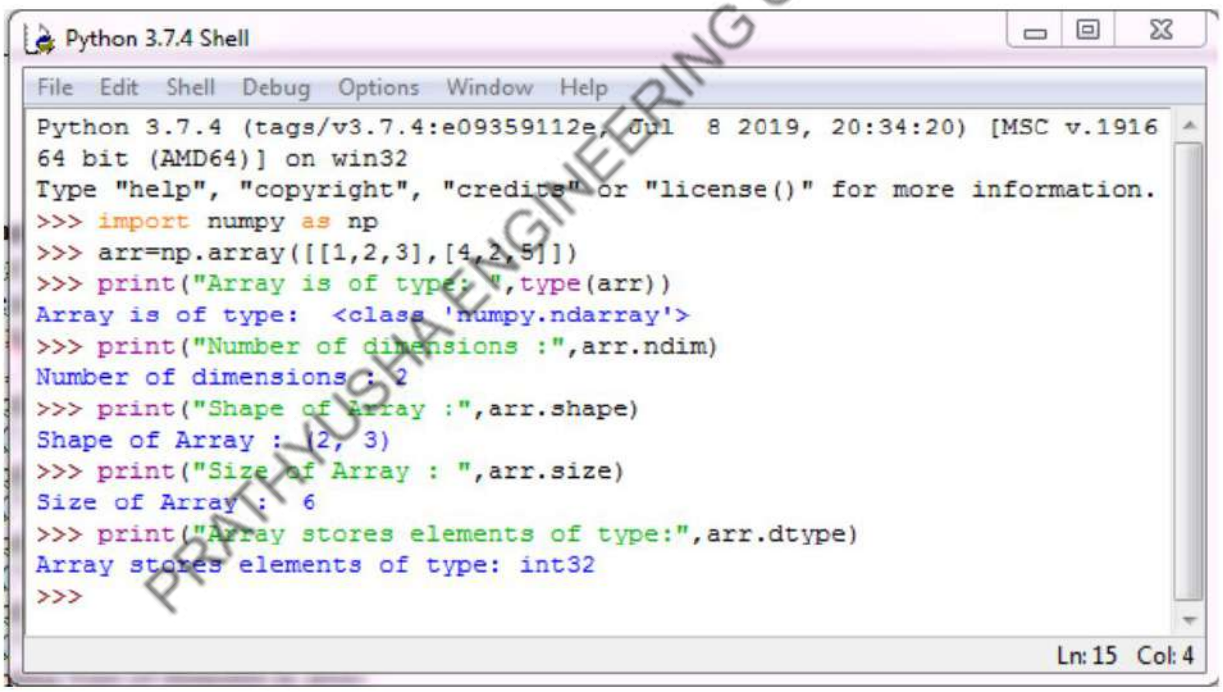
output:



- **Sample python program using numpy**:

```
import numpy as np
# Creating array object
arr = np.array( [[ 1, 2, 3],
[ 4, 2, 5]] )
# Printing type of arr object
print("Array is of type: ", type(arr))
# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)
# Printing shape of array
print("Shape of array: ", arr.shape)
# Printing size (total number of elements) of array
print("Size of array: ", arr.size)
# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

OUTPUT



RESULT:

Thus the NumPy package is downloaded, installed and the features are explored.

1 b. **Aim** : To download, install and explore the features of Jupyter packages.

Data Science:

Data science combines math and statistics, specialized programming, advanced analytics, artificial intelligence (AI), and machine learning with specific subject matter expertise to uncover actionable insights hidden in an organization's data.

Jupyter:

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

Jupyter has support for over 40 different programming languages and Python is one of them. Python is a requirement (Python 3.3 or greater, or Python 2.7) for installing the Jupyter Notebook itself.

Procedure:

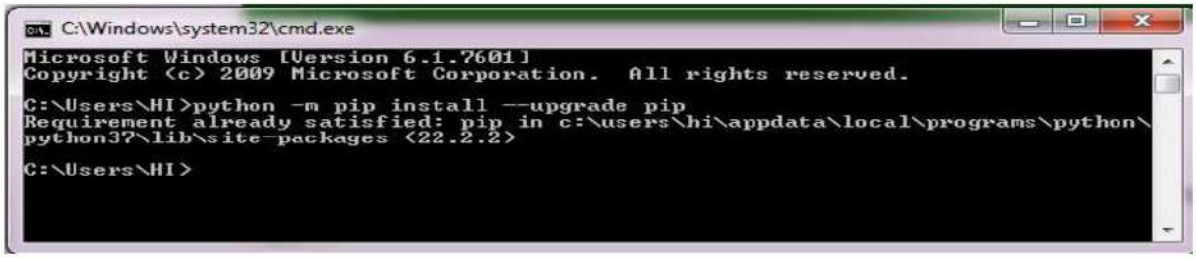
PIP is a package management system used to install and manage software packages/libraries written in Python. These files are stored in a large "on-line repository" termed as Python Package Index (PyPI).

pip uses PyPI as the default source for packages and their dependencies.

Installing Jupyter Notebook using pip:

To install Jupyter using pip, we need to first check if pip is updated in our system. Use the following command to update pip:

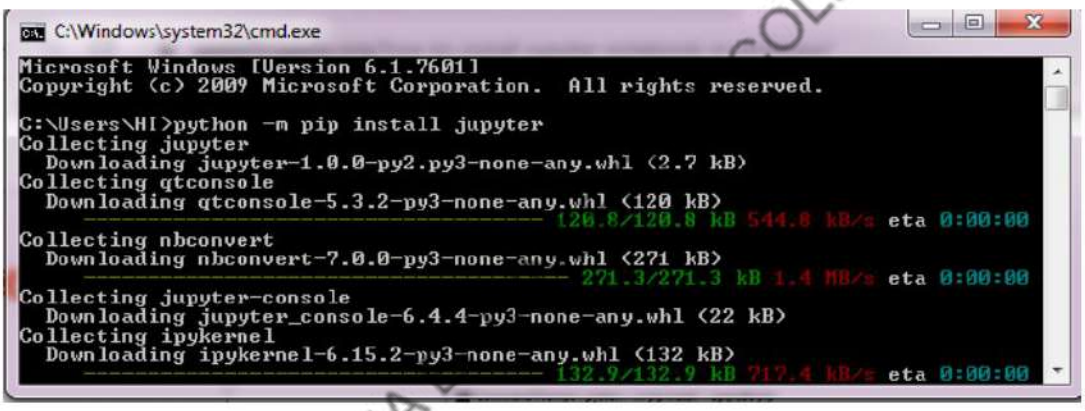
python -m pip install --upgrade pip



After updating the pip version, follow the instructions provided below to install Jupyter:

Command to install Jupyter:

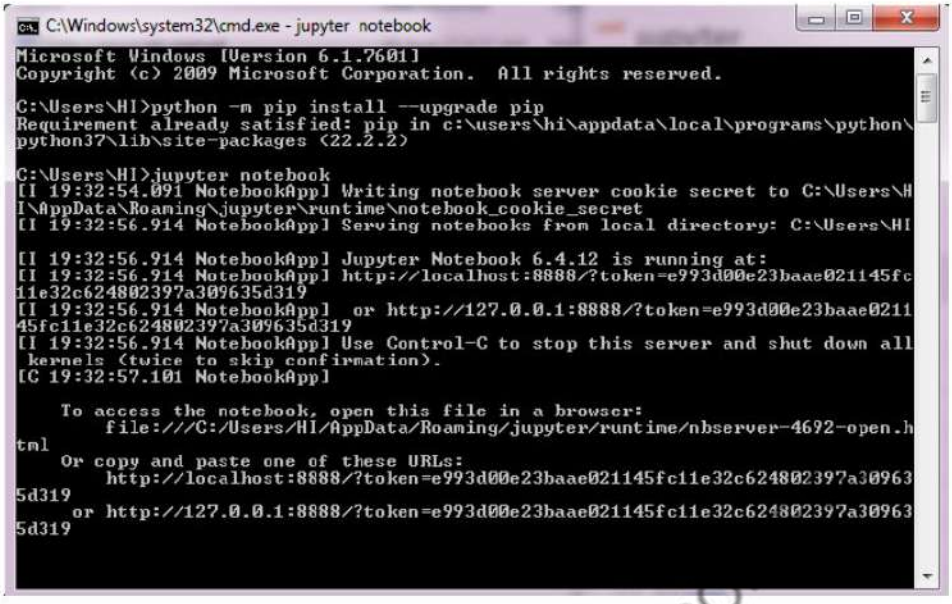
python -m pip install jupyter



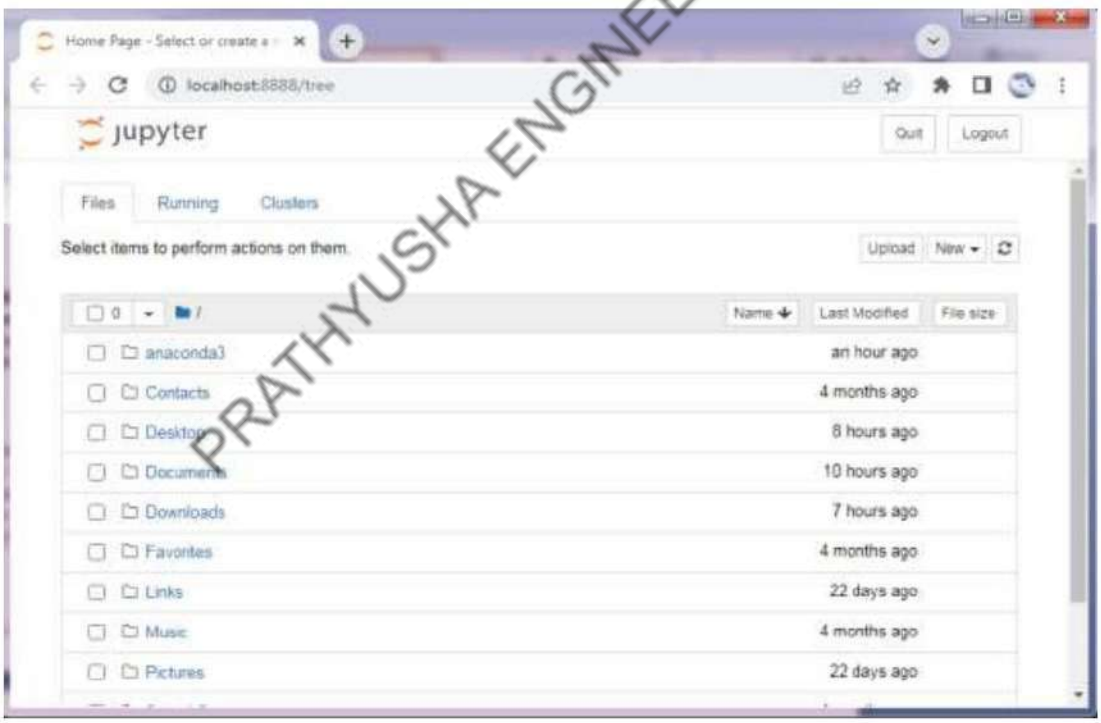
· · **Finished Installation:**

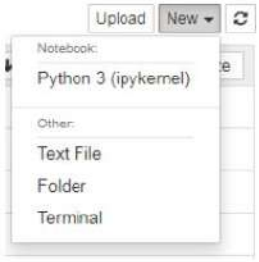
Use the following command to launch Jupyter using command-line:

jupyter notebook



Launching Jupyter Notebook





- Click New and select python 3(ipykernal) and type the following program. Click run to execute the program.

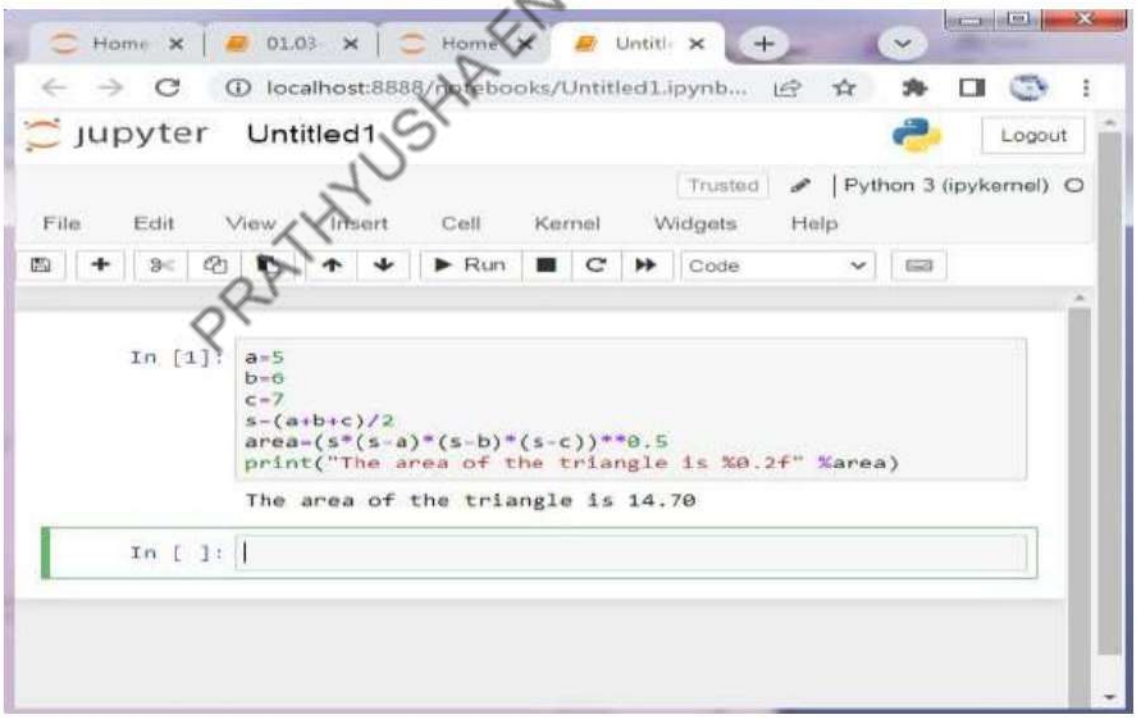
Running the Python program:

Python code:

Program to find the area of a triangle

```
# Python Program to find the area of triangle
a = 5
b = 6
c = 7
# calculate the semi-perimeter
s = (a + b + c) / 2
# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Output:



Result:

Thus the jupyter package is downloaded, installed and the features are explored.

1 C Aim:

To download, install and explore the features of Scipy package.

Problem Description

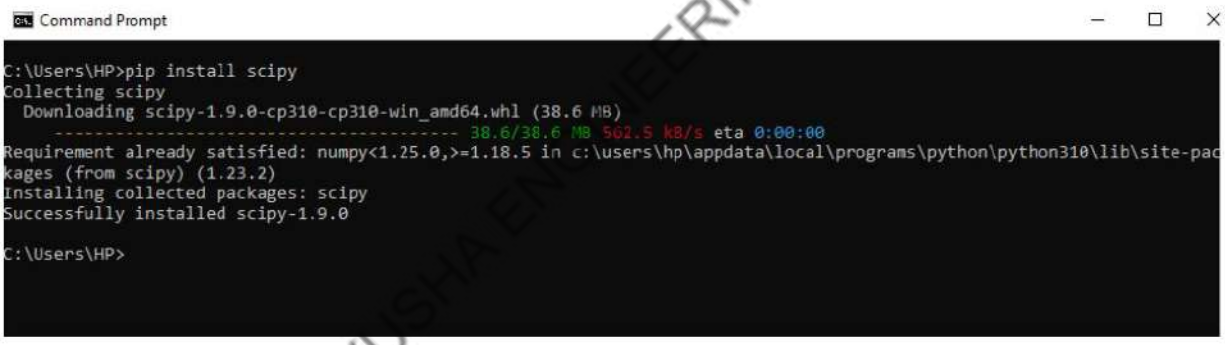
Scipy is a python library that is useful in solving many mathematical equations and algorithms. It is designed on the top of Numpy library that gives more extension of finding scientific mathematical formulae like Matrix Rank, Inverse, polynomial equations, LU Decomposition, etc. Using its high-level functions will significantly reduce the complexity of the code and helps in better analyzing the data.

Downloading and Installing Scipy:

pip use the below command to install Scipy package on Windows:

pip install scipy

output

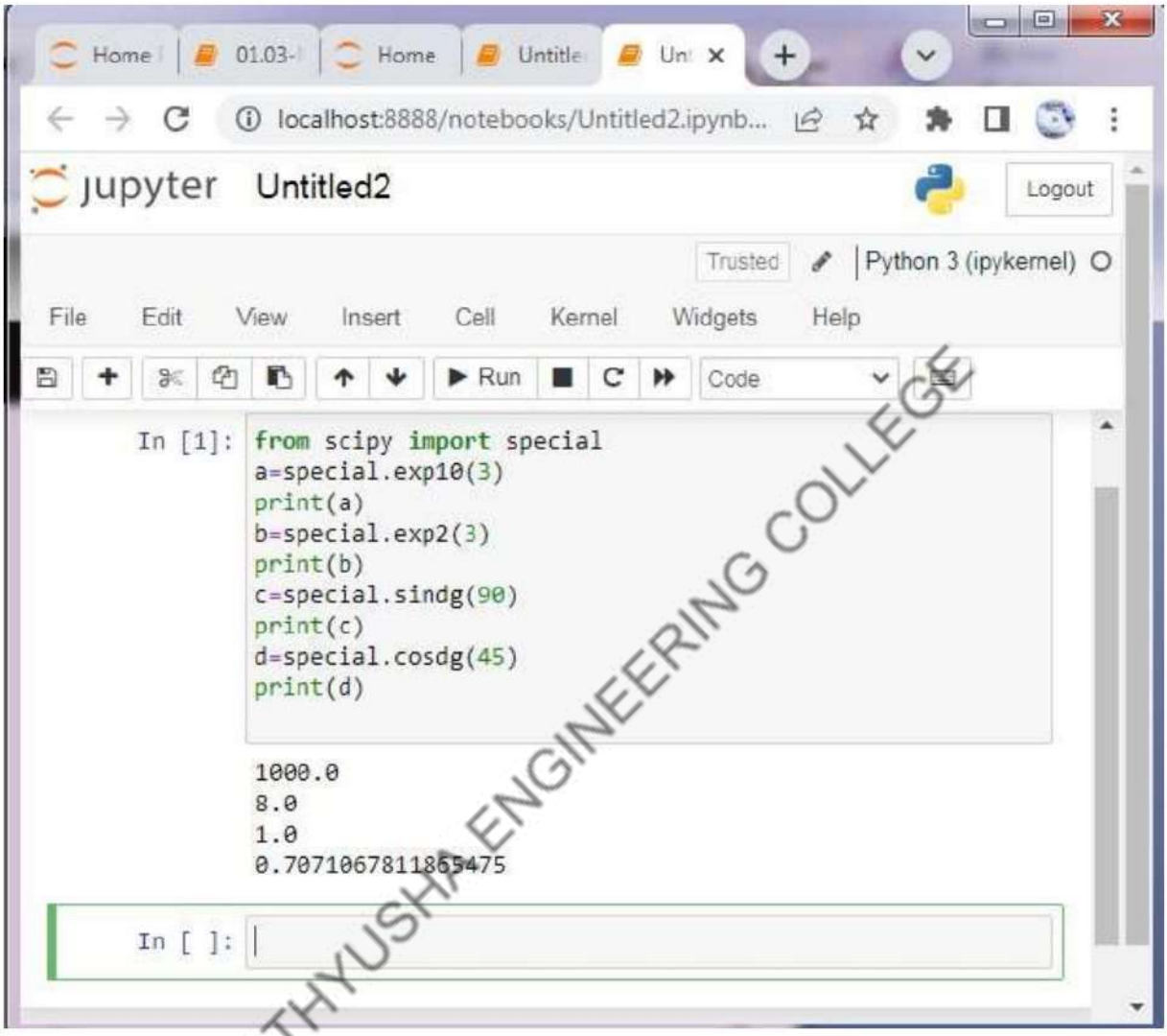


Sample python code using Scipy:

Type the program in Jupyter notebook

```
from scipy import special
a = special.exp10(3)
print(a)
b = special.exp2(3)
print(b)
c = special.sindg(90)
print(c)

d = special.cosdg(45)

print(d)
```

Output:



RESULT

Thus the SciPy package is downloaded, installed and the features are explored.

1 D). **Aim** :

To download, install and explore the features of Panda packages.

Problem Description

Pandas is one of the most popular open-source frameworks available for Python. It is among the fastest and most easy-to-use libraries for data analysis and manipulation. Pandas dataframes are some of the most useful data structures available in any library. It has uses in every data-intensive field, including but not limited to scientific computing, data science, and machine learning.

The library does not come included with a regular install of Python. To use it, you must install the Pandas framework separately.

Installing Pandas on Windows

There are two ways of installing Pandas on Windows.

Method #1: Installing with pip

It is a package installation manager that makes installing Python libraries and frameworks straightforward.

As long as you have a newer version of Python installed (> Python 3.4), pip will be installed on your computer along with Python by default.

However, if you're using an older version of Python, you will need to install pip on your computer before installing Pandas.

Step #1: Launch Command Prompt

Press the Windows key on your keyboard or click on the Start button to open the start menu. Type **cmd**, and the Command Prompt app should appear as a listing in the start menu.

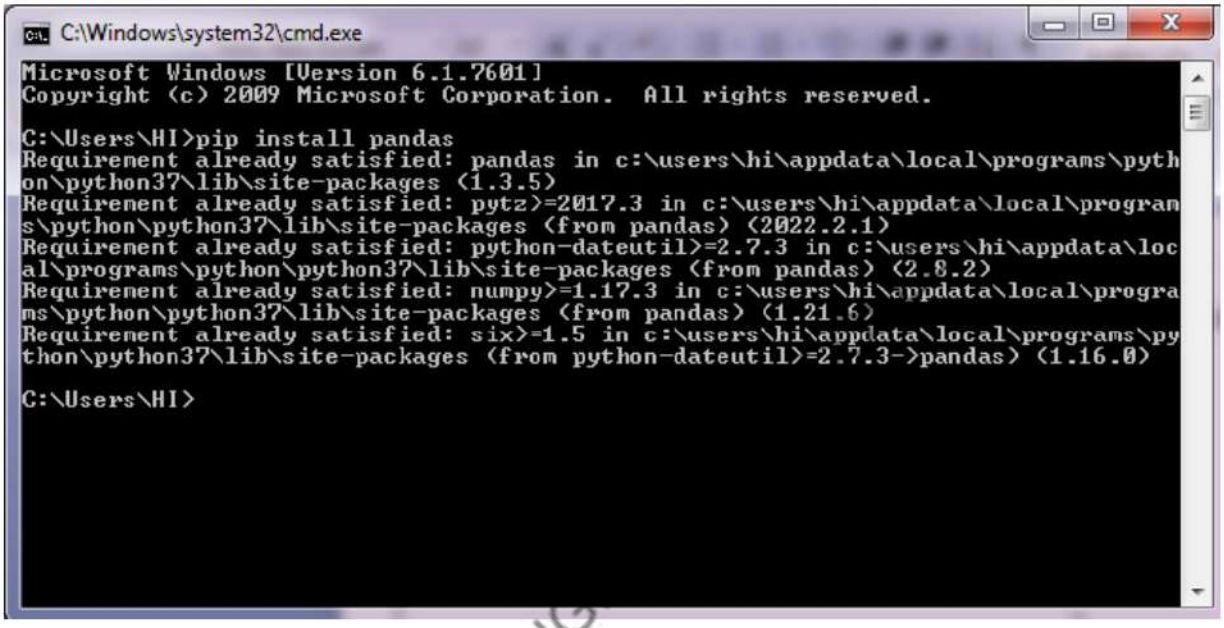
Step #2: Enter the Required Command

After you launch the command prompt, the next step in the process is to type in the required command to initialize pip installation.

Enter the command

pip install pandas

on the terminal. This should launch the pip installer. The required files will be downloaded, and Pandas will be ready to run on your computer.



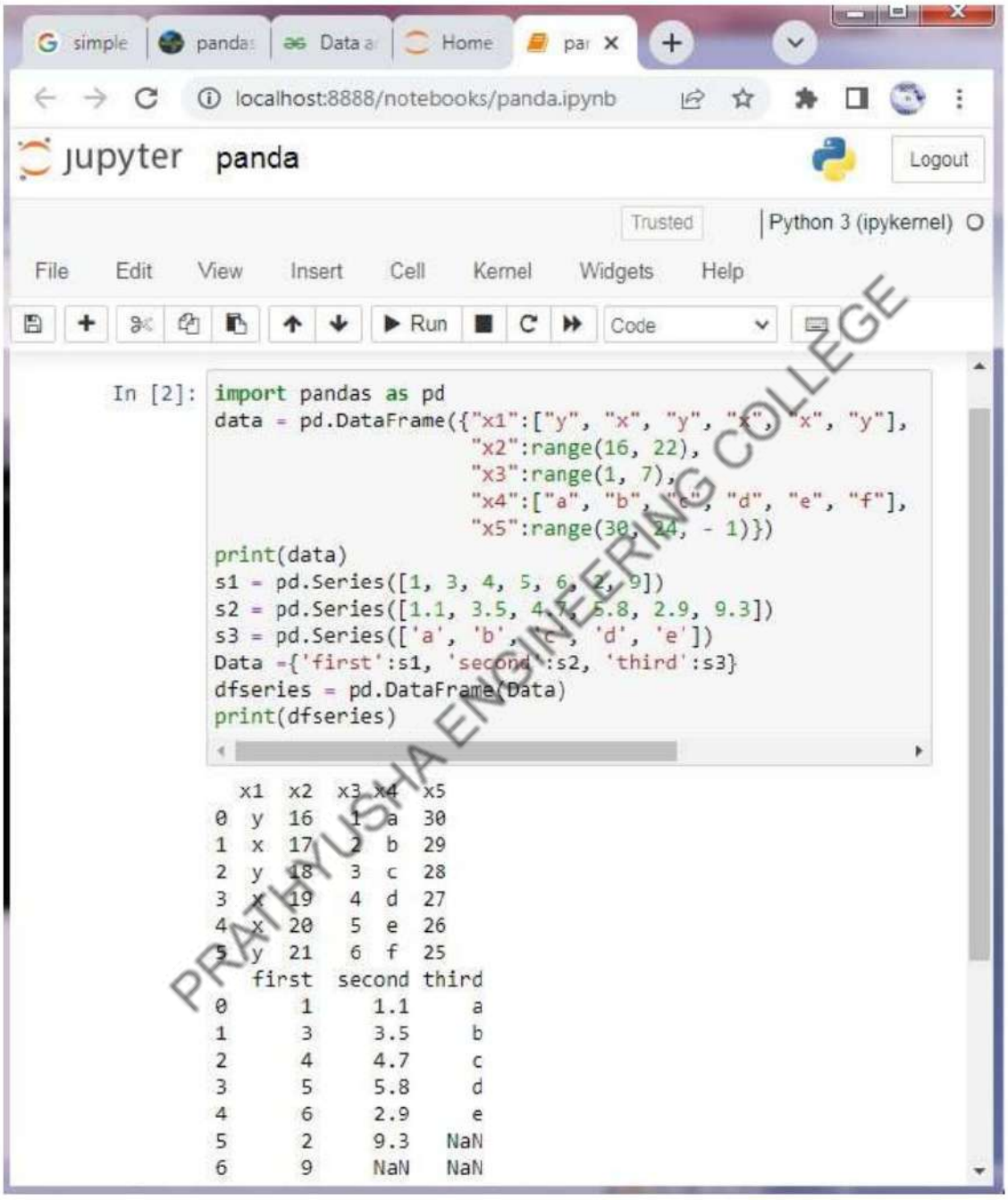
Panda package is successfully installed.

Sample program

// to be typed in Jupyter notebook

```
import pandas as pd
data = pd.DataFrame({"x1":["y", "x", "y", "x", "x", "y"],
                     "x2":range(16, 22),
                     "x3":range(1, 7),
                     "x4":["a", "b", "c", "d", "e", "f"],
                     "x5":range(30, 24, - 1)})
print(data)
s1 = pd.Series([1, 3, 4, 5, 6, 2, 9])
s2 = pd.Series([1.1, 3.5, 4.7, 5.8, 2.9, 9.3])
s3 = pd.Series(['a', 'b', 'c', 'd', 'e'])
Data ={'first':s1, 'second':s2, 'third':s3}
```

```
dfseries = pd.DataFrame(Data)
print(dfseries)
```



Result:

Thus the Panda package is downloaded, installed and the features are explored.

1E). Aim:

To download, install and explore the features of Statsmodels package.

Problem Description:

Statsmodels is a popular library in Python that enables us to estimate and analyze various statistical models. It is built on numeric and scientific libraries like NumPy and SciPy.

Some of the essential features of this package are-

1. It includes various models of linear regression like ordinary least squares, generalized least squares, weighted least squares, etc.
2. It provides some efficient functions for time series analysis.
3. It also has some datasets for examples and testing.
4. Models based on survival analysis are also available.
5. All the statistical tests that we can imagine for data on a large scale are present.

Installing Statsmodels

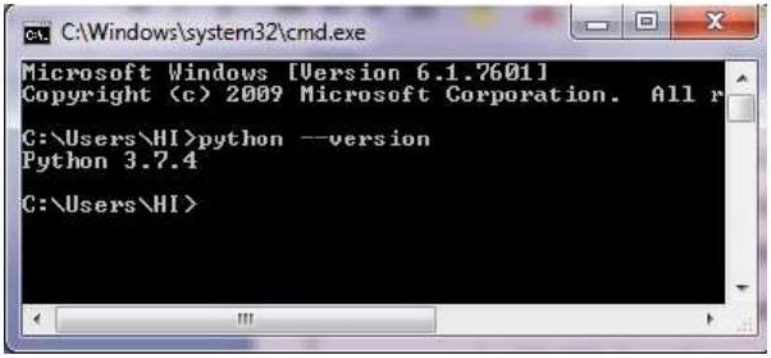
Check the version of python installed in the PC.

Using Command Prompt

Type 'Command Prompt' on the taskbar's search pane and you'll see its icon. Click on it to open the command prompt.

Also, you can directly click on its icon if it is pinned on the taskbar.

1. Once the 'Command Prompt' screen is visible on your screen.
2. Type *python -version* and click on 'Enter'.
3. The version installed in your system would be displayed in the next line.

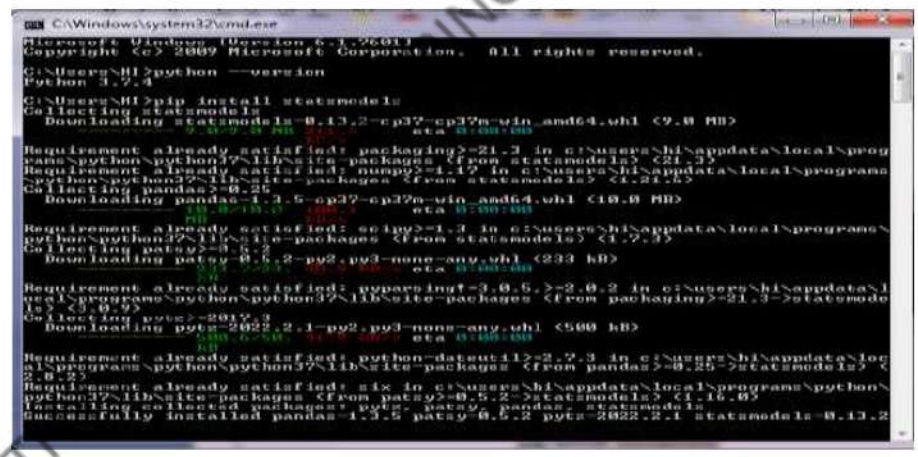


Installation of statsmodels

Now for installing **statsmodels** in our system, Open the Command Prompt, type the following command and click on **'Enter'.**

pip install statsmodels

Output



It's time to look have a program in which we will import statsmodels-

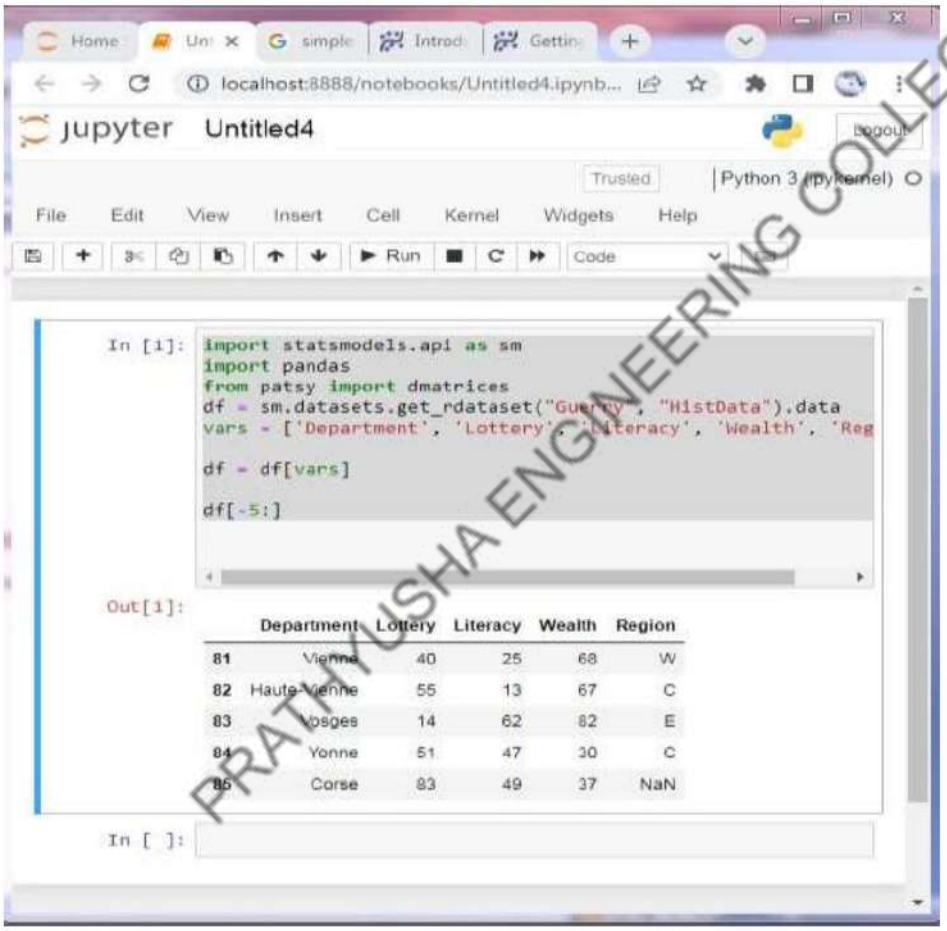
Here, we will perform OLS(Ordinary Least Squares) regression, in this technique we will try to minimize the net sum of squares of difference between the calculated value and observed value.

Program

```
import statsmodels.api as sm

import pandas
```

```
from patsy import dmatrices

df = sm.datasets.get_rdataset("Guerry", "HistData").data

vars = ['Department', 'Lottery', 'Literacy', 'Wealth', 'Region']

df = df[vars]

df[-5:]
```

OUTPUT



Result:

Thus the Statsmododels package is downloaded, installed and the features are explored.

Ex.No.2	WORKING WITH NUMPY ARRAYS

Aim :

Write a python program to show the woking of NumPy Arrays in Python.

2a) Use Numpy array to demonstrate basic array characteristics

b) Create Numpy array using list and tuple

c) Apply basic operations (+,_,*./) and find the transpose of the matrix

d) Perform sorting operation with Numpy arrays

Problem Description

Arrays in NumPy: NumPy's main object is the homogeneous multidimensional array.

- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called *axes*. The number of axes is *rank*.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.

Example 1:

Write a python program to demonstrate the basic NumPy array characteristics

```
import numpy as np

# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)
```

```
# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

Output :

```
Array is of type: <class 'numpy.ndarray'>

No. of dimensions: 2
Shape of array: (2, 3)
Size of array: 6
Array stores elements of type: int64
```

2. Array creation:

There are various ways to create arrays in NumPy.

- For example, you can create an array from a regular Python **list** or **tuple** using the **array** function. The type of the resulting array is deduced from the type of the elements in the sequences.
- Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with **initial placeholder content**. These minimize the necessity of growing arrays, an expensive operation. **For example:** np.zeros, np.ones, np.full, np.empty, etc.
- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.
- **arange:** returns evenly spaced values within a given interval. **step** size is specified.
- **linspace:** returns evenly spaced values within a given interval. **num** no. of elements are returned.
- **Reshaping array:** We can use **reshape** method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is: a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM . (i.e original size of array remains unchanged.)
- **Flatten array:** We can use **flatten** method to get a copy of array collapsed into **one dimension**. It accepts *order* argument. Default value is 'C' (for row-major order). Use 'F' for column major order.

Example 2:

```
import numpy as np

# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype = 'float')
print ("Array created using passed list:\n", a)
```

```
# Creating array from tuple
b = np.array((1 , 3, 2))
print ("\nArray created using passed tuple:\n", b)

# Creating a 3X4 array with all zeros
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

# Create a constant value array of complex type
d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s.")
 print( "Array type is complex:\n", d)

# Create an array with random values
e = np.random.random((2, 2))
print ("\nA random array:\n", e)

# Create a sequence of integers
# from 0 to 30 with steps of 5
f = np.arange(0, 30, 5)
print ("\nA sequential array with steps of 5:\n", f)

# Create a sequence of 10 values in range 0 to 5
g = np.linspace(0, 5, 10)
print ("\nA sequential array with 10 values between"
                        "0 and 5:\n", g)

# Reshaping 3X4 array to 2X2X3 array
arr = np.array([[1, 2, 3, 4],
            [5, 2, 4, 2],
            [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)

# Flatten array
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Fattened array:\n", flarr)
```

OUTPUT

Array created using passed list:

```
[[ 1. 2. 4.]
[ 5.  8.  7.]]
```

Array created using passed tuple:

```
[1 3 2]
```

An array initialized with all zeros:

```
[[ 0. 0. 0. 0.]
[ 0.  0.  0.  0.]

[ 0.  0.  0.  0.]]
```

An array initialized with all 6s. Array type is complex:

```
[[ 6.+0.j 6.+0.j 6.+0.j]
[ 6.+0.j 6.+0.j  6.+0.j]
[ 6.+0.j 6.+0.j  6.+0.j]]
```

A random array:

```
[[ 0.46829566  0.67079389]
[ 0.09079849  0.95410464]]
```

A sequential array with steps of 5:

```
[ 0 5 10 15 20 25]
```

A sequential array with 10 values between 0 and 5:

```
[ 0.         0.55555556 1.11111111  1.66666667  2.22222222  2.77777778
3.33333333  3.88888889  4.44444444  5.        ]
```

Original array:

[[1 2 3 4]

[5 2 4 2]

[1 2 0 1]]

Reshaped array:

[[[1 2 3]

[4 5 2]]

[[4 2 1]

[2 0 1]]]

Original array:

[[1 2 3]

[4 5 6]]

Fattened array:

[1 2 3 4 5 6]

3. Basic operations:

Arithmetic oprations on NumPy Array:

Program 3:

```
import numpy as np

array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9], [10, 11, 12]])

print("Addition")
print(array1 + array2)
print("-" * 20)
print("Subtraction")
print(array1 - array2)
print("-" * 20)
print("Multiplication")
print(array1 * array2)
print("-" * 20)
print("Division")
```

```
print(array2 / array1)
print("-" * 40)
print(array1 ** array2)
print("-" * 40)
a = np.array([1, 2, 5, 3])
print ("Adding 1 to every element:", a+1)
print ("Subtracting 3 from each element:", a-3)
print ("Multiplying each element by 10:", a*10)
print ("Squaring each element:", a**2)
a *= 2
print ("Doubled each element of original array:", a)
a = np.array([[1, 2, 3], [3, 4, 5], [9, 6, 0]])
print ("\nOriginal  array:\n",  a)
print ("Transpose of array:\n", a.T)
```

Output

```
Addition
[[ 8 10 12]
 [14 16 18]]
----------------------------
Subtraction
[[-6 -6 -6]
 [-6 -6 -6]]
----------------------------
Multiplication
[[ 7 16 27]
 [40 55 72]]
----------------------------
Division
[[7.  4.  3. ]
 [2.5 2.2 2. ]]
------------------------------------------------------
[[          1         256       19683]
 [    1048576    48828125 -2118184960]]
------------------------------------------------------
```


Adding 1 to every element: [2 3 6 4]

Subtracting 3 from each element: [-2 -1 2 0]

Multiplying each element by 10: [10 20 50 30]

Squaring each element: [1 4 25 9]

Doubled each element of original array: [2 4 10 6]

Original array:

```
[[1 2 3]
[3 4 5]
[9 6 0]]
```

Transpose of array:

```
[[1 3 9]
[2 4 6]
[3 5 0]]
```

4. Sorting array: There is a simple **np.sort** method for sorting NumPy arrays. Let's explore it a bit.

Program 4:

```
import numpy as np


a = np.array([[1, 4, 2],
              [3, 4, 6],
             [0, -1, 5]])

# sorted array
print ("Array elements in sorted order:\n",
                np.sort(a, axis = None))

# sort array row-wise
print ("Row-wise sorted array:\n",
              np.sort(a, axis = 1))

# specify sort algorithm
print ("Column wise sort by applying merge-sort:\n",
          np.sort(a, axis = 0, kind = 'mergesort'))

# Example to show sorting of structured array
# set alias names for dtypes
dtypes = [('name', 'S10'), ('grad_year', int), ('cgpa', float)]

# Values to be put in array
values = [('Hrithik', 2009, 8.5), ('Ajay', 2008, 8.7),
          ('Pankaj', 2008, 7.9), ('Aakash', 2009, 9.0)]

# Creating array
arr = np.array(values, dtype = dtypes)
print ("\nArray sorted by names:\n",
```

```
        np.sort(arr, order = 'name'))

print ("Array sorted by graduation year and then cgpa:\n",
            np.sort(arr, order = ['grad_year', 'cgpa']))
```

OUTPUT

Array elements in sorted order:

[-1 0 1 2 3 4 4 5 6]

Row-wise sorted array:

[[1 2 4]
[3 4 6]
[-1 0 5]]

Column wise sort by applying merge-sort:

[[0 -1 2]
[1 4 5]
[3 4 6]]

Array sorted by names:

[('Aakash', 2009, 9.0) ('Ajay', 2008, 8.7) ('Hrithik', 2009, 8.5)
('Pankaj', 2008, 7.9)]

Array sorted by graduation year and then cgpa:

[('Pankaj', 2008, 7.9) ('Ajay', 2008, 8.7) ('Hrithik', 2009, 8.5)
('Aakash', 2009, 9.0)]

Result

Thus the python programs are written and executed to explain the features of NumPy array.

Ex.No.3	WORKING WITH PANDAS DATA FRAMES

Aim:

Write a python program to work with Panda data frames.

Pandas

Pandas is an open-source library that is built on top of NumPy library. It is a Python package that offers various data structures and operations for manipulating numerical data and time series. It is mainly popular for importing and analyzing data much easier. Pandas is fast and it has high-performance & productivity for users.

Pandas DataFrame

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc.

Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the **data**, **rows**, and **columns**.

Creating a Panda Data Frames

A pandas DataFrame can be created using the following constructor −

pandas.DataFrame(data, index, columns, dtype, copy)

.

Creating an empty dataframe :
A basic DataFrame, which can be created is an Empty Dataframe. An Empty Dataframe is created just by calling a dataframe constructor.

Creating a dataframe using List:

DataFrame can be created using a single list or a list of lists.

Creating dataframe from dict of ndarray/lists:

To create dataframe from dict of narray/list, all the narray must be of same length. If index is passed then the length index should be equal to the length of arrays. If no index is passed, then by default, index will be range(n) where n is the array length.

Iterating over rows :

In order to iterate over rows, we can use three function iteritems(), iterrows(), itertuples() . These three function will help in iteration over rows.

Program

```
import pandas as pd


# Calling DataFrame constructor
print("Empty dataframe")
df = pd.DataFrame()
print(df)
print("Dataframe creation using List")
# list of strings
lst = ['Geeks', 'For', 'Geeks', 'is', 'portal', 'for', 'Geeks']
# Calling DataFrame constructor on list
df = pd.DataFrame(lst)
print(df)
# initialise data of lists.
Data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}
# Create dataframe
```

```
df = pd.DataFrame(Data)
# Print the output.
print(df)
print("Create dataframe from dictionoary of lists")
# dictionary of lists
dict = {'name':["aparna", "pankaj", "sudhir", "Geeku"],
        'Degree': ["MBA", "BCA", "M.Tech", "MBA"],
        'Score':[90, 40, 80, 98]}

# creating a dataframe from a dictionary
df = pd.DataFrame(dict)
print(df)
# iterating over rows using iterrows() function
for i, j in df.iterrows():
    print(i, j)
    print()
```

OUTPUT

```
Empty dataframe
Empty DataFrame
Columns: []
Index: []

Dataframe creation using List
      0
0  Geeks
1    For
2  Geeks
3     is
4 portal
5    for
6 Geeks
   Name  Age
0   Tom  20
1  nick  21
2 krish  19
```

3 jack 18

Create dataframe from dictionoary of lists

```
    name Degree  Score
0  aparna    MBA     90
1  pankaj    BCA     40
2  sudhir  M.Tech     80
3   Geeku    MBA     98
```

```
0 name      aparna
Degree       MBA
Score         90
Name: 0, dtype: object
```

```
1 name      pankaj
Degree       BCA
Score         40
Name: 1, dtype: object
```

```
2 name      sudhir
Degree    M.Tech
Score  80
Name: 2, dtype: object
```

```
3 name     Geeku
Degree      MBA
Score       98
Name: 3, dtype: object
```

Pandas Dataframe visualization
Retrieving data from the web

In[1]:

```
import pandas as pd
url =
'https://github.com/chris1610/pbpython/blob/master/data/2018_Sales_Total_v2.xlsx?raw=True'
df = pd.read_excel(url)
df
```

OUTPUT

	account number	name	sku	quantity	unit price	ext price	date
0	740150	Barton LLC	B1-20000	39	86.69	3380.91	2018-01-01 07:21:51
1	714466	Trantow-Barrows	S2-77896	-1	63.16	-63.16	2018-01-01 10:00:47
2	218895	Kulas Inc	B1-69924	23	90.70	2086.10	2018-01-01 13:24:58
3	307599	Kassulke, Ondricka and Metz	S1-65481	41	21.05	863.05	2018-01-01 15:05:22
4	412290	Jerde-Hilpert	S2-34077	6	83.21	499.26	2018-01-01 23:26:55
...	...	...	...	...	...	...	...
1502	424914	White-Trantow	B1-69924	37	42.77	1582.49	2018-11-27 14:29:02
1503	424914	White-Trantow	S1-47412	16	65.58	1049.28	2018-12-19 15:15:41
1504	424914	White-Trantow	B1-86481	75	28.89	2166.75	2018-12-29 13:03:54
1505	424914	White-Trantow	S1-82801	20	95.75	1915.00	2018-12-22 03:31:36
1506	424914	White-Trantow	S2-83881	100	88.19	8819.00	2018-12-16 00:46:26

1507 rows × 7 columns

Pandas for retrieving data from the csv file

In[2]:

```
import pandas as pd

data = pd.read_csv(r'C:\Users\HI\Downloads\PythonDataScienceHandbook-
master\notebooks\data\iris.csv')
df = pd.DataFrame(data)

print (df)
```

Out[2]:

```
     sepallength  sepalwidth  petallength  petalwidth           class
0            5.1         3.5          1.4         0.2     Iris-setosa
1            4.9         3.0          1.4         0.2     Iris-setosa
2            4.7         3.2          1.3         0.2     Iris-setosa
3            4.6         3.1          1.5         0.2     Iris-setosa
4            5.0         3.6          1.4         0.2     Iris-setosa
..           ...         ...          ...         ...             ...
145          6.7         3.0          5.2         2.3  Iris-virginica
146          6.3         2.5          5.0         1.9  Iris-virginica
147          6.5         3.0          5.2         2.0  Iris-virginica
148          6.2         3.4          5.4         2.3  Iris-virginica
149          5.9         3.0          5.1         1.8  Iris-virginica

[150 rows x 5 columns]
```

Result:

Thus the python program is written to show the working of pandas dataframes

Ex.No.4	READING DATA FROM TEXT FILES, EXCEL AND THE WEB AND EXPLORING VARIOUS COMMANDS FOR DOING DESCRIPTIVE ANALYTICS ON THE IRIS DATA SET

Aim:

Reading data from text files, excel and the web and exploring various commands for doing descriptive analytics on the Iris data set.

What is Exploratory Data Analysis?

Exploratory Data Analysis (EDA) is a technique to analyze data using some visual Techniques. With this technique, we can get detailed information about the statistical summary of the data. We will also be able to deal with the duplicates values, outliers, and also see some trends or patterns present in the dataset.
Now let's see a brief about the Iris dataset.

Iris Dataset
If you are from a data science background you all must be familiar with the Iris Dataset. If you are not then don't worry we will discuss this here.

Iris Dataset is considered as the Hello World for data science. It contains five columns namely – Petal Length, Petal Width, Sepal Length, Sepal Width, and Species Type. Iris is a flowering plant, the researchers have measured various features of the different iris flowers and recorded them digitally.

4A). Aim:

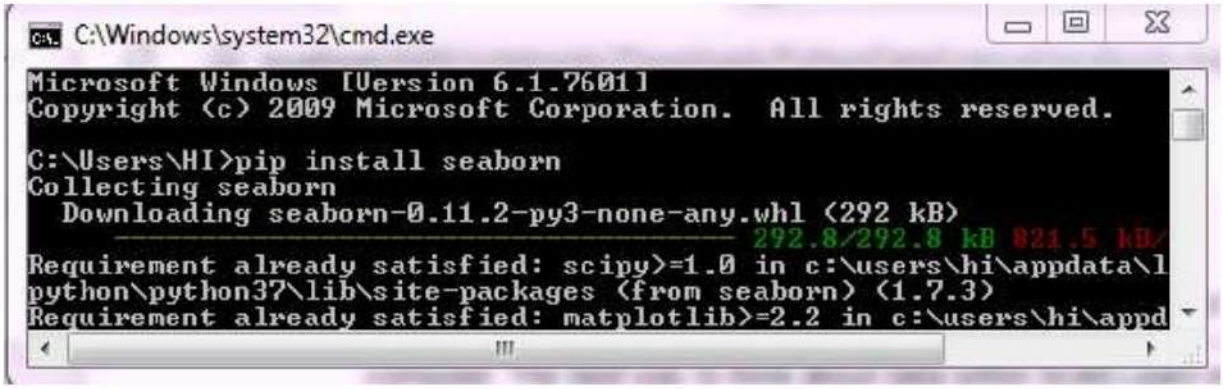
Reading data from Text file and exploring various commands for doing descriptive analytics on the Iris data set.

Seaborn Package:

Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's set() method. By convention, Seaborn is imported as sns:

Seaborn package is installed by typing the following command in the command prompt

pip install seaborn



Step 2:

After the successful insertion of seaborn package launch into jupyter using jupyter notebook command. Type the following program. Give the correct path name for iris dataset. The Iris dataset, which lists measurements of petals and sepals of three iris species.

Step 1:

Import Packages

In[1]:

```
import numpy as np
import pandas as pd # package for working with data frames in python
import seaborn as sns # package for visualization (more on seaborn later)
import matplotlib.pyplot as plt
%matplotlib inline
```

Step 2:

Import iris dataset

In[2]:

```
iris = sns.load_dataset('iris')
my_data_frame = pd.DataFrame(iris)
my_data_frame.head()
```


OUTPUT

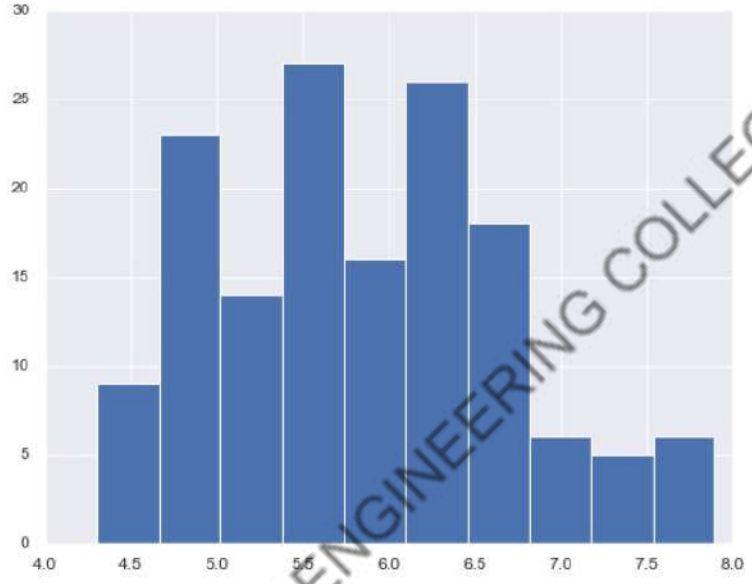
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Step 3: Simple plot

In[3]:

```
p=plt.hist(my_data_frame.sepal_length)
```

OUTPUT

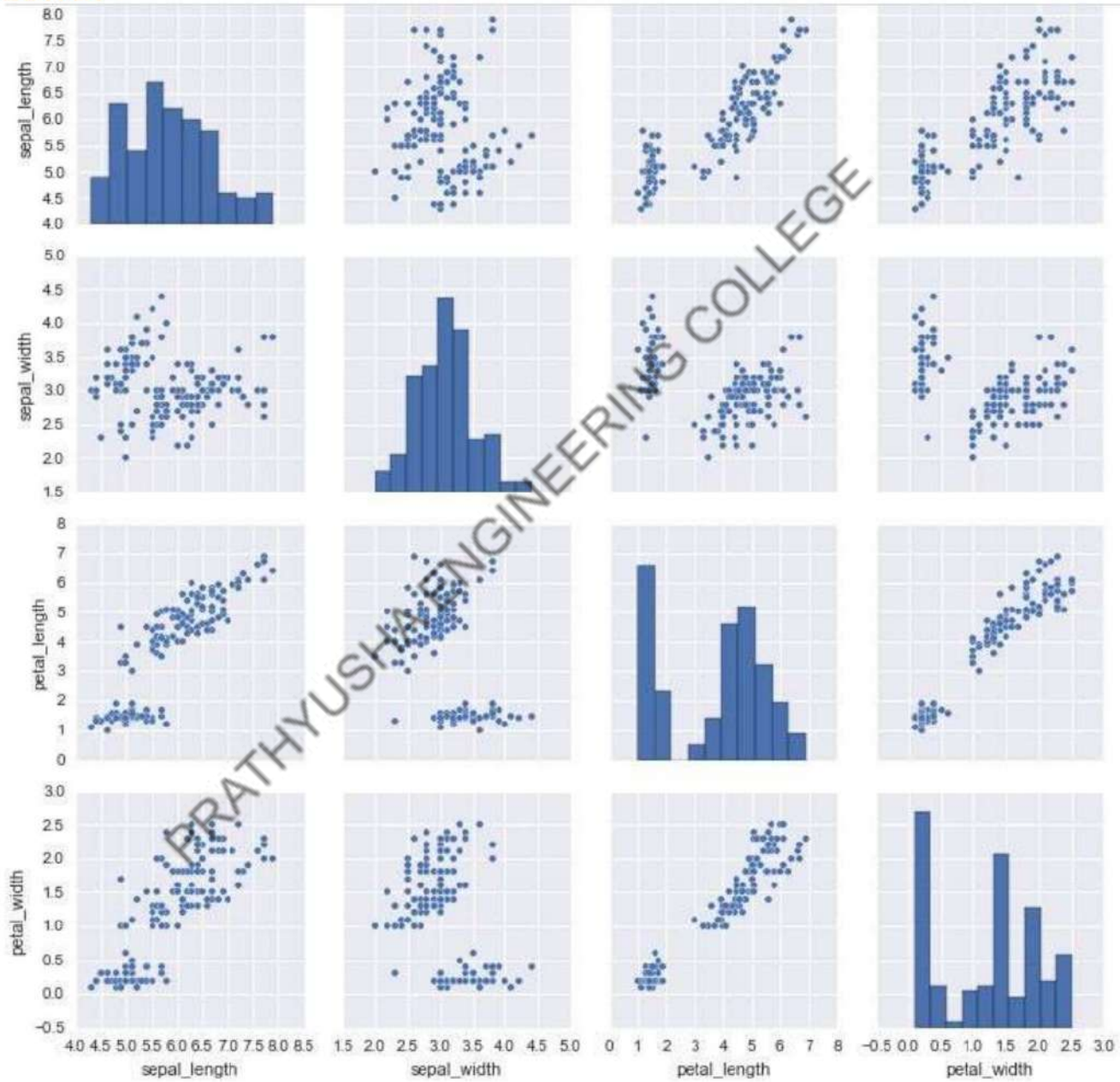


Step: 4 Plot using Seaborn

In[4]:

```
g = sns.pairplot(my data frame)
```

OUTPUT

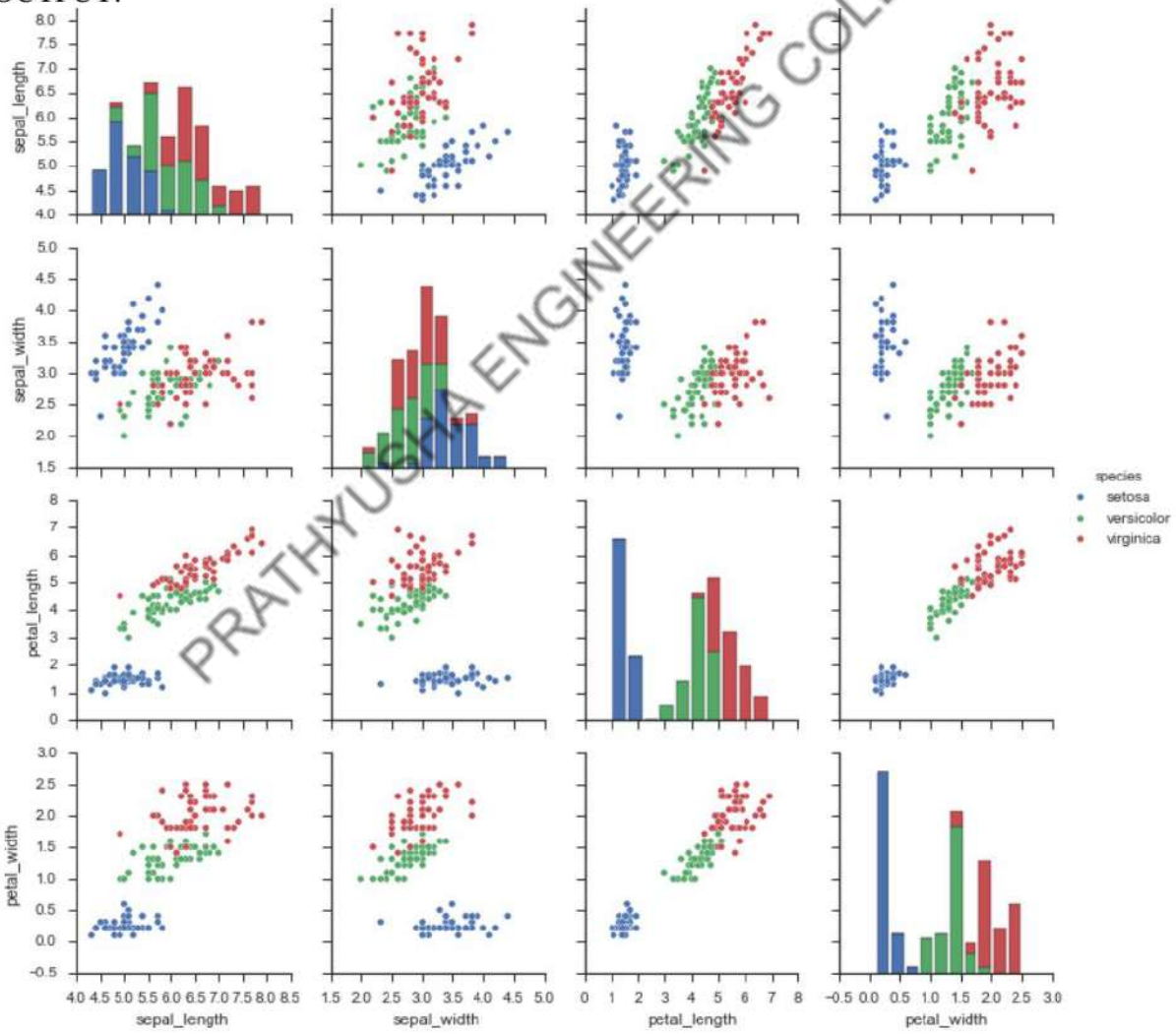


Step 5: Colour plot

In[5]:

```
sns.set(style="ticks", color_codes=True) # change style g =
sns.pairplot(iris, hue="species")
```

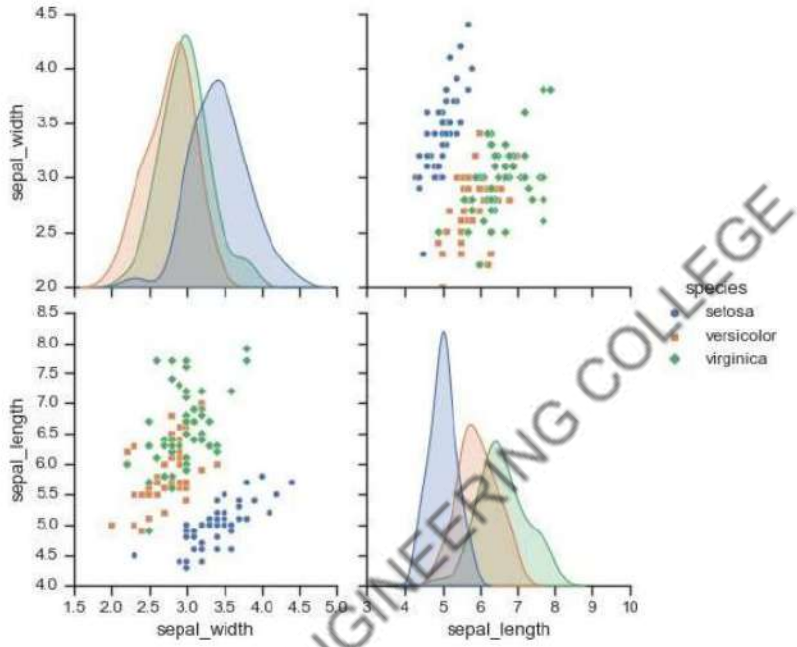
OUTPUT:



In [6]:

