Charting in Collaboratory

A common use for notebooks is data visualization using charts. Collaboratory makes this easy with several charting tools available as Python imports.

Matplotlib

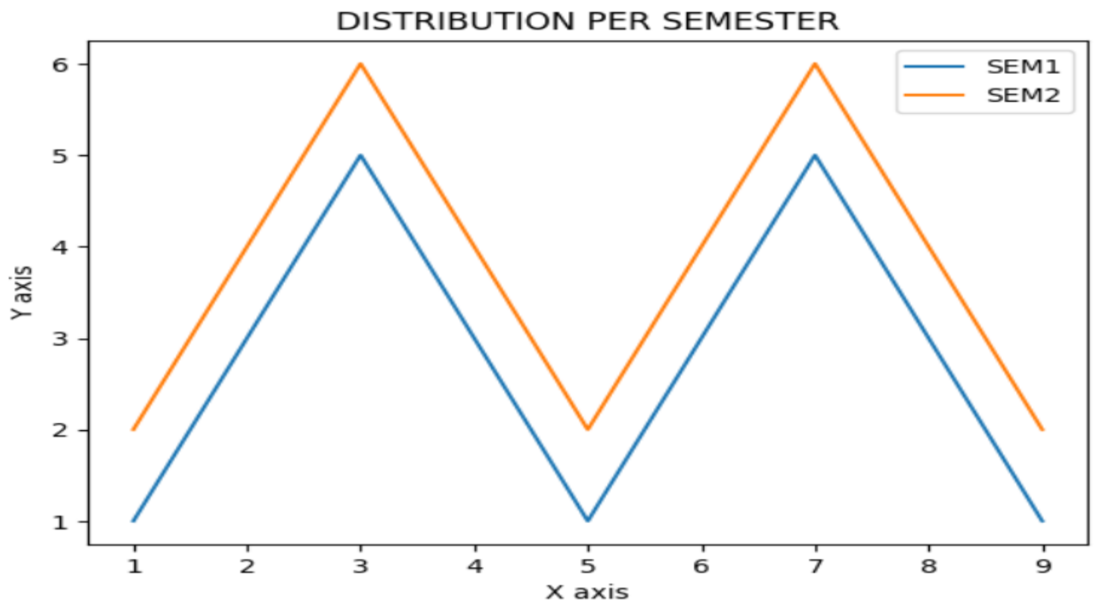
Matplotlib is the most common charting package.

---

1) **Line Plots**

EXAMPLE 1:

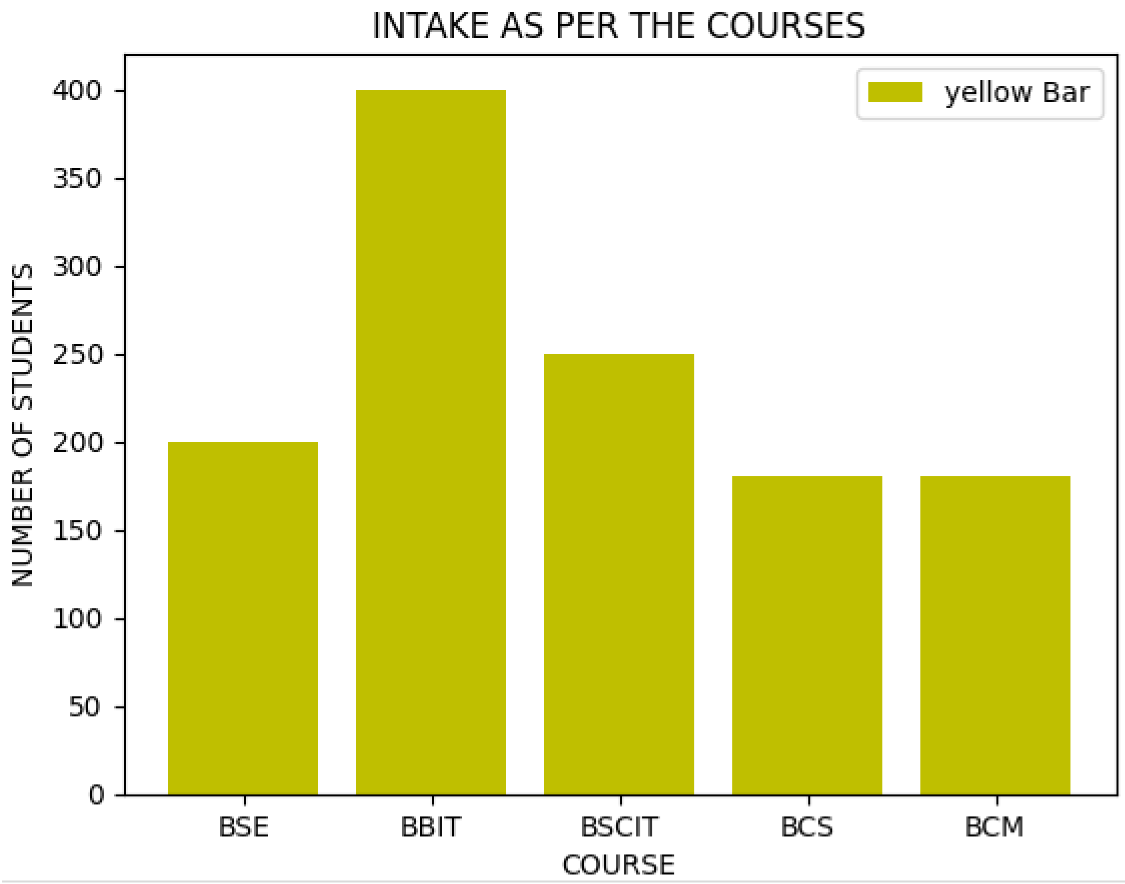
```python
import matplotlib.pyplot as plt #creating static, animated, and interactive visualizations

x  = [1, 2, 3, 4, 5, 6, 7, 8, 9]
y1 = [1, 3, 5, 3, 1, 3, 5, 3, 1]
y2 = [2, 4, 6, 4, 2, 4, 6, 4, 2]
plt.plot(x, y1, label="SEM1")
plt.plot(x, y2, label="SEM2")
plt.plot()

plt.xlabel("X axis")
plt.ylabel("Y axis")
plt.title("DISTRIBUTION PER SEMESTER")
plt.legend() #used to describe elements for a particular area of a graph
plt.show()
```



2) Bar Plots

EXAMPLE 1:

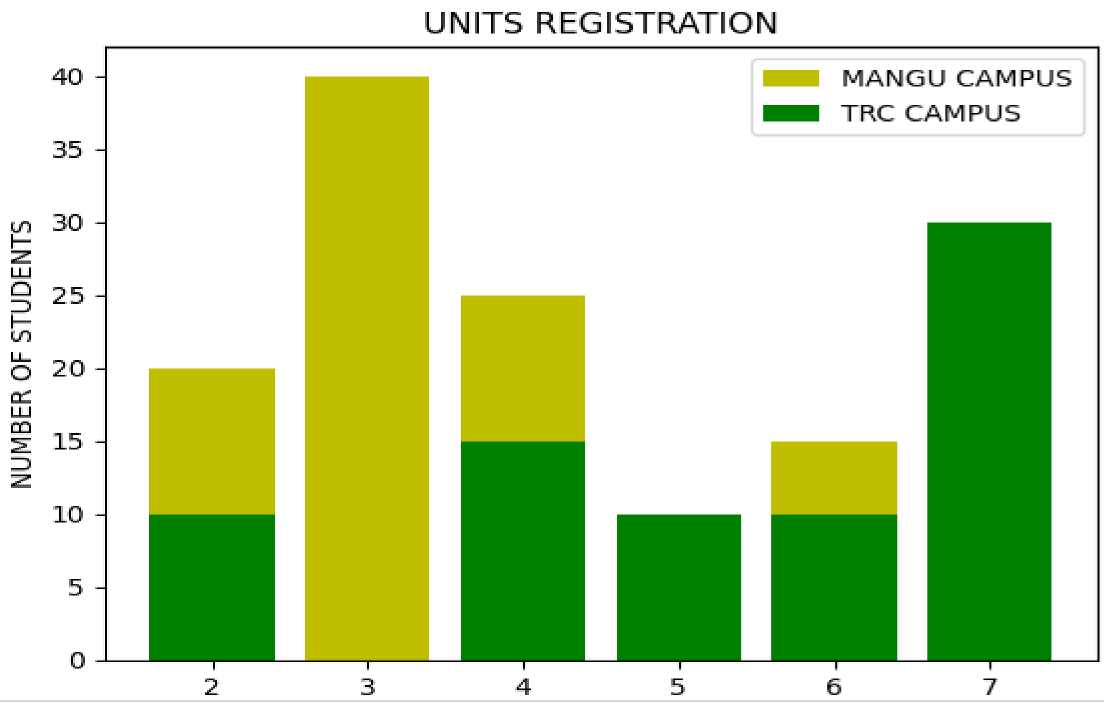
```
import matplotlib.pyplot as plt
x1 = ["BSE","BBIT", "BSCIT", "BCS", "BCM"]
y1 = [200, 400, 250, 180, 180]

plt.bar(x1, y1, label="yellow Bar", color='y')
plt.plot()
plt.xlabel("COURSE")
plt.ylabel("NUMBER OF STUDENTS")
plt.title("INTAKE AS PER THE COURSES")
plt.legend()
plt.show()
```



EXAMPLE 2:

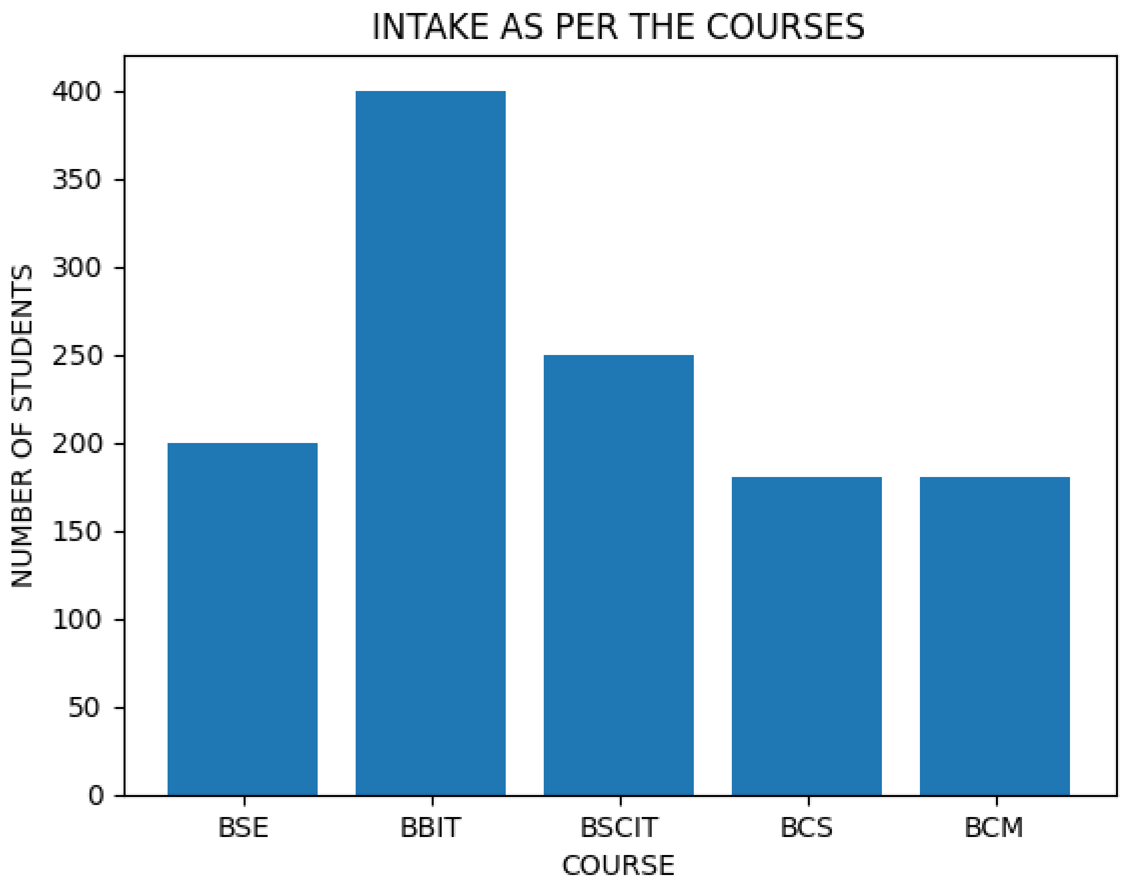
```
import matplotlib.pyplot as plt
x1 = [2,3,4,5,6]
y1 = [20, 40, 25, 10, 15]

x2 = [2, 4, 5, 6, 7]
y2 = [10, 15, 10, 10, 30]
```

```
plt.bar(x1, y1, label="MANGU CAMPUS", color='y')
plt.bar(x2, y2, label="TRC CAMPUS", color='g')
plt.plot()
plt.xlabel("UNITS REGISTERED")
plt.ylabel("NUMBER OF STUDENTS")
plt.title("UNITS REGISTRATION")
plt.legend()
plt.show()
```



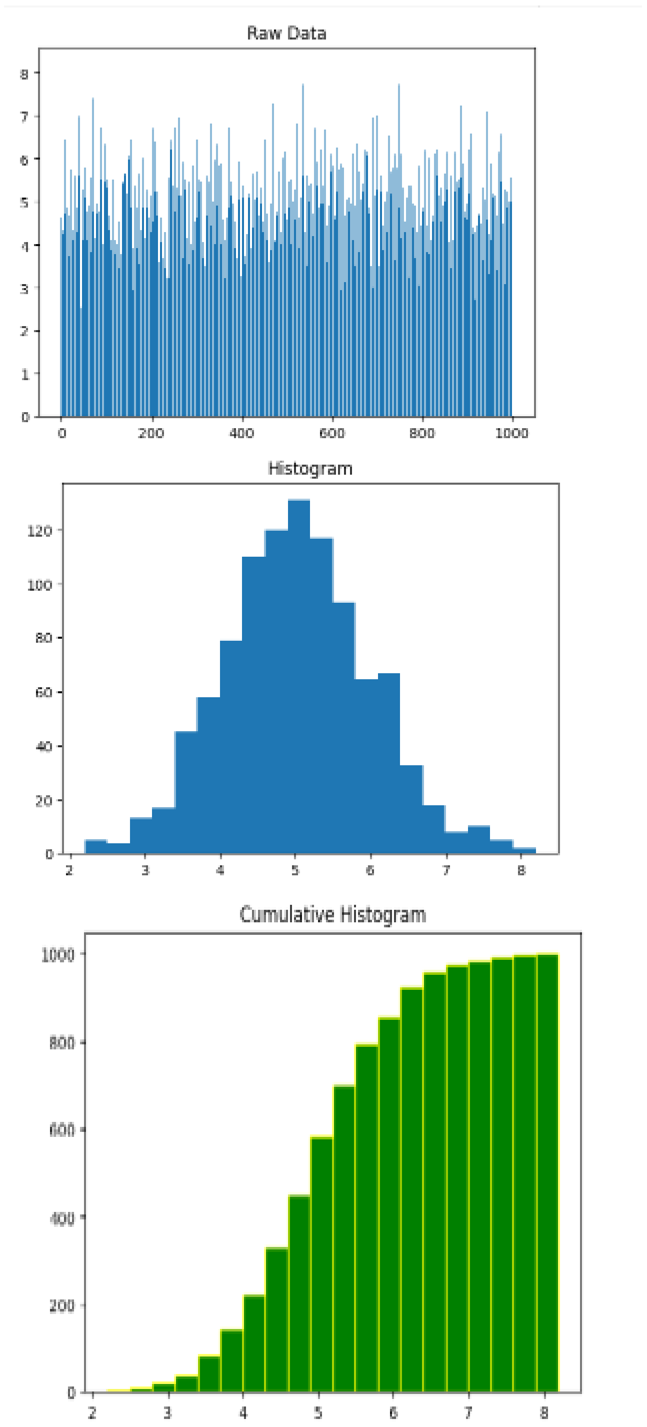
USING ARRAY FUNCTION IN BAR GRAPHS

```
# importing the necessary libraries and modules
import matplotlib.pyplot as plt
import numpy as np
# creating the data values for the vertical y and horisontal x axis
x = np.array(["BSE","BBIT", "BSCIT", "BCS", "BCM"])
y = np.array([200, 400, 250, 180, 180])
# using the pyplot.bar funtion
plt.bar(x,y)
plt.plot()
plt.xlabel("COURSE")
plt.ylabel("NUMBER OF STUDENTS")
plt.title("INTAKE AS PER THE COURSES")
# to show our graph
plt.show()
```



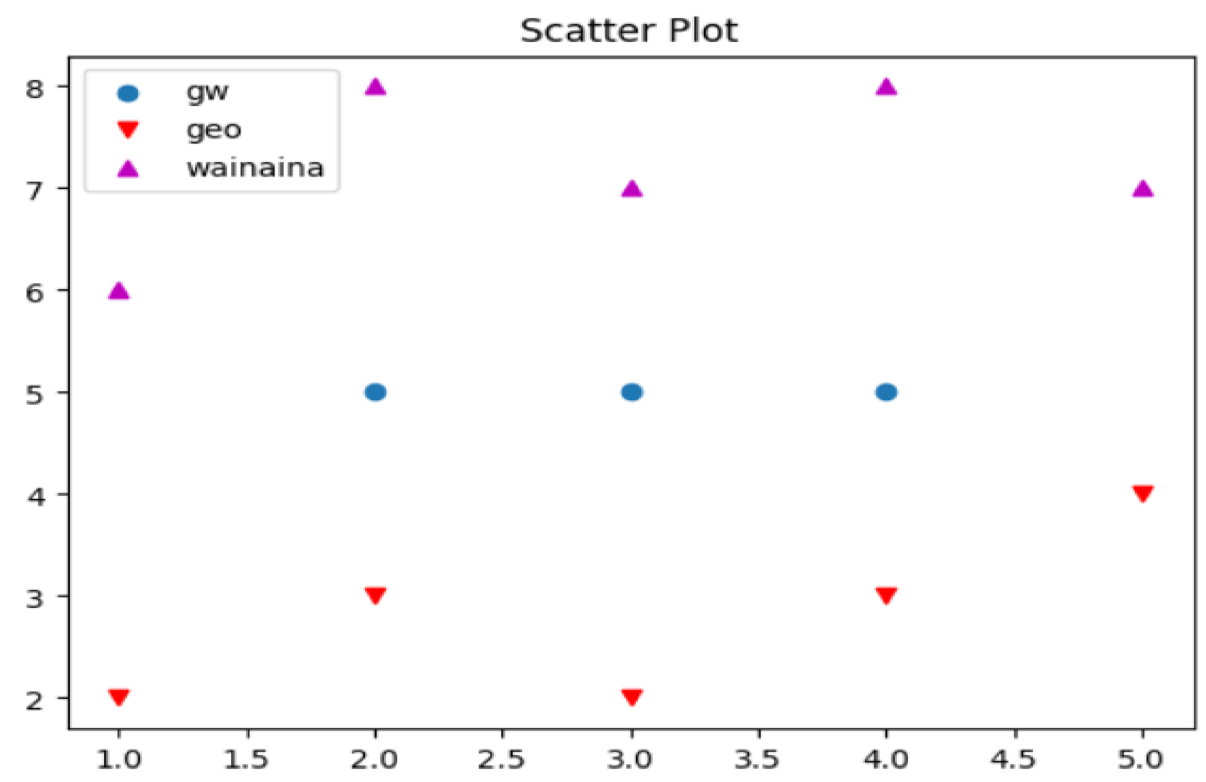
3) <u>Histograms</u>

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

# Use numpy to generate a bunch of random data in a bell curve around 5.
n = 5 + np.random.randn(1000) #is used to generate an array of random numbers from a standard
normal distribution (also known as a Gaussian distribution or a bell curve)

m = [m for m in range(len(n))]
plt.bar(m, n)
plt.title("Raw Data")
plt.show()

plt.hist(n, bins=20) #Bins are the number of intervals you want to divide all of your data into, such
that it can be displayed as bars on a histogram.
#width = (1000 – 0 ) / 20 = 200
plt.title("Histogram")
plt.show()
plt.hist(n, cumulative=True, bins=20, edgecolor="yellow", color="green") #When cumulative is set to True in these functions, it means that the operation will be performed cumulatively on the elements of an array or a sequence.
plt.title("Cumulative Histogram")
plt.show()
```



4) Scatter Plots

```
import matplotlib.pyplot as plt
x1 = [2, 3, 4]
y1 = [5, 5, 5]
x2 = [1, 2, 3, 4, 5]
y2 = [2, 3, 2, 3, 4]
y3 = [6, 8, 7, 8, 7]
plt.scatter(x1, y1, label="gw")
plt.scatter(x2, y2, label="geo", marker='v', color='r')
plt.scatter(x2, y3, label="wainaina", marker='^', color='m')
plt.title('Scatter Plot')
plt.legend()
plt.show()
```

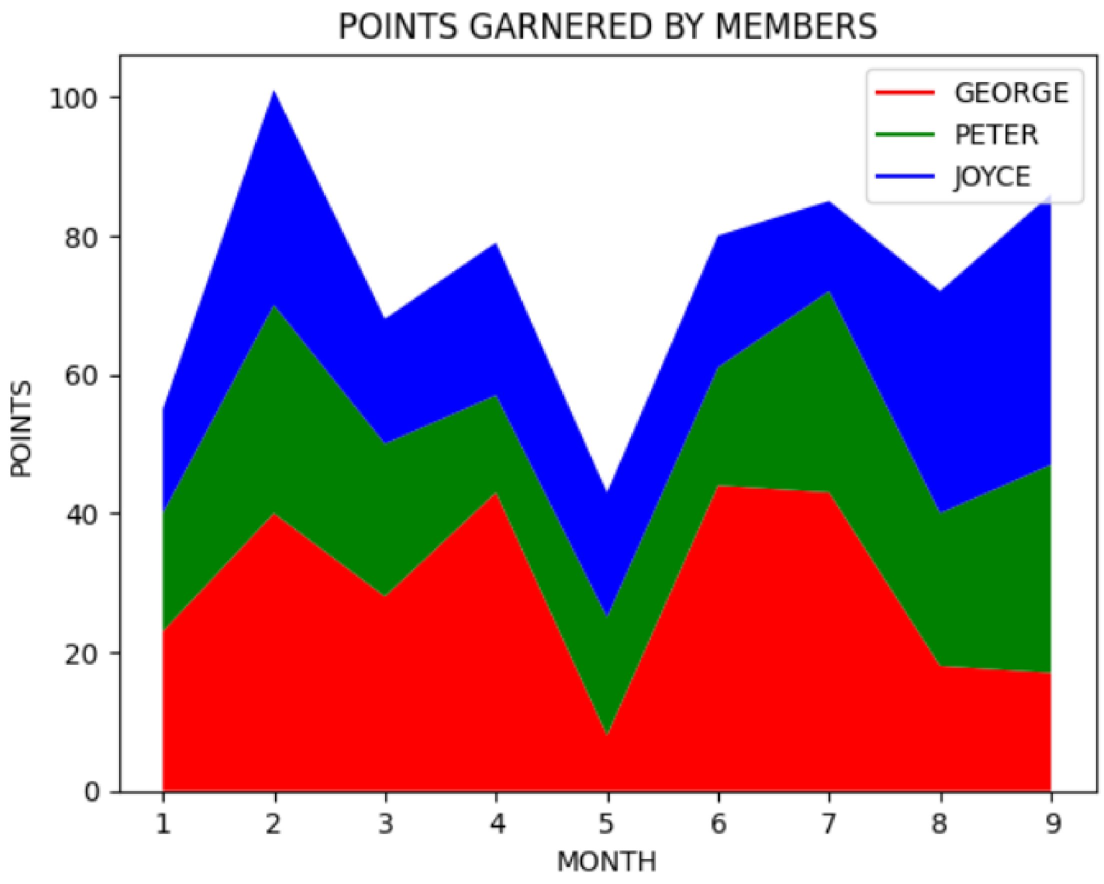


marker	symbol	description
"."	●	point
","	·	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up

marker	symbol	description
"<"	◀	triangle_left
">"	▶	triangle_right
"1"	Y	tri_down
"2"	⅄	tri_up
"3"	≺	tri_left
"4"	≻	tri_right
"8"	●	octagon
"s"	■	square
"p"	⬟	pentagon
"P"	✚	plus (filled)
"*"	★	star
"h"	⬢	hexagon1
"H"	⬣	hexagon2
"+"	+	plus
"x"	×	x
"X"	✖	x (filled)
"D"	◆	diamond
"d"	♦	thin_diamond
"|"	|	vline
"_"	—	hline
0 (TICKLEFT)	−	tickleft
1 (TICKRIGHT)	−	tickright
2 (TICKUP)	|	tickup
3 (TICKDOWN)	|	tickdown
4 (CARETLEFT)	◀	caretleft
5 (CARETRIGHT)	▶	caretright
6 (CARETUP)	▲	caretup
7 (CARETDOWN)	▼	caretdown
8 (CARETLEFTBASE)	◀	caretleft (centered at base)
9 (CARETRIGHTBASE)	▶	caretright (centered at base)
10 (CARETUPBASE)	▲	caretup (centered at base)
11 (CARETDOWNBASE)	▼	caretdown (centered at base)
"none" or "None"		nothing
" " or ""		nothing

5) Stack Plots

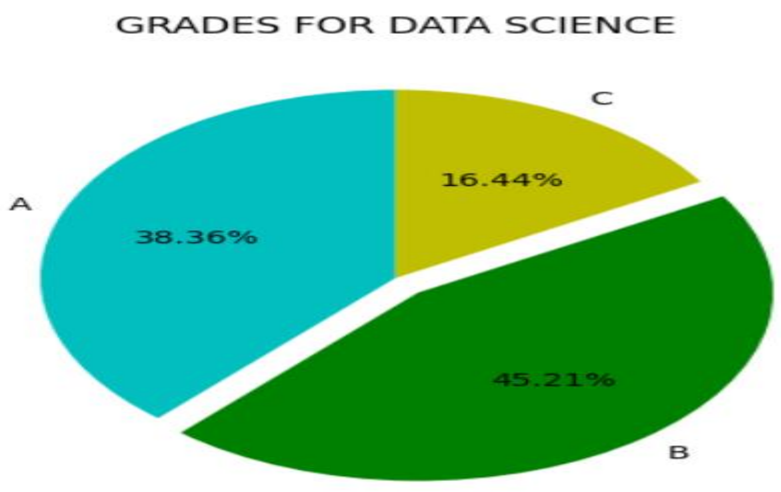
```
import matplotlib.pyplot as plt

MONTHS = [ 1,  2,  3,  4,  5,  6,  7,  8,  9]
GEORGE  = [23, 40, 28, 43,  8, 44, 43, 18, 17]
PETER  = [17, 30, 22, 14, 17, 17, 29, 22, 30]
JOYCE  = [15, 31, 18, 22, 18, 19, 13, 32, 39]

# Adding legend for stack plots is tricky.
plt.plot([], [], color='r', label = 'GEORGE')
plt.plot([], [], color='g', label = 'PETER')
plt.plot([], [], color='b', label = 'JOYCE')

plt.stackplot(MONTHS, GEORGE, PETER, JOYCE, colors= ['r', 'g', 'b'])
plt.xlabel("MONTH")
plt.ylabel("POINTS")
plt.title('POINTS GARNERED BY MEMBERS')
plt.legend()
plt.show()
```



6) Pie Charts

```
import matplotlib.pyplot as plt

labels = 'A', 'B', 'C'
sections = [56, 66, 24]
colors = ['c', 'g', 'y']
plt.pie(sections, labels=labels, colors=colors,startangle=90,explode = (0, 0.1, 0),autopct =
'%1.2f%%')
plt.title('GRADES FOR DATA SCIENCE')
plt.show()
```

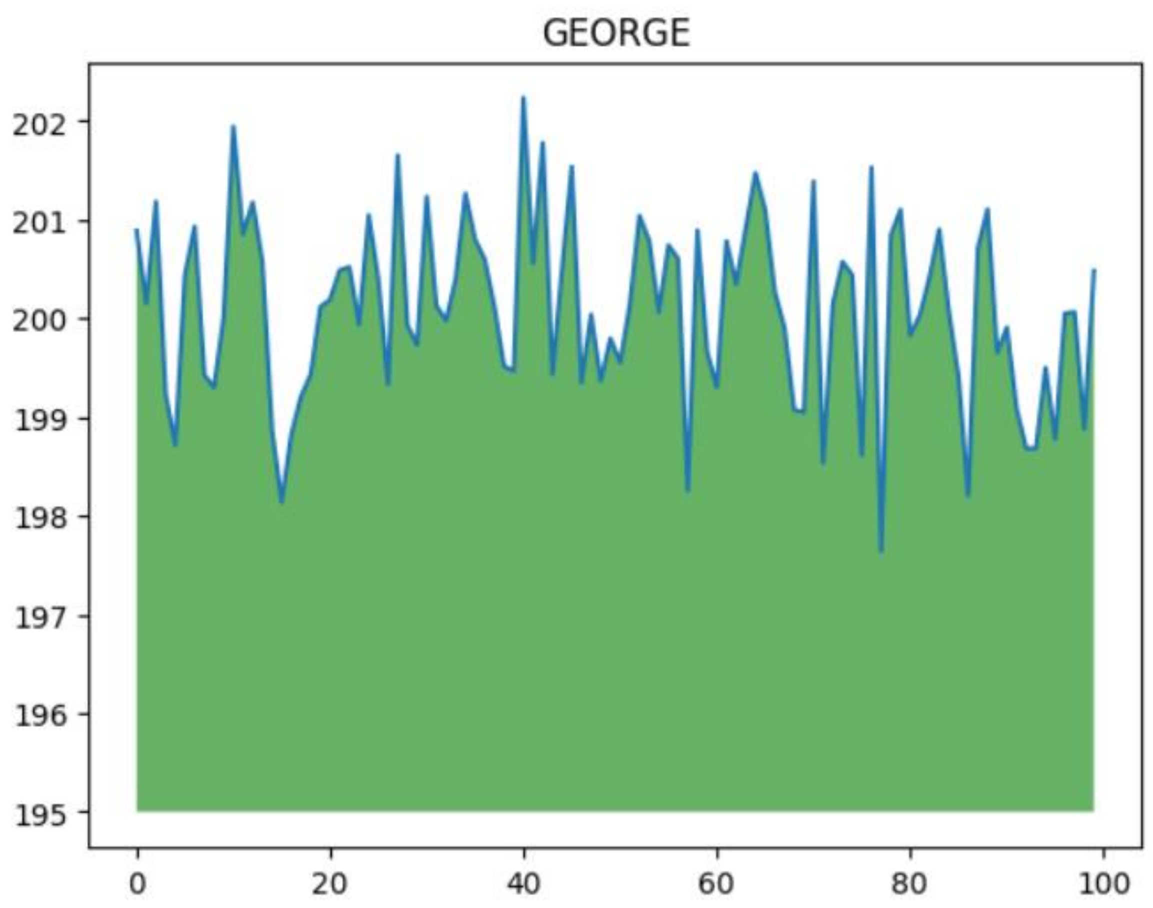
1. **plt.pie**: This is a function call to create a pie chart using Matplotlib. **plt** is typically an alias for **matplotlib.pyplot**, which is commonly used to create plots and charts in Python.
2. **sections**: This is a list or array containing the data values that will be used to create the pie chart. Each value in this list represents a portion or a slice of the pie chart.
3. **labels**: This is a list of labels that correspond to the sections in the **sections** list. It provides text labels for each slice of the pie chart. The labels will be displayed adjacent to their respective slices.
4. **colors**: This is a list of colors that correspond to the sections in the **sections** list. It specifies the colors of each slice in the pie chart.
5. **startangle=90**: This parameter sets the angle at which the first slice of the pie chart starts. In this case, it's set to 90 degrees, which means that the first slice will start from the top of the chart and proceed in a clockwise direction.
6. **explode=(0, 0.1, 0)**: This parameter is used to "explode" or separate one or more slices from the pie chart. It is a tuple where each value corresponds to a slice in the **sections** list. A non-zero value will push the respective slice away from the center of the pie chart. In this example, the second slice (index 1) is exploded by 10% of the radius of the pie chart.
7. **autopct='%1.2f%%'**: This parameter specifies how to format the autopct (automatic percentage) labels that are displayed on each slice of the pie chart. The **'1.2f%%'** format string means that percentages will be displayed with two decimal places.



7) fill_between and alpha.

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

ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)

plt.title("GEORGE")
plt.show()
```

1. Import necessary libraries:
   - Import the Matplotlib library and alias it as **plt**.
   - Import the NumPy library and alias it as **np**.
2. Generate random data:
   - Create an array **ys** containing 100 random values.
   - These values are generated using NumPy's **randn** function, which produces random numbers following a normal distribution (Gaussian) with a mean of 200 and a standard deviation of 1.
   - This simulates a dataset of 100 data points with random noise around a mean value of 200.
3. Generate x-coordinates:
   - Create a list **x** that represents the x-coordinates for the data points.
   - It generates a list of numbers from 0 to 99 (the length of **ys**) using a list comprehension.
4. Create a line plot:
   - Use Matplotlib's **plot** function to create a line plot.
   - The **x** values are used as the x-coordinates, and **ys** provides the y-coordinates for the data points.
   - The **'-'** argument specifies that a simple line should connect the data points.
5. Fill the area between the line and a horizontal line:
   - Use Matplotlib's **fill_between** function to add shaded regions to the plot.
   - It fills the area between the **x** values, the **ys** values, and a horizontal line at y = 195.
   - The **where** parameter specifies that the fill should occur only where **ys** is greater than 195.
   - The **facecolor** parameter sets the color of the filled region to green ('g').
   - The **alpha** parameter sets the transparency level of the fill to 0.6, making it somewhat transparent.
6. Set the plot title:
   - Use **plt.title** to set the title of the plot to "GEORGE"
7. Display the plot:
   - Use **plt.show()** to display the plot on the screen.
   - Without this line, the plot would be created but not shown to the user.



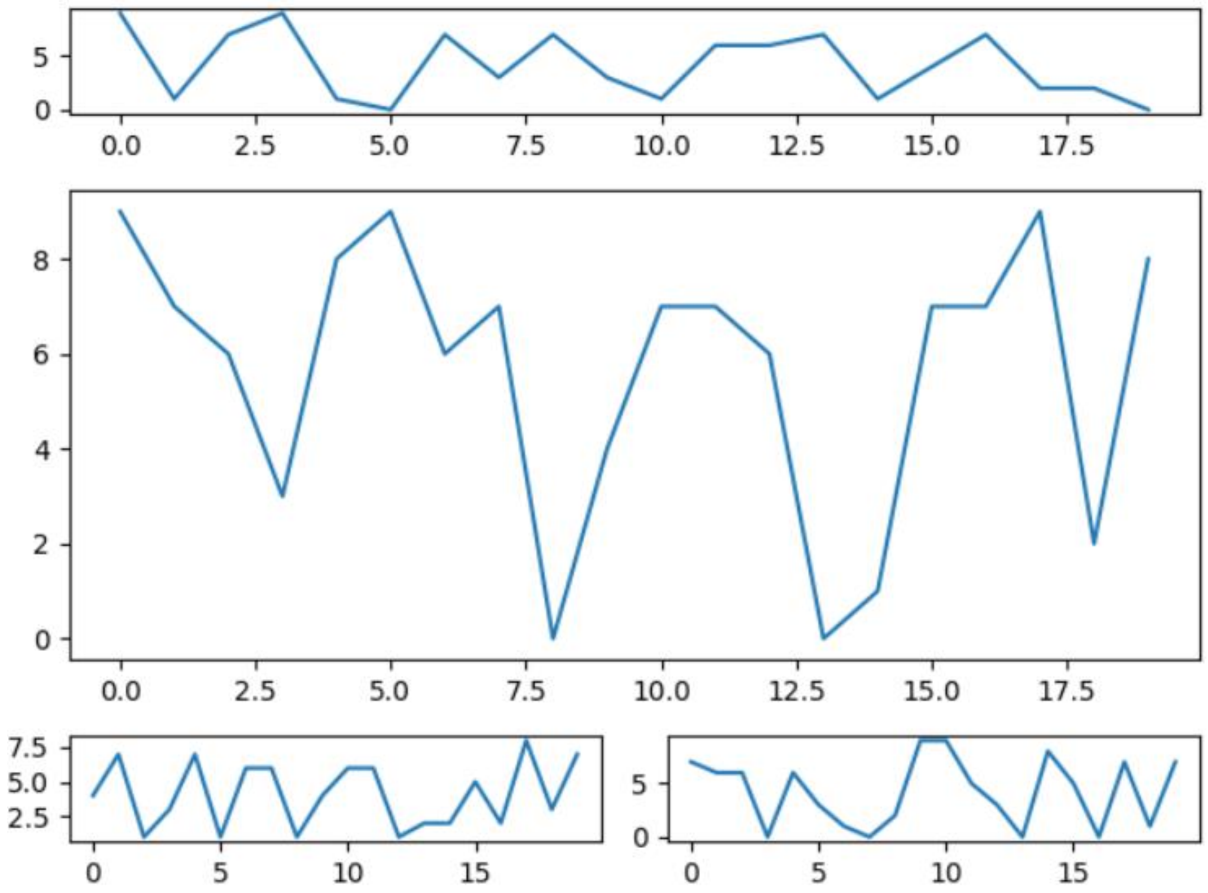
8) **Subplotting using Subplot2grid**

A code to creates a figure with four subplots arranged in a grid. Each subplot displays a line plot of random data generated by the random_plots() function. The plt.tight_layout() function is used to improve the subplot layout, and plt.show() displays the figure with the subplots.

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

def random_plots():
  xs = []
  ys = []

  for i in range(20):
    x = i
    y = np.random.randint(10)

    xs.append(x)
    ys.append(y)

  return xs, ys
```

```
fig = plt.figure()
ax1 = plt.subplot2grid((5, 2), (0, 0), rowspan=1, colspan=2)
ax2 = plt.subplot2grid((5, 2), (1, 0), rowspan=3, colspan=2)
ax3 = plt.subplot2grid((5, 2), (4, 0), rowspan=1, colspan=1)
ax4 = plt.subplot2grid((5, 2), (4, 1), rowspan=1, colspan=1)

x, y = random_plots()
ax1.plot(x, y)

x, y = random_plots()
ax2.plot(x, y)

x, y = random_plots()
ax3.plot(x, y)
x, y = random_plots()
ax4.plot(x, y)

plt.tight_layout()
plt.show()
```

1. Import necessary libraries:
   - Import the Matplotlib library and alias it as **plt**.
   - Import the NumPy library and alias it as **np**.
2. Define a function to generate random data:
   - Create a function called **random_plots()** that generates random data points.
   - Initialize empty lists **xs** and **ys** to store x and y values.
   - Use a for loop to generate 20 data points.
   - For each data point, set **x** to the loop index **i** and generate a random integer **y** between 0 and 9.
   - Append **x** and **y** to the respective lists.
   - Return the lists **xs** and **ys** containing the generated data.
3. Create a figure and subplot grid:
   - Create a new Matplotlib figure using **plt.figure()**. This figure will contain multiple subplots.
   - Define four subplots within a 5x2 grid using **plt.subplot2grid()**. These subplots are assigned to variables **ax1**, **ax2**, **ax3**, and **ax4**.
   - The **(5, 2)** argument specifies the size of the grid, and the following arguments specify the position and size of each subplot within the grid.
4. Generate random data and plot in subplots:
   - Repeatedly call the **random_plots()** function to generate random data and assign it to **x** and **y**.
   - Use each subplot (**ax1**, **ax2**, **ax3**, and **ax4**) to create line plots of the generated data.
5. Adjust subplot layout:
   - Use **plt.tight_layout()** to automatically adjust the spacing and layout of the subplots to prevent overlapping and ensure a neat appearance.
6. Display the figure:
   - Use **plt.show()** to display the entire figure containing the subplots.



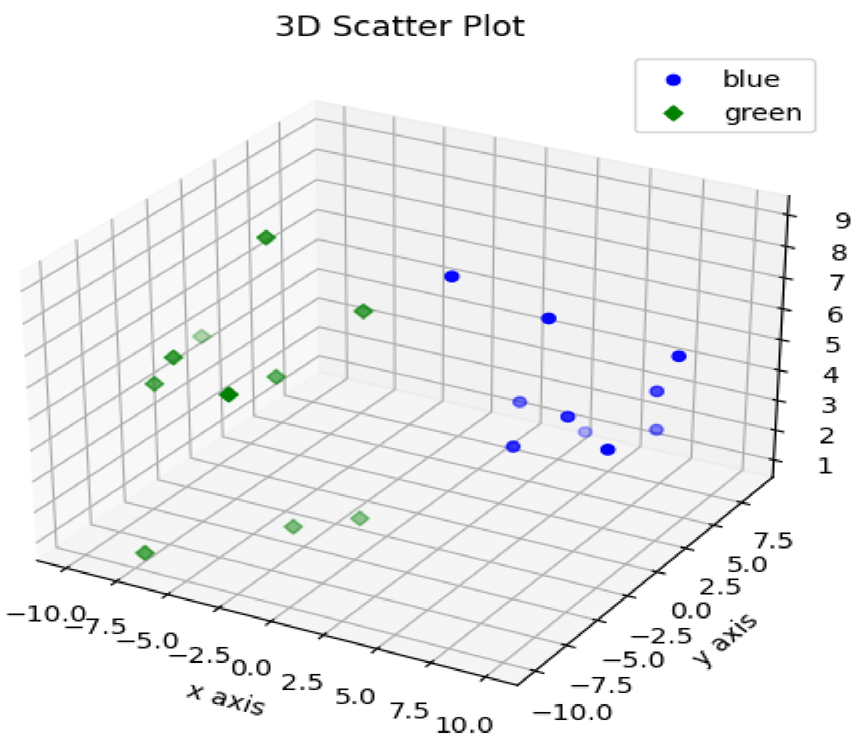
9) <u>3D Scatter Plots</u>

Code that creates a 3D scatter plot using Matplotlib. It generates random data points for two sets of points (blue and green) and visualizes them in a 3D space, with labels for the axes, a legend, and a title.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axes3d

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')

x1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y1 = np.random.randint(10, size=10)
z1 = np.random.randint(10, size=10)

x2 = [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
y2 = np.random.randint(-10, 0, size=10)
z2 = np.random.randint(10, size=10)
```

```
ax.scatter(x1, y1, z1, c='b', marker='o', label='blue')
ax.scatter(x2, y2, z2, c='g', marker='D', label='green')

ax.set_xlabel('x axis')
ax.set_ylabel('y axis')
ax.set_zlabel('z axis')
plt.title("3D Scatter Plot")
plt.legend()
plt.tight_layout()
plt.show()
```

1. Import necessary libraries:
   - Import the Matplotlib library and alias it as **plt**.
   - Import the NumPy library and alias it as **np**.
   - Import the **axes3d** module from **mpl_toolkits.mplot3d**, which is used to create 3D plots.
2. Create a 3D figure and subplot:
   - Create a new Matplotlib figure using **plt.figure()**.
   - Add a 3D subplot to the figure using **fig.add_subplot(111, projection='3d')**.
   - The **projection='3d'** argument specifies that this subplot should be a 3D plot.
3. Define data for two sets of points:
   - Define **x1**, **y1**, and **z1** as lists representing the x, y, and z coordinates for the first set of points. These points are in blue.
   - Generate random integers for **y1** and **z1** using NumPy. **y1** contains values between 0 and 9, and **z1** contains values between 0 and 9.
   - Define **x2**, **y2**, and **z2** for the second set of points. These points are in green.
   - Generate random integers for **y2** and **z2** using NumPy. **y2** contains values between -10 and -1, and **z2** contains values between 0 and 9.
4. Create scatter plots for the data:
   - Use **ax.scatter()** to create scatter plots for both sets of points in the 3D space.
   - **x1**, **y1**, and **z1** are used for the blue points, and they are marked with blue circles ('o').
   - **x2**, **y2**, and **z2** are used for the green points, and they are marked with green diamonds ('D').
5. Set labels for the axes:
   - Use **ax.set_xlabel()**, **ax.set_ylabel()**, and **ax.set_zlabel()** to label the x, y, and z axes, respectively.
6. Set the plot title and legend:
   - Use **plt.title()** to set the title of the 3D scatter plot to "3D Scatter Plot."
   - Use **plt.legend()** to add a legend to the plot, which shows labels for the blue and green points.
7. Adjust subplot layout:
   - Use **plt.tight_layout()** to automatically adjust the spacing and layout of the plot to prevent overlapping and ensure a neat appearance.
8. Display the 3D scatter plot:
   - Use **plt.show()** to display the figure containing the 3D scatter plot on the screen.

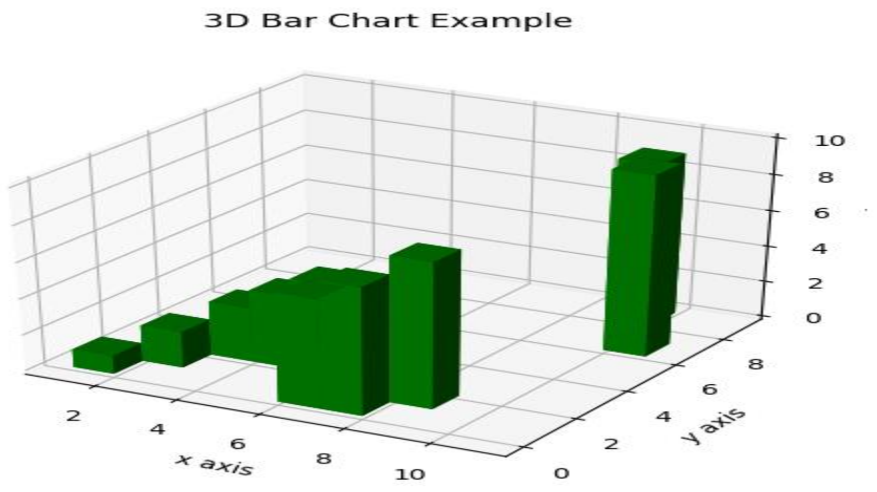


10) <u>3D Bar Plots</u>

Code to creates a 3D bar chart with bars of varying heights, represented by the dz values. The bars are positioned in 3D space with labels for the axes, a title, and a green color.

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

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = np.random.randint(10, size=10)
z = np.zeros(10)

dx = np.ones(10)
dy = np.ones(10)
dz = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ax.bar3d(x, y, z, dx, dy, dz, color='g')

ax.set_xlabel('x axis')
ax.set_ylabel('y axis')
ax.set_zlabel('z axis')
plt.title("3D Bar Chart Example")
plt.tight_layout()
plt.show()
```

1. Import necessary libraries:
   - Import the Matplotlib library and alias it as **plt**.
   - Import the NumPy library and alias it as **np**.
2. Create a 3D figure and subplot:
   - Create a new Matplotlib figure using **plt.figure()**.
   - Add a 3D subplot to the figure using **fig.add_subplot(111, projection='3d')**.
   - The **projection='3d'** argument specifies that this subplot should be a 3D plot.
3. Define data for the bar chart:
   - Define **x**, **y**, and **z** as lists representing the x, y, and z coordinates for the bars.
   - **x** contains values from 1 to 10.
   - **y** is generated with 10 random integers between 0 and 9 using NumPy.
   - **z** is initialized as a list of zeros.
4. Define dimensions for the bars:
   - Define **dx**, **dy**, and **dz** to represent the dimensions of the bars.
   - **dx** and **dy** are arrays with all elements equal to 1, representing the width and depth of the bars, respectively.
   - **dz** is a list containing values from 1 to 10, representing the height of each bar.
5. Create the 3D bar chart:
   - Use **ax.bar3d()** to create the 3D bar chart. This function takes the x, y, and z coordinates of the bars, as well as their dimensions (**dx**, **dy**, **dz**).
   - The bars are colored green using the **color** parameter.
6. Set labels for the axes:
   - Use **ax.set_xlabel()**, **ax.set_ylabel()**, and **ax.set_zlabel()** to label the x, y, and z axes, respectively.
7. Set the plot title:
   - Use **plt.title()** to set the title of the 3D bar chart to "3D Bar Chart Example."
8. Adjust subplot layout:
   - Use **plt.tight_layout()** to automatically adjust the spacing and layout of the plot to prevent overlapping and ensure a neat appearance.
9. Display the 3D bar chart:
   - Use **plt.show()** to display the figure containing the 3D bar chart on the screen.

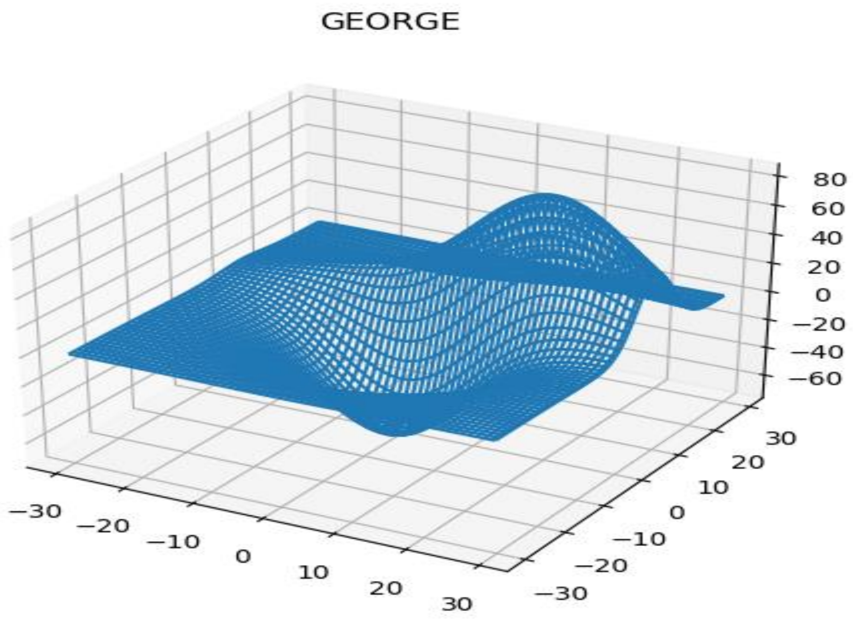


11) Wireframe Plots

A code that creates a 3D wireframe plot using test data obtained from axes3d.get_test_data(). The wireframe represents a surface using lines in a 3D space. It includes a title and ensures proper layout before displaying the plot.

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')

x, y, z = axes3d.get_test_data()

ax.plot_wireframe(x, y, z, rstride = 2, cstride = 2)

plt.title("Wireframe Plot Example")
plt.tight_layout()
plt.show()
```

1. Import necessary libraries:
    - Import the Matplotlib library and alias it as **plt**.
2. Create a 3D figure and subplot:
    - Create a new Matplotlib figure using **plt.figure()**.
    - Add a 3D subplot to the figure using **fig.add_subplot(111, projection='3d')**.
    - The **projection='3d'** argument specifies that this subplot should be a 3D plot.
3. Get test data:
    - Use **axes3d.get_test_data()** to obtain test data for a 3D plot.
    - This function provides sample data for demonstration purposes. It returns **x**, **y**, and **z** arrays representing 3D coordinates.
4. Create a wireframe plot:
    - Use **ax.plot_wireframe()** to create the wireframe plot.
    - The **x**, **y**, and **z** arrays obtained from **axes3d.get_test_data()** are used as input data.
    - The **rstride** (row stride) and **cstride** (column stride) parameters are set to 2, which controls the spacing between the lines in the wireframe. A value of 2 means that every second row and column is included in the wireframe.
5. Set the plot title:
    - Use **plt.title()** to set the title of the wireframe plot to "GEORGE."
6. Adjust subplot layout:
    - Use **plt.tight_layout()** to automatically adjust the spacing and layout of the plot to prevent overlapping and ensure a neat appearance.
7. Display the wireframe plot:
    - Use **plt.show()** to display the figure containing the wireframe plot on the screen.


GEORGE
80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
30
30
20
10
0
−10
−20
−30