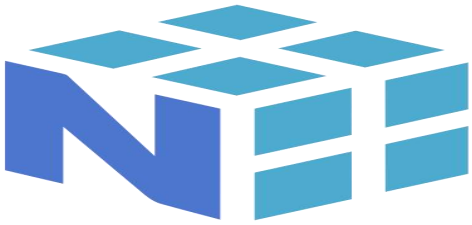
python

Stay
Refreshed



NumPy

100%

Matplotlib Graphical Functions

Guide to optimise your
Visualization skills using
matplotlib functions

Solomon Akintola

Data visualisation describes any effort to help people understand the significance of data by placing it in a visual context.

Using Matplotlib is a powerful technique that helps people understand the significance of data by presenting it in a visual context. By leveraging visual elements like charts, graphs, and plots, data visualization enables us to explore patterns, trends, and relationships within the data more effectively than just examining raw numbers or tables.

5 benefits of data visualization using Matplotlib:

1. Clarity and Simplification: Visualizing data allows us to simplify complex information and present it in a more digestible format. By using graphical representations, we can quickly grasp patterns, outliers, and other important insights that might not be immediately apparent in raw data.
2. Patterns and Trends Identification: With data visualization, we can identify patterns and trends more easily. Line plots, bar charts, scatter plots, and other visualizations help us identify correlations, variations, and relationships between different variables, enabling us to make data-driven decisions.
3. Data Exploration and Analysis: Visualizing data provides an interactive and exploratory environment for data analysis. Matplotlib's interactive features allow users to zoom in, pan, and interact with the plots to explore specific data points or regions of interest, facilitating a deeper understanding of the data.
4. Communication and Presentation: Visualizing data makes it easier to communicate findings and insights to others. By using compelling visual representations, we can effectively convey complex information to stakeholders, clients, or colleagues who may not have a technical background. Visualizations help tell a story and facilitate data-driven discussions.
5. Decision Making and Planning: Data visualization assists in making informed decisions and strategic planning. By visualizing data, we can identify trends, spot outliers, and uncover hidden patterns that influence decision-making processes across various domains, such as business, finance, healthcare, and more.

Matplotlib, as a popular data visualization library in Python, provides a wide range of plot types, customization options, and tools to create visually appealing and informative visualizations. It empowers analysts, data scientists, and developers to present data in a compelling manner, enabling better insights and informed decision-making.

Effective data visualization is not just about creating visually appealing plots, but about conveying the story within the data, making complex information more accessible, and facilitating data-driven understanding and decision-making.

1. Line Plot ('plot()'):

Line plot is a type of plot that represents data points connected by straight lines. It is commonly used to visualize the trend, patterns or relationship between two continuous variables. The x-axis represents the independent variable, and the y-axis represents the dependent variable.

Line plots are versatile and widely used for visualizing various types of data. They provide a clear and concise representation of trends and patterns, making them valuable tools for data analysis and communication.

Line plots are suitable for various types of data, including:

1. Time Series Data: Line plots are commonly used to display trends over time, such as stock prices, temperature variations, or population growth.

2. Scientific Data: Line plots can be used to represent scientific data, such as experimental results, measurements, or sensor readings.

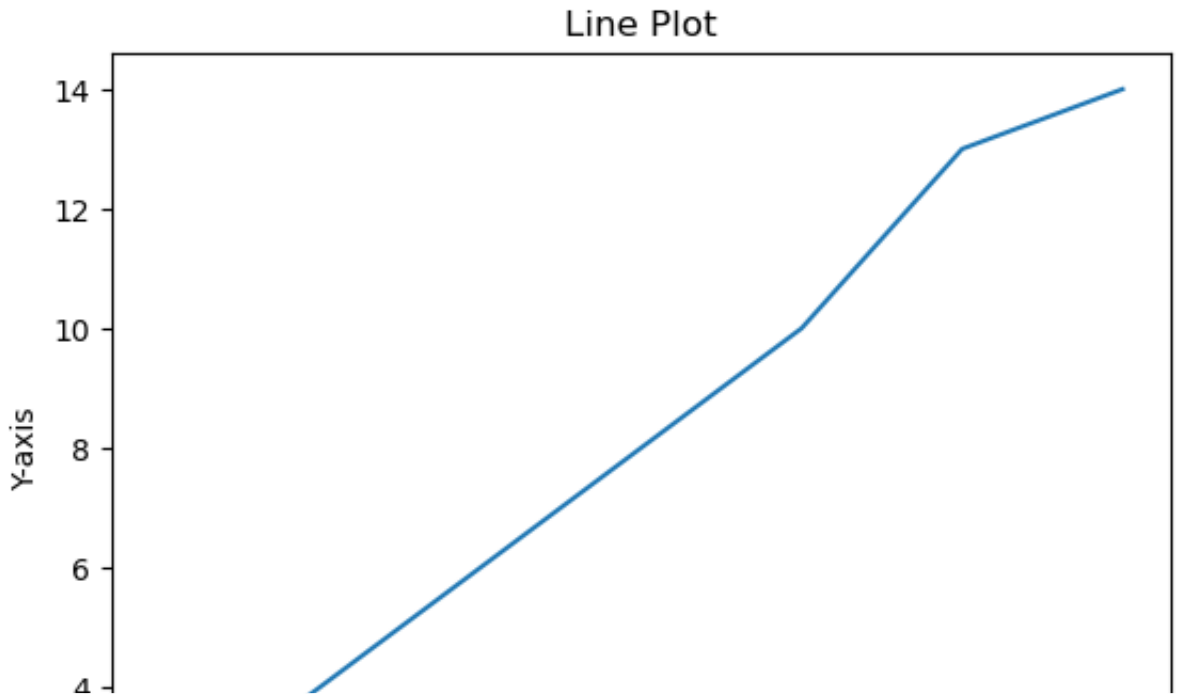
3. Financial Data: Line plots are often employed to show financial data, such as stock market trends, exchange rates, or portfolio performance.

4. Statistical Data: Line plots can be used to visualize statistical data, such as trends in survey responses, sales figures, or population statistics.

Example 1:

In this example, we have two lists `x` and `y` representing the x-axis and y-axis values, respectively. The `plot()` function is used to create the line plot by passing in the `x` and `y` values. The plot is then customized using the `plt_xlabel()`, `plt_ylabel()`, and `plt_title()` functions. Finally, `plt.show()` is called to display the plot.

In [1]:
```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5, 6, 7]
y = [2, 4, 6, 8, 10,13, 14]
plt.plot(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Plot")
plt.show()
```



2. Scatter Plot (`scatter()`):

Matplotlib scatter plot is a type of plot used to visualize the relationship or distribution between two numeric variables. It displays data points as individual markers on a two-dimensional coordinate system, with one variable represented on the x-axis and the other on the y-axis. Each data point is represented by a marker, typically a circle, whose position on the plot corresponds to the values of the two variables for that data point.

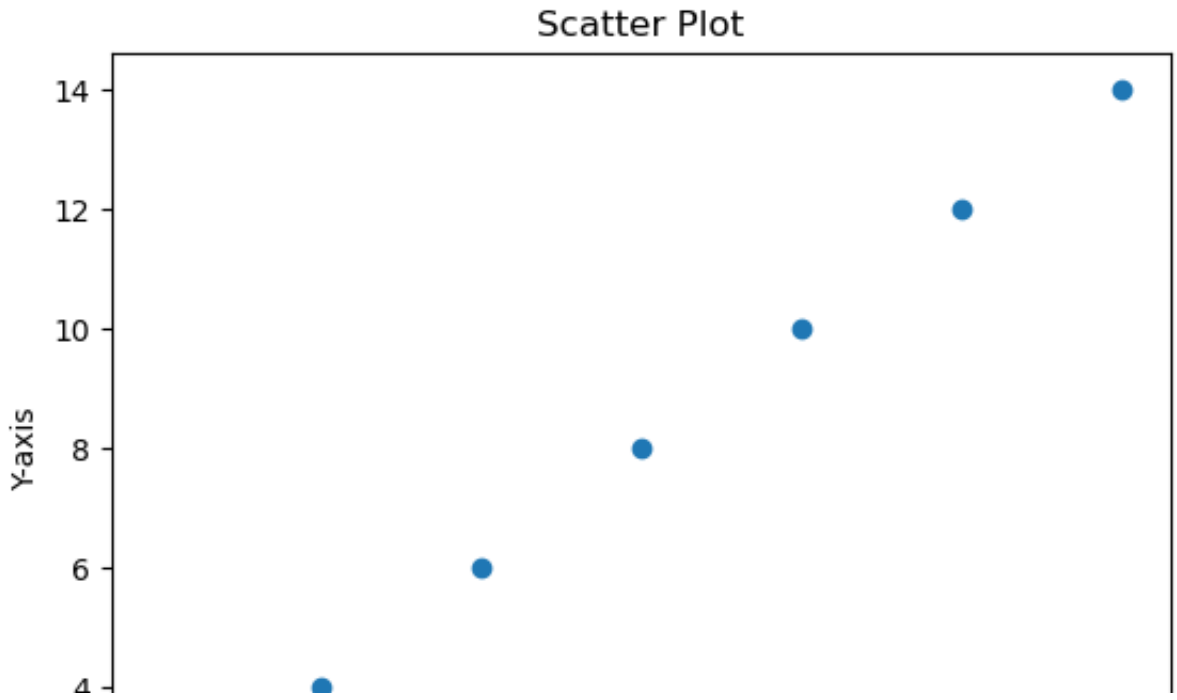
The scatter plot is particularly useful when you want to examine the correlation or pattern between two continuous variables. It helps identify trends, clusters, outliers, or any other relationship between the variables.

You can use various types of data to plot a scatter plot, such as:

- Examining the relationship between a student's study time and their exam score.
- Analyzing the correlation between a company's advertising expenditure and its sales revenue.
- Investigating the connection between temperature and ice cream sales.

Example 1:

In [2]:
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5, 6, 7]
y = [2, 4, 6, 8, 10, 12,14]
plt.scatter(x, y)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scatter Plot")
plt.show()
```



3. Bar Plot (`bar()`):

Matplotlib's bar plot is a type of chart that represents data using rectangular bars. It is commonly used to display and compare categorical data or numerical data grouped into categories. Bar plots are particularly useful for visualizing frequency, count, or aggregated data.

In a bar plot, the height of each bar corresponds to the value of the data being represented. The bars can be oriented vertically (vertical bar plot) or horizontