Box plot,heapmap and 3D plot by Mrittika Megaraj

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Box plot

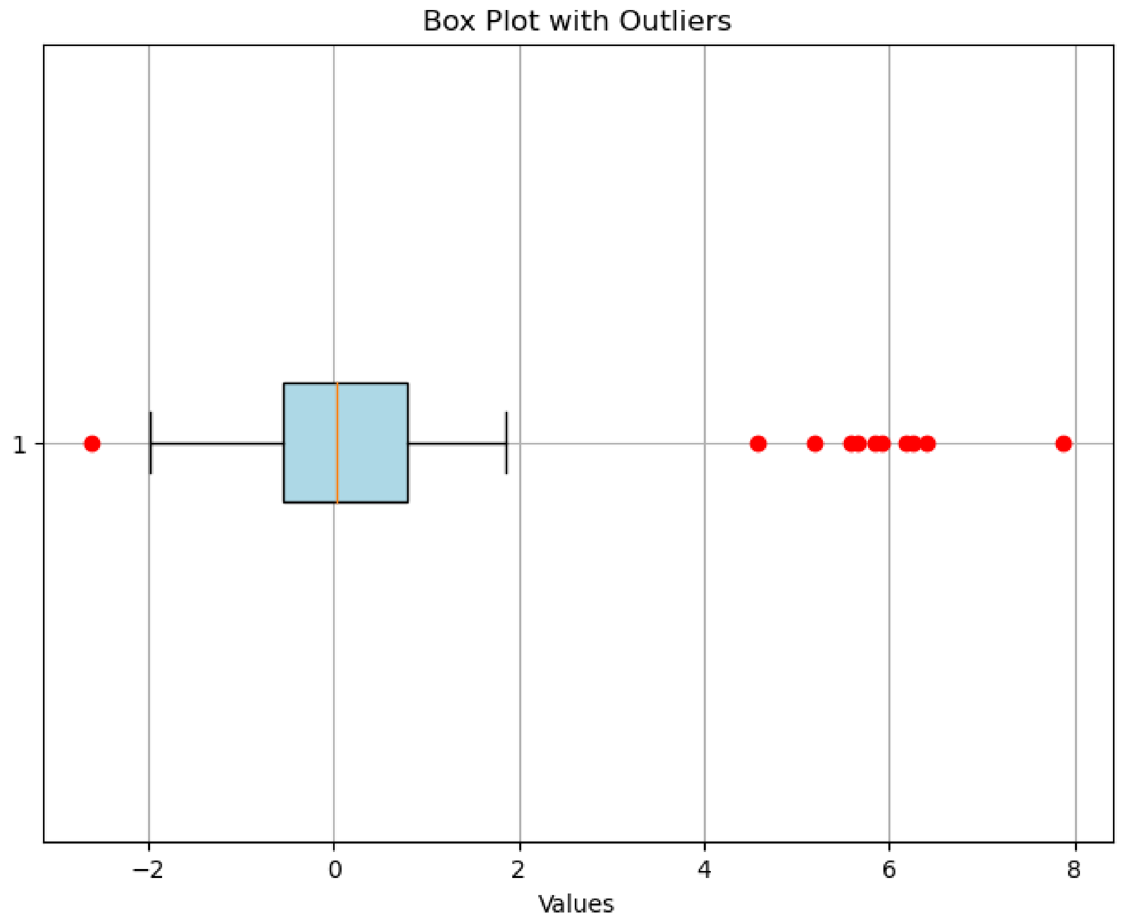
Box plot is the distribution of numerical data by displaying quartiles, outliers, and potential skewness. They provide insights into data spread, central tendency, and variability. Box plots are especially useful for identifying outliers and comparing distributions.

You can use plt.boxplot(data) to plot the box plot. You can customize the appearance of the box and outliers using boxprops and flierprops , use vert=False to make the box plot horizontal and patch_artist=True to fill the box with color.

Use Cases:

1. Analyzing the distribution of salaries in a company.
2. Assessing the variability of housing prices in different neighborhoods.

In [2]:
```
# Generate random data with outliers
np.random.seed(42)
data = np.concatenate([np.random.normal(0, 1, 100), np.random.normal(6, 1,

# Create a box plot with outliers
plt.figure(figsize=(8, 6))  # Set the figure size
plt.boxplot(data, vert=False, patch_artist=True, boxprops={'facecolor': 'li

# Add labels and a title
plt.xlabel('Values')
plt.title('Box Plot with Outliers')

# Display the plot
plt.grid(True)  # Add a grid for better readability
plt.show()
```

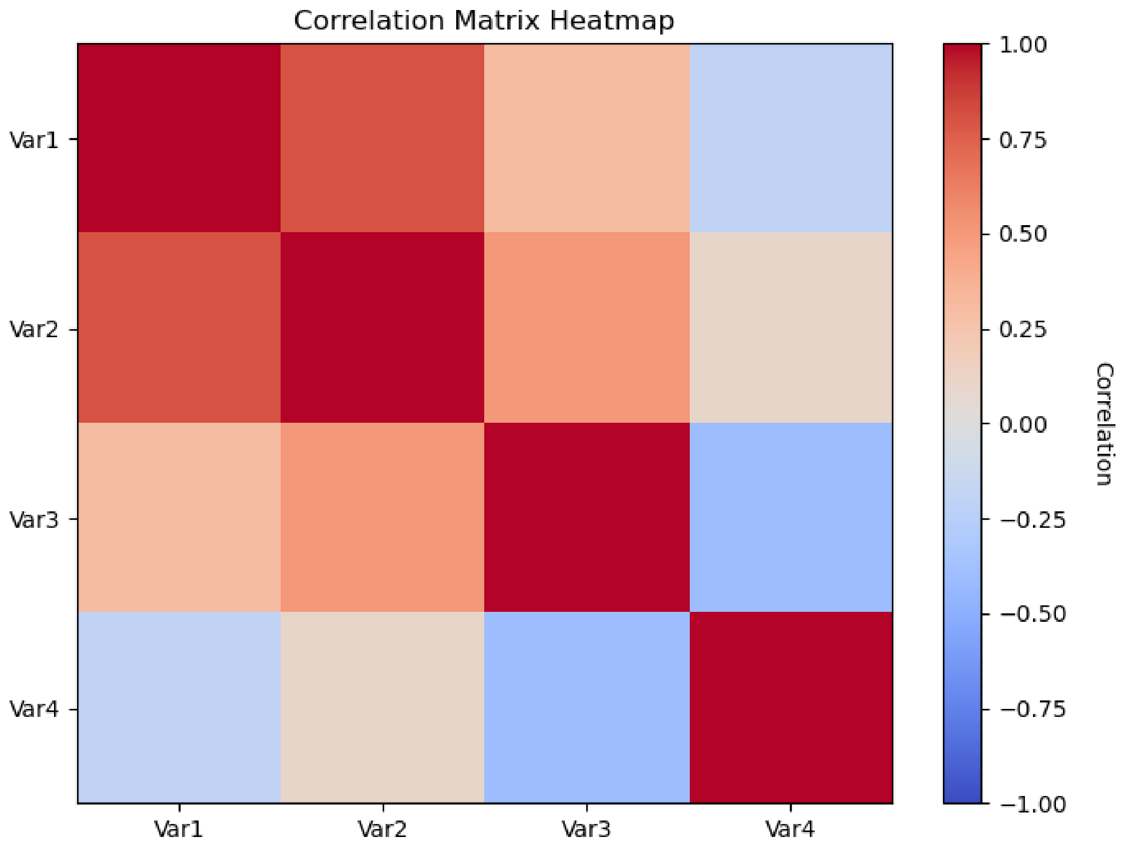


Displaying Images, Array Visualization

plt.imshow() is a Matplotlib function that is used for displaying 2D image data, visualizing 2D arrays, or showing images in various formats.

Using imshow for heatmap: Heatmap is a visualization for correlation matrix, which will give us a sense of how each variable is correlated with the other variable. Here, we’ll create a heatmap to visualize a correlation matrix, and we’ll use a color map to show this relationship visually. Pass the correlation matrix to imshow to visualize the heatmap.

In [3]:
```
# Create a sample correlation matrix
correlation_matrix = np.array([[1.0, 0.8, 0.3, -0.2],
                               [0.8, 1.0, 0.5, 0.1],
                               [0.3, 0.5, 1.0, -0.4],
                               [-0.2, 0.1, -0.4, 1.0]])

# Create a heatmap for the correlation matrix
plt.figure(figsize=(8, 6))  # Set the figure size
plt.imshow(correlation_matrix, cmap='coolwarm', vmin=-1, vmax=1, aspect='au

# Add a colorbar
cbar = plt.colorbar()
cbar.set_label('Correlation', rotation=270, labelpad=20)

# Add labels and a title
plt.title('Correlation Matrix Heatmap')
plt.xticks(range(len(correlation_matrix)), ['Var1', 'Var2', 'Var3', 'Var4']
plt.yticks(range(len(correlation_matrix)), ['Var1', 'Var2', 'Var3', 'Var4']

plt.show()
```

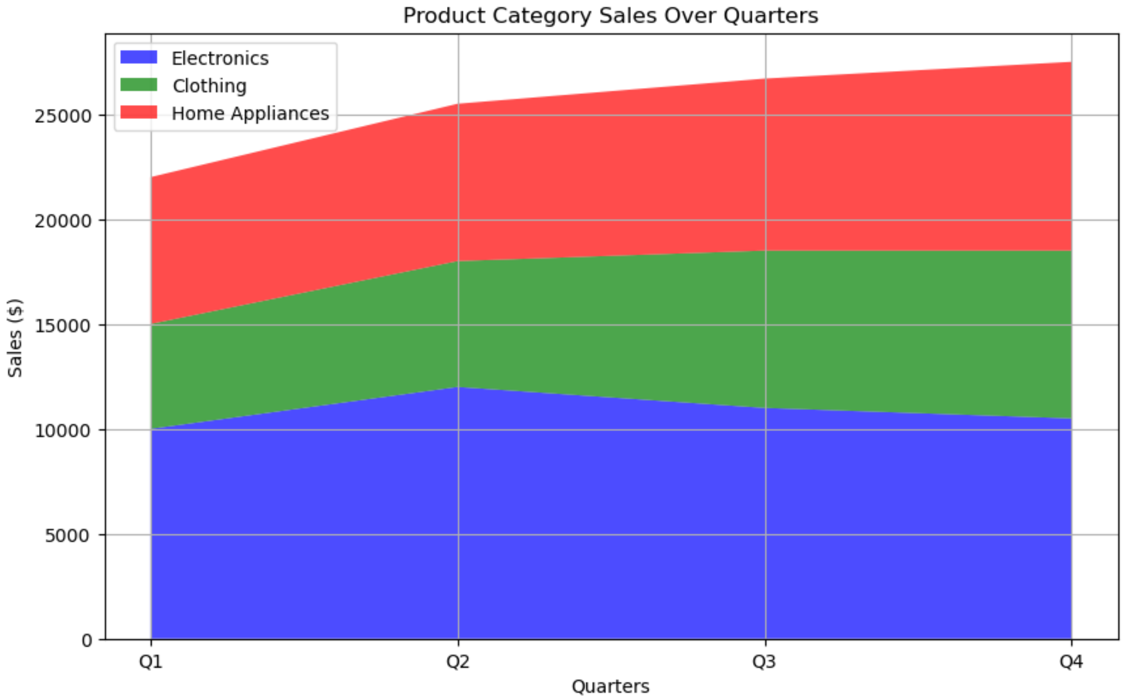


Stack Plot

Imagine you want to visualize how three product categories (electronics, clothing, and home appliances) contribute to total sales over four quarters (Q1 to Q4). Then you can represent each category’s sales as layers in the plot, and the plot helps us understand their contributions and trends over time. That’s exactly what the stack plot does.

A stack plot, which is also known as a stacked area plot, is a type of data visualization that displays multiple datasets as layers stacked on top of one another, with each layer representing a different category or component of the data. Stack plots are particularly useful for visualizing how individual components contribute to a whole over a continuous time period or categorical domain. Use it as plt.stackplot(x,y1,y2) , as many stacks as you

In [4]:
```
import matplotlib.pyplot as plt

# Sample data for stack plot
quarters = ['Q1', 'Q2', 'Q3', 'Q4']
electronics = [10000, 12000, 11000, 10500]
clothing = [5000, 6000, 7500, 8000]
home_appliances = [7000, 7500, 8200, 9000]

# Create a stack plot
plt.figure(figsize=(10, 6))  # Set the figure size
plt.stackplot(quarters, electronics, clothing, home_appliances, labels=['El
              colors=['blue', 'green', 'red'], alpha=0.7)

# Add labels, legend, and title
plt.xlabel('Quarters')
plt.ylabel('Sales ($)')
plt.title('Product Category Sales Over Quarters')
plt.legend(loc='upper left')

# Display the plot
plt.grid(True)

plt.show()
```



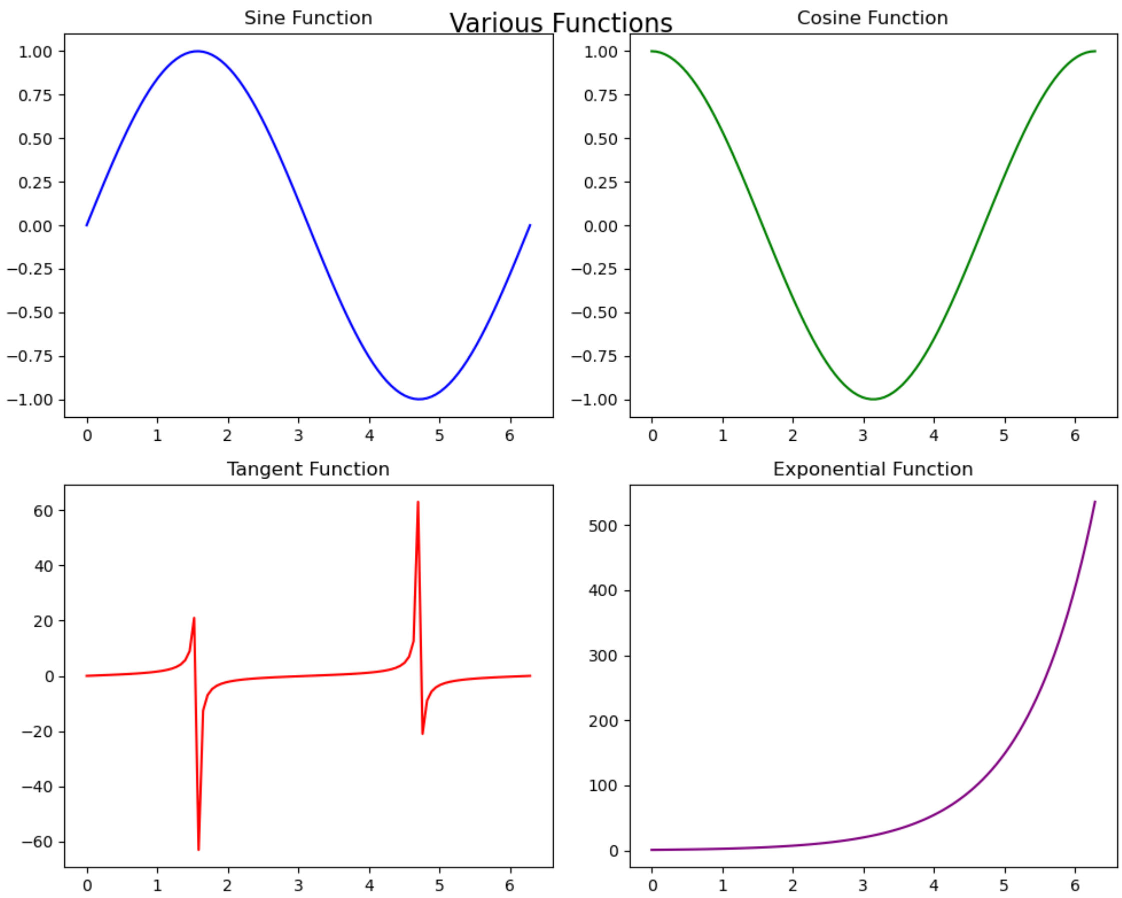
Multiple Subplots

Say, You are working with a dataset that has the age of a person, the software they are working on, and their salary. You want to visualize the Python developers' ages and salaries and then compare them with Java developers. By Now, you know you can do that by making two plots one in each cell of the notebook. But then, you have to move back and forth to compare, we better not talk about what if there are 4 things to compare!!

To Ease this issue, we have a feature called subplots, in the same plot there will be different subplots of each. You can create the subplots using plt.subplots(nrows=x,ncols=y) . By default nrows=1, and ncols=1. plt.subplots() returns two things one(fig) is to style the entire plot, and the other(axes) is to make subplots. Plot an each subplot using axes[row, column], where row and column specify the location of the subplot in the grid. You can use the sharex or sharey parameters when you have common axes for the subplots. Let's see a few examples to make it clear.