```
g = sns.pairplot(iris, height=3, vars=["sepal_width", "sepal_length"], \
                 markers=["o", "s", "D"], hue="species")
```

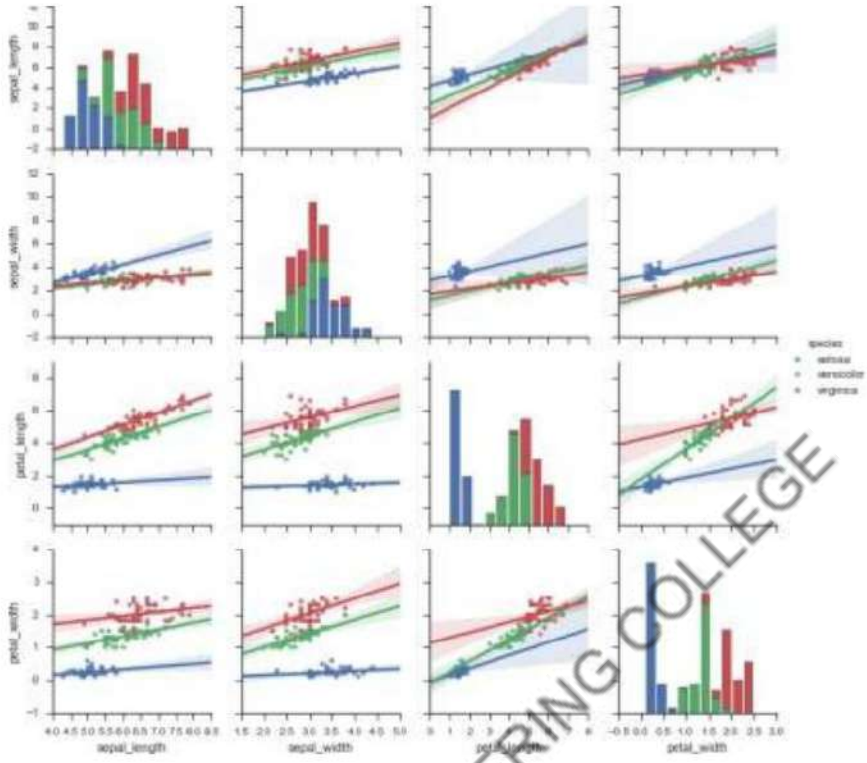
OUTPUT



In [7]:

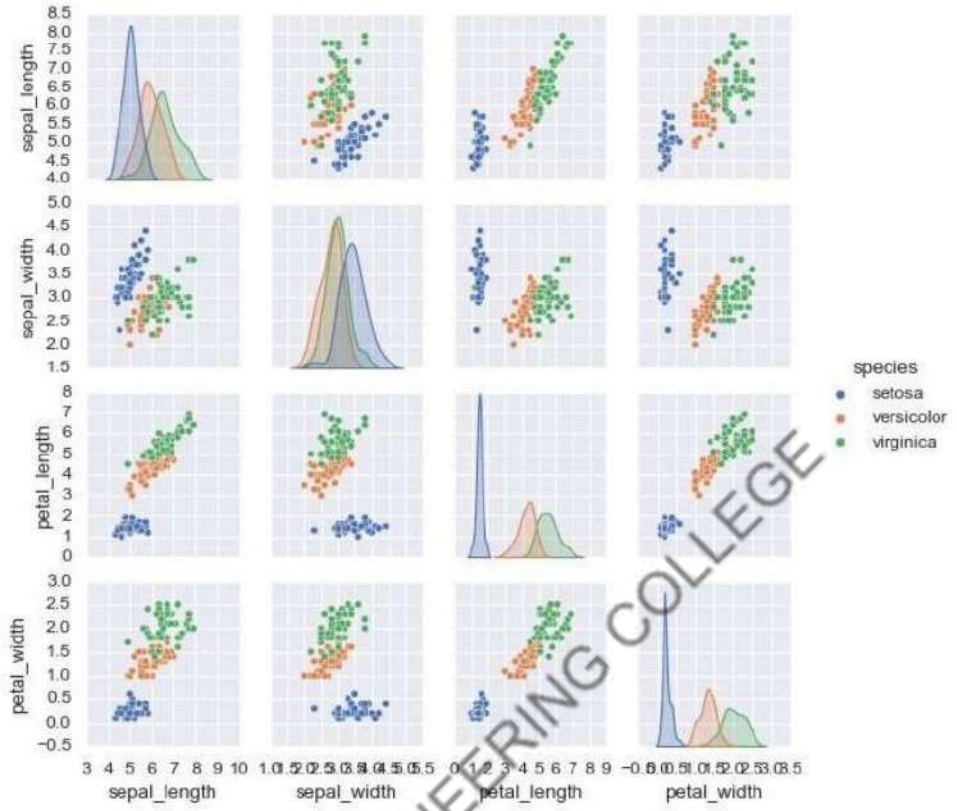
```
g = sns.pairplot(iris, kind="reg", hue="species")
```

OUTPUT



In[8]:

```
sns.set(style="ticks", color_codes=True) # change style
g = sns.pairplot(iris, hue="species")
```



Result:

Thus reading data from Text file and exploring various commands for doing descriptive analytics on the Iris data set is executed.

4 B). Aim:

Reading data from web and exploring various commands for doing descriptive analytics on the iris dataset.

Step 1:

Download the Iris dataset from the UCI machine learning repository.by providind the corresponding URL.

In [1]:

```
import pandas as pd

data = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data',header=None)
data.columns = ['sepal length', 'sepal width', 'petal length', 'petal
width', 'class']

data.head()
```

Out[1]:

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Step 2:

For each quantitative attribute, calculate its average, standard deviation, minimum, and maximum values.

In [2]:

```
from pandas.api.types import is_numeric_dtype

for col in data.columns:
    if is_numeric_dtype(data[col]):
        print('%s:' % (col))
        print('\t Mean = %.2f' % data[col].mean())
        print('\t Standard deviation = %.2f' % data[col].std())
        print('\t Minimum = %.2f' % data[col].min())
```

```
        print('\t Maximum = %.2f' % data[col].max())
```

Out[2]:

```
sepal length:
        Mean = 5.84
        Standard deviation = 0.83
        Minimum = 4.30
        Maximum = 7.90
sepal width:
        Mean = 3.05
        Standard deviation = 0.43
        Minimum = 2.00
        Maximum = 4.40
petal length:
        Mean = 3.76
        Standard deviation = 1.76
        Minimum = 1.00
        Maximum = 6.90
petal width:
        Mean = 1.20
        Standard deviation = 0.76
        Minimum = 0.10
        Maximum = 2.50
```

Step 3:

For the qualitative attribute (class), count the frequency for each of its distinct values.

In [3]:

```
data['class'].value_counts()
```

Out [3]:

```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: class, dtype: int64
```

Step 4:

It is also possible to display the summary for all the attributes simultaneously in a table using the describe() function. If an attribute is quantitative, it will display its mean, standard deviation and various quantiles (including minimum, median, and maximum) values. If an attribute is qualitative, it will display its number of unique values and the top (most frequent) values.

In [4]:

```
data.describe(include='all')
describe()
```

Out [4]:

	sepal length	sepal width	petal length	petal width	class
count	150.000000	150.000000	150.000000	150.000000	150
unique	NaN	NaN	NaN	NaN	3
top	NaN	NaN	NaN	NaN	Iris-setosa
freq	NaN	NaN	NaN	NaN	50
mean	5.843333	3.054000	3.758667	1.198667	NaN
std	0.828066	0.433594	1.764420	0.763161	NaN
min	4.300000	2.000000	1.000000	0.100000	NaN
25%	5.100000	2.800000	1.600000	0.300000	NaN
50%	5.800000	3.000000	4.350000	1.300000	NaN
75%	6.400000	3.300000	5.100000	1.800000	NaN
max	7.900000	4.400000	6.900000	2.500000	NaN

Step :5

For multivariate statistics, you can compute the covariance and correlation between pairs of attributes.

In [5]:

```
print('Covariance:')
data.cov()
```

Out[5]:

Covariance:

	sepal length	sepal width	petal length	petal width
sepal length	0.685694	-0.039268	1.273682	0.516904
sepal width	-0.039268	0.188004	-0.321713	-0.117981
petal length	1.273682	-0.321713	3.113179	1.296387
petal width	0.516904	-0.117981	1.296387	0.582414

In [6]:

```
print('Correlation:')
data.corr()
```

Out[6]:

Correlation:

	sepal length	sepal width	petal length	petal width
sepal length	1.000000	-0.109369	0.871754	0.817954
sepal width	-0.109369	1.000000	-0.420516	-0.356544
petal length	0.871754	-0.420516	1.000000	0.962757
petal width	0.817954	-0.356544	0.962757	1.000000

Result:

Thus the reading data from web and exploring various commands for doing descriptive analytics on the iris dataset is executed.

4C). Aim:

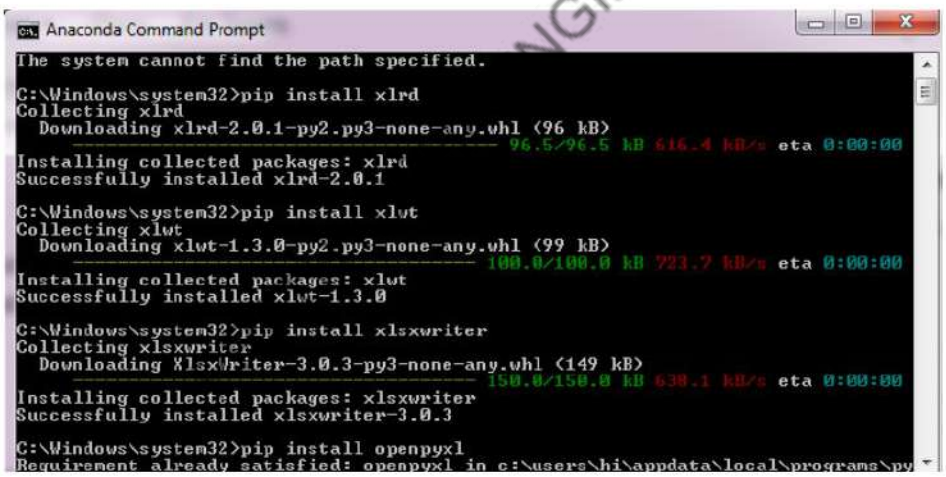
Reading data from Excel file and exploring various commands for doing descriptive analytics on the Iris data set.

Required python packages

- matplotlib – data visualization
- NumPy – numerical data functionality
- OpenPyXL – read/write Excel 2010 xlsx/xlsm files
- pandas – data import, clean-up, exploration, and analysis
- xlrd – read Excel data
- xlwt – write to Excel
- XlsxWriter – write to Excel (xlsx) files

You can install the required modules using pip. Open your command line program and execute command pip install <module name> to install a module. You should replace <module name> with the actual name of the module you are trying to install. For example, to install pandas, you would execute command –

eg: pip install pandas.



Step 1:

Create a excel file by name dept.. It can be saved as **dept.xlsx**

Sheet 1

	A	B	C	D	E
1	Sl.No	Stream	Name	Height	Weight
2	1	Science	Ayush	5.6	67
3	2	Science	Aniban	6	56
4	3	Commerce	Saurav	5.7	76
5	4	Humanities	laxman	5.8	58
6	5	Commerce	Rahul	6.1	49
7	6	Science	Sneha	6	55
8	7	Hotel Management	Harshit	5.11	68
9	8	Humanitics	Biiju	5.9	63
10	9	Aviation	karan	5.8	59
11	10	Physical Education	Rinu	5.5	65

dept1 dept2 Sheet3

Sheet 2

	A	B	C	D	E
1	Sl.No	Stream	Name	Height	Weight
2	1	Humanitie	Ankitha	5.1	67
3	2	Aviation	Nakul	5	64
4	3	Business	Rohan	5.6	56
5	4	Science	Satheesh	6	47
6	5	Science	Yuva	6.3	55
7	6	Commerc	Piyush	5.4	58
8	7	Hotel Mar	Sreeram	5.3	6
9	8	Hotel Mar	Rakul	5.7	54
10	9	Science	Rani	6	66
11	10	Humanitie	Yogesh	5.9	57

dept1 dept2 Sheet3

Step 2:

Now we can import the excel file using the read_excel function in pandas, as shown below: #place "r" before the path string to address special character, such as '\'. Don't forget to put the file name at the end of the path + '.xlsx'

In [1]:

```
import pandas as pd
df = pd.read_excel (r'C:\Users\HI\Downloads\dept.xlsx')

print (df)
```

Out [1]:

	Sl.No	Stream	Name	Height	Weight
0	1	Science	Ayush	5.60	67
1	2	Science	Aniban	6.00	56
2	3	Commerce	Saurav	5.70	76
3	4	Humanities	laxman	5.80	58
4	5	Commerce	Rahul	6.10	49
5	6	Science	Sneha	6.00	55
6	7	Hotel Management	Harshit	5.11	68
7	8	Humanitics	Biiju	5.90	63
8	9	Aviation	karan	5.80	59
9	10	Physical Education	Rinu	5.50	65

Step 2:

The second statement reads the data from excel and stores it into a pandas Data Frame which is represented by the variable newData. If there are multiple sheets in the excel workbook, the command will import data of the first sheet. To make a data frame with all the sheets in the workbook, the easiest method is to create different data frames separately and then concatenate them.

The read_excel method takes argument sheet_name and index_col where we can specify the sheet of which the data frame should be made of and specifies the title column.

In [2]:

```
sheet1 = pds.read_excel(file, sheet_name = 0, index_col = 0)

sheet2 = pds.read_excel(file, sheet_name = 1, index_col = 0)

newData = pds.concat([sheet1, sheet2])

newData
```

Out [2]:

Sl.No				
1	Science	Ayush	5.60	67
2	Science	Aniban	6.00	56
3	Commerce	Saurav	5.70	76
4	Humanities	laxman	5.80	58
5	Commerce	Rahul	6.10	49
6	Science	Sneha	6.00	55
7	Hotel Management	Harshit	5.11	68
8	Humanitics	Biiju	5.90	63
9	Aviation	karan	5.80	59
10	Physical Education	Rinu	5.50	65
1	Humanities	Ankitha	5.10	67
2	Aviation	Nakul	5.00	64
3	Business	Rohan	5.60	56
4	Science	Satheesh	6.00	47
5	Science	Yuva	6.80	55
6	Commerce	Piyush	5.40	58
7	Hotel Management	Sreeram	5.30	6
8	Hotel Management	Rakul	5.70	54
9	Science	Rani	6.00	66
10	Humanities	Yogesh	5.90	57

Step 3:

To view 5 columns from the top and from the bottom of the data frame, we can run the command

In [3] :

```
newData.tail()
```

Out[3]:

Sl.No	Stream	Height	Weight	Name
6	Commerce	5.4	58	Piyush
7	Hotel Management	5.3	6	Sreeram
8	Hotel Management	5.7	54	Rakul
9	Science	6.0	66	Rani
10	Humanities	5.9	57	Yogesh

In [4] :

```
newData.head()
```

Out[4]:

Sl.No	Stream	Name	Height	Weight
1	Science	Ayush	5.6	67
2	Science	Aniban	6.0	56
3	Commerce	Saurav	5.7	76
4	Humanities	laxman	5.8	58
5	Commerce	Rahul	6.1	49

Step 5:

If any column contains numerical data, we can sort that column using the sort_values() method in pandas as follows:

In[5]:

```
sorted_column = newData.sort_values(['Weight'], ascending = True)

sorted_column.head(5)
```

Out[5]:

Sl.No	Stream	Name	Height	Weight
7	Hotel Management	Sreeram	5.3	6
4	Science	Satheesh	6.0	47
5	Commerce	Rahul	6.1	49
8	Hotel Management	Rakul	5.7	54
6	Science	Sneha	6.0	55

Step 6:

Our data is mostly numerical. We can get the statistical information like mean, max, min, etc. about the data frame using the describe() method as shown below:

In [6]:

```
newData.describe()
```

Out [6]:

	Height	Weight
count	20.000000	20.000000
mean	5.690500	57.300000
std	0.361742	13.955191
min	5.000000	6.000000
25%	5.475000	55.000000
50%	5.750000	58.000000
75%	6.000000	65.250000
max	6.300000	76.000000

Result:

Thus reading data from Excel file and exploring various commands for doing descriptive analytics has been executed.

Ex.No.5	USE THE DIABETES DATA SET FROM UCI AND PIMA INDIANS DIABETES

Aim:

Use the data set from UCI and Pima Indians diabetes and find the diabetic patients.

Coding:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
dataset=pd.read_csv("E:\\ds\diabetes.csv")
dataset.head()
```

Output:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	2	138	62	35	0	33.6	0.127	47	1
1	0	84	82	31	125	38.2	0.233	23	0
2	0	145	0	0	0	44.2	0.630	31	1
3	0	135	68	42	250	42.3	0.365	24	1
4	1	139	62	41	480	40.7	0.536	21	0


```
dataset.shape
```

output:

```
(2000, 9)
```

```
dataset.describe()
```

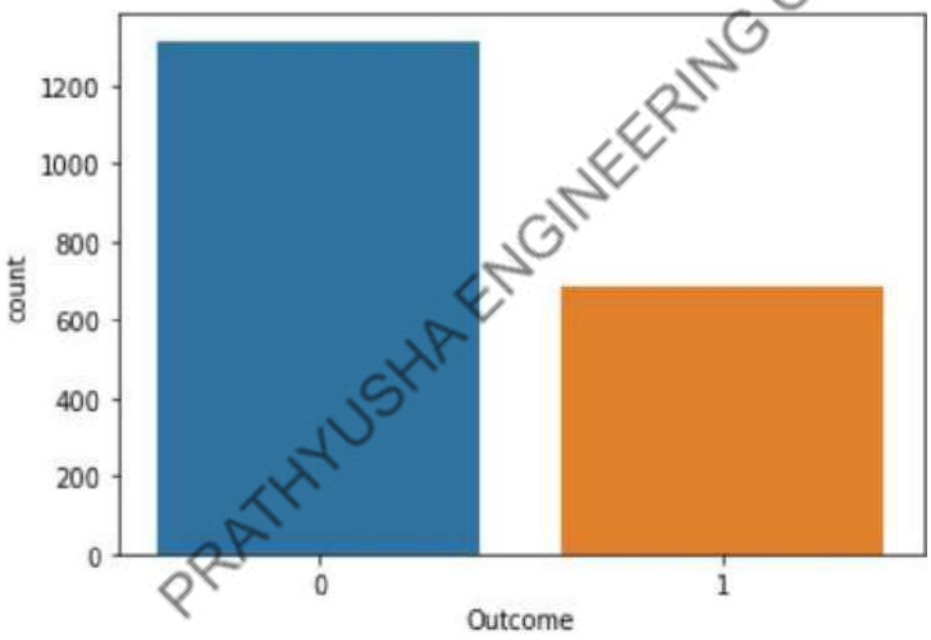
output:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	3.703500	121.182500	69.145500	20.935000	80.254000	32.193000	0.470930	33.090500	0.342000
std	3.306063	32.068636	19.188315	16.103243	111.180534	8.149901	0.323553	11.786423	0.474498
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	63.500000	0.000000	0.000000	27.375000	0.244000	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	40.000000	32.300000	0.376000	29.000000	0.000000
75%	6.000000	141.000000	80.000000	32.000000	130.000000	36.800000	0.624000	40.000000	1.000000
max	17.000000	199.000000	122.000000	110.000000	744.000000	80.600000	2.420000	81.000000	1.000000

sns.countplot(x='Outcome',data=dataset)

output:

```
<AxesSubplot:xlabel='Outcome', ylabel='count'>
```



#

dataset['Outcome'].value_counts()

OUTPUT

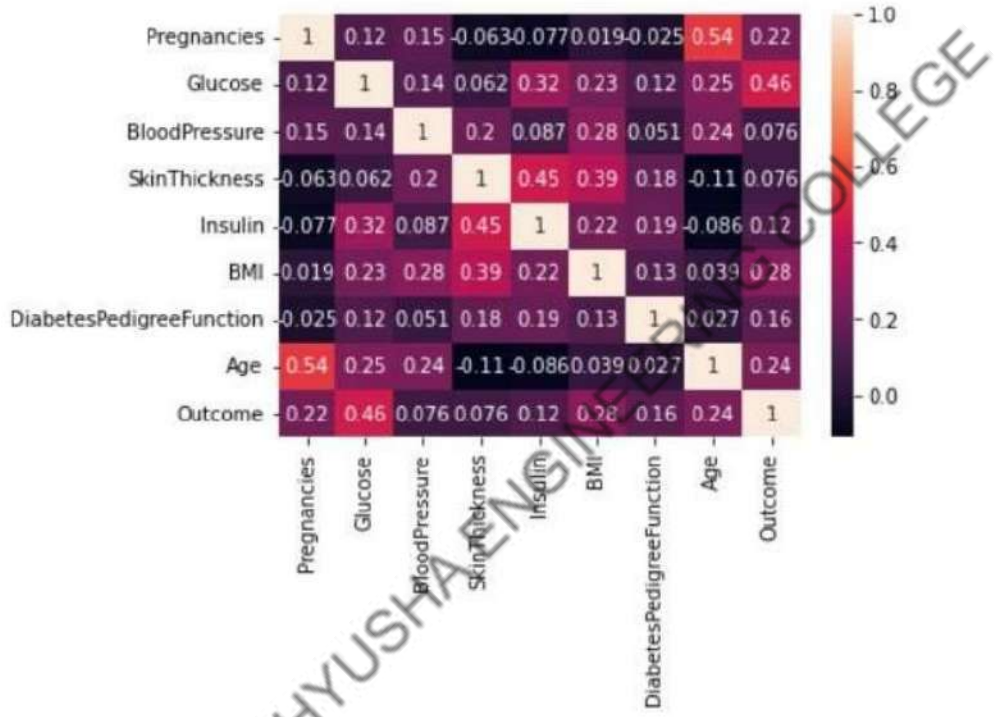
```
Out[31]: 0    1316
         1     684
         Name: Outcome, dtype: int64
```

corr_mat=dataset.corr()

sns.heatmap(corr_mat,annot=True)

output:

```
<AxesSubplot:>
```



Data cleaning

#check any null or empty data is present in the dataset

dataset.isna().sum()

output:

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Feature matrix -Taking all our independent columns into single array and dependent values into another array

x=dataset.iloc[:,:-1].values

y=dataset.iloc[:,-1].values

x.shape

output

```
(2000, 8)
```

#

x[0]

OUTPUT

```
array([2.00e+00, 1.38e+02, 6.20e+01, 3.50e+01, 0.00e+00, 3.36e+01,
       1.27e-01, 4.70e+01])
```

#

y

Output

array([1, 0, 1, ..., 0, 1, 0], dtype=int64)

#

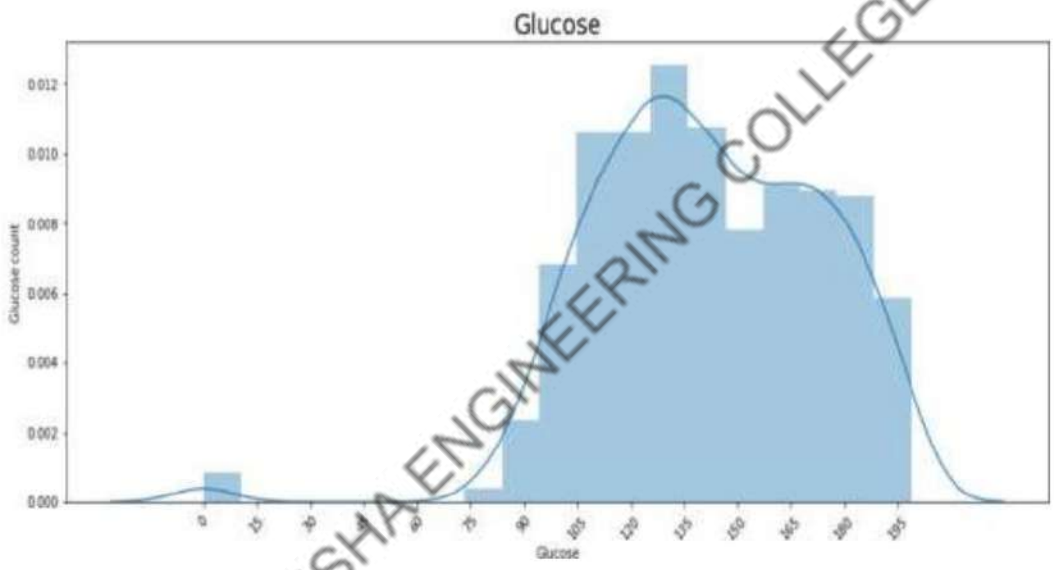
fig = plt.figure(figsize=(16,6))

sns.distplot(dataset["Glucose"][dataset["Outcome"]==1])

```
plt.xticks([i for i in range(0,201,15)],rotation=45)
plt.ylabel("Glucose count")
plt.title("Glucose",fontsize=20)
#
```

OUTPUT

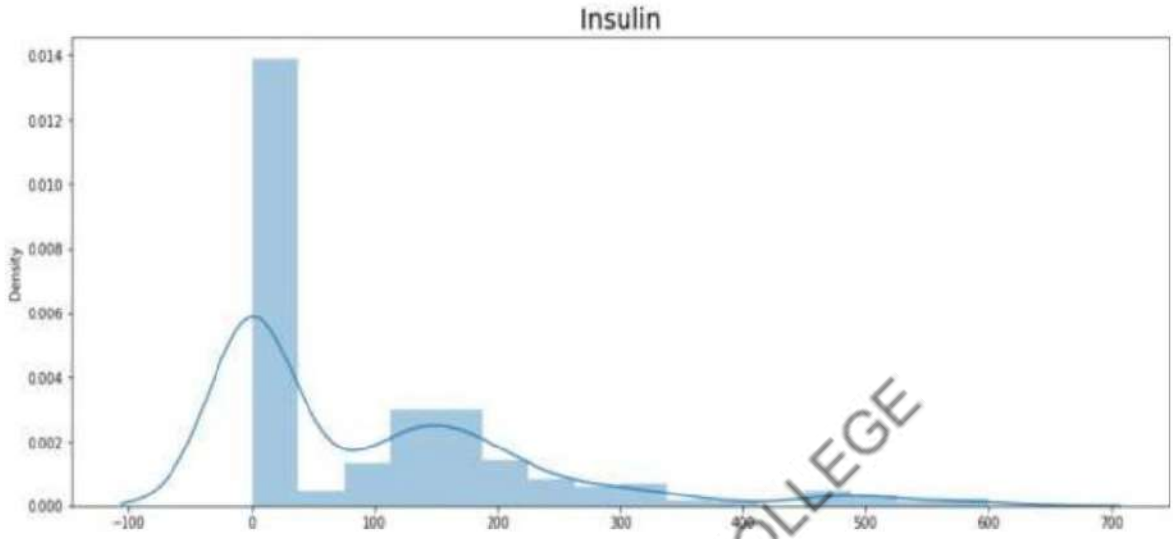
```
Out[17]: Text(0.5, 1.0, 'Glucose')
```




```
#
fig = plt.figure(figsize=(16,6))
sns.distplot(dataset["Insulin"][dataset["Outcome"]==1])
plt.xticks()
plt.title("Insulin",fontsize=20)
```

OUTPUT

```
Out[18]: Text(0.5, 1.0, 'Insulin')
```

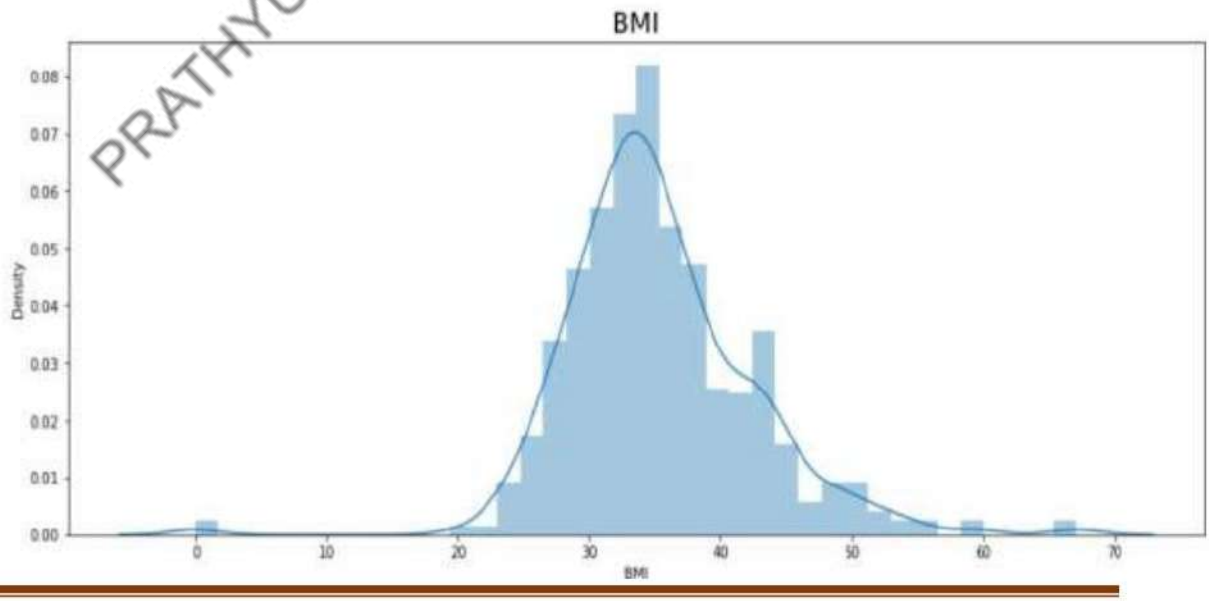


#

```
fig = plt.figure(figsize=(16,6))
sns.distplot(dataset["BMI"][dataset["Outcome"]==1])
plt.xticks()
plt.title("BMI",fontsize=20)
```

OUTPUT

```
Out[19]: Text(0.5, 1.0, 'BMI')
```

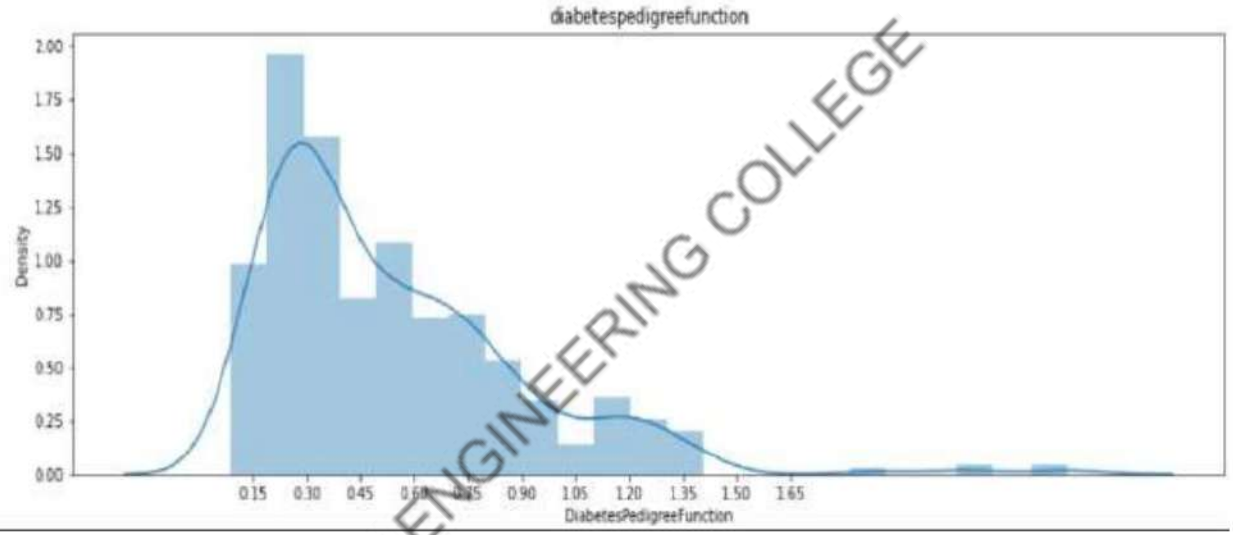


#

```
fig = plt.figure(figsize=(16,5))
sns.distplot(dataset["DiabetesPedigreeFunction"][dataset["Outcome"]==1])
plt.xticks([i*0.15 for i in range(1,12)])
plt.title("diabetespedigreefunction")
```

Output

```
Out[21]: Text(0.5, 1.0, 'diabetespedigreefunction')
```

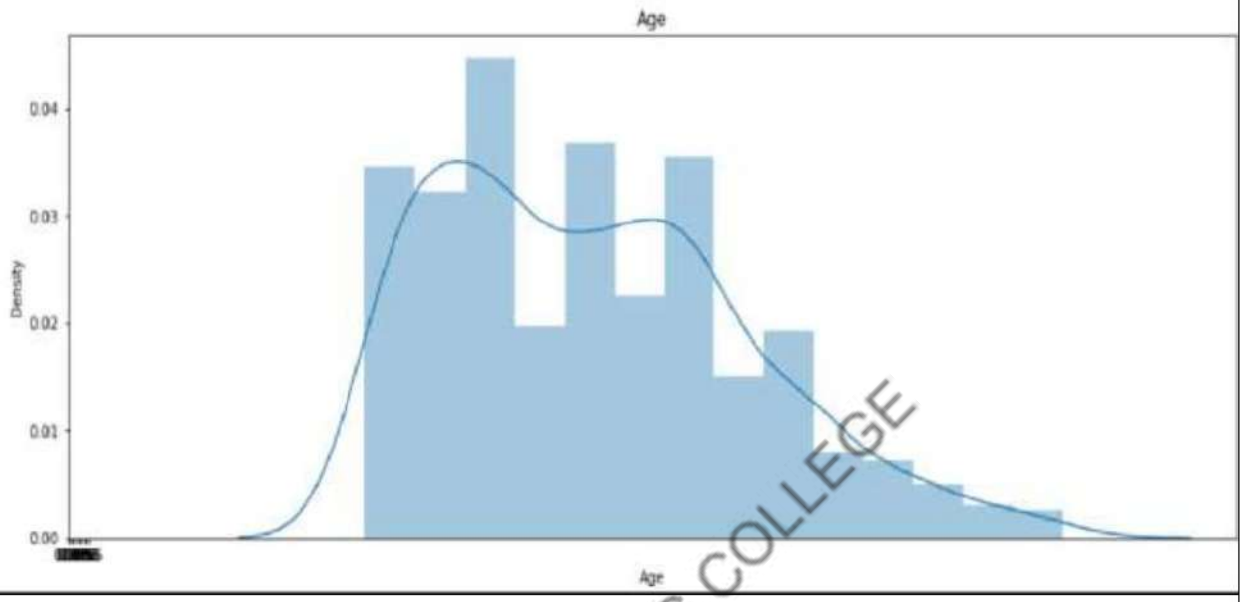


#

```
fig = plt.figure(figsize=(16,6))
sns.distplot(dataset["Age"][dataset["Outcome"]==1])
plt.xticks([i*0.15 for i in range(1,12)])
plt.title("Age")
```

Output

```
Out[22]: Text(0.5, 1.0, 'Age')
```




```
#
x = dataset.drop(["Pregnancies","BloodPressure","SkinThickness","Outcome"],axis = 1)
y = dataset.iloc[:,-1]
#
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2, random_state=0)
#
x_train.shape
```

Output

```
Out[41]: (1600, 5)
```

```
#
x_test.shape
```

```
Out[42]: (400, 5)
```

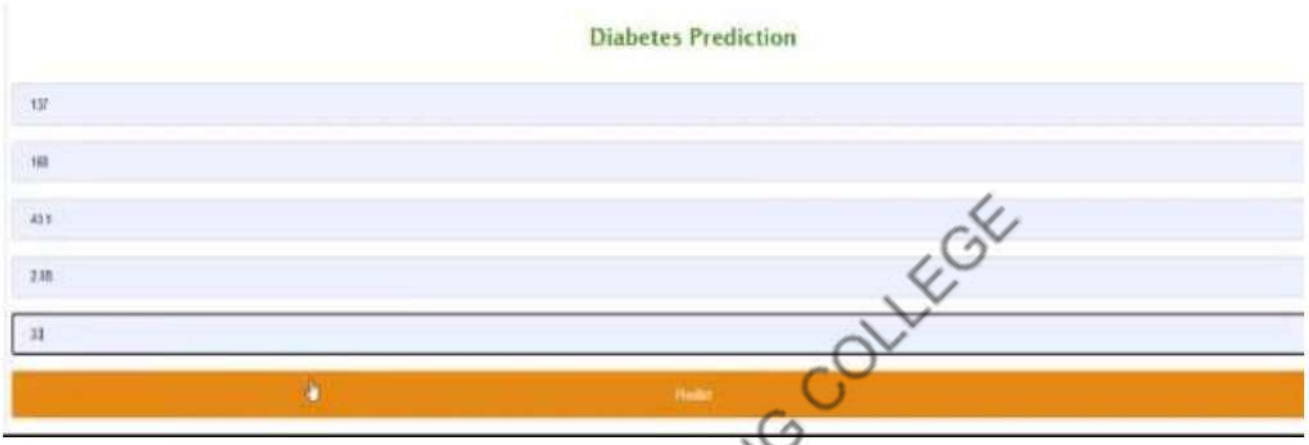
Saving the classifier:

Import pickle

Pickle.dump(svc,open('classifier.pkl',wb))

Pickle.dump(sc,open('sc.pkl','wb'))

Output:



Results:

Chances of having Diabetes is more, please consult a Doctor.

Result:

Ex.No.6	**APPLY AND EXPLORE VARIOUS PLOTTING FUNCTIONSON UCI DATA SETS.** **A. NORMAL CURVES** **B. DENSITY AND CONTOUR PLOTS** **C. CORRELATION AND SCATTER PLOTS** **D. HISTOGRAMS** **E. THREE DIMENSIONAL PLOTTING**

Aim:

To apply and explore various plotting functions on UCI data sets.
A. Normal curves
B. Density and contour plots
C. Correlation and scatter plots
D. Histograms
E. Three-dimensional plotting

6A) Normal curves

Aim:

To apply and explore the probability density function on normal curves.

Normal Distribution is a probability function used in statistics that tells about how the data values are distributed. It is the most important probability distribution function used in statistics because of its advantages in real case scenarios. For example, the height of the population, shoe size, IQ level, rolling a die, and many more.
The probability density function of normal or Gaussian distribution is given by:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Probability Density Function

Where, x is the variable, mu is the mean, and sigma standard deviation

Modules Needed

- ***Matplotlib*** is python's data visualization library which is widely used for the purpose of data visualization.
- ***Numpy*** is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

- ***Scipy*** is a python library that is useful in solving many mathematical equations and algorithms.
- ***Statistics*** module provides functions for calculating mathematical statistics of numeric data.

Functions used

- To calculate mean of the data

mean(data)

- To calculate standard deviation of the data

stdev(data)

- To calculate normal probability density of the data norm.pdf is used, it refers to the normal probability density function which is a module in scipy library that uses the above probability density function to calculate the value.

Syntax:

norm.pdf(Data, loc, scale)

Here, loc parameter is also known as the mean and the scale parameter is also known as standard deviation.

Approach

- Import module
- Create data
- Calculate mean and deviation
- Calculate normal probability density
- Plot using above calculated values
- Display plot

Implementation

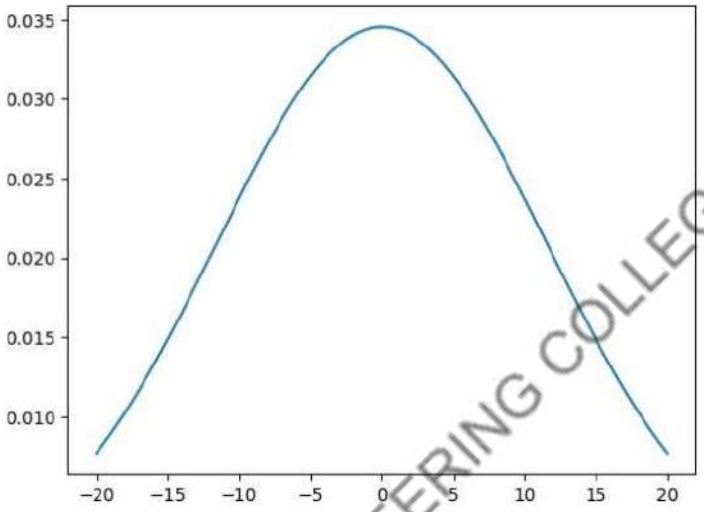
Step 1: Draw a normal Curve

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import statistics

# Plot between -10 and 10 with .001 steps.
```

```
x_axis = np.arange(-20, 20, 0.01)
# Calculating mean and standard deviation
mean = statistics.mean(x_axis)
sd = statistics.stdev(x_axis)
plt.plot(x_axis, norm.pdf(x_axis, mean, sd))
plt.show()
```

Out [1]:



Step 2:

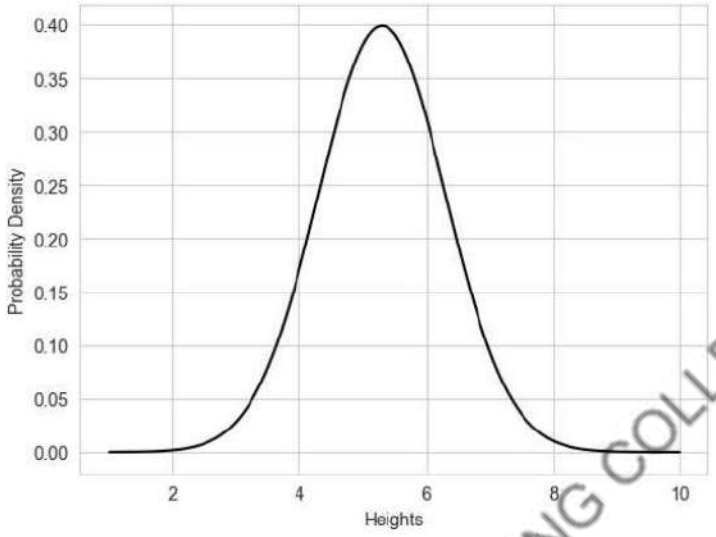
Find the probability density function

In [2]:

```
# import required libraries

from scipy.stats import norm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb

# Creating the distribution
data = np.arange(1,10,0.01)
pdf = norm.pdf(data , loc = 5.3 , scale = 1 )

#Visualizing the distribution

sb.set_style('whitegrid')
sb.lineplot(data, pdf , color = 'black')
plt.xlabel('Heights')
plt.ylabel('Probability Density')
```

Out [2]:

```
Text(0, 0.5, 'Probability Density')
```



Result:

Thus the probability density function on normal curves is applied and executed..

6B) Density and contour plots:

Aim:

To apply and explore Density and Contour Plotting function on UCI Datasets

Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task:

plt.contour for contour plots,
plt.contourf for filled contour plots, and
plt.imshow for showing images.

Start with **Jupyter notebook** in command prompt.

In [1] :

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
```

Visualizing a Three-Dimensional Function

We'll start by demonstrating a contour plot using a function $z=(x,y)$z=f(x,y), using the following particular choice for fff .

In [2]:

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

A contour plot can be created with the **plt.contour** function.

It takes three arguments:

a grid of x values,
a grid of y values, and
a grid of z values.

The x and y values represent positions on the plot, and the z values will be represented by the contour levels.

Perhaps the most straightforward way to prepare such data is to use the **np.meshgrid** function, which builds two-dimensional grids from one-dimensional arrays:

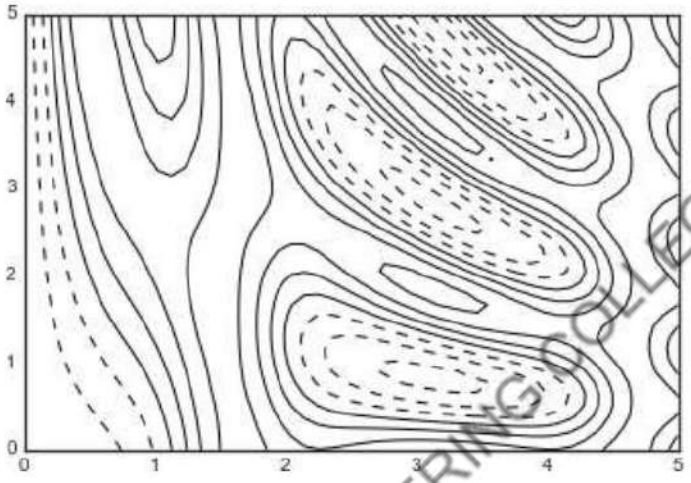
In [3]:

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

In [4]:

```
plt.contour(X, Y, Z, colors='black');
```

OUTPUT



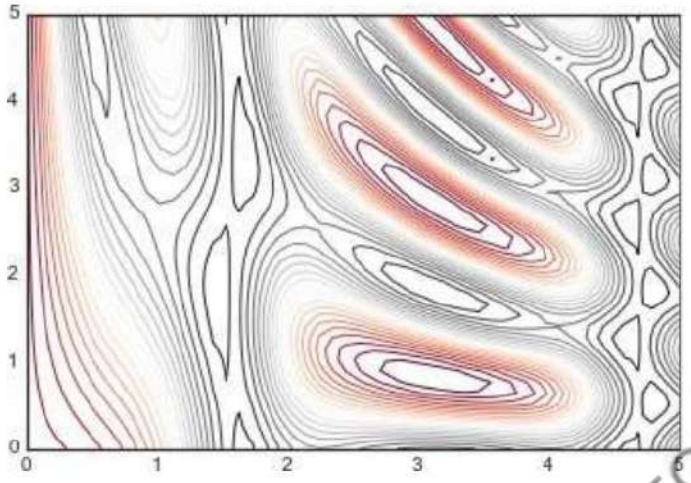
Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.

Alternatively, the lines can be color-coded by specifying a colormap with the **cmap** argument. Here, we'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range:

In [5]:

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

OUTPUT



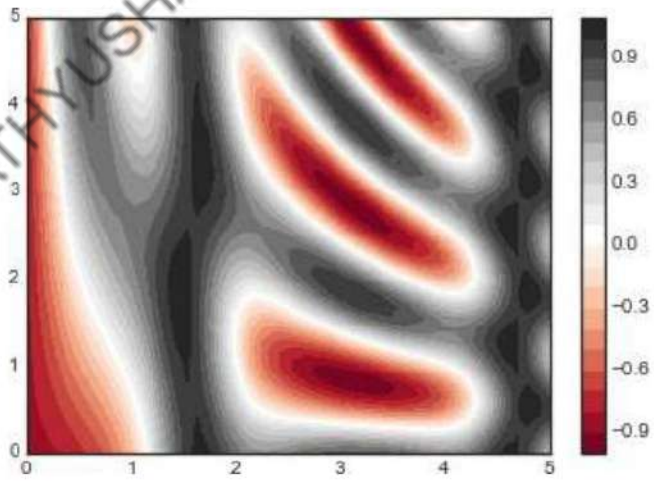
The spaces between the lines may be a bit distracting. We can change this by switching to a filled contour plot using the **plt.contourf()** function (notice the f at the end), which uses largely the same syntax as **plt.contour().**

Add a **plt.colorbar()** command, which automatically creates an additional axis with labeled color information for the plot:

In [6]:

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```

OUTPUT



The colorbar makes it clear that the black regions are "peaks," while the red regions are "valleys."

One potential issue with this plot is that it is a bit "splotchy." That is, the color steps are discrete rather than continuous, which is not always what is desired.

This could be remedied by setting the number of contours to a very high number, but this results in a rather inefficient plot:

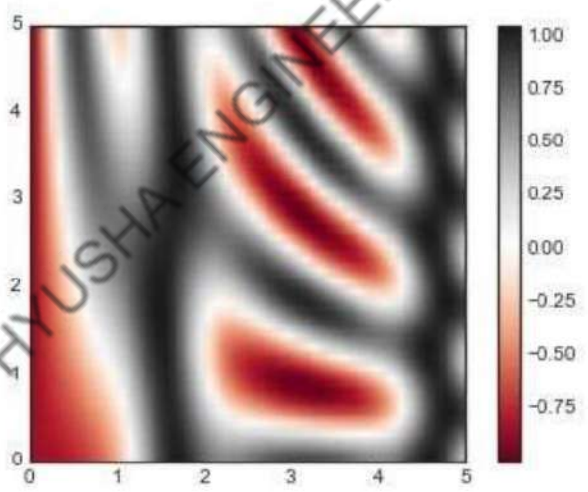
Matplotlib must render a new polygon for each step in the level. A better way to handle this is to use the **plt.imshow()** function, which interprets a two-dimensional grid of data as an image.

The following code shows this:

In [7]:

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image');
```

OUTPUT



There are a few potential gotchas with **imshow(), however:**

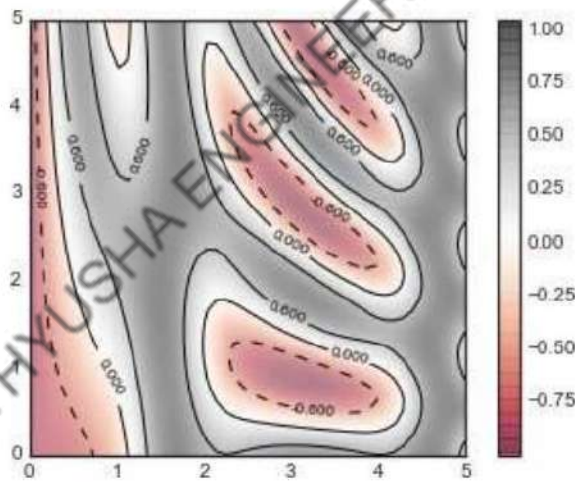
- **plt.imshow()** doesn't accept an *x* and *y* grid, so you must manually specify the *extent* [*xmin*, *xmax*, *ymin*, *ymax*] of the image on the plot.
- **plt.imshow()** by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.

- **plt.imshow()** will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, for example, plt.axis(aspect='image') to make *x* and *y* units match.
- Finally, it can sometimes be useful to combine contour plots and image plots. For example, here we'll use a partially transparent background image (with transparency set via the **alpha** parameter) and overplot contours with labels on the contours themselves (using the **plt.clabel()** function):

In [8]:

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
           cmap='RdGy', alpha=0.5)
plt.colorbar();
```

OUTPUT



The combination of these three functions—**plt.contour**, **plt.contourf**, and **plt.imshow**—gives nearly limitless possibilities for displaying this sort of three-dimensional data within a two-dimensional plot.

Result:

Thus the probability density function on density and contour plotting is done.

6 C.Correlation and Scatterplots

Aim:

To apply and explore Correlation and Scatterplots function on UCI Datasets

i) Simple Scatter Plots

Commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape. We'll start by setting up the notebook for plotting and importing the functions we will use:

In [1]:

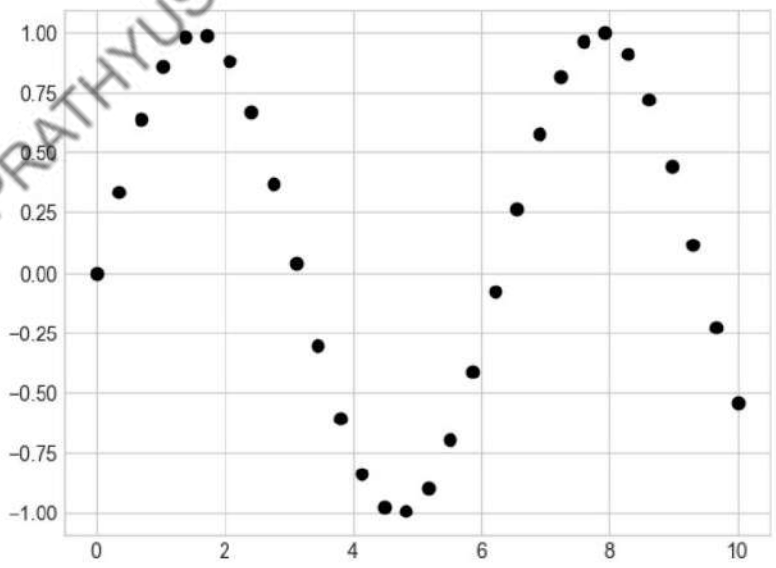
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

Run the file and then type next line.

In [2]:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```

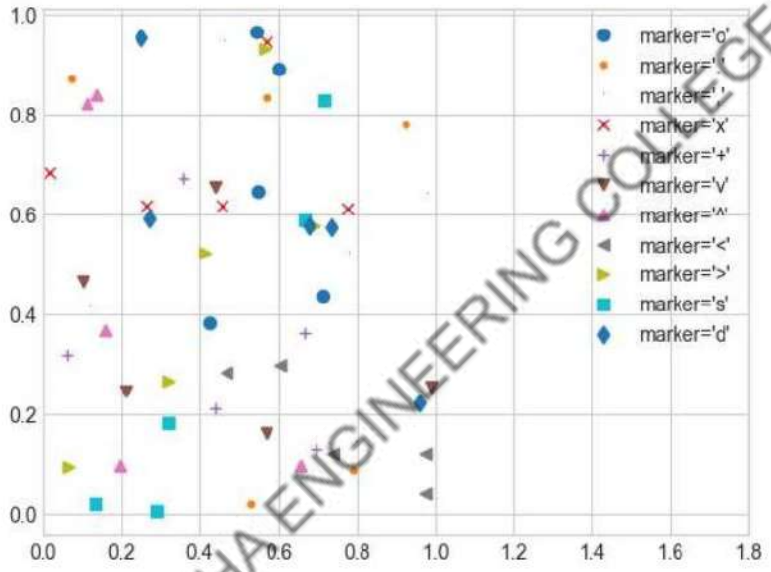
OUTPUT



In [3]:

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
             label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

OUTPUT

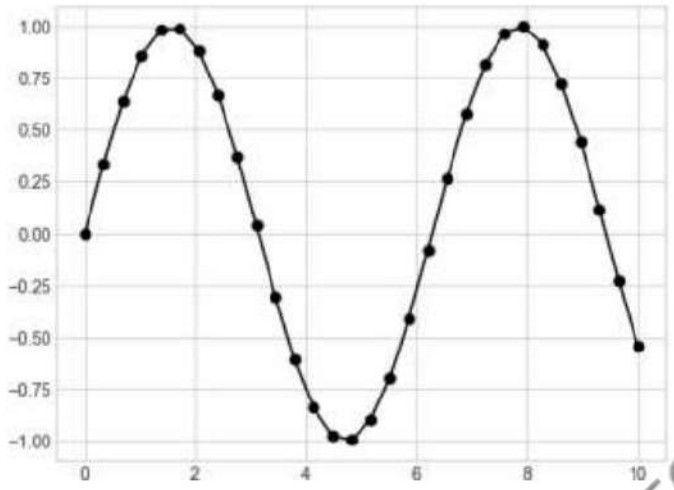


For even more possibilities, these character codes can be used together with line and color codes to plot points along with a line connecting them:

In [4]:

```
plt.plot(x, y, '-ok');
```

OUTPUT

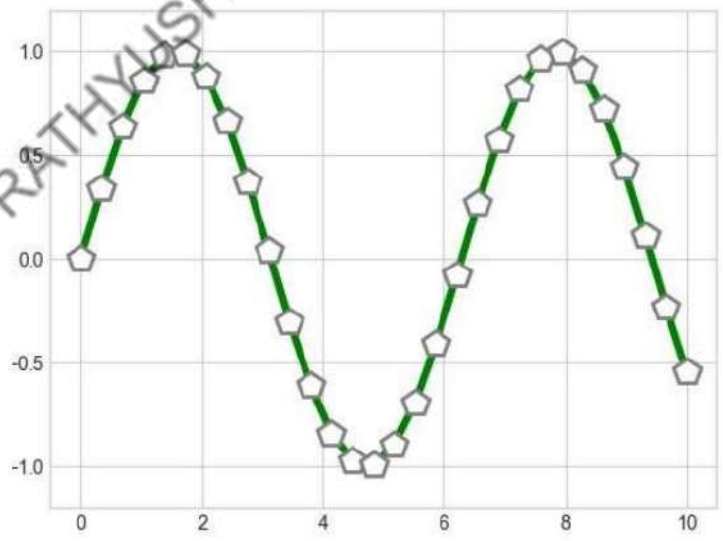


Additional keyword arguments to **plt.plot** specify a wide range of properties of the lines and markers:

In [5]:

```
plt.plot(x, y, '-p', color='gray',
         markersize=15, linewidth=4,
         markerfacecolor='white',
         markeredgecolor='gray',
         markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

OUTPUT



This type of flexibility in the **plt.plot** function allows for a wide variety of possible visualization options. For a full description of the options available, refer to the **plt.plot** documentation.

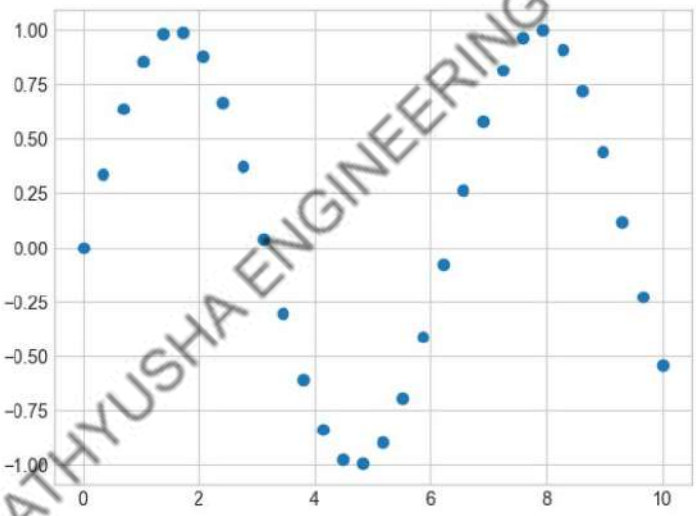
Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the ``**plt.scatter**`` function, which can be used very similarly to the ``**plt.plot**`` function:

In [6]:

```
plt.scatter(x, y, marker='o');
```

OUTPUT



The primary difference of **plt.scatter** from **plt.plot** is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

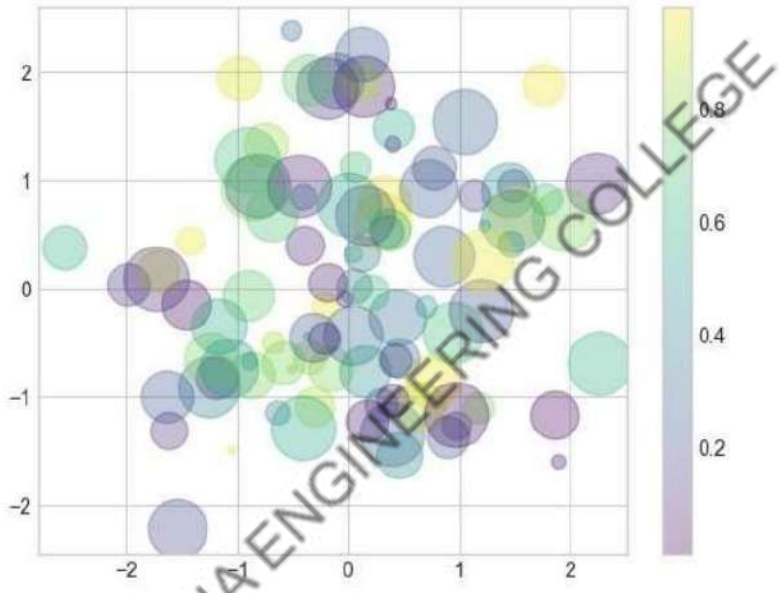
Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the **alpha** keyword to adjust the transparency level:

In [7]:

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
```

```
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar();  # show color scale
```

OUTPUT

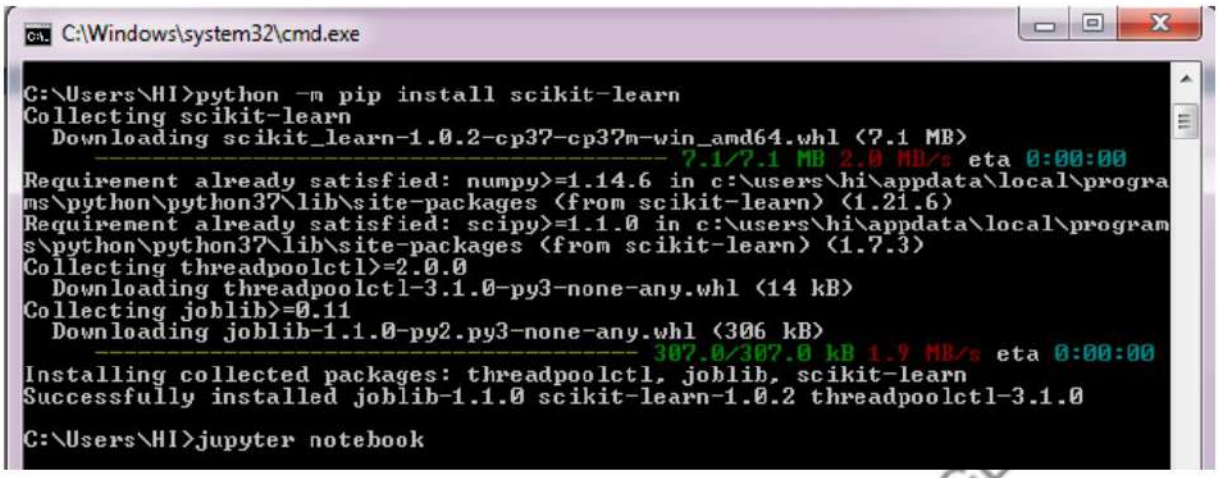


Notice that the color argument is automatically mapped to a color scale (shown here by the **colorbar**() command), and that the size argument is given in pixels. In this way, the color and size of points can be used to convey information in the visualization, in order to visualize multidimensional data.

For example, we might use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully measured:

For running this module **scikit** package should be installed. The command used is

```
python -m pip install scikit-learn
```

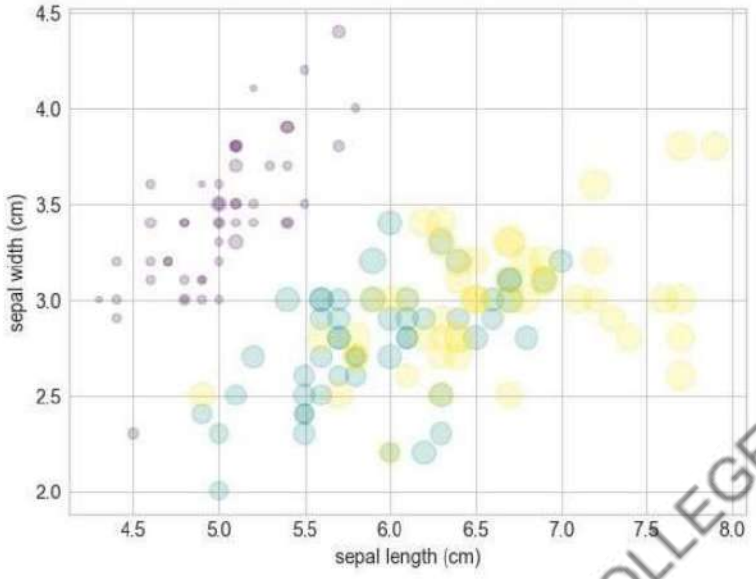


Then launch to jupyter notebook

In [8]:

```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T

plt.scatter(features[0], features[1], alpha=0.2,
            s=100*features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
```

OUTPUT



We can see that this scatter plot has given us the ability to simultaneously explore four different dimensions of the data: the (x, y) location of each point corresponds to the sepal length and width, the size of the point is related to the petal width, and the color is related to the particular species of flower. Multicolor and multifeature scatter plots like this can be useful for both exploration and presentation of data.

plot Versus scatter: A Note on Efficiency

Aside from the different features available in **plt.plot** and **plt.scatter**, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, **plt.plot** can be noticeably more efficient than **plt.scatter**. The reason is that **plt.scatter** has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In **plt.plot**, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, **plt.plot** should be preferred over **plt.scatter** for large datasets.

Result:

Thus the Correlation and Scatterplots function on UCI Datasets are executed.

6 D.Histograms

Aim:

To apply and explore Histograms function on UCI Datasets

Preliminaries

A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function, which creates a basic histogram in one line, once the normal boiler-plate imports are done:

Program

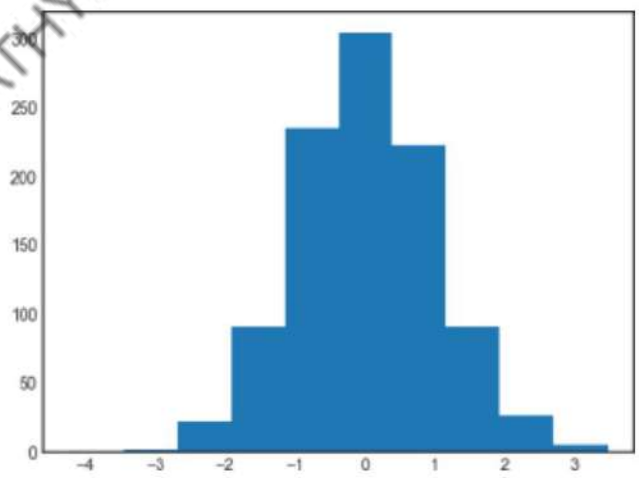
In [1]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

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

In [2]:

```
plt.hist(data);
```

OUTPUT

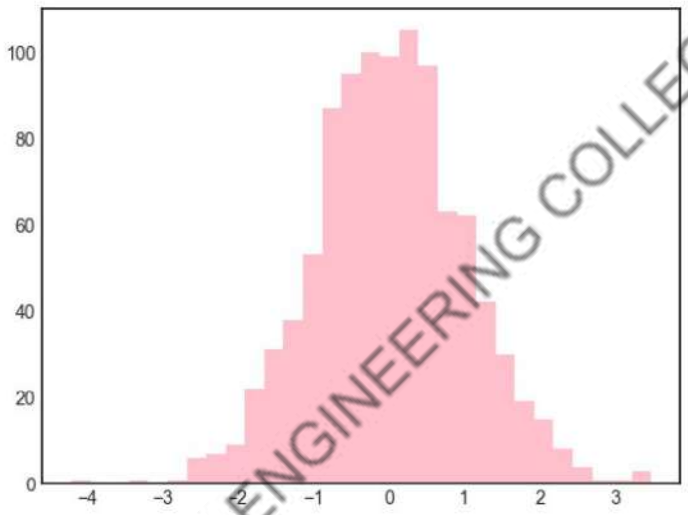


The **hist()** function has many options to tune both the calculation and the display; here's an example of a more customized histogram:

In [3]:

```
plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

OUTPUT

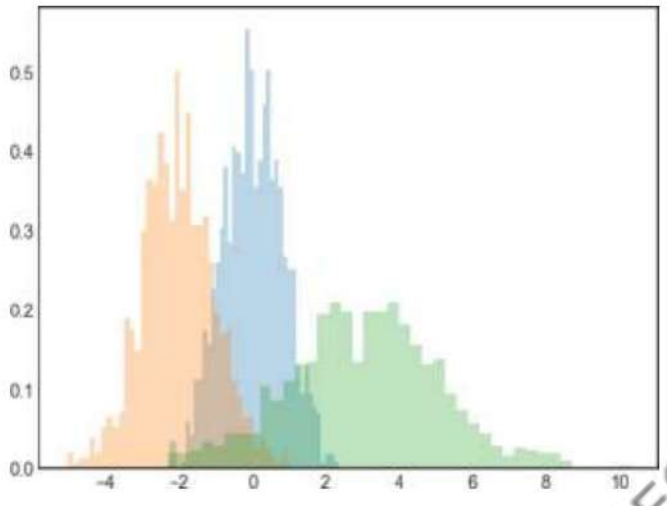


The **plt.hist** docstring has more information on other customization options available. I find this combination of **histtype='stepfilled'** along with some transparency **alpha** to be very useful when comparing histograms of several distributions:

In [4]:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

OUTPUT



If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the **np.histogram()** function is available:

In [5]:

```
counts, bin_edges = np.histogram(data, bins=5)
print(counts)
```

OUTPUT

```
In [12]: counts, bin_edges = np.histogram(data, bins=5)
         print(counts)

         [  3 113 539 313  32]
```


Two-Dimensional Histograms and Binnings

Just as we create histograms in one dimension by dividing the number-line into bins, we can also create histograms in two-dimensions by dividing points among two-dimensional bins. We'll take a brief look at several ways to do this here. We'll start by defining some data—an **x** and **y** array drawn from a multivariate Gaussian distribution:

In [6]:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

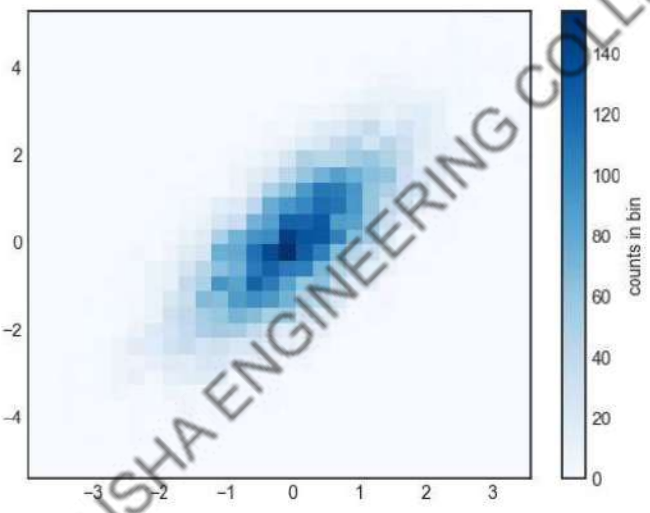
plt.hist2d: Two-dimensional histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's **plt.hist2d** function:

In [7]:

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```

OUTPUT



Just as with **plt.hist, plt.hist2d** has a number of extra options to fine-tune the plot and the binning, which are nicely outlined in the function docstring. Further, just as **plt.hist** has a counterpart in **np.histogram**, **plt.hist2d** has a counterpart **in np.histogram2d**, which can be used as follows:

In [8]:

```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

For the generalization of this histogram binning in dimensions higher than two, see the **np.histogramdd** function.

plt.hexbin: Hexagonal binnings

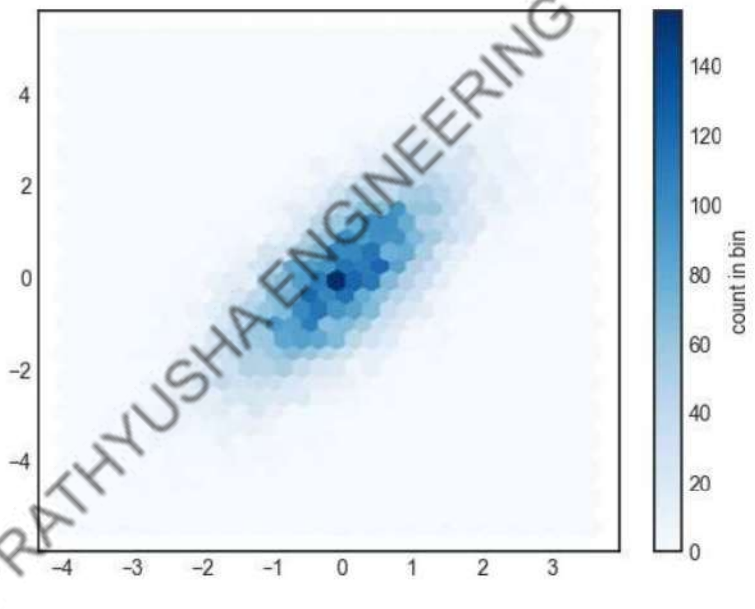
The two-dimensional histogram creates a tesselation of squares across the axes. Another natural shape for such a tesselation is the regular hexagon.

For this purpose, Matplotlib provides the **plt.hexbin** routine, which will represents a two-dimensional dataset binned within a grid of hexagons:

In [9]:

```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```

OUTPUT



plt.hexbin has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

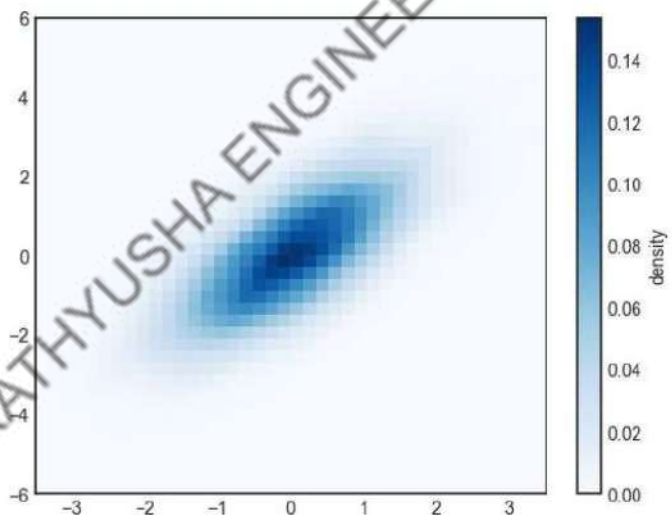
Kernel density estimation

Another common method of evaluating densities in multiple dimensions is *kernel density estimation* (KDE). One extremely quick and simple KDE implementation exists in the scipy.stats package. Here is a quick example of using the KDE on this data:

In [10]:

```
from scipy.stats import gaussian_kde

# fit an array of size [Ndim, Nsamples]
data = np.vstack([x, y])
kde = gaussian_kde(data)

# evaluate on a regular grid
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),
           origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6],
           cmap='Blues')
cb = plt.colorbar()
cb.set_label("density")
```

OUTPUT



Result:

Thus the Histograms function on UCI Datasets are executed.

6 E Three dimensional plotting

Aim:

To apply and explore Three Dimensional plotting function on UCI Datasets

Three-Dimensional Plotting in Matplotlib

Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. three-dimensional plots are enabled by importing the mplot3d toolkit, included with the main Matplotlib installation:

Program

In [1]:

```
from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword **projection='3d'** to any of the normal axes creation routines:

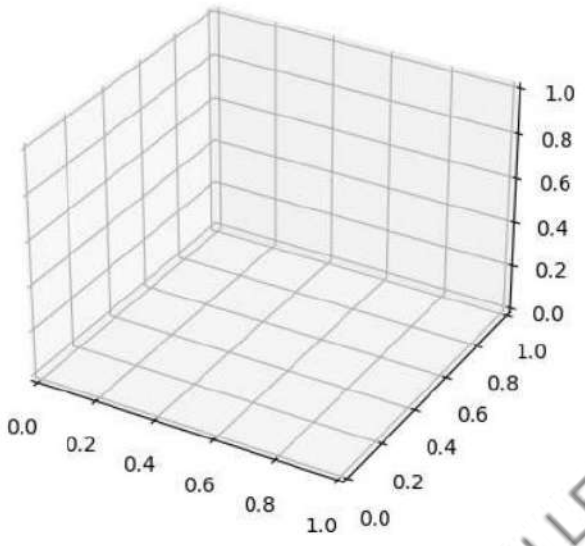
In [2]:

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

In [3]:

```
fig = plt.figure()
ax = plt.axes(projection='3d')
```

OUTPUT



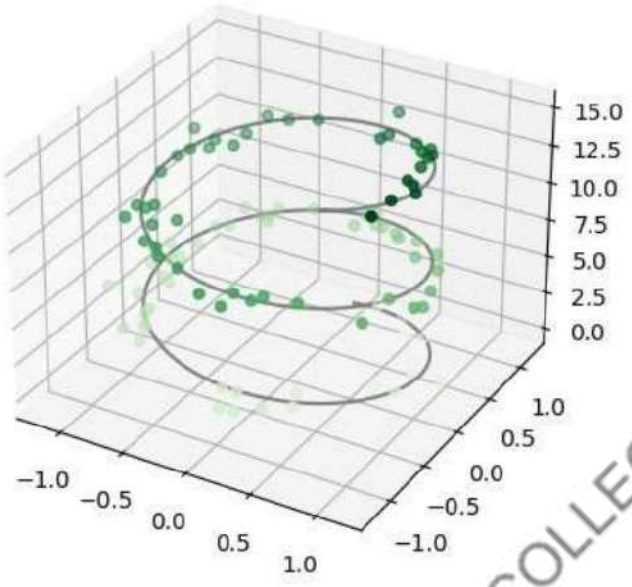
Three-dimensional Points and Lines

The most basic three-dimensional plot is a line or collection of scatter plot created from sets of (x, y, z) triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the ax.plot3D and ax.scatter3D functions. The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots and Simple Scatter Plots for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line:

In [4]:

```
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline  =  np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

OUTPUT



Three-dimensional Contour Plots

Analogous to the contour plots we explored in Density and Contour Plots, **mplot3d** contains tools to create three-dimensional relief plots using the same inputs. Like two-dimensional **ax.contour** plots**, ax.contour3D** requires all the input data to be in the form of two-dimensional regular grids, with the Z data evaluated at each point. Here we'll show a three-dimensional contour diagram of a three-dimensional sinusoidal function

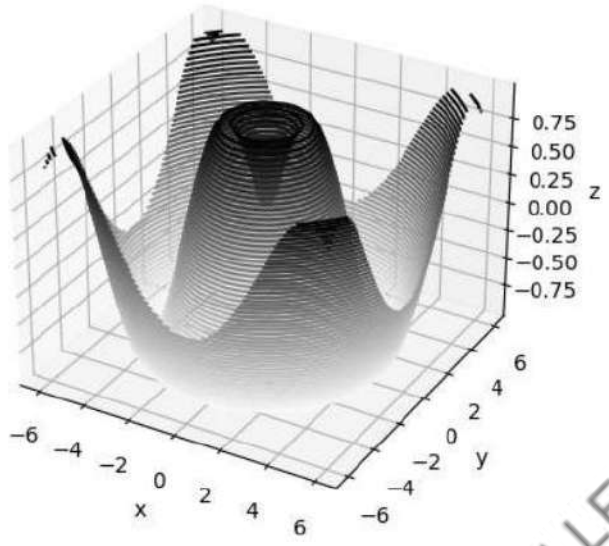
In [5]:

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

In [6]:

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```

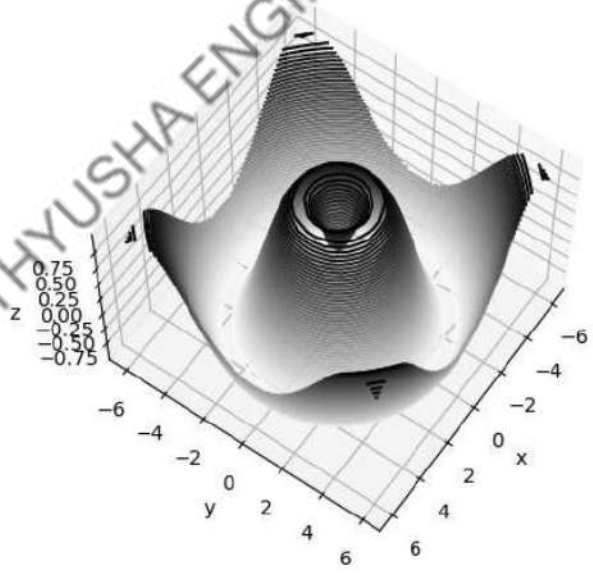
OUTPUT



In [7]:

```
ax.view_init(60, 35)
fig
```

OUTPUT



Wireframes and Surface Plots

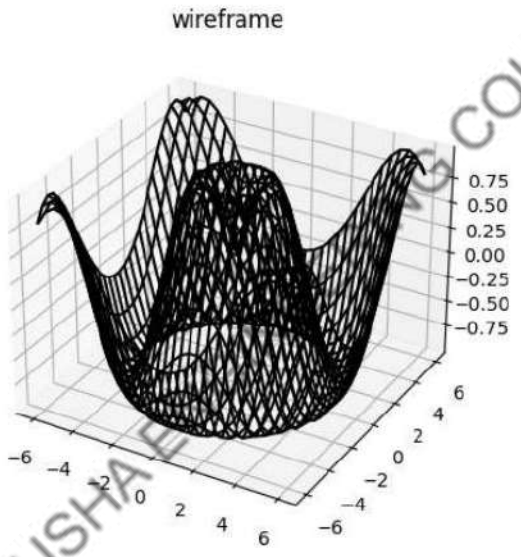
Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots. These take a grid of values and project it onto the specified three-dimensional surface, and

can make the resulting three-dimensional forms quite easy to visualize. Here's an example of using a wireframe:

In [7]:

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```

OUTPUT

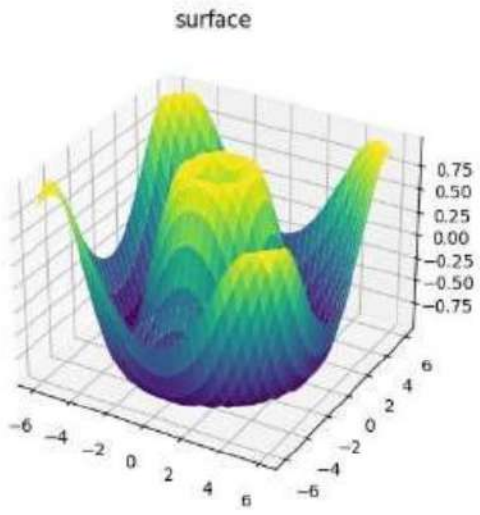


A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized:

In [8]:

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface');
```

OUTPUT



Surface Triangulations

For some applications, the evenly sampled grids required by the above routines is overly restrictive and inconvenient. In these situations, the triangulation-based plots can be very useful. What if rather than an even draw from a Cartesian or a polar grid, we instead have a set of random draws?

In [9]:

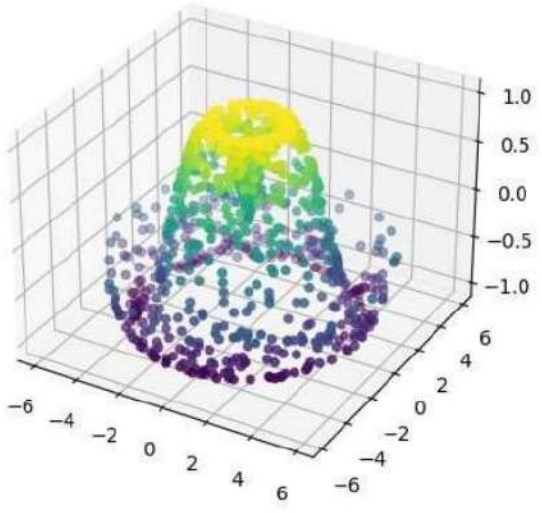
```
theta = 2 * np.pi * np.random.random(1000)
r = 6 * np.random.random(1000)
x = np.ravel(r * np.sin(theta))
y = np.ravel(r * np.cos(theta))
z = f(x, y)
```

We could create a scatter plot of the points to get an idea of the surface we're sampling from:

In [10]:

```
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
```

OUTPUT

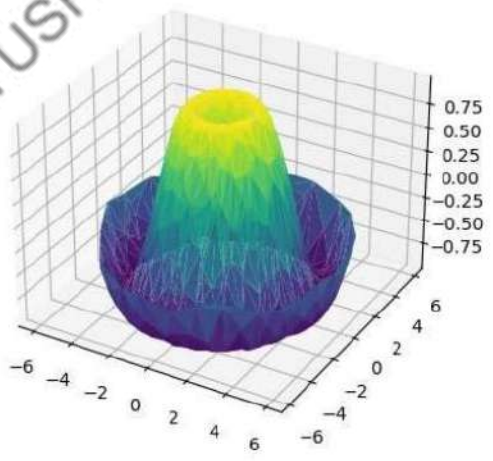


This leaves a lot to be desired. The function that will help us in this case is **ax.plot_trisurf,** which creates a surface by first finding a set of triangles formed between adjacent points (remember that x, y, and z here are one-dimensional arrays):

In [11]:

```
ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z,
                cmap='viridis', edgecolor='none');
```

OUTPUT



Result:

Thus the three dimensional plotting on UCI Datasets are executed.

Ex.No.7	VISUALIZING GEOGRAPHIC DATA WITH BASEMAP

Aim:

To visualizing geographic data with basemap

Basemap

Basemap is a great tool for creating maps using python in a simple way. It's a matplotlib extension, so it has got all its features to create data visualizations, and adds the geographical projections and some datasets to be able to plot coast lines, countries, and so on directly from the library.

- **Physical boundaries and bodies of water**
    - drawcoastlines(): Draw continental coast lines
    - drawlsmask(): Draw a mask between the land and sea, for use with projecting images on one or the other
    - drawmapboundary(): Draw the map boundary, including the fill color for oceans.
    - drawrivers(): Draw rivers on the map
    - fillcontinents(): Fill the continents with a given color; optionally fill lakes with another color
- **Political boundaries**
    - drawcountries(): Draw country boundaries
    - drawstates(): Draw US state boundaries
    - drawcounties(): Draw US county boundaries
- **Map features**
    - drawgreatcircle(): Draw a great circle between two points
    - drawparallels(): Draw lines of constant latitude
    - drawmeridians(): Draw lines of constant longitude
    - drawmapscale(): Draw a linear scale on the map
- **Whole-globe images**

- bluemarble(): Project NASA's blue marble image onto the map
- shadedrelief(): Project a shaded relief image onto the map
- etopo(): Draw an etopo relief image onto the map
- warpimage(): Project a user-provided image onto the map

Installation

Step 1: Use the Anaconda Navigator to install basemap. Go to start and click Anaconda command prompt.

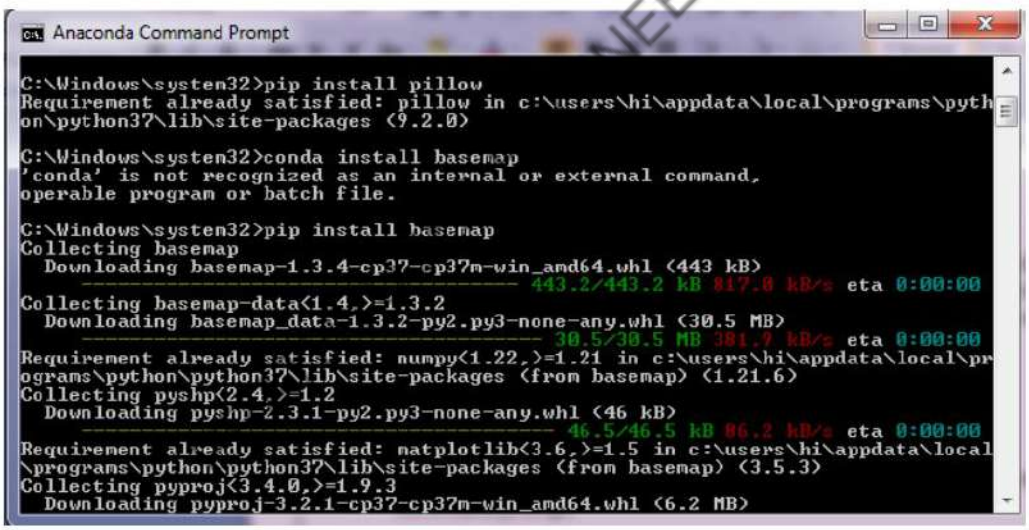
Step 2: Before installing Basemap, be sure to install pillow package. Install the pillow package using the command line

pip install pillow

Step 3: Next step is to install the Basemap using the following command

pip install basemap

The anaconda command prompt will look like



Step 4: After successfully installing basmape package navigate to jupyter notebook using the following command

jupyter notebook

Step 4: The above cmd will open a new webpage with address http://localhost:8888/tree.

Step 5: Click **New→Python 3 (ipykernal)**.

Step 6: Start the program for visualizing geographical data using basemap. Click run to run the pogram.

Some of these map-specific methods are:

- contour()/contourf() : Draw contour lines or filled contours
- imshow(): Draw an image
- pcolor()/pcolormesh() : Draw a pseudocolor plot for irregular/regular meshes
- plot(): Draw lines and/or markers.
- scatter(): Draw points with markers.
- quiver(): Draw vectors.
- barbs(): Draw wind barbs.
- drawgreatcircle(): Draw a great circle.

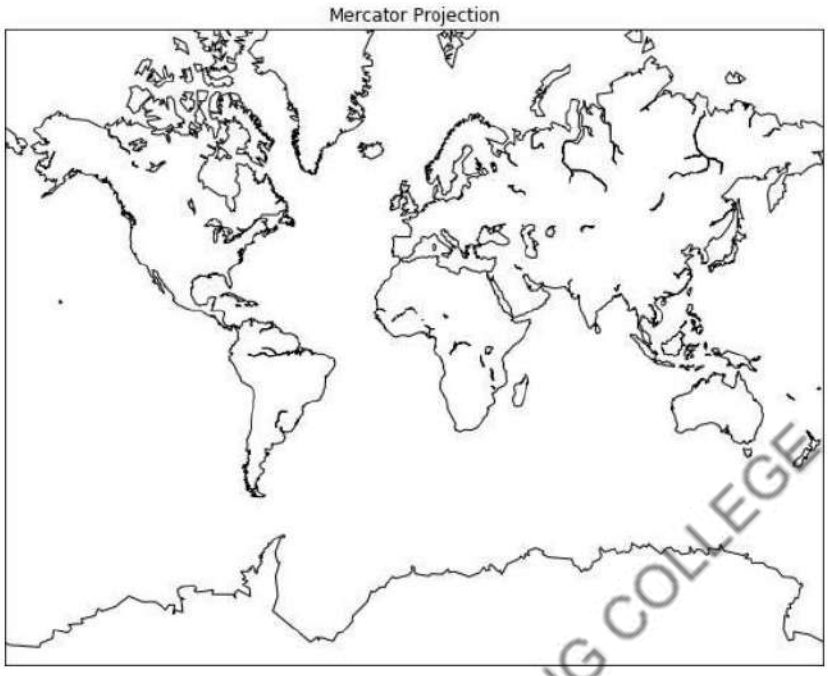
PROGRAM

1. **Simple Maps and color it**

To start, import Basemap as well as matplotlib and numpy:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
import warnings
import matplotlib.cbook
warnings.filterwarnings("ignore",category=matplotlib.cbook.mplDeprecation)
Basemap?
fig = plt.figure(num=None, figsize=(12, 8) )
m = Basemap(projection='merc',llcrnrlat=-80,urcrnrlat=80,llcrnrlon=-
180,urcrnrlon=180,resolution='c')
m.drawcoastlines()
plt.title("Mercator Projection")
plt.show()
```

OUTPUT:

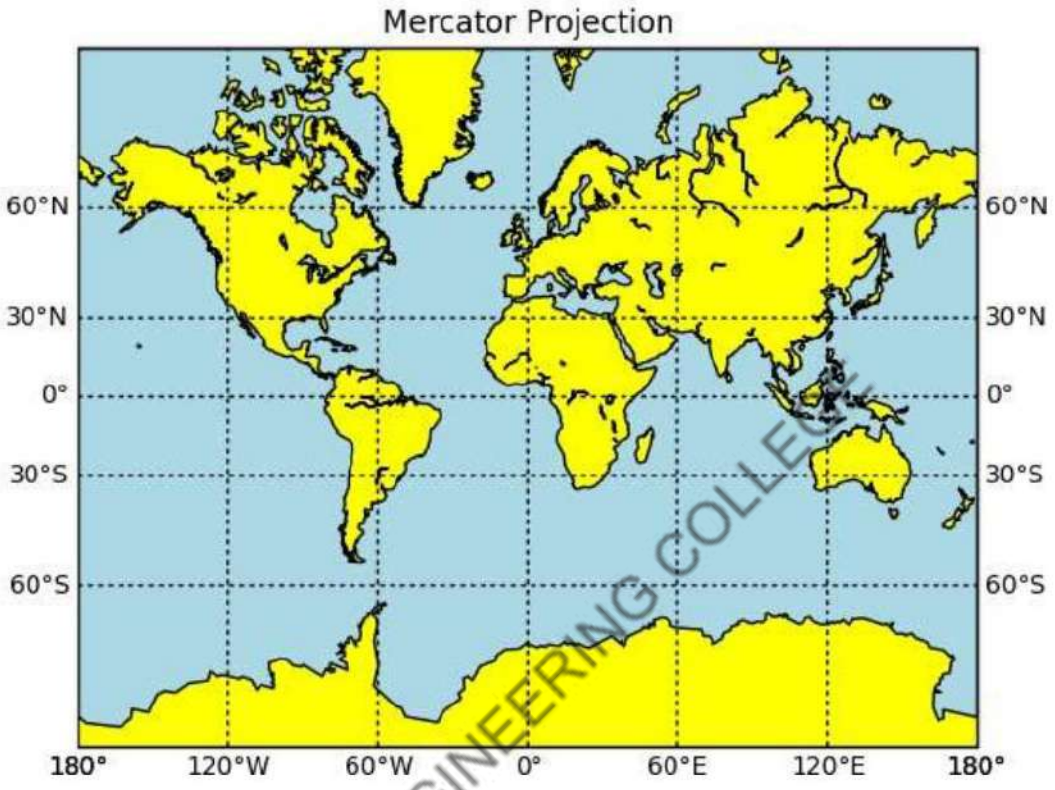


1a. Coding for Coloring

```
fig = plt.figure(num=None, figsize=(12, 8) )
m = Basemap(projection='merc',llcrnrlat=-80,urcrnrlat=80,llcrnrlon=-
180,urcrnrlon=180,resolution='c')
m.drawcoastlines()
m.fillcontinents(color='tan',lake_color='lightblue')
# draw parallels and meridians.
m.drawparallels(np.arange(-90.,91.,30.),labels=[True,True,False,False],dashes=[2,2])
m.drawmeridians(np.arange(-180.,181.,60.),labels=[False,False,False,True],dashes=[2,2])
m.drawmapboundary(fill_color='lightblue')
plt.title("Mercator Projection")
```

Output

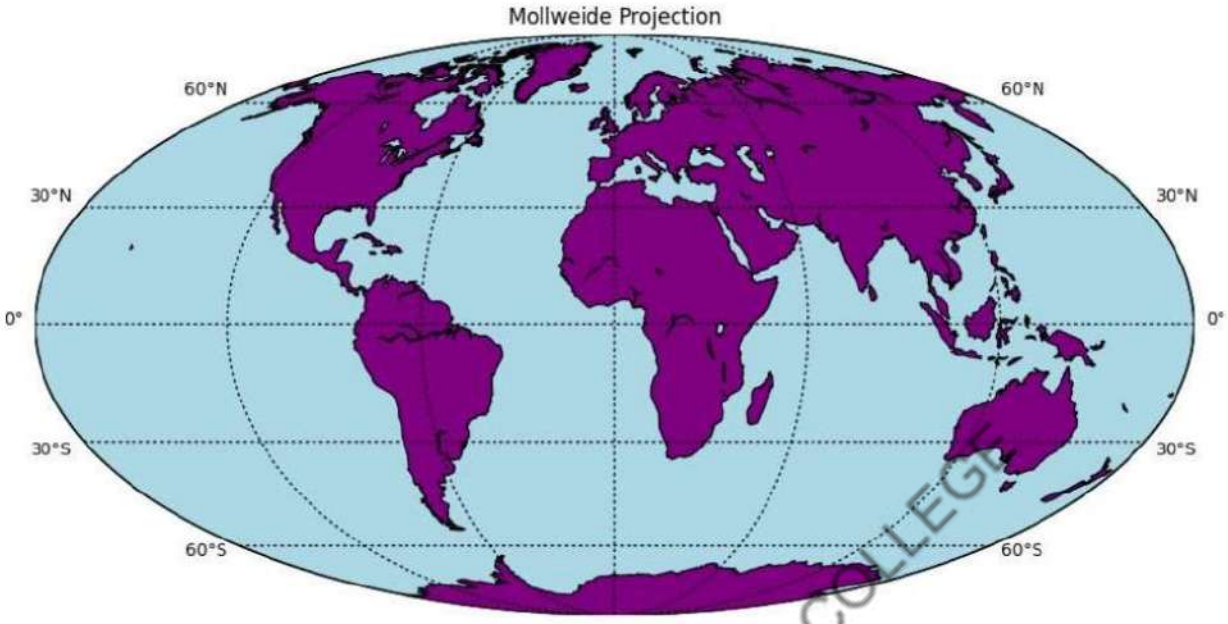
```
Out[5]: Text(0.5, 1.0, 'Mercator Projection')
```



1b. Same sequence of commands but with a different projection:

```
fig = plt.figure(num=None, figsize=(12, 8) )
m = Basemap(projection='moll',lon_0=0,resolution='c')
m.drawcoastlines()
m.fillcontinents(color='purple',lake_color='lightblue')
# draw parallels and meridians.
m.drawparallels(np.arange(-90.,91.,30.),labels=[True,True,False,False],dashes=[2,2])
m.drawmeridians(np.arange(-180.,181.,60.),labels=[False,False,False,False],dashes=[2,2])
m.drawmapboundary(fill_color='lightblue')
plt.title("Mollweide Projection");
```

Output



2 a. Create a map centered on North America with lines showing the country and state boundaries as well as rivers:

```
fig = plt.figure(num=None, figsize=(12, 8) )
m =
Basemap(width=6000000,height=4500000,resolution='c',projection='aea',lat_1=35.,lat_2=45,lon
_0=-100,lat_0=40)
m.drawcoastlines(linewidth=0.5)
m.fillcontinents(color='tan',lake_color='lightblue')
# draw parallels and meridians.
m.drawparallels(np.arange(-90.,91.,15.),labels=[True,True,False,False],dashes=[2,2])
m.drawmeridians(np.arange(-180.,181.,15.),labels=[False,False,False,True],dashes=[2,2])
m.drawmapboundary(fill_color='lightblue')
m.drawcountries(linewidth=2, linestyle='solid', color='k' )
m.drawstates(linewidth=0.5, linestyle='solid', color='k')
m.drawrivers(linewidth=0.5, linestyle='solid', color='blue')
```

Out[6]:

2b. Use a different map projection, zoom-in to North America and plot the location of Seattle

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
            width=8E6, height=8E6,
            lat_0=45, lon_0=-100,)
m.etopo(scale=0.5, alpha=0.5)

# Map (long, lat) to (x, y) for plotting
x, y = m(-122.3, 47.6)
plt.plot(x, y, 'ok', markersize=5)
plt.text(x, y, ' Seattle', fontsize=12);
```

Output

2. Map Projections

The Basemap package implements several dozen such projections, all referenced by a short format code. Here we'll briefly demonstrate some of the more common ones.

We'll start by defining a convenience routine to draw our world map along with the longitude and latitude lines:

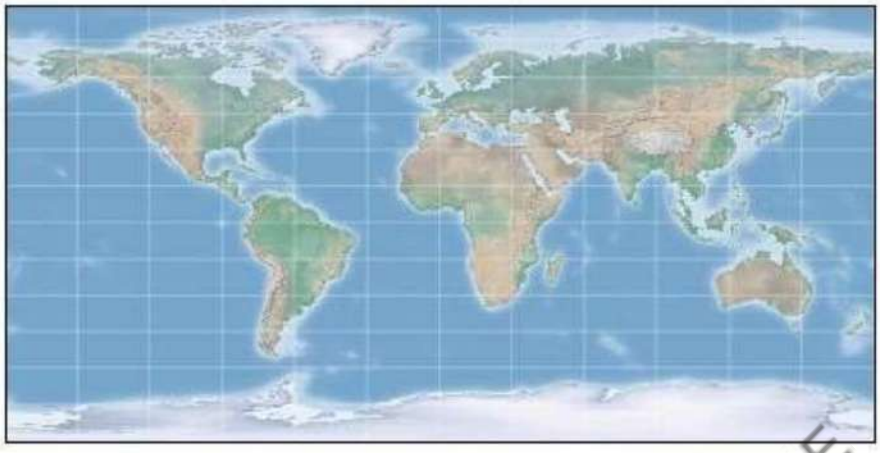
```
from itertools import chain

def draw_map(m, scale=0.2):
    # draw a shaded-relief image
    m.shadedrelief(scale=scale)

    # lats and longs are returned as a dictionary
    lats = m.drawparallels(np.linspace(-90, 90, 13))
    lons = m.drawmeridians(np.linspace(-180, 180, 13))

    # keys contain the plt.Line2D instances
    lat_lines = chain(*(tup[1][0] for tup in lats.items()))
    lon_lines = chain(*(tup[1][0] for tup in lons.items()))
    all_lines = chain(lat_lines, lon_lines)

    # cycle through these lines and set the desired style
    for line in all_lines:
        line.set(linestyle='-', alpha=0.3, color='w')
```

Cylindrical projections

The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitude are mapped to horizontal and vertical lines, respectively. This type of mapping represents equatorial regions quite well, but results in extreme distortions near the poles. The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles. In the following figure we show an example of the *equidistant cylindrical projection*, which chooses a latitude scaling that preserves distances along meridians. Other cylindrical projections are the Mercator (projection='merc') and the cylindrical equal area (projection='cea') projections.

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
            llcrnrlat=-90, urcrnrlat=90,
            llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)
```

OUTPUT



orthographic projection or Perspective projections

Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lies within the Earth!). One common example is the orthographic projection (projection='ortho'), which shows one side of the globe as seen from a viewer at a very long distance. As such, it can show only half the globe at a time. Other perspective-based projections include the gnomonic projection (projection='gnom') and stereographic projection (projection='stere'). These are often the most useful for showing small portions of the map.

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
            lat_0=50, lon_0=0)
draw_map(m);
```

OUTPUT

Conic projections

A Conic projection projects the map onto a single cone, which is then unrolled. This can lead to very good local properties, but regions far from the focus point of the cone may become very distorted. One example of this is the Lambert Conformal Conic projection (`projection='lcc'`), which we saw earlier in the map of North America. It projects the map onto a cone arranged in such a way that two standard parallels (specified in Basemap by `lat_1` and `lat_2`) have well-represented distances, with scale decreasing between them and increasing outside of them. Other useful conic projections are the equidistant conic projection (`projection='eqdc'`) and the Albers equal-area projection (`projection='aea'`). Conic projections, like perspective projections, tend to be good choices for representing small to medium patches of the globe.

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
            lon_0=0, lat_0=50, lat_1=45, lat_2=55,
            width=1.6E7, height=1.2E7)
draw_map(m)
```

OUTPUT

Result:

Thus visualizing geographic data with basemap is implemented.