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

Table of Contents

2.Plot Types

- 2.1 Bar Chart
- 2.2 Histograms
- 2.3 Scatter plots
- 2.4 Pie Charts
- 2.5 Box Plot (Box and Whisker Plot)
- 2.6 Heatmap, and Displaying Images
- 2.7 Stack Plot

3.Multiple Subplots

- 3.1 Creating Multiple Plots in a Single Figure
- 3.2 Combining Different Types of Plots

4.Advanced Features

- 4.1 Adding annotations and text
- 4.2 Fill the Area Between Plots
- 4.3 Plotting Time Series Data
- 4.4 Creating 3D Plots
- 4.5 Live Plot - Incorporating Animations and Interactivity.

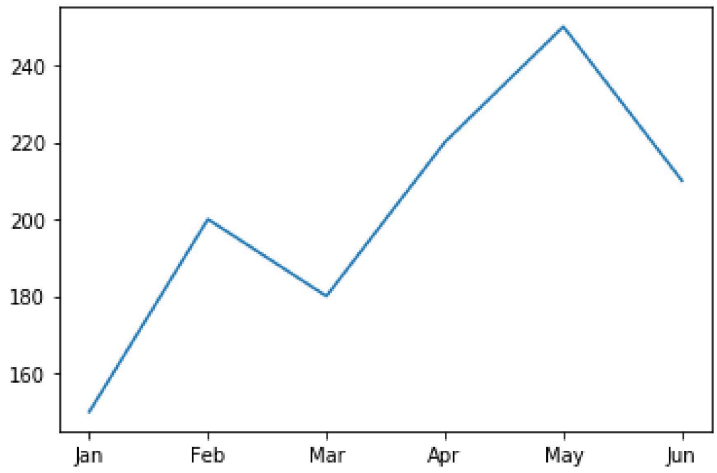
1-Basic Plotting

First things first, we need to import matplotlib.pyplot to access the plotting functions.

1.1 Creating Simple Line Plots

As the name suggests, data points are connected by straight lines which are useful for displaying data that varies continuously over a range, making it easy to identify patterns and trends. Use `plt.plot(x,y)` for a simple line plot, and `plt.show()` to show the plot.

In [2]:
```
# Sample data representing monthly website traffic (in thousands)
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
traffic = [150, 200, 180, 220, 250, 210]

# Create a line plot
plt.plot(months, traffic)

plt.show()
```

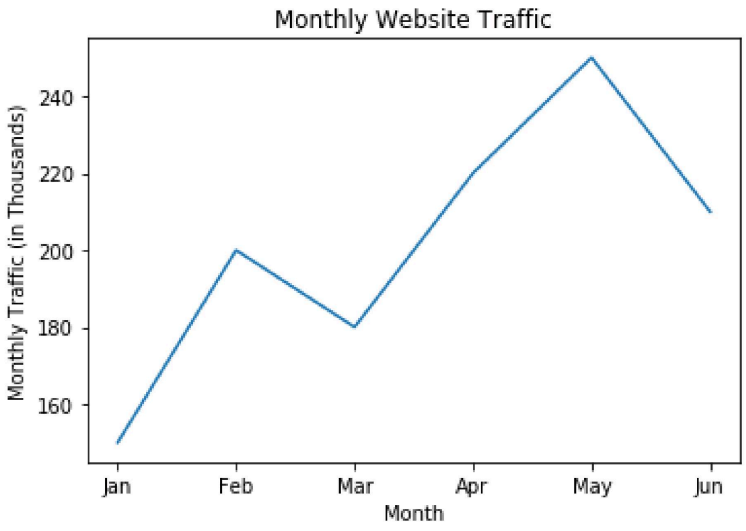


- But as we can see, it doesn't have any labels or titles as such.

1.2 Adding Labels and Titles

We can convey the information with much clarity by customizing the plots. Matplotlib offers numerous customization options, allowing you to control color, line style, markers, and more.

In [3]:
```
import matplotlib.pyplot as plt

# Sample data representing monthly website traffic (in thousands)
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
traffic = [150, 200, 180, 220, 250, 210]

# Create a line plot
plt.plot(months, traffic)

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

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



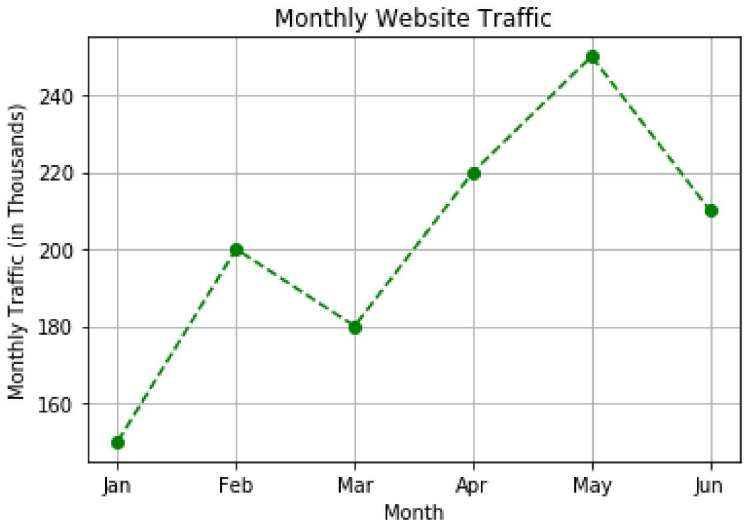
Note: Remember that plt.show() should always be at the end of your plot settings. If you give label commands after the plt.show()then they won't be displayed!

1.3 Customizing the plot

Use the parameter `marker` to mark the points, linetsyle to change the styling of line, and add grid to the plots by using `plt.grid(True)`

Adding styles to the graph

In [4]:
```
# Create a line plot with custom appearance
plt.plot(months, traffic, marker='o', linestyle='--', color='g')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

plt.grid(True)

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



Changing the Entire Plot Style

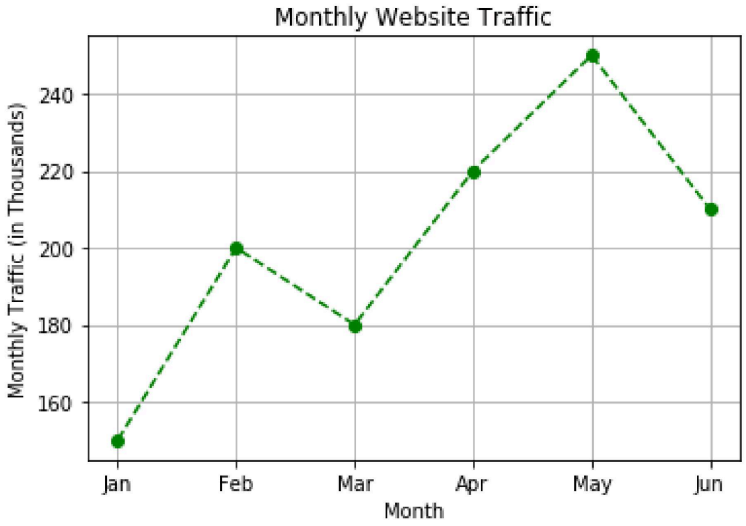
There are various styles available in Matplotlib, to check the available styles, use the command `plt.style.available` . Use plt.style.use(desired_style) to change the style of the entire plot. To use a comic-style plot, you can use plt.xkcd() , this will give a cool plot like below.

In [5]:
```
plt.style.available
```

Out[5]:
```
['bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark-palette',
 'seaborn-dark',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'seaborn',
 'Solarize_Light2',
 'tableau-colorblind10',
 '_classic_test']
```

These are all the available styles of the plots.

In [6]:
```
# Create a line plot with custom appearance
plt.plot(months, traffic, marker='o', linestyle='--', color='g')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

plt.style.use('fivethirtyeight')
plt.grid(True)
# Display the plot
plt.show()
```



To bring the plot changes to default, use plt.style.use("default"). And to use a comic style plot, use `plt.xkcd()`

Note: Make sure to use these style commands before the plt.show()

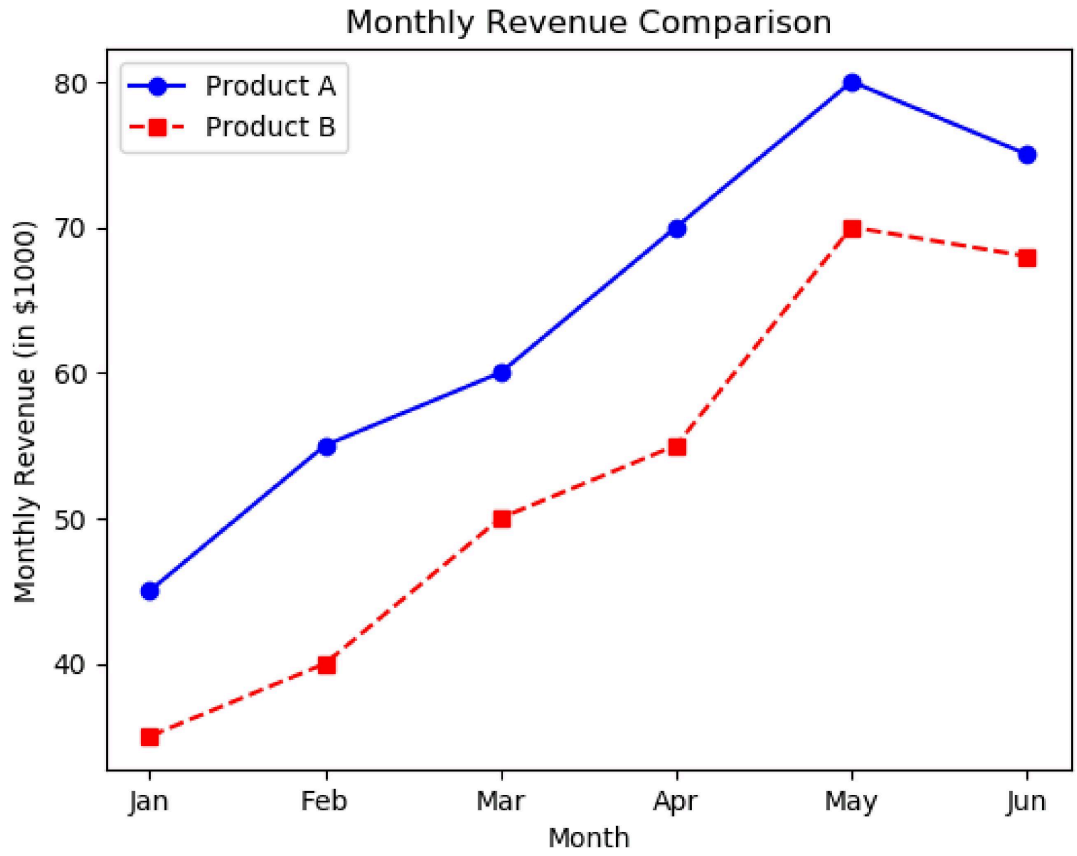
In [7]:
```
plt.style.use("default")
```

We often have to adjust the plot size, right? And to do that, we need to use plt.figure(figsize=(x_size,y_size)) , Make sure to use this before the .plot command.

1.4 Multiple Line Plot

- In the case of plotting multiple lines in the same graph, You can do so by using the plot command two times for the variables you want. But the issue is to differentiate them properly, for this, we have a parameter called a label , along with that you also need to use plt.legend()

In [8]:
```
# Sample data for two products' monthly revenue (in thousands of dollars)
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
product_a_revenue = [45, 55, 60, 70, 80, 75]
product_b_revenue = [35, 40, 50, 55, 70, 68]

# Create a line plot for Product A with a blue line and circular markers
plt.plot(months, product_a_revenue, marker='o', linestyle='-', color='blue', lab
el='Product A')

# Create a line plot for Product B with a red dashed line and square markers
plt.plot(months, product_b_revenue, marker='s', linestyle='--', color='red', lab
el='Product B')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Revenue (in $1000)')
plt.title('Monthly Revenue Comparison')

# Display a legend to differentiate between Product A and Product B
plt.legend()

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



1.5 Saving the plot as an image

In Jupyter Notebook: When working in Jupyter Notebook if you wish to save the plot as an image file, you have to use `plt.savefig(path/to/directory/plot_name.png)` . You can specify the complete file path, and you can specify the desired file name and format(Eg: .jpg, .png, .pdf)

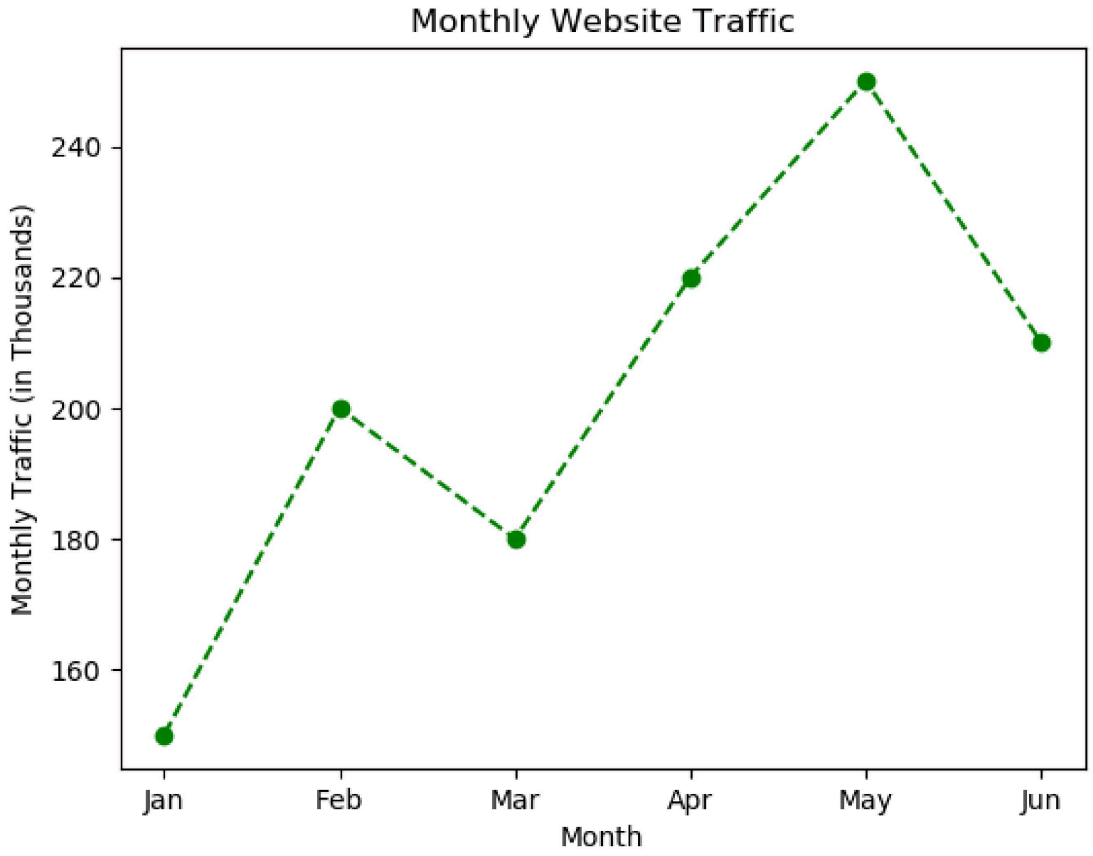
In Google Colab: When working in Google Colab, if you wish to save the plot as an image file, you have to first mount the drive and use plt.savefig().

from google.colab import drive

Mount Google Drive using `drive.mount('/content/drive')` `path = '/content/drive/My Drive/'`

Save the plot as an image file in Colab `plt.savefig(path+'saving_line_plot.png')`

In [9]:
```
# Create a line plot with custom appearance
plt.plot(months, traffic, marker='o', linestyle='--', color='g')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

# Save the plot as an image file in Colab
plt.savefig('saving_line_plot.png')  # Specify the complete file path

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



It will be saved now :)

2-Plot Types

We have seen the basic line plot in the previous section, but Matplotlib has a lot more kinds of plots to offer such as Bar Charts, Histograms, Scatter Plots, Pie Charts, Box Plot (Box and whisker Plot), Heatmaps, Displaying images, etc. Now, Let's understand when to use them along with a few use cases.

2.1 Bar Plots

Bar charts represent categorical data with rectangular bars, where the length or height of each bar represents a value. You can use the command plt.bar(x,y) to generate vertical bar charts and plt.barh(x,y) for horizontal bar charts.

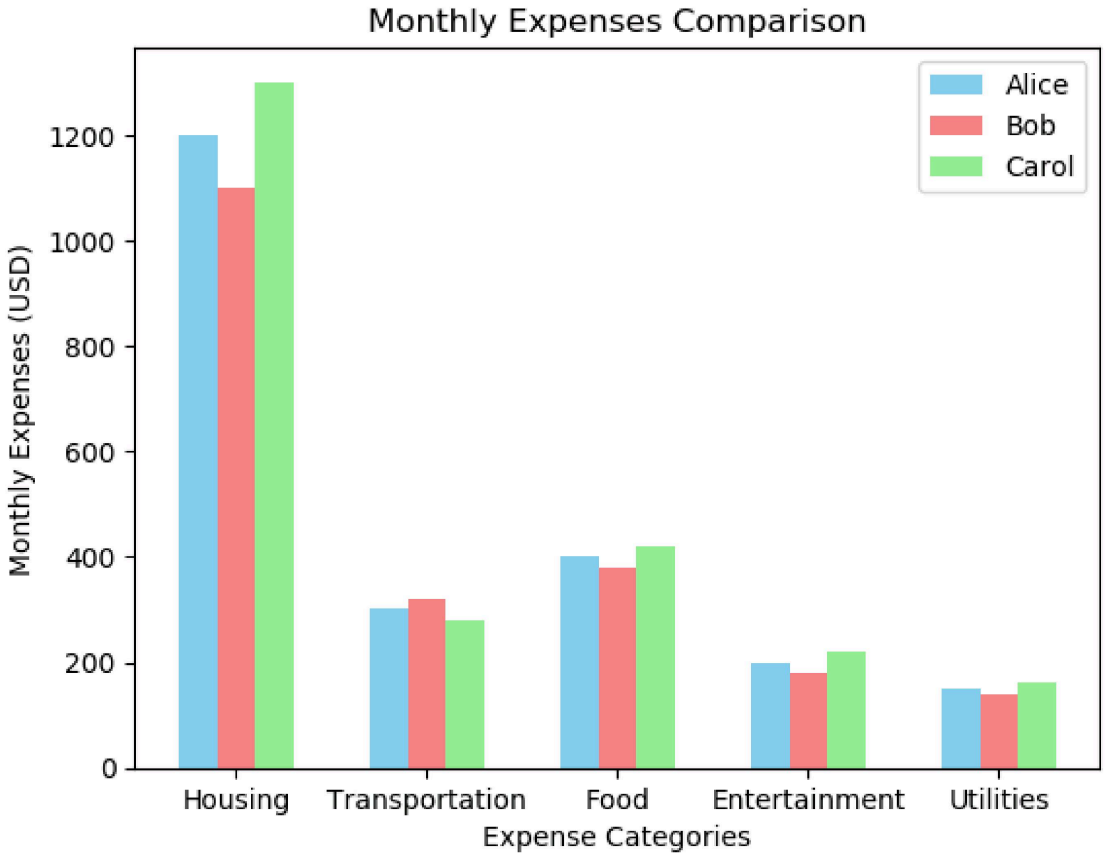
Few Use Cases:

1. Comparing sales performance of different products.
2. Showing population distribution by country.

Eg: Multi Bar plot in a single Graph

In [10]:

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

# Expense categories
categories = ['Housing', 'Transportation', 'Food', 'Entertainment', 'Utilities']

# Monthly expenses for Alice, Bob, and Carol
alice_expenses = [1200, 300, 400, 200, 150]
bob_expenses = [1100, 320, 380, 180, 140]
carol_expenses = [1300, 280, 420, 220, 160]

# Create an array for the x-axis positions
x = np.arange(len(categories))

# Width of the bars
bar_width = 0.2

# Create bars for Alice's expenses
plt.bar(x - bar_width, alice_expenses, width=bar_width, label='Alice', color='sk
yblue')

# Create bars for Bob's expenses
plt.bar(x, bob_expenses, width=bar_width, label='Bob', color='lightcoral')

# Create bars for Carol's expenses
plt.bar(x + bar_width, carol_expenses, width=bar_width, label='Carol', color='li
ghtgreen')

# Add labels, a title, and a legend
plt.xlabel('Expense Categories')
plt.ylabel('Monthly Expenses (USD)')
plt.title('Monthly Expenses Comparison')
plt.xticks(x, categories)
plt.legend()

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



- so, to get these bars, for the first bar we subtracted the x-labels with the bar width, and for the last bar, we added the label with the bar width. We set the width parameter to be equal to the bar width for all.

2.2 Histograms

Histograms are used to visualize the distribution of continuous or numerical data and they help us identify patterns in data. In a histogram plot the data is grouped into “bins,” and the height of each bar represents the frequency or count of data points in that bin. It takes the lower and upper limits of the given data and divides it into the no of bins given.

You can use the command plt.hist(x) to generate histograms. Unlike the bar plot, here you don’t need the ‘y’, as it only represents the frequency of one continuous data. The default bins are 10, and they can be changed. You can override the bins range as well with your desired bins range. You can also add the edgecolor for bars.

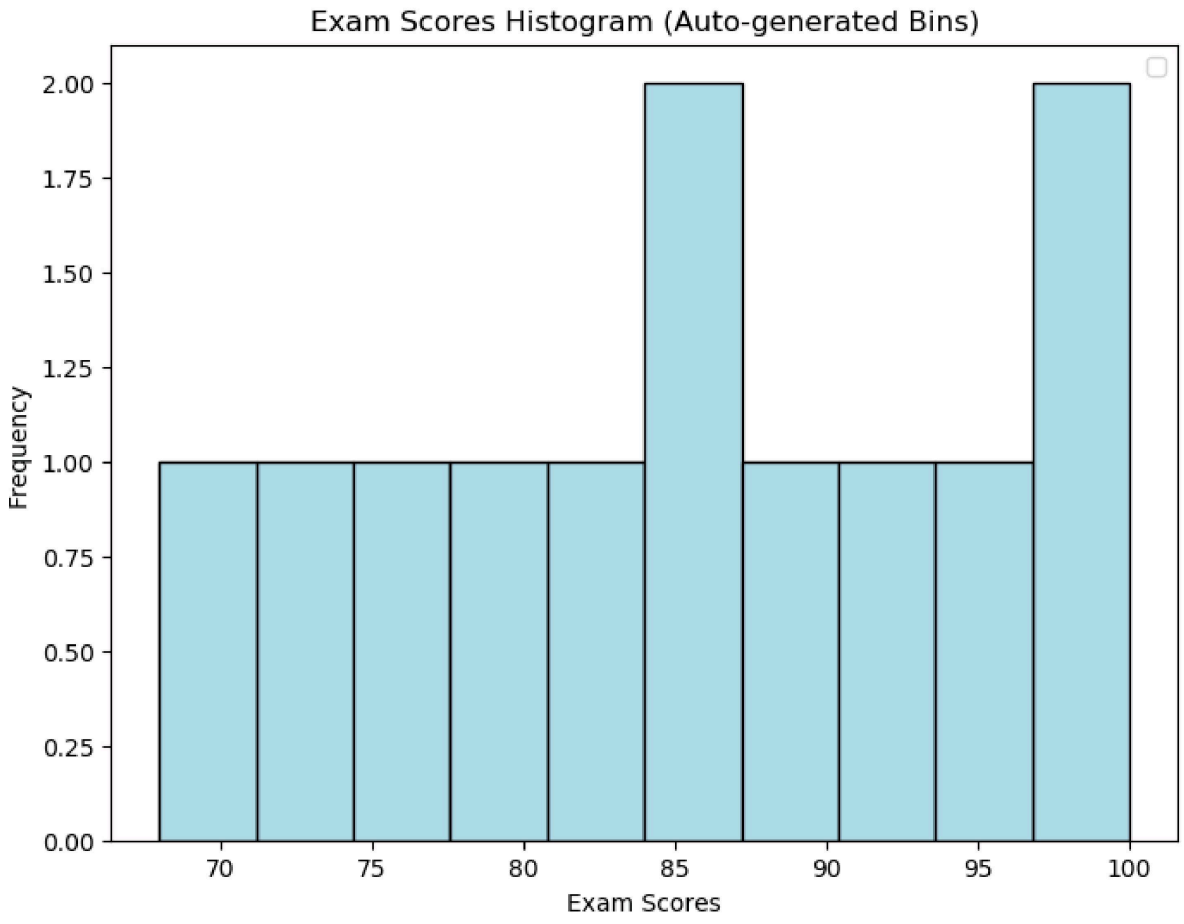
Along with the histogram plot, in the same graph if you want to add any line, say the mean or median, you can do so by calculating the value and passing to to plt.axvline(calculated_mean,label=desired_label) . This can be used with any other plot.

Few Use Cases:

1. Analyzing age distribution in a population.
2. Examining exam score distribution in a classroom.

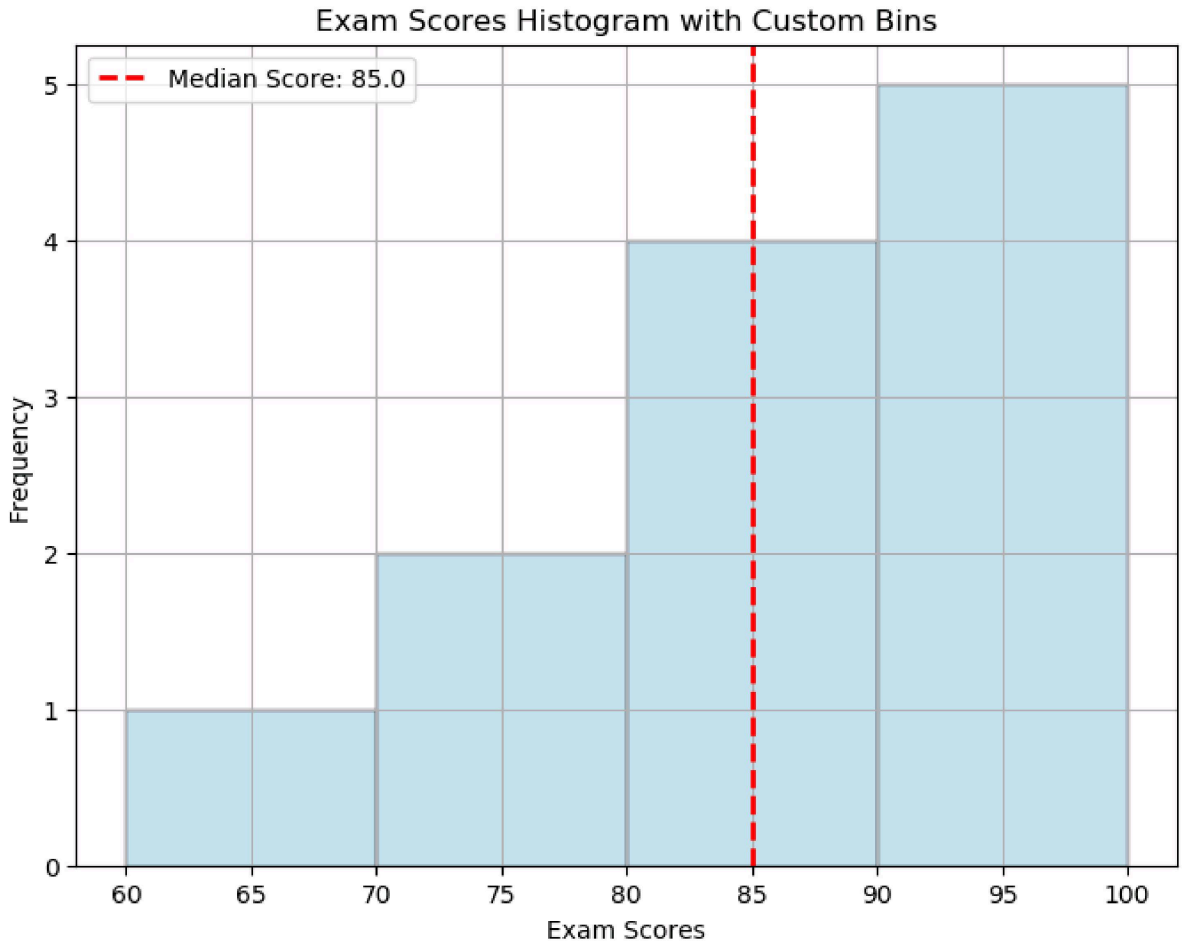
In [11]:
```
# Sample exam scores data
exam_scores = [68, 72, 75, 80, 82, 84, 86, 90, 92, 95, 98, 100]

# Create a histogram without specifying bin ranges
plt.figure(figsize=(8, 6))  # Set the figure size
plt.hist(exam_scores, color='lightblue', edgecolor='black')

# Add labels and a title
plt.xlabel('Exam Scores')
plt.ylabel('Frequency')
plt.title('Exam Scores Histogram (Auto-generated Bins)')

# Add a legend
plt.legend()
plt.show()
```

```
No handles with labels found to put in legend.
```



In [12]:

```
# Sample exam scores data
exam_scores = [68, 72, 75, 80, 82, 84, 86, 90, 92, 95, 98, 100]

# Custom bin ranges
bin_ranges = [60, 70, 80, 90, 100]

# Create a histogram with custom bin ranges
plt.figure(figsize=(8, 6))  # Set the figure size
plt.hist(exam_scores, bins=bin_ranges, color='lightblue', edgecolor='black', alp
ha=0.7)

# Add labels and a title
plt.xlabel('Exam Scores')
plt.ylabel('Frequency')
plt.title('Exam Scores Histogram with Custom Bins')

# Calculate and add a median line
median_score = np.median(exam_scores)
plt.axvline(median_score, color='red', linestyle='dashed', linewidth=2, label=
f'Median Score: {median_score}')

# Add a legend
plt.legend()

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



In this histogram example:

- We have a dataset of exam scores in the exam_scores list.
- We specify custom bin ranges using the bin_ranges list, which defines the edges of the bins.
- The plt.hist() function is used to create the histogram with custom bins. We set the color, edge color, and transparency (alpha) for the bars.
- We calculate the median score using np.median() and add it as a dashed red line using plt.axvline().
- Labels, a title, and a legend are added for better understanding.

2.3 Scatter Plot

Scatter plots display individual data points as dots on a two-dimensional plane. And they are used to explore relationships or correlations between two numerical variables. In this, each axis represents one variable, and the dots represent data points.

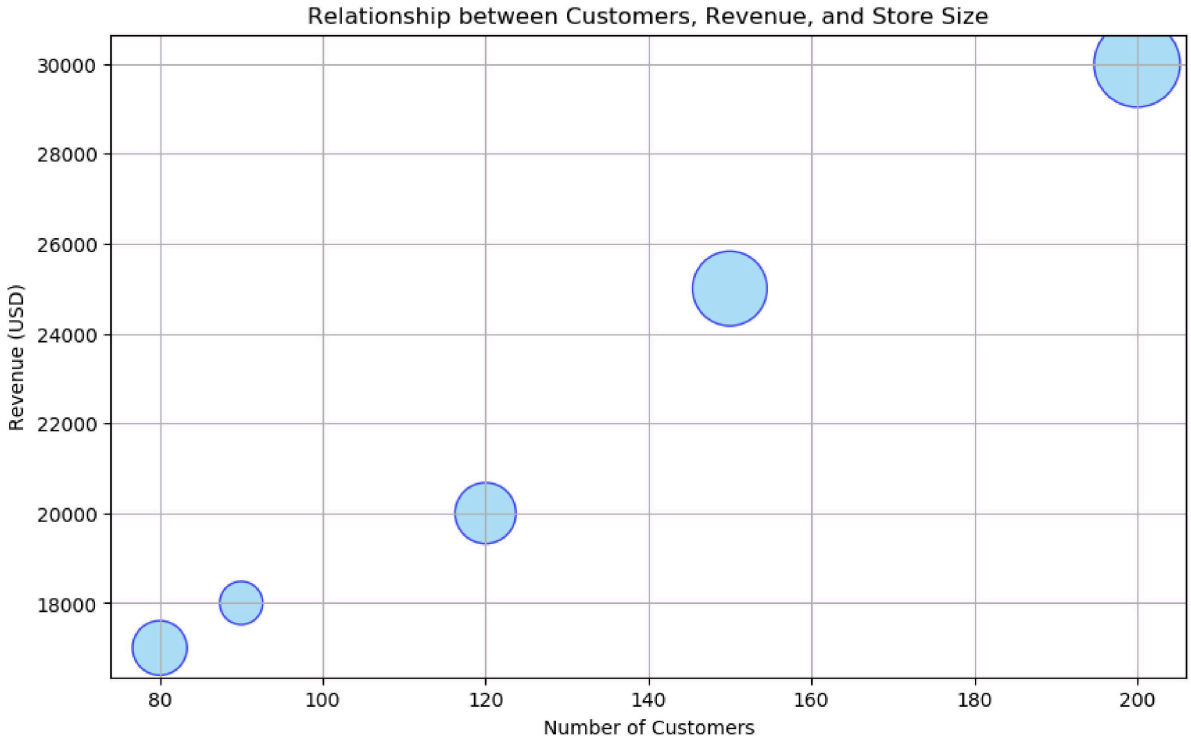
You can use plt.scatter(x,y) to generate scatter plots. To change the size of the points use the parameter s , c for the color, and marker to change the marker instead of a dot. And alpha parameter controls the intensity of the color. For the size, you can even send a different list of sizes for each point.

Few Use Cases:

1. Investigating the relationship between study hours and exam scores.
2. Analyzing the correlation between temperature and ice cream sales.

In [13]:

```
# Sample data for stores
stores = ['Store A', 'Store B', 'Store C', 'Store D', 'Store E']
customers = [120, 90, 150, 80, 200]
revenue = [20000, 18000, 25000, 17000, 30000]
store_size = [10, 5, 15, 8, 20]  # Represents the size of each store in 100sq.ft

# Scale the store sizes for point sizes in the scatter plot
point_sizes = [size * 100 for size in store_size]

# Create a scatter plot with different point sizes
plt.figure(figsize=(10, 6))  # Set the figure size
plt.scatter(customers, revenue, s=point_sizes, c='skyblue', alpha=0.7, edgecolor
s='b')

# Add labels, a title, and a legend
plt.xlabel('Number of Customers')
plt.ylabel('Revenue (USD)')
plt.title('Relationship between Customers, Revenue, and Store Size')

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



2.4 Pie Charts

Pie charts represent parts of a whole as slices of a circular pie. They are suitable for showing the composition of a single categorical variable in terms of percentages. But this won’t look good when there are more than six categories as they get clumsy, in such cases horizontal bar might be preferred.

Use the command plt.pie(x,labels=your_category_names, colors=desired_colors_list) , if you have a desired colors list, you can provide that and you can also change the edge color of the pie chart with the parameter wedgeprops= {'edgecolor':your_color} .

We can also highlight particular segments using explode parameter by passing a tuple where each element is the amount by which it has to explode. And autopct parameter enables you to choose how many values after the decimal are to be shown in the plot.

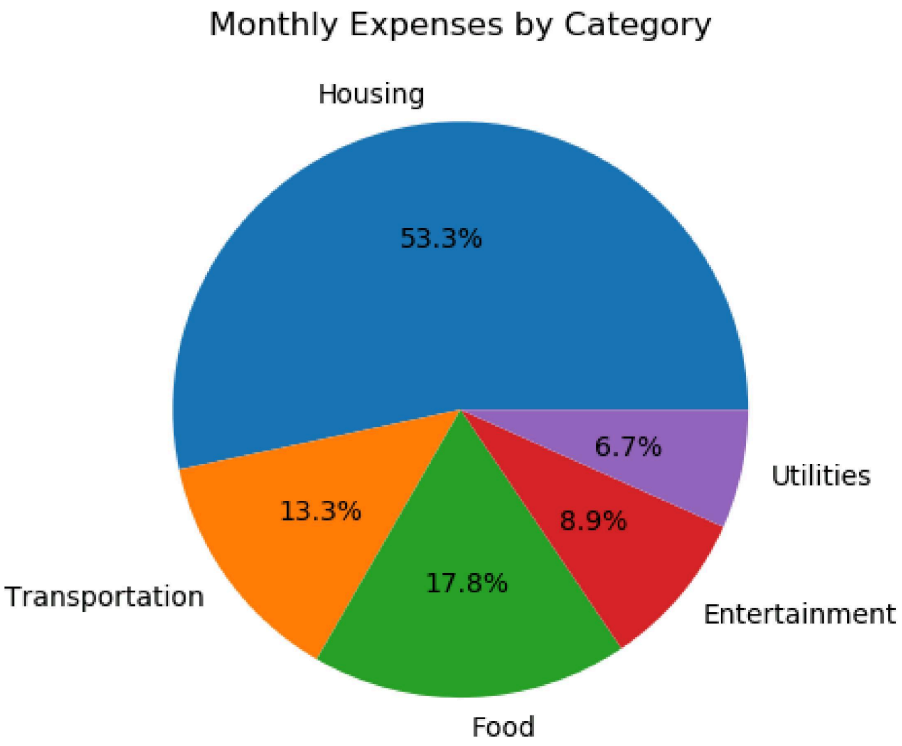
Few Use Cases:

1. Displaying the distribution of a budget by expense categories.
2. Showing the market share of various smartphone brands.

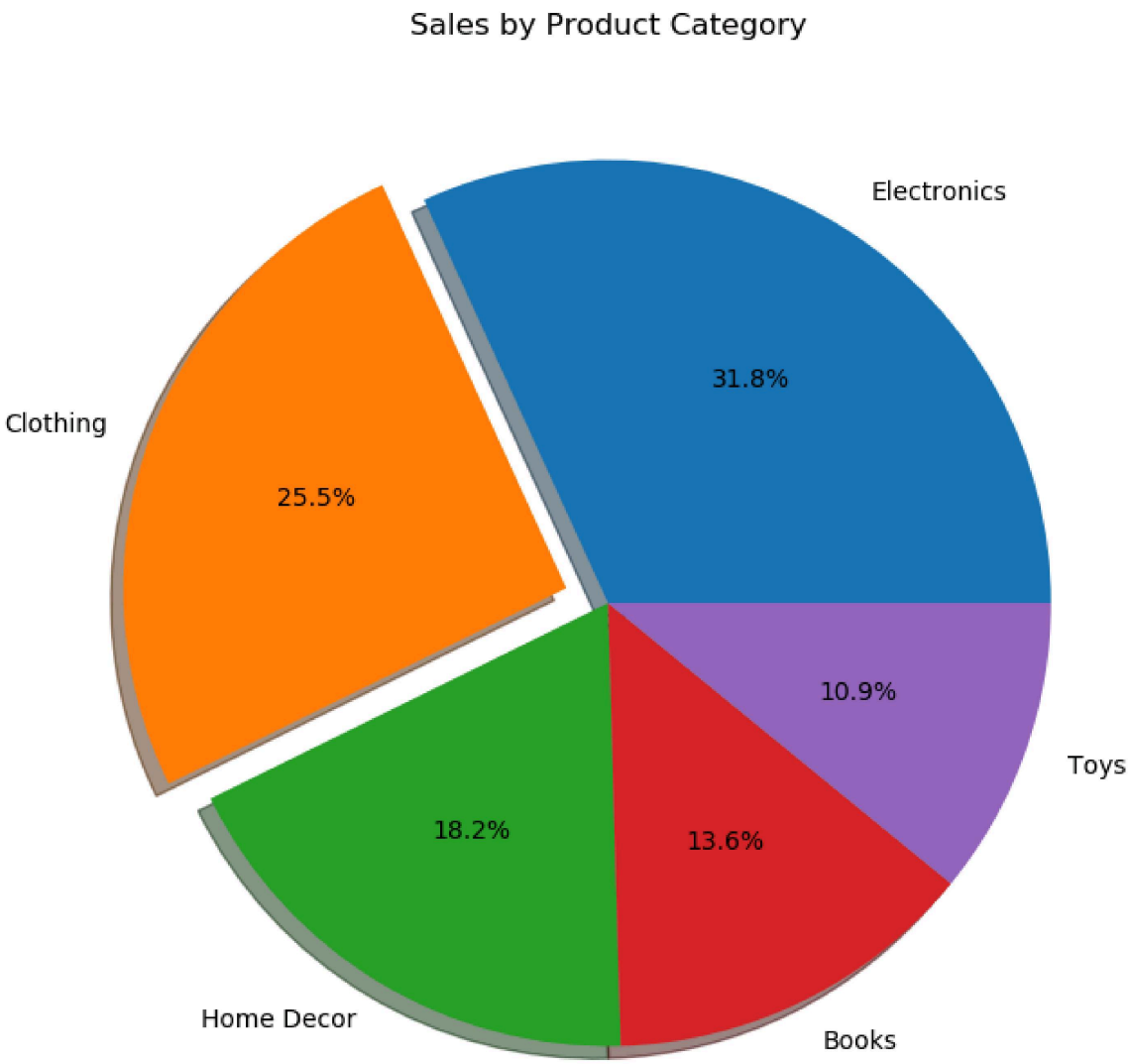
Eg: Exploding a particular segment for better storytelling during presentations.

In [14]:

```
# Sample data (expenses)
categories = ['Housing', 'Transportation', 'Food', 'Entertainment', 'Utilities']
expenses = [1200, 300, 400, 200, 150]

# Create a pie chart
plt.pie(expenses, labels=categories, autopct='%1.1f%%')

# Add a title
plt.title('Monthly Expenses by Category')

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



In [15]:
```
# Product categories
categories = ['Electronics', 'Clothing', 'Home Decor', 'Books', 'Toys']

# Sales data for each category
sales = [3500, 2800, 2000, 1500, 1200]

# Explode a specific segment (e.g., 'Clothing')
explode = (0, 0.1, 0, 0, 0)  # The second value (0.1) is the amount by which the segment 'Clothing' will be exploded

# Create a pie chart with explode and shadow
plt.figure(figsize=(8, 8))  # Set the figure size
plt.pie(sales, labels=categories, explode=explode, shadow=True, autopct='%1.1f%%')
plt.title('Sales by Product Category')

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



2.5 Box plot

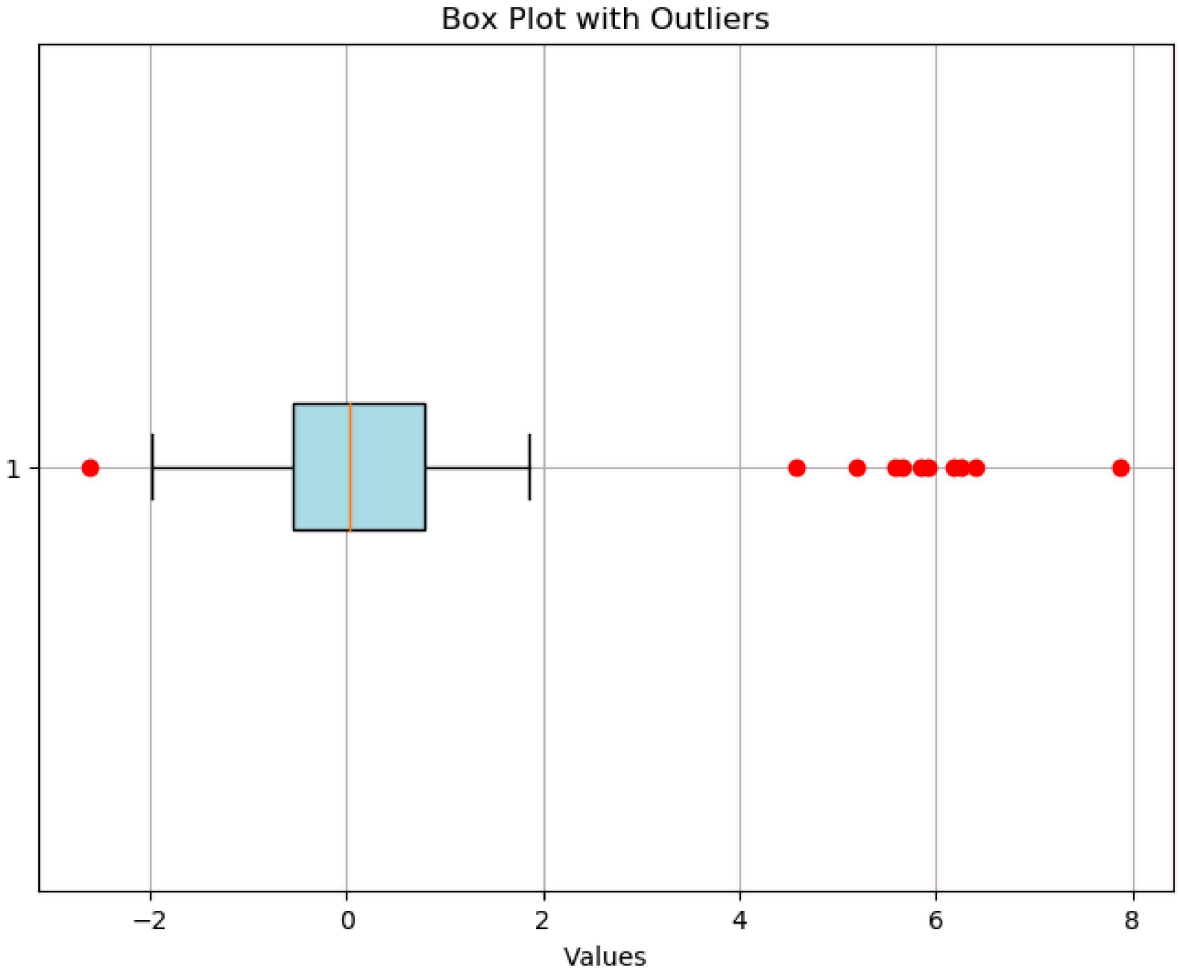
Box plots are the ones that look complicated, right? Simply put they summarize the distribution of numerical data by displaying quartiles, outliers, and potential skewness. They provide insights into data spread, central tendency, and variability. Box plots are especially useful for identifying outliers 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.

Few Use Cases:

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

In [16]:
```
# Generate random data with outliers
np.random.seed(42)
data = np.concatenate([np.random.normal(0, 1, 100), np.random.normal(6, 1, 10)])

# 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': 'lightblue'}, flierprops={'marker': 'o', 'markerfacecolor': 'red', 'markeredgecolor': 'red'})

# 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()
```



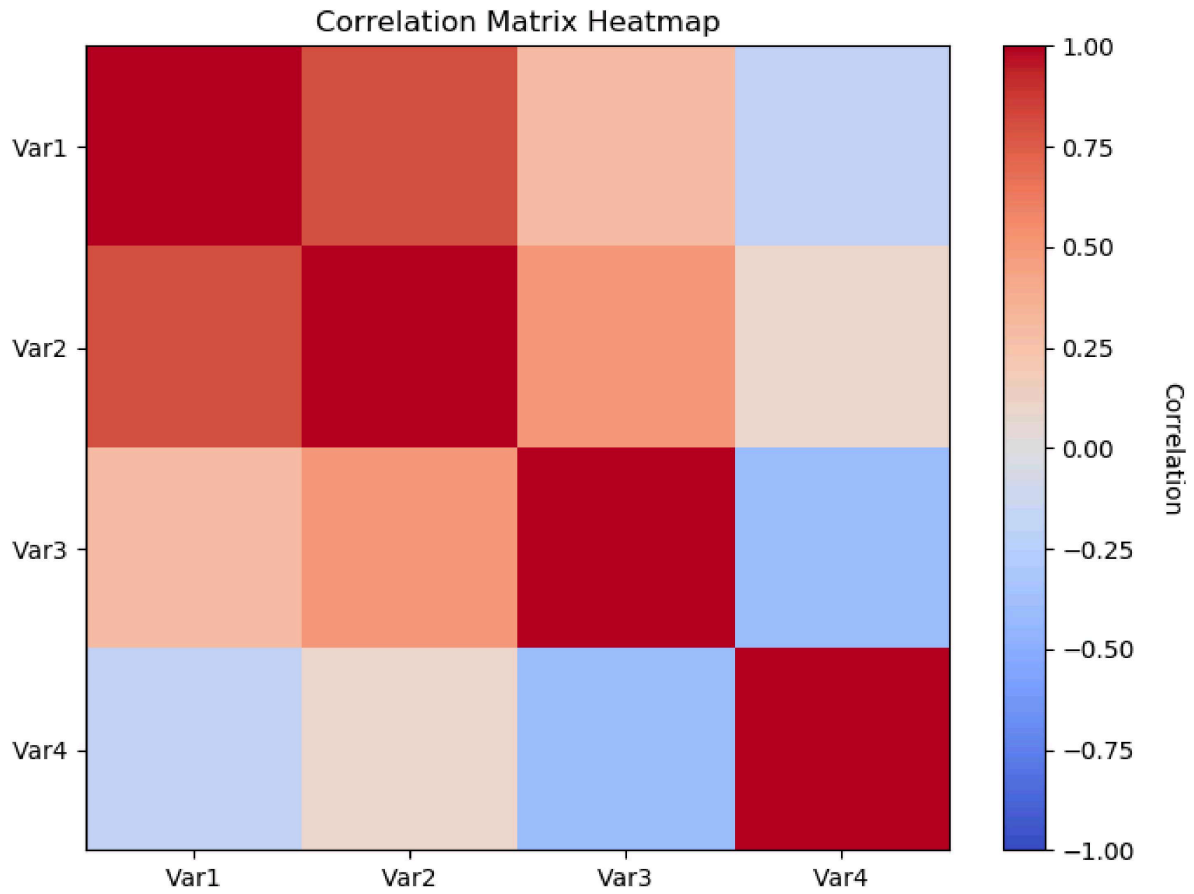
2.6 Displaying Images, Array Visualization

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 [17]:

```
# 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='auto',
origin='upper')

# 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()
```



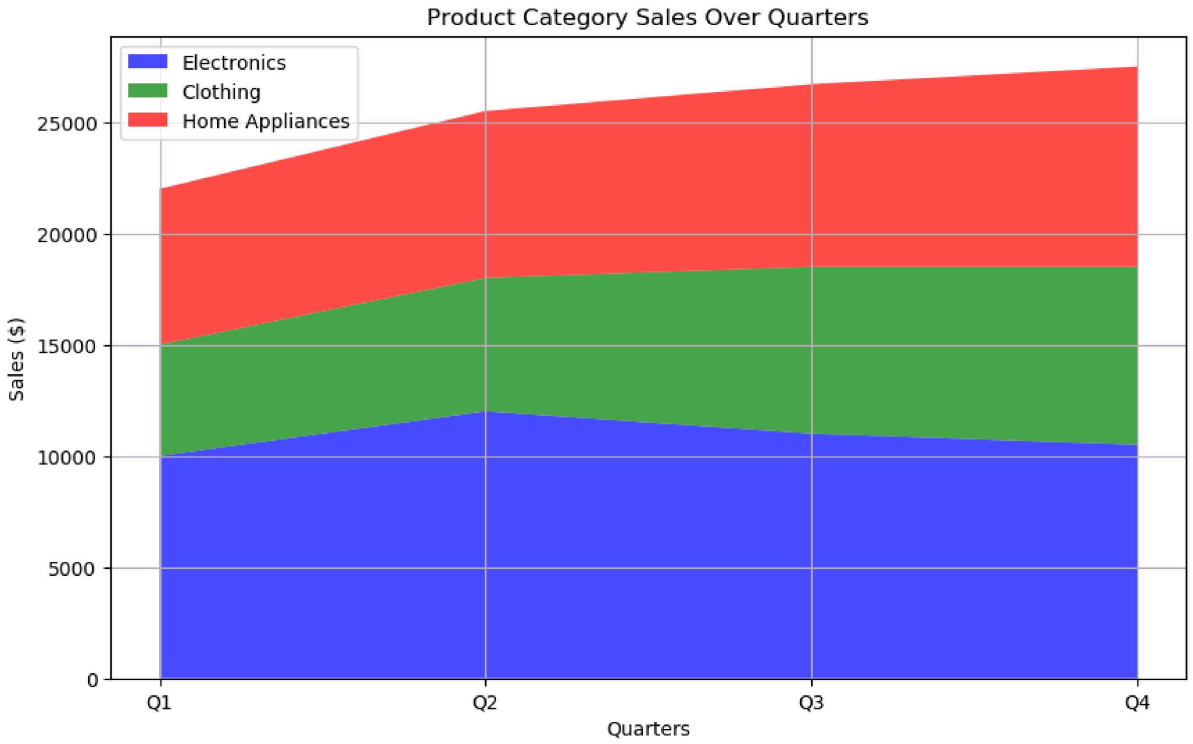
2.7 Stack Plot

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 want!

In [18]:

```
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=['Electro
nics', 'Clothing', 'Home Appliances'],
              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()
```



In this code:

We have sample data for three product categories: electronics, clothing, and home appliances, with sales data for each quarter (Q1, Q2, Q3, Q4).

We use plt.stackplot() to create the stack plot. Each category's sales are represented as a separate argument, and we provide labels and colors for better visualization.

Labels, legends, and a title are added to enhance the plot's readability and context.

The resulting stack plot visually shows how each product category's sales contribute to the total sales over the four quarters of the year. It's a helpful tool for business analysts to track and understand category performance.

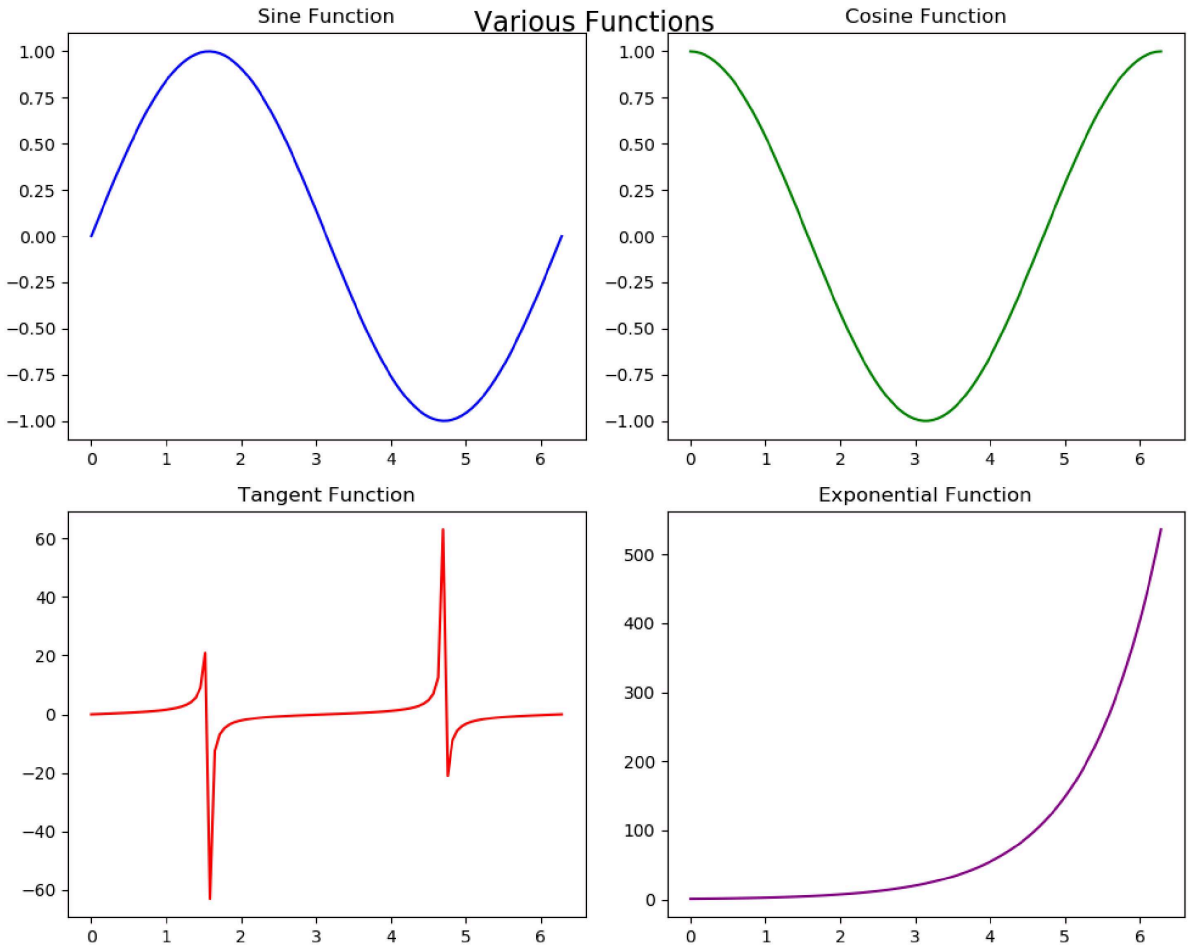
3-Multiple Subplots

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.

3.1 Creating Multiple Plots in a single figure

In [19]:
```
# 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()
```

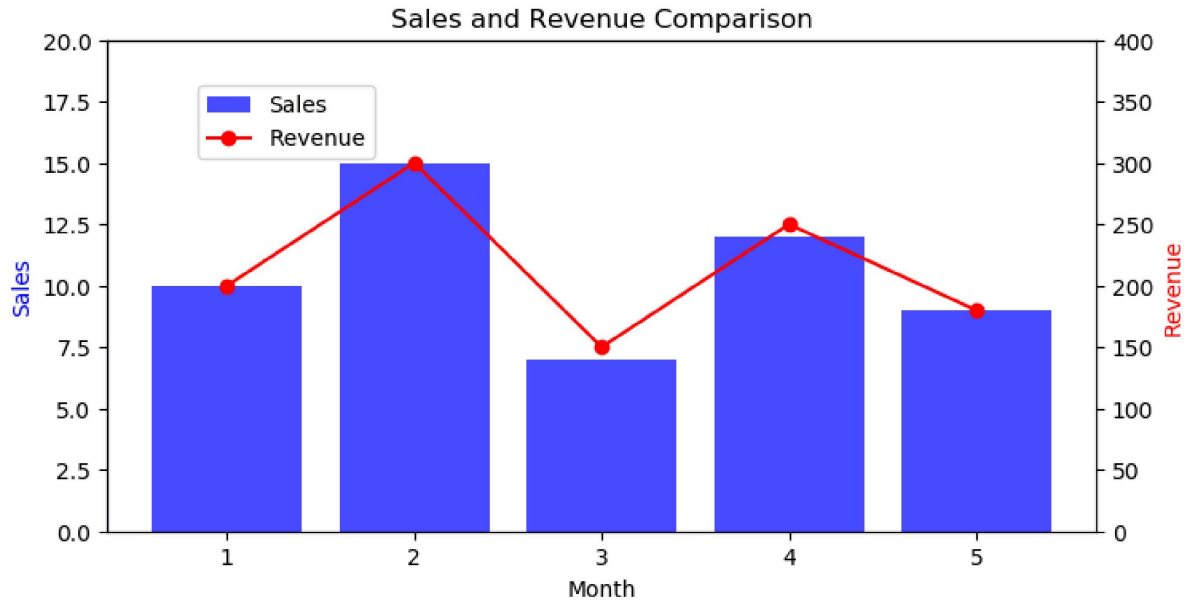


3.2 Combining Different Types of plots

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 [20]:
```
# 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()
```



4-Advanced Features

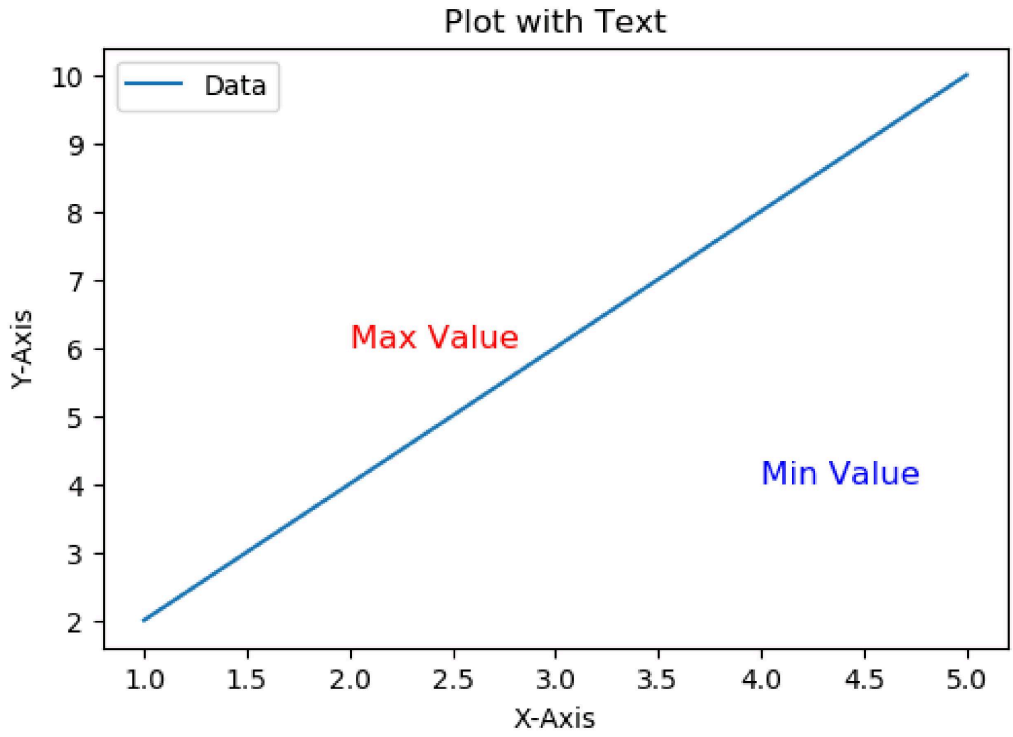
4.1 Annoate and Text for the plots

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()

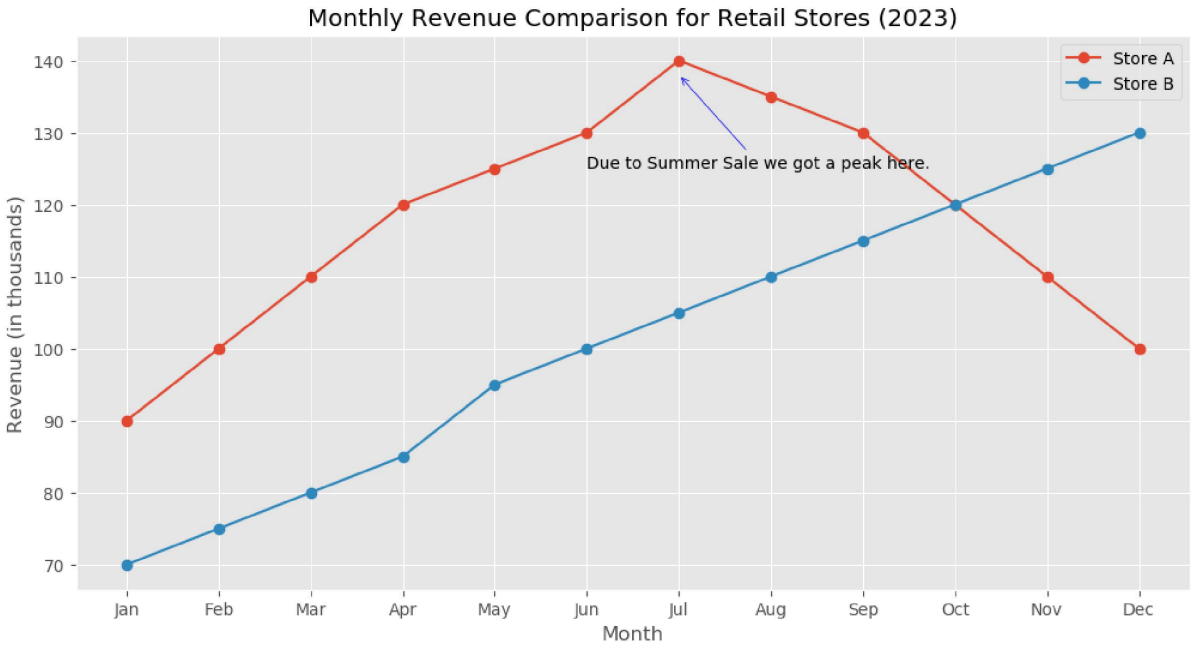
In [21]:

```
# 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()
```

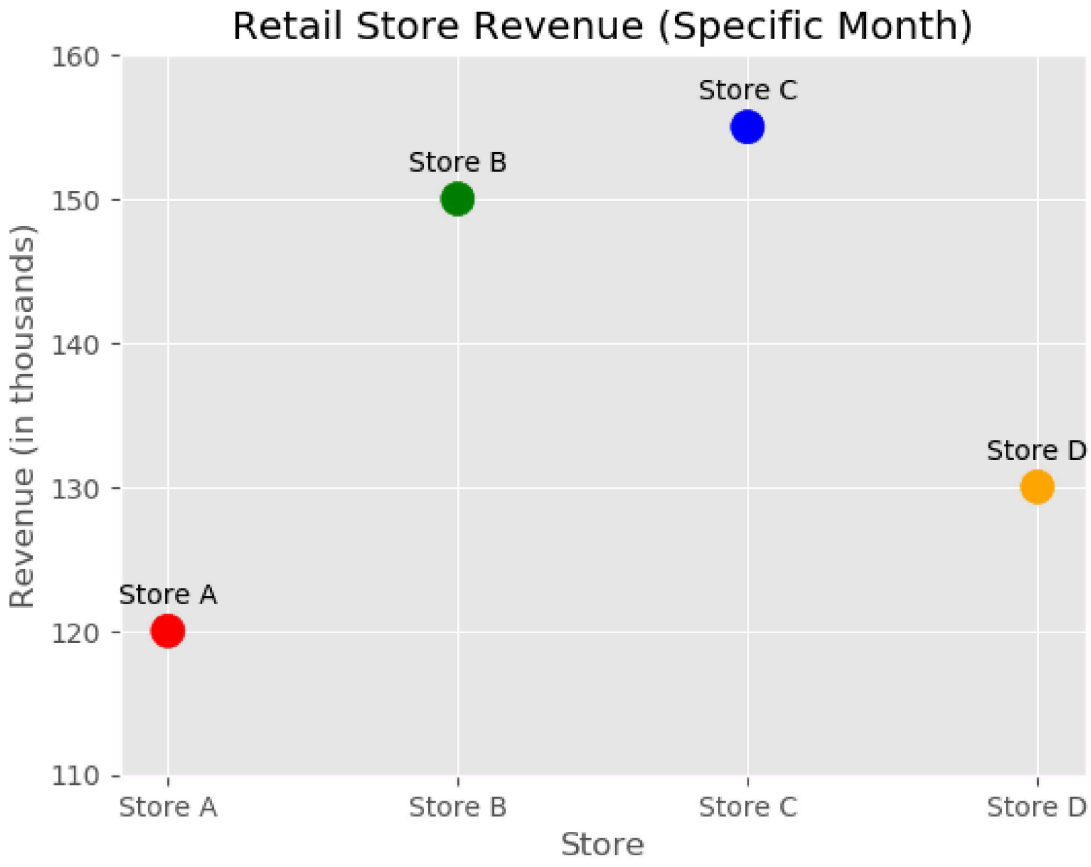


In [22]:

```
# Sample data for retail shop revenue
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec']
store_locations = ['Store A', 'Store B']
revenue_data = np.array([[90, 100, 110, 120, 125, 130, 140, 135, 130, 120, 110,
100],
                         [70, 75, 80, 85, 95, 100, 105, 110, 115, 120, 125, 13
0]])

# 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), xytext=
('Jun', 125),
             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 [23]:
```
# 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='Reven
ue')

# Label each store
for i, store in enumerate(store_names):
    plt.annotate(store, (store, revenue_data[i]), textcoords="offset points", xy
text=(0, 10), ha='center')
    # 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 [24]: plt.style.use('default')
```

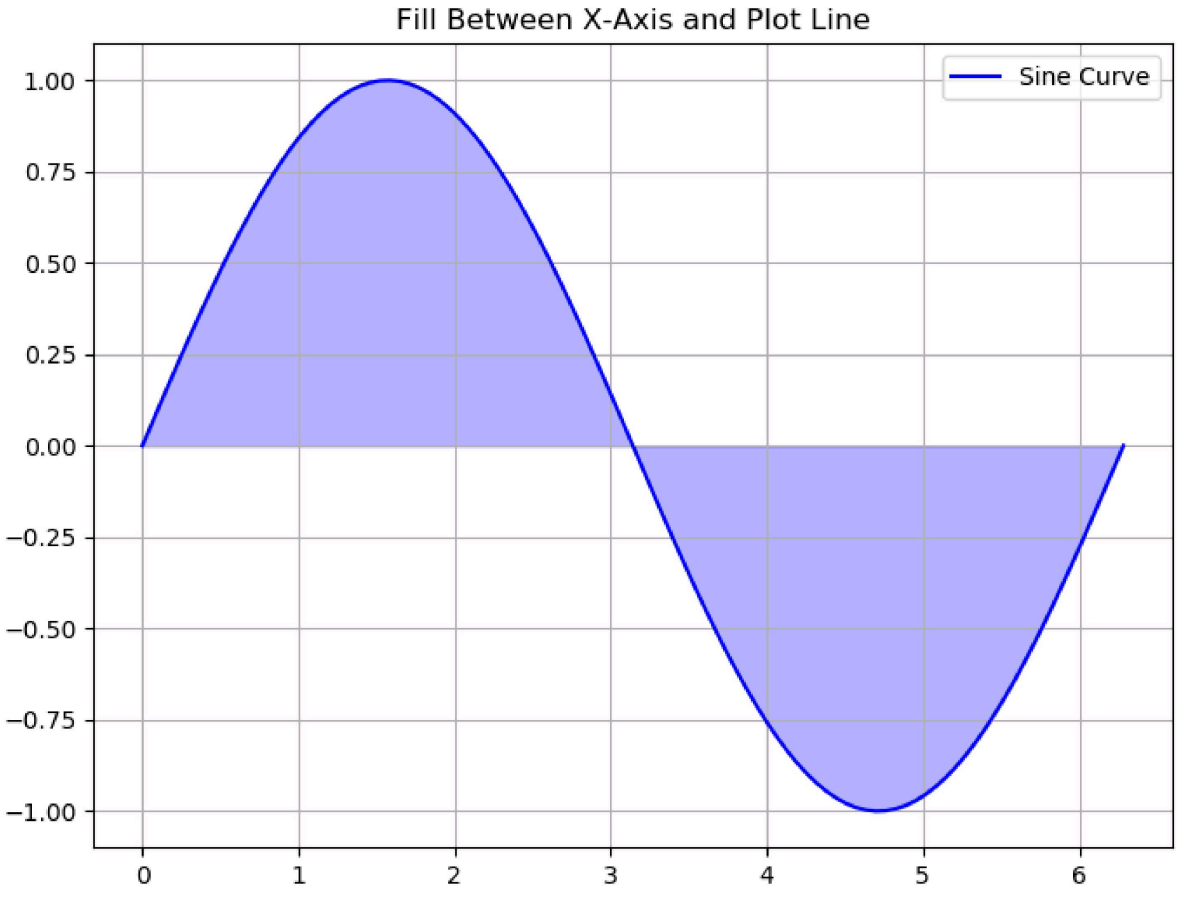
4.2 Fill the Area Between the plots

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 [25]:
```
# 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()
```

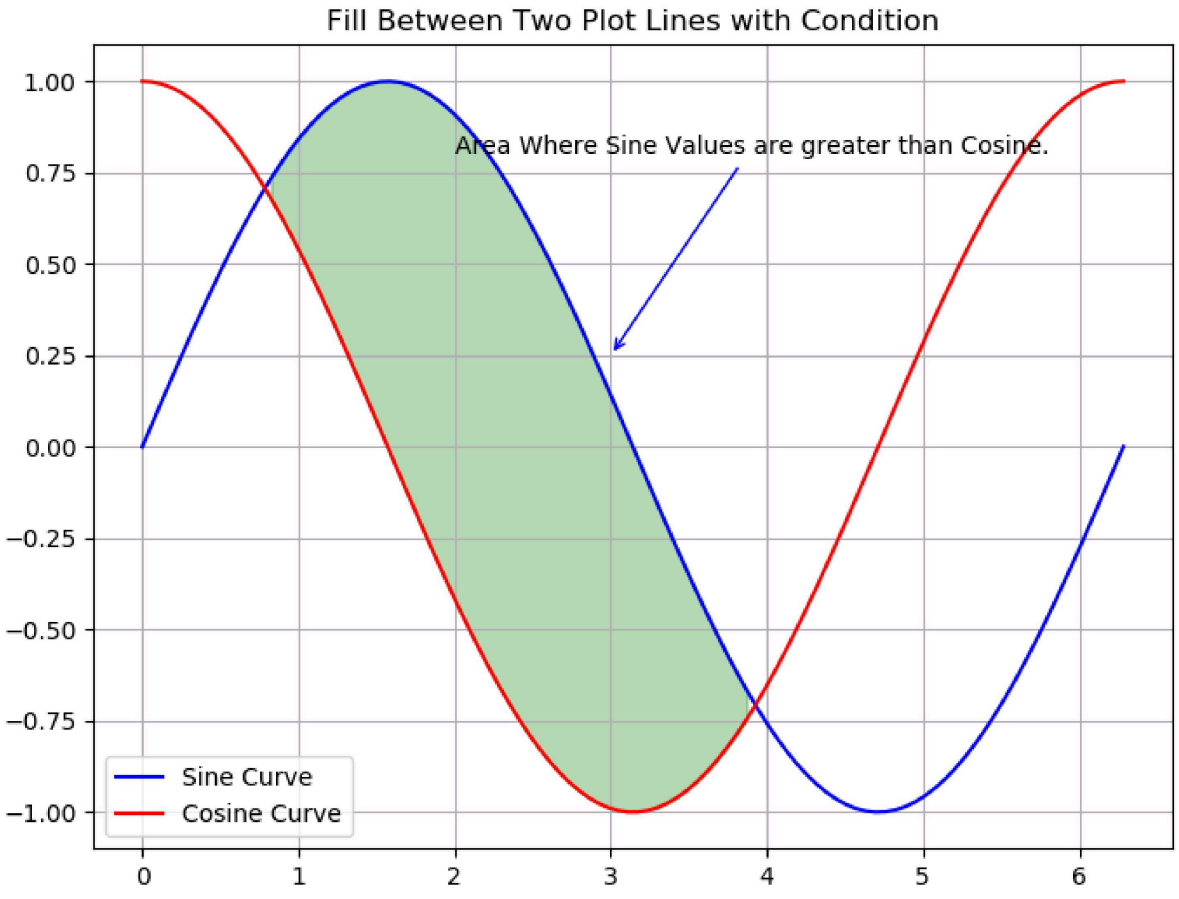


In [26]:

```
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()
```

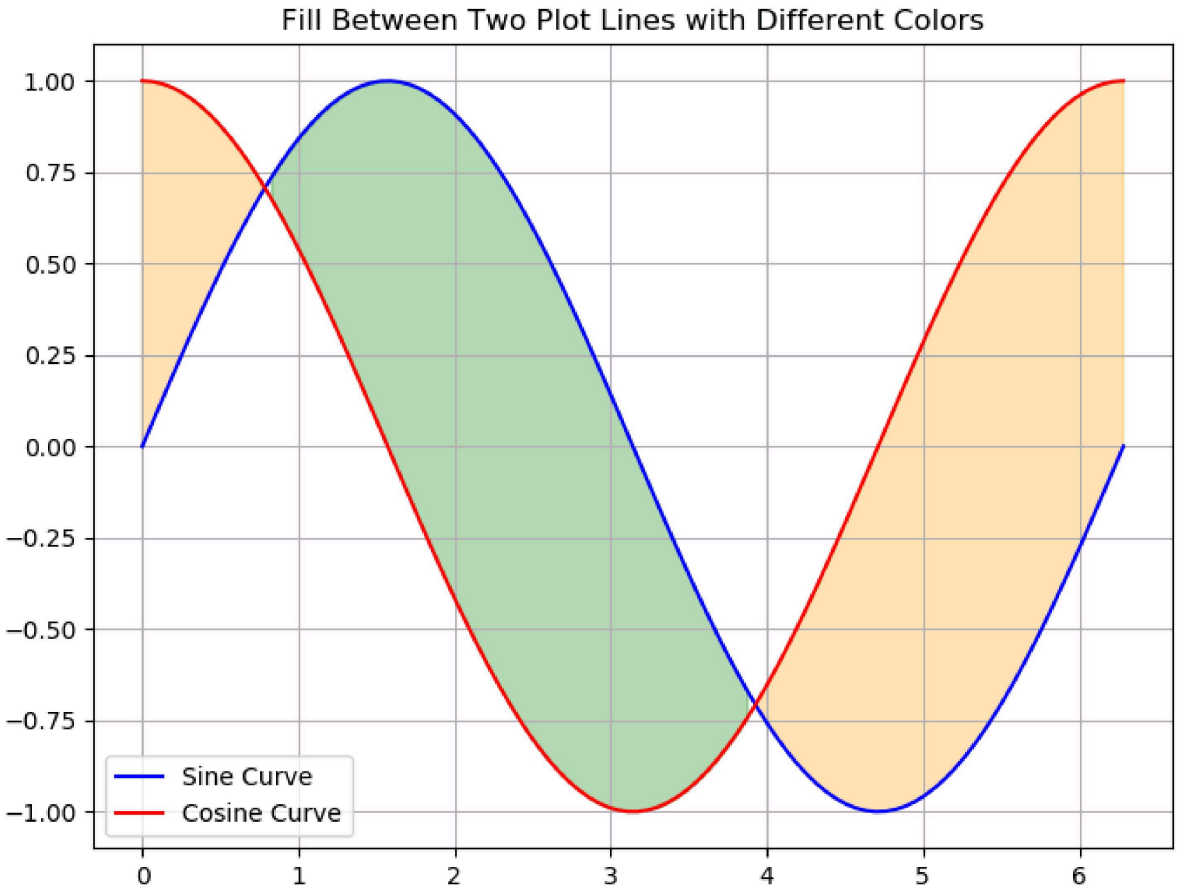
In [27]:

```
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), xy
text=(2, 0.80),
             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 [28]:

```
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()
```



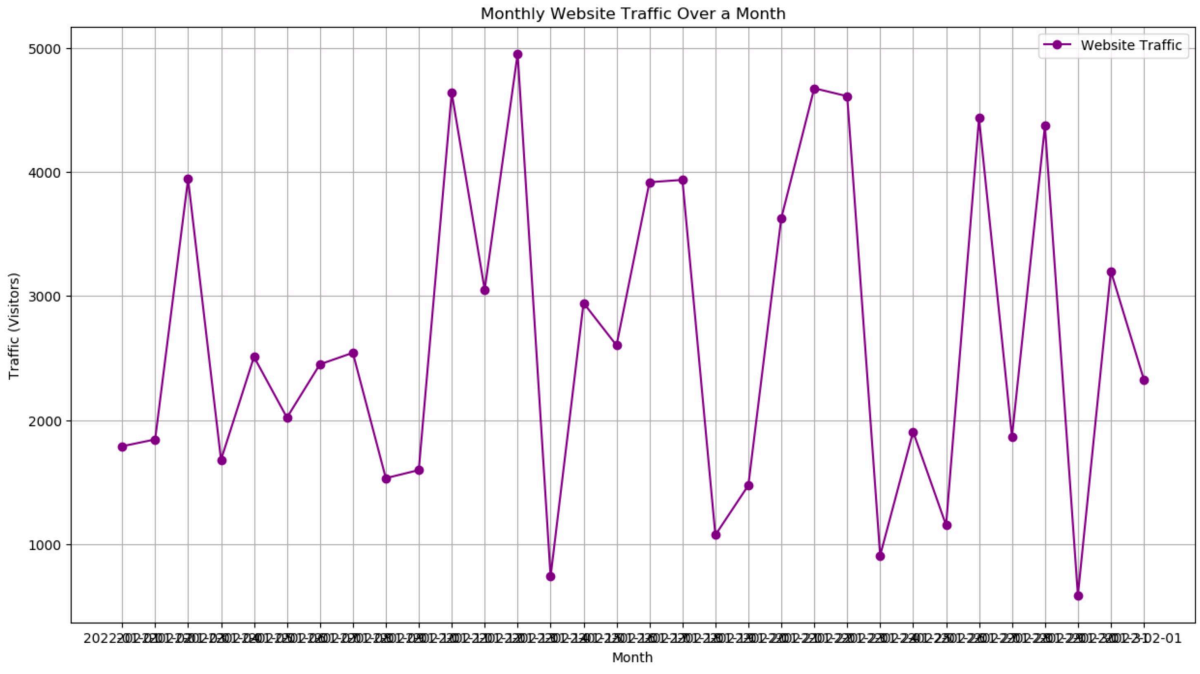
4.3 Plotting Time Series Data

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.

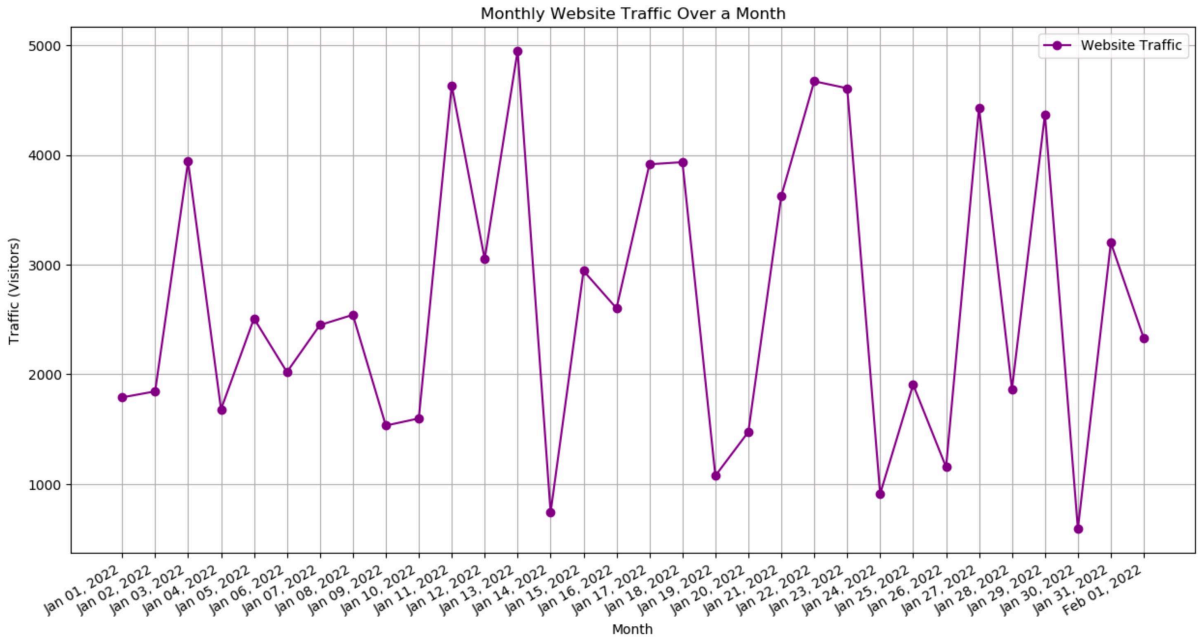
In [29]:

```
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_dat
a})

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

# Now, let's create a time series plot to visualize the monthly website traffic.
plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label='Webs
ite Traffic', color='purple', linestyle='-')

# 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 [30]:
```
# Set the figure size
plt.figure(figsize=(15, 8))

# Now, let's create a time series plot to visualize the monthly website traffic.
plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label='Webs
ite Traffic', color='purple', linestyle='-')

# 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()
```



4.4 3D Plots

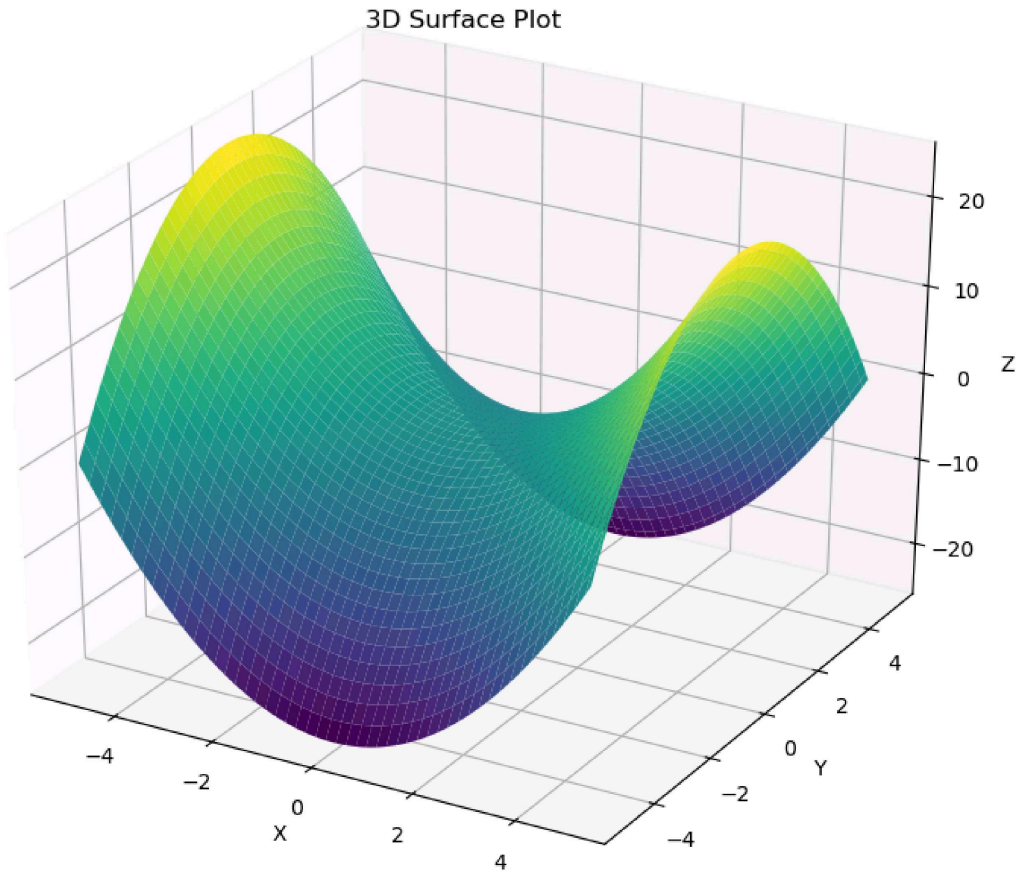
Creating 3D plots using Matplotlib involves using the mpl_toolkits.mplot3d toolkit, which provides functions for creating various types of 3D visualizations. you need to import the Axes3D to visualize the plots in 3D with the following command from mpl_toolkits.mplot3d import Axes3D .

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.

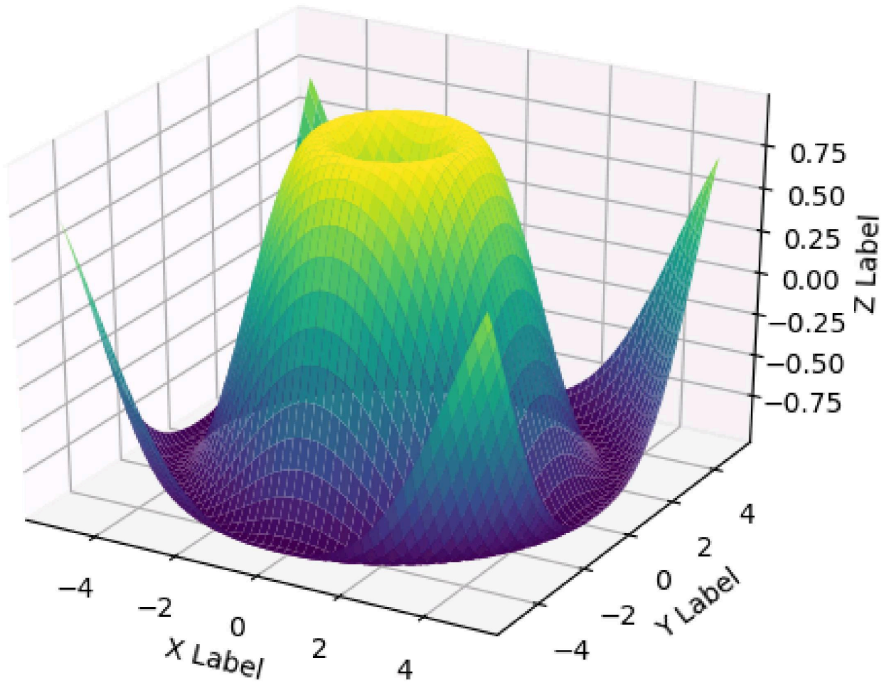
In [31]:

```
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()
```

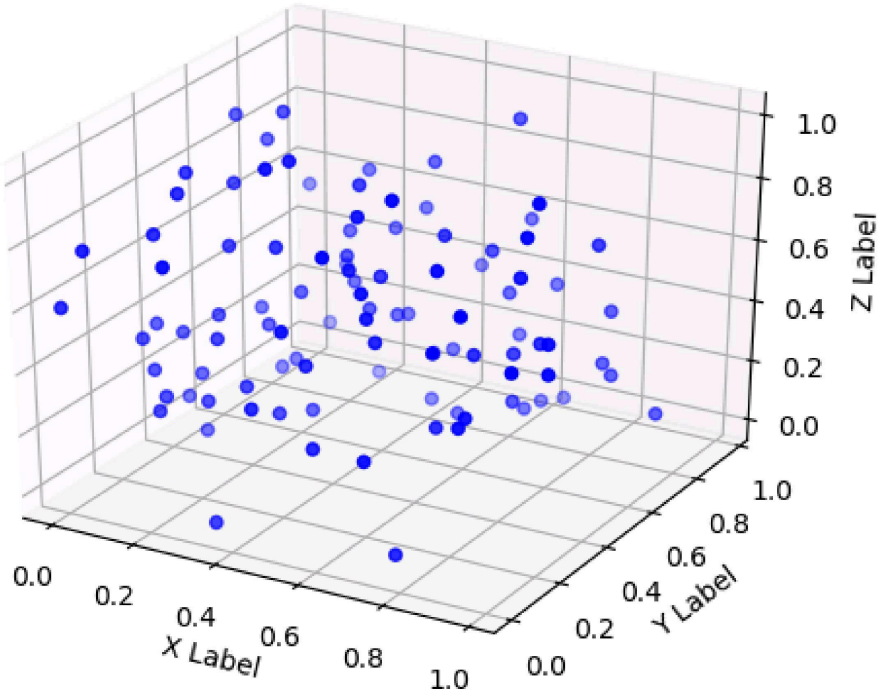


In [32]:

```
# 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 [33]:
```
# 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()
```



4.5 Animation

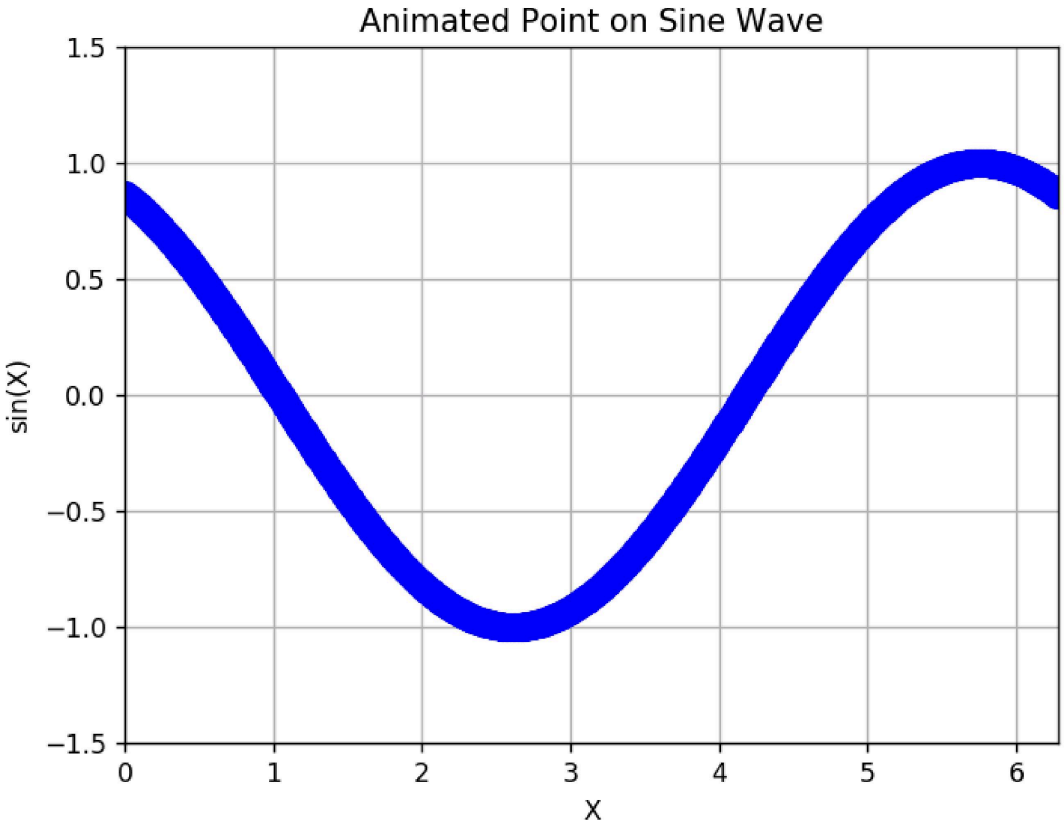
It would be nice to rotate, zoom, and hover to see the location of the above 3D plot, right? Guess what, we can actually do that in one line! Use the command %matplotlib notebook in your Jupyter Notebook to make the plot interactive. If you want to change it back to static plots use %matplotlib inline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

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 figure object, so as it happens continuously we get an animation plot.

In [34]:

```
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, interva
l=50)

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

plt.show()
```



As you continue your journey in the world of data science and analysis, remember that Matplotlib provides the foundation for creating compelling visuals that tell your data's story, so just play with it. I hope you find this guide useful.

Throughout this two-part series, we have used Numpy often to create test data, if you are curious about learning more about Numpy, check out the below article.

Mastering Numpy - https://medium.com/python-in-plain-english/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84 (https://medium.com/python-in-plain-english/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84)

Mastering Pandas - https://medium.com/python-in-plain-english/pandas-demystified-a-comprehensive-handbook-for-data-enthusiasts-part-1-136127e407f (https://medium.com/python-in-plain-english/pandas-demystified-a-comprehensive-handbook-for-data-enthusiasts-part-1-136127e407f)

Happy Learning!

RaviTeja G

Follow me for more Detailed Notes and Articles Related to Data Science :)