Creating Multiple Plots in a single figure

In [5]:
```
# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)
y4 = np.exp(x)

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Plot the first subplot (top-left)
axes[0, 0].plot(x, y1, color='blue')
axes[0, 0].set_title('Sine Function')

# Plot the second subplot (top-right)
axes[0, 1].plot(x, y2, color='green')
axes[0, 1].set_title('Cosine Function')

# Plot the third subplot (bottom-left)
axes[1, 0].plot(x, y3, color='red')
axes[1, 0].set_title('Tangent Function')

# Plot the fourth subplot (bottom-right)
axes[1, 1].plot(x, y4, color='purple')
axes[1, 1].set_title('Exponential Function')

# Adjust spacing between subplots
plt.tight_layout()

# Add a common title for all subplots
fig.suptitle('Various Functions', fontsize=16)

# Display the subplots
plt.show()
```



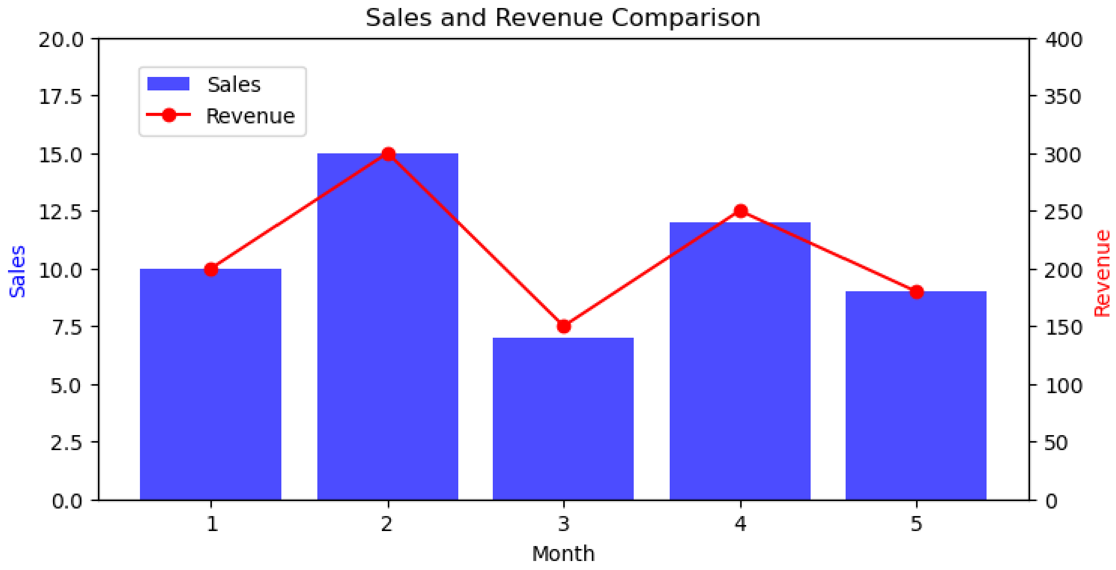
Combining Different Types of plots

When talking about comparing plots, we will not always wish to have the same axes for both plots, right? There will be cases where we have one common axis and other different!

In such cases, You can combine these different plots within a single figure using the twinx() or twiny() functions to share one axis while having independent y or x-axes. For example, you can combine a line plot of Month vs. Revenue, with a bar plot of Month vs. Sales, to visualize two related datasets with different scales. Here we have a common x-axis but a different y-axis.

In [6]:

```
# Sample data
x = np.arange(1, 6)
y1 = np.array([10, 15, 7, 12, 9])
y2 = np.array([200, 300, 150, 250, 180])

# Create a bar plot
fig, ax1 = plt.subplots(figsize=(8, 4))
ax1.bar(x, y1, color='b', alpha=0.7, label='Sales')
ax1.set_xlabel('Month')
ax1.set_ylabel('Sales', color='b')
ax1.set_ylim(0, 20)  # Set y-axis limits for the left y-axis

# Create a line plot sharing the x-axis
ax2 = ax1.twinx()
ax2.plot(x, y2, color='r', marker='o', label='Revenue')
ax2.set_ylabel('Revenue', color='r')
ax2.set_ylim(0, 400)  # Set y-axis limits for the right y-axis

# Add a legend
fig.legend(loc='upper left', bbox_to_anchor=(0.15, 0.85))

# Add a title
plt.title('Sales and Revenue Comparison')

# Show the plot
plt.show()
```



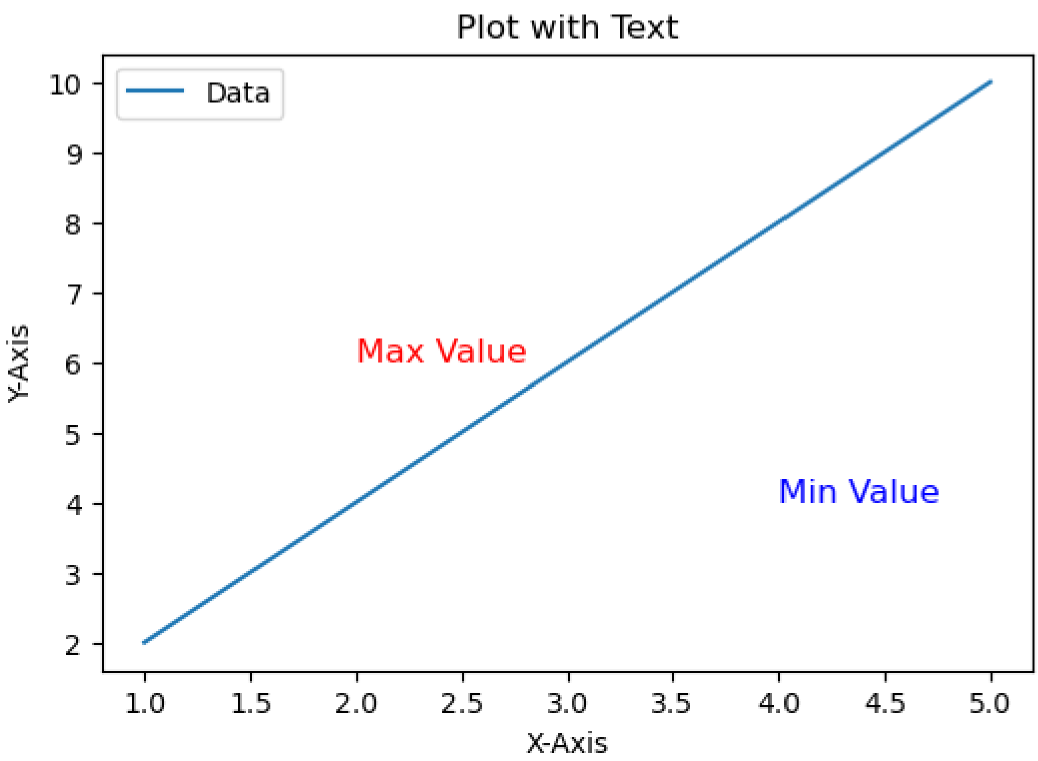
Advanced Features

Annoate and Text for the plots

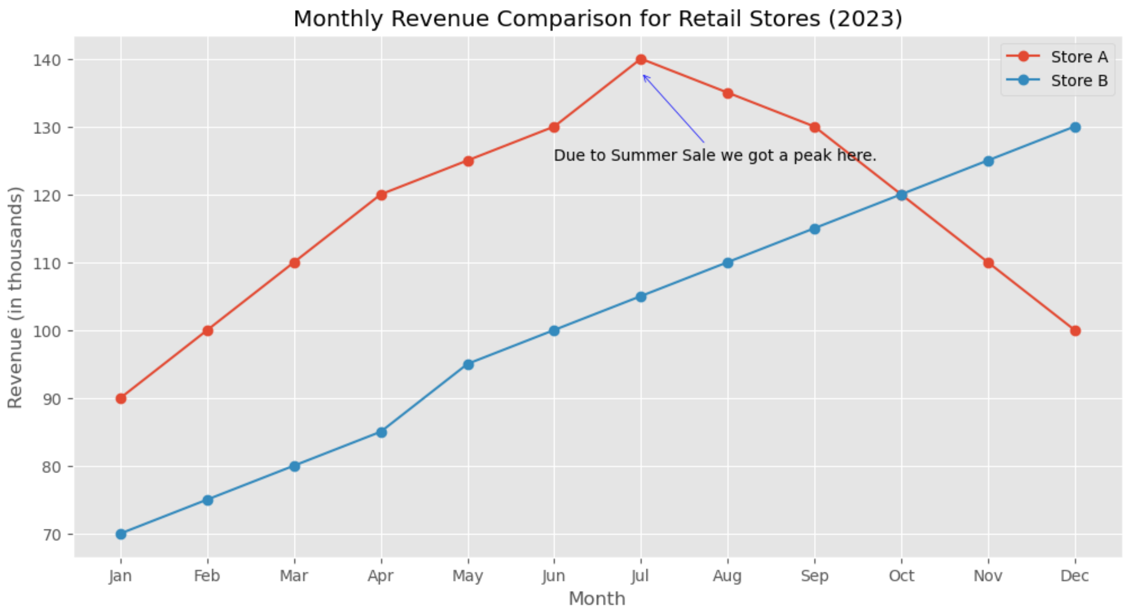
In Matplotlib, you can incorporate annotations and text using various methods. This is very useful during presentations, it is a powerful technique to enhance the communication of insights and highlight key points in your plots.

- **Adding text with text functions:** The plt.text(x_pos,y_pos,desired_text,fontsize=desired_size) function allows you to add custom text at specified coordinates on the plot.
- **Annotating with annotate() Function:** The plt.annotate(desired_text,xy=arrow_pos,xytext=text_post) function allows you to add text with an associated arrow or marker pointing to a specific location on the plot.
- **Labeling Data Points:** You can label individual data points in a scatter plot using text() or annotate()

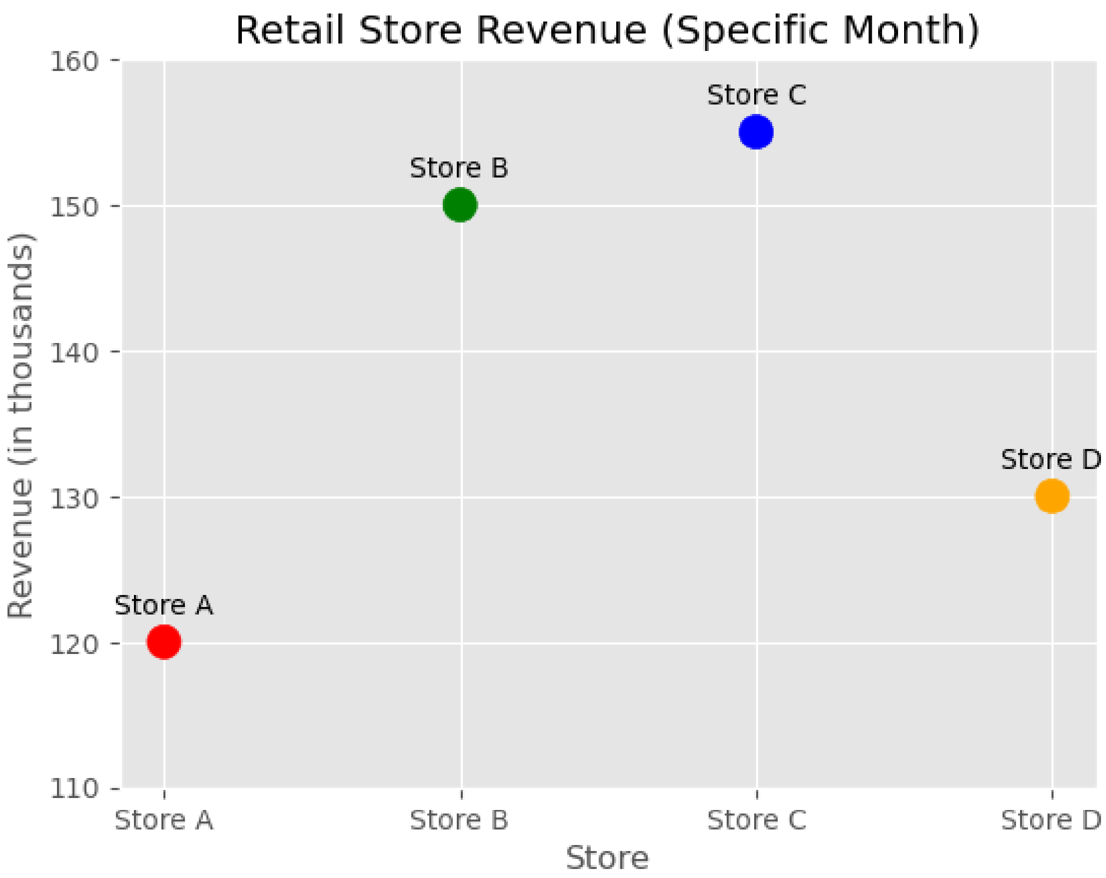
In [7]:
```
# Create a simple plot
plt.figure(figsize=(6, 4))
plt.plot([1, 2, 3, 4, 5], [2, 4, 6, 8, 10], label='Data')

# Add text to the plot
plt.text(2, 6, 'Max Value', fontsize=12, color='red')
plt.text(4, 4, 'Min Value', fontsize=12, color='blue')

# Customize the text
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.title('Plot with Text')
plt.legend()

# Show the plot
plt.show()
```



```
In [8]: # Sample data for retail shop revenue
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', '(
store_locations = ['Store A', 'Store B']
revenue_data = np.array([[90, 100, 110, 120, 125, 130, 140, 135, 130, 120,
                         [70, 75, 80, 85, 95, 100, 105, 110, 115, 120, 125,

# Create the plot
plt.figure(figsize=(12, 6))

plt.style.use('ggplot')

# Plot monthly revenue for each store location
for i in range(len(store_locations)):
    plt.plot(months, revenue_data[i], marker='o', label=store_locations[i]

# Highlight special promotions
plt.annotate('Due to Summer Sale we got a peak here.',  xy=('Jul', 138), xy
             arrowprops=dict(arrowstyle='->', color='blue'))

# Add title and labels
plt.title('Monthly Revenue Comparison for Retail Stores (2023)')
plt.xlabel('Month')
plt.ylabel('Revenue (in thousands)')
plt.grid(True)
plt.legend()

plt.show()
```



In [9]:
```
# Sample data for retail shop revenue in a specific month
store_names = ['Store A', 'Store B', 'Store C', 'Store D']
revenue_data = np.array([120, 150, 155, 130])  # Revenue in thousands
colors = ['red', 'green', 'blue', 'orange']

# Plot store revenue as points
plt.scatter(store_names, revenue_data, s=150, c=colors, marker='o', label='

# Label each store
for i, store in enumerate(store_names):
    plt.annotate(store, (store, revenue_data[i]), textcoords="offset points
    # plt.text(store, revenue_data[i]+1,store)

# Add title and labels
plt.title('Retail Store Revenue (Specific Month)')
plt.xlabel('Store')
plt.ylabel('Revenue (in thousands)')
plt.ylim(110,160)
plt.grid(True)
plt.show()
```



In the above example of the plot with labels, To add labels, we need to iterate through the names and use the annotate method for each point of the name. In this case, you don't need to use xy and arrowprops parameters. But you do need to use textcoords='offset points' , this ensures that the positions specified for the label (in this case, xytext) are interpreted in a coordinate system where the origin (0, 0).

In [10]:
```
plt.style.use('default')
```

4.2 Fill the Area Between the plots

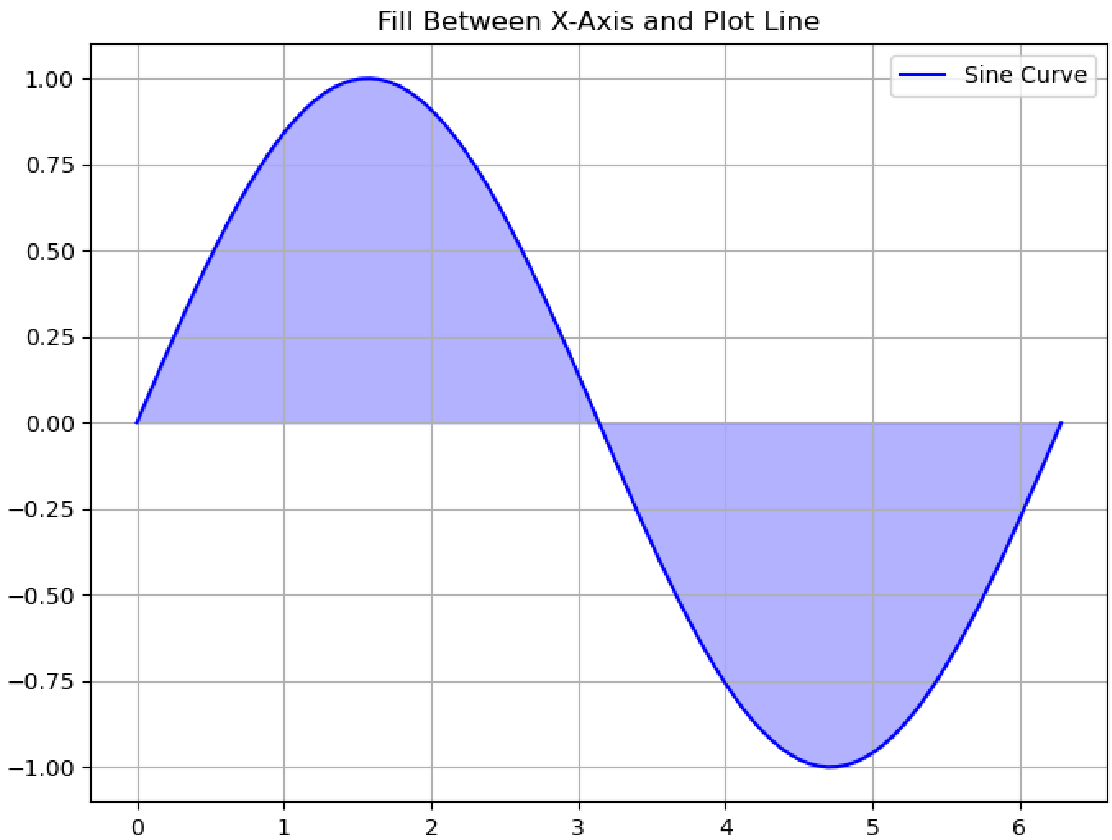
Sometimes we need to highlight the regions between two line plots, which can help viewers understand where one curve surpasses another. And this can be achieved through fill_between method in Matplotlib. The intensity of the fill color can be controlled through alpha parameter.

- To Fill all the Region between the x-axis and the plot line, you can use the command plt.fill_between(x,y)
- To Fill the intersection between two plot lines, you can use the command plt.fill_between(x,y1,y2)
- To Fill the intersection between two plot lines only if they satisfy a specified condition, you can use the command plt.fill_between(x,y1,y2,where=condition)
- To Fill more than one region of the plot with different conditions and different colors.

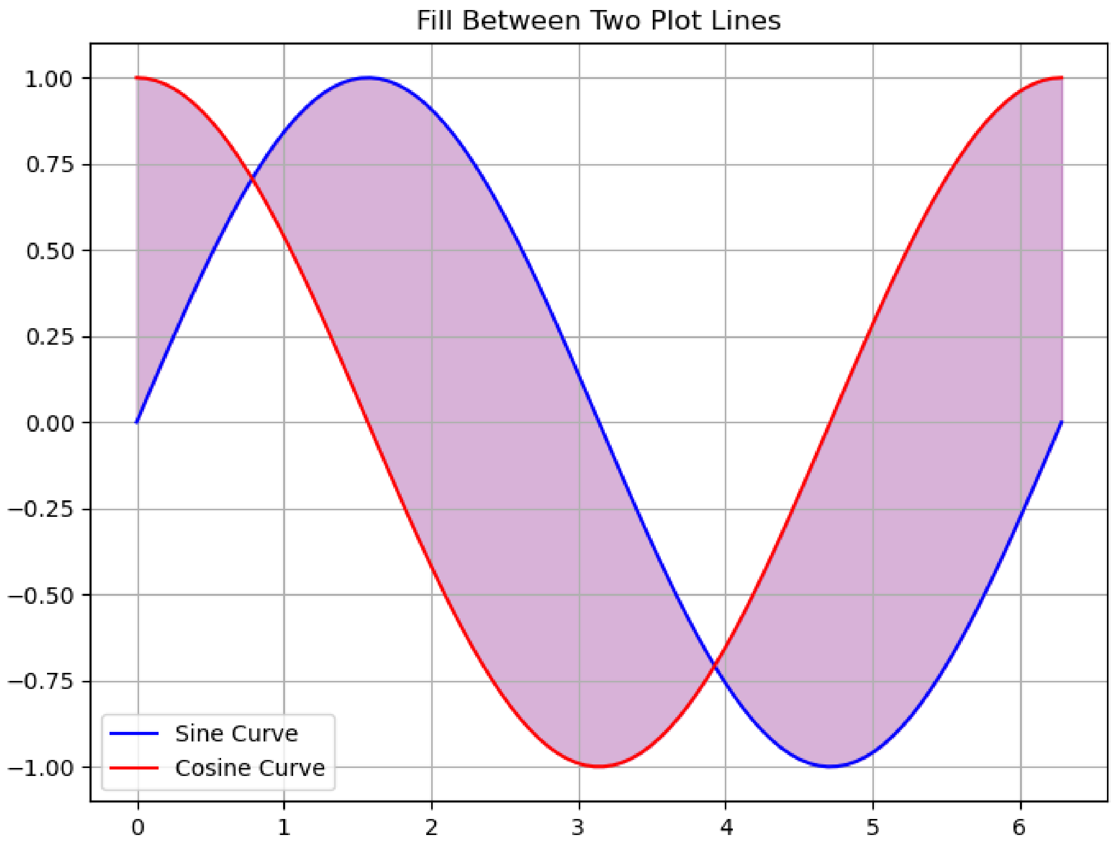
Here are a Few Examples of the above cases.

In [11]:

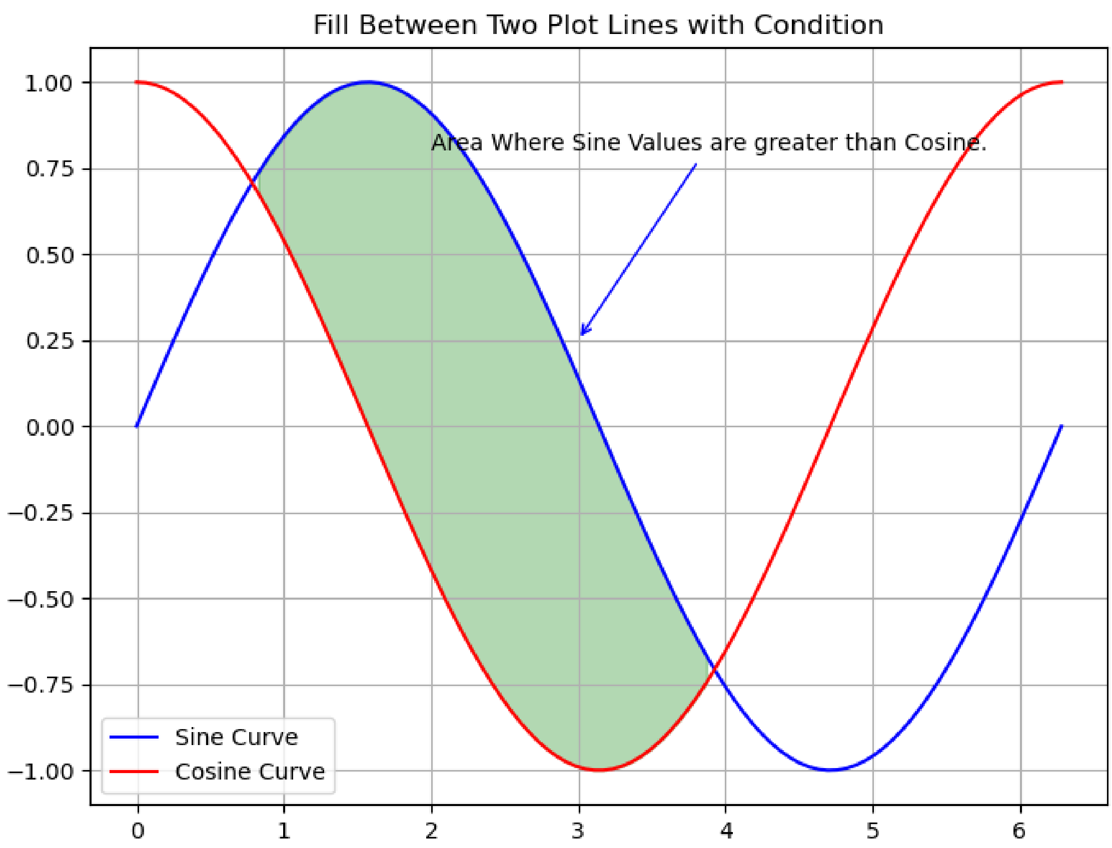
```
# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the curve
plt.plot(x, y, label='Sine Curve', color='blue')

# Fill the region between the curve and the x-axis
plt.fill_between(x, 0, y, alpha=0.3, color='blue')

# Add title and labels
plt.title('Fill Between X-Axis and Plot Line')
plt.grid(True)
plt.legend()
plt.show()
```



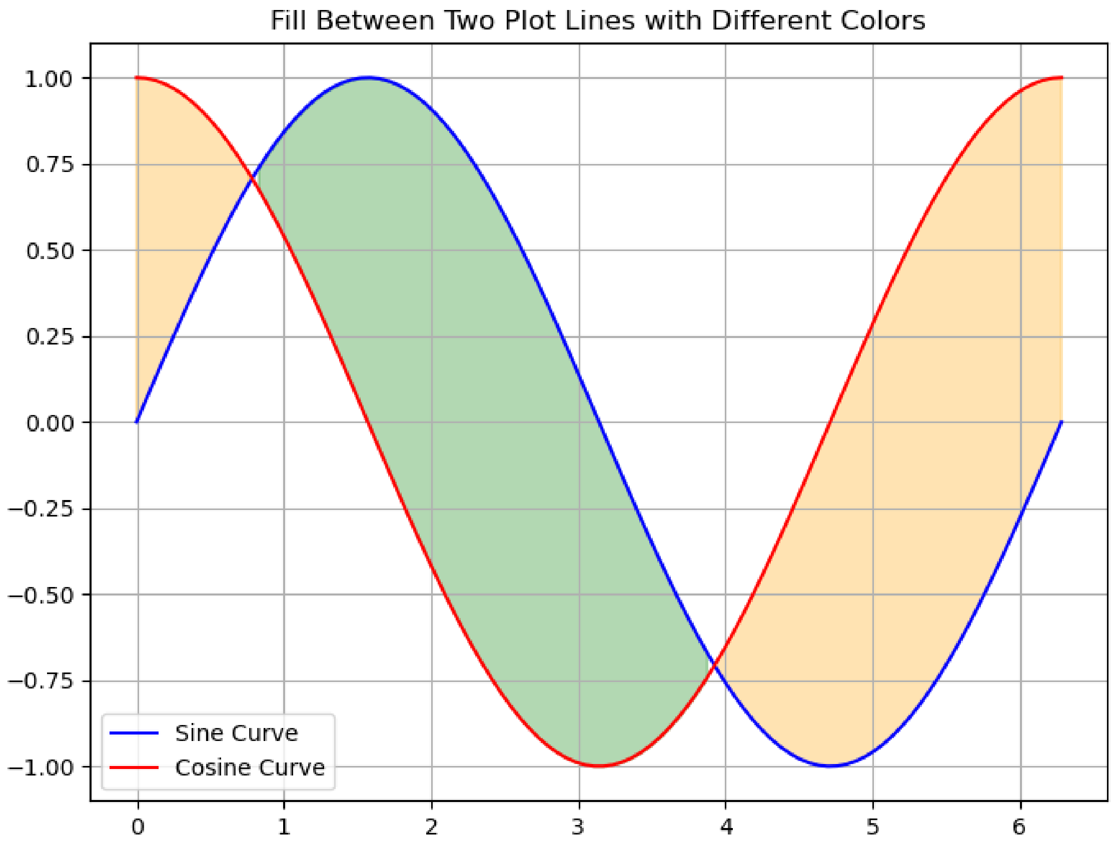
In [12]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill the region between the two curves
plt.fill_between(x, y1, y2, alpha=0.3, color='purple')

# Add title and labels
plt.title('Fill Between Two Plot Lines')
plt.grid(True)
plt.legend()
plt.show()
```



In [13]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill the region between the two curves where y1 > y2
plt.fill_between(x, y1, y2, where=(y1 > y2), alpha=0.3, color='green')

# Highlight special promotions
plt.annotate('Area Where Sine Values are greater than Cosine.', xy=(3, 0.25
             arrowprops=dict(arrowstyle='->', color='blue'))

# Add title and labels
plt.title('Fill Between Two Plot Lines with Condition')
plt.grid(True)
plt.legend()
plt.show()
```



In [14]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill multiple regions with different colors
plt.fill_between(x, y1, y2, where=(y1 > y2), alpha=0.3, color='green')
plt.fill_between(x, y1, y2, where=(y1 <= y2), alpha=0.3, color='orange')

# Add title and labels
plt.title('Fill Between Two Plot Lines with Different Colors')
plt.grid(True)
plt.legend()
plt.show()
```

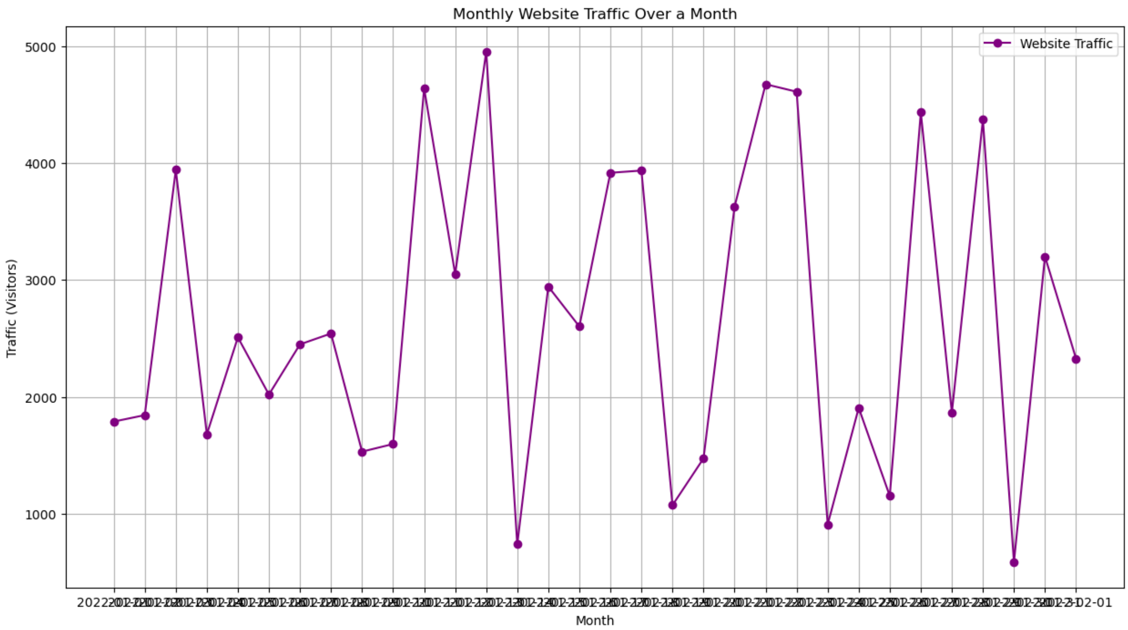


Plotting Time Series Data

We all know that Time series data is very common in many fields such as finance, climate science, business analytics, etc. And also the data will be very huge in these cases, we can't make the most out of data by just doing some aggregations! Matplotlib offers us ways to easily interpret the time-series data.

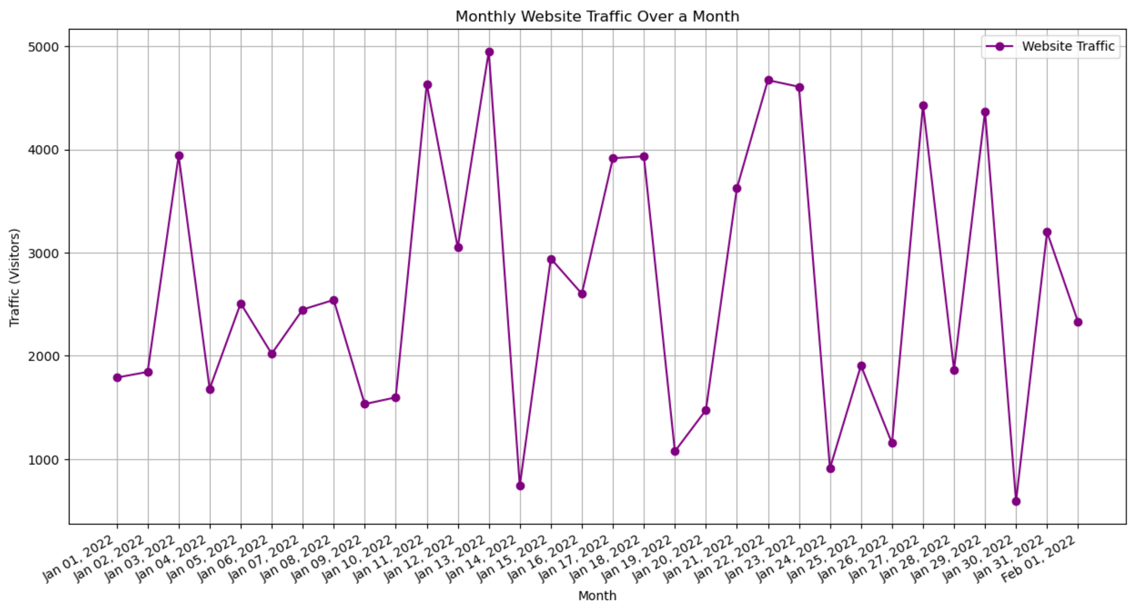
Imagine, you want to plot website traffic over one month. If you make a line plot, the x-axis will be very clumsy with all the dates and you can't see any dates properly! Something like below.

In [15]:
```
import pandas as pd
import matplotlib.dates as mdates
import numpy as np
np.random.seed(10)

# Let's generate sample time series data for one month
date_rng = pd.date_range(start='2022-01-01', end='2022-02-01')

# Generate random website traffic values for one month
traffic_data = np.random.randint(500, 5000,len(date_rng))

# Create a DataFrame
traffic_df = pd.DataFrame({'Month': date_rng, 'Traffic (Visitors)': traffic

# Set the figure size
plt.figure(figsize=(15, 8))

# Now, let's create a time series plot to visualize the monthly website tra
plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label=

# To show all the dates of one month
plt.xticks(traffic_df['Month'])

# Adding Labels and Title:
plt.xlabel('Month')
plt.ylabel('Traffic (Visitors)')
plt.title('Monthly Website Traffic Over a Month')

# Adding Grid Lines and Legends:
plt.grid(True)
plt.legend(['Website Traffic'], loc='upper right')

# Display the plot
plt.show()
```



Let's see the same example but with just three additional lines of customization that make the time series plot easily interpretable!

In [16]:
```
# Set the figure size
plt.figure(figsize=(15, 8))

# Now, let's create a time series plot to visualize the monthly website tra
plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label=

# To show all the dates of one month
plt.xticks(traffic_df['Month'])

# Adding Labels and Title:
plt.xlabel('Month')
plt.ylabel('Traffic (Visitors)')
plt.title('Monthly Website Traffic Over a Month')

# Set x-axis Date Format: Month Day, Year
date_format = mdates.DateFormatter('%b %d, %Y')
# Customize date formatting by using DateFormatter
plt.gca().xaxis.set_major_formatter(date_format)
# Autoformatting the x-axis
plt.gcf().autofmt_xdate()

# Adding Grid Lines and Legends:
plt.grid(True)
plt.legend(['Website Traffic'], loc='upper right')

# Display the plot
plt.show()
```



3D Plots

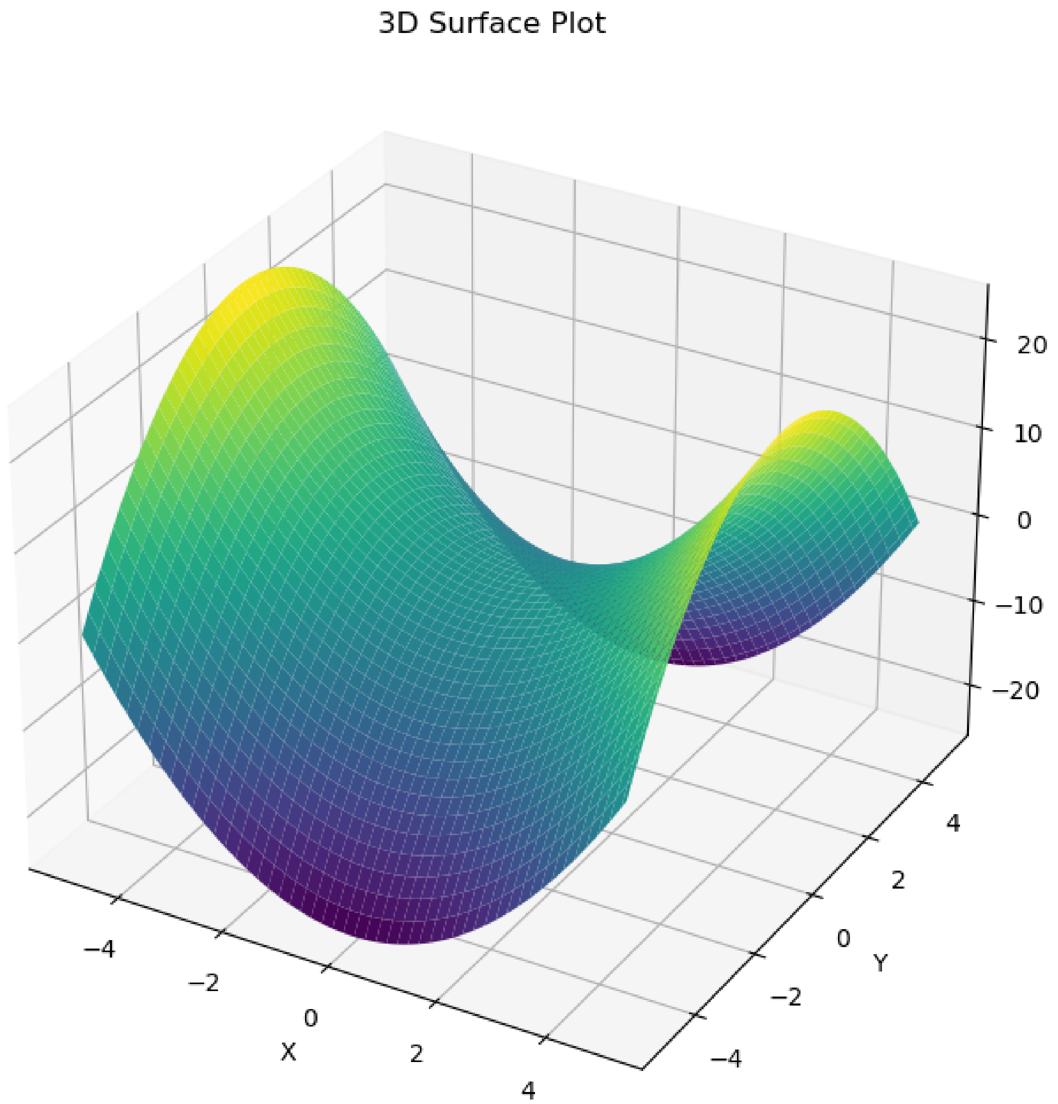
Creating 3D plots using Matplotlib involves using the mpl_toolkits.mplot3d toolkit, which provides functions for creating various types of 3D visualizations. you need to import the Axes3D to visualize the plots in 3D with the following command from mpl_toolkits.mplot3d import Axes3D .

First, we need to create a Matplotlib figure object using fig=plt.figure() . To add a 3D subplot to the figure we need to use the add_subplot method, axes=fig.add_subplt(111,projection='3d') . In this case, (1, 1, 1) means there is only one row, and one column, and the current subplot is in the first (and only) position.

Let's create a 3D surface plot, you can also create a 3D line plot, 3D scatter plot, etc.

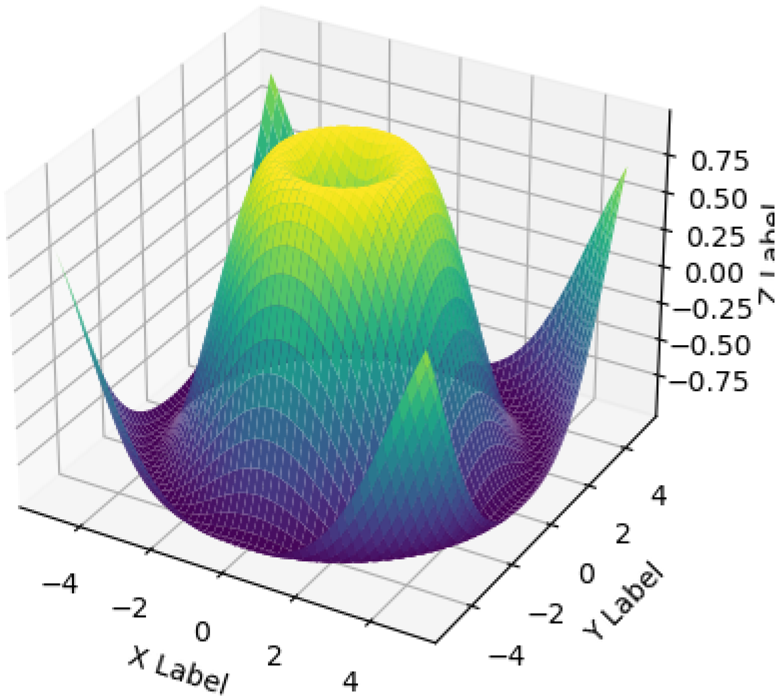
In [17]:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Create a meshgrid of X and Y values
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

# Define the function to plot (example: a saddle shape)
Z = X**2 - Y**2

# Create a 3D surface plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, Z, cmap='viridis')

# Add title and labels
ax.set_title('3D Surface Plot')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```

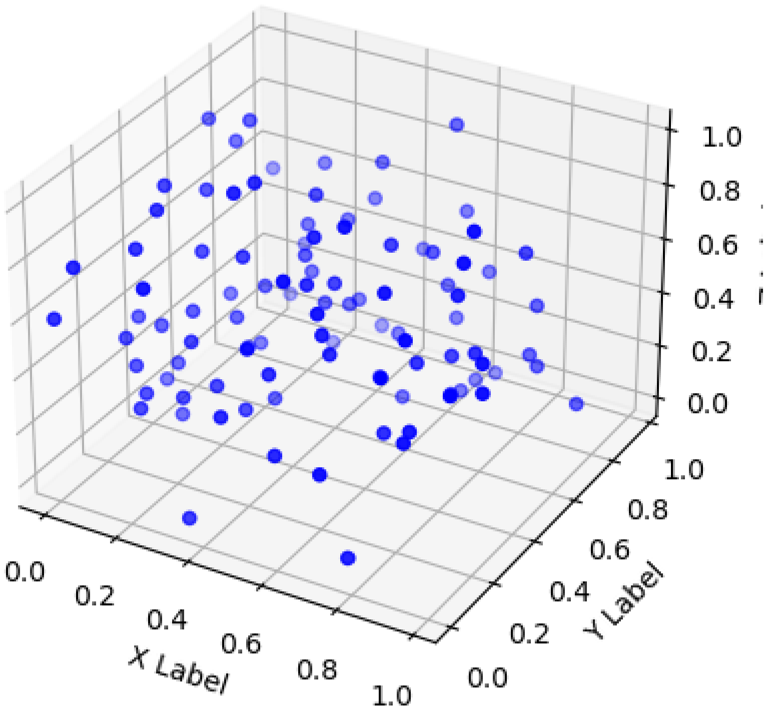

3D Surface Plot
X
Y
−4
−2
0
2
4
20
10
0
−10
−20

In [18]:

```python
# Create a 3D surface plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Generate 3D data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create the surface plot
ax.plot_surface(X, Y, Z, cmap='viridis')

# Add labels
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```



In [19]:
```
# Create a 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Generate random 3D data
x = np.random.rand(100)
y = np.random.rand(100)
z = np.random.rand(100)

# Create the scatter plot
ax.scatter(x, y, z, c='b', marker='o')

# Add labels
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```



Animation

It would be nice to rotate, zoom, and hover to see the location of the above 3D plot, right? Guess what, we can actually do that in one line! Use the command %matplotlib notebook in your Jupyter Notebook to make the plot interactive. If you want to change it back to static plots use %matplotlib inline .

When the interactive plots are enabled, you can also create nice animation plots, like a moving sine wave, etc. That can be achieved by using FuncAnimation methods from matplotlib.animation module.

The FncAnimation method takes in the figure object, the function to call repeatedly, interval time to call the function. The below code will create an animated sine wave. So, Here animate function will be called every 1 second, and the resulting plot will be plotted in the

In [20]:

```
from matplotlib.animation import FuncAnimation

%matplotlib notebook
# Create a figure and axis
fig, ax = plt.subplots()
ax.set_xlim(0, 2 * np.pi)
ax.set_ylim(-1.5, 1.5)

# Initialize the point to be animated
point, = ax.plot([], [], 'bo', markersize=10)

# Function to initialize the plot
def init():
    point.set_data([], [])
    return point,

# Function to update the animation
def animate(frame):
    x = np.linspace(0, 2 * np.pi, 1000)
    y = np.sin(x + 0.1 * frame)  # Vary the phase to create animation
    point.set_data(x, y)
    return point,

# Create the animation
ani = FuncAnimation(fig, animate, frames=100, init_func=init, blit=True, ir

plt.title('Animated Point on Sine Wave')
plt.xlabel('X')
plt.ylabel('sin(X)')
plt.grid(True)

plt.show()
```

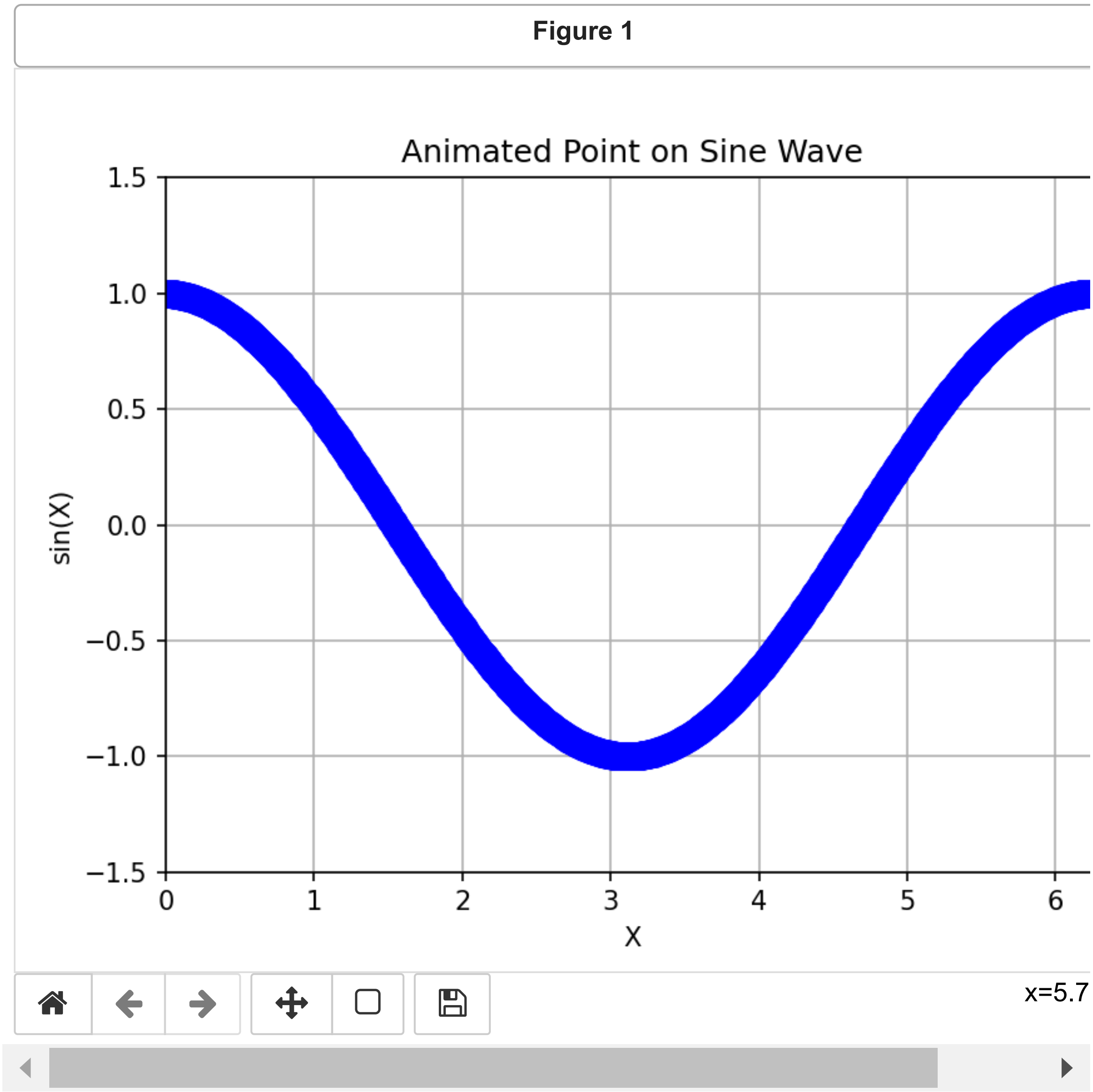


Figure 1
Animated Point on Sine Wave
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
sin(X)
0
1
2
3
4
5
6
X
x=5.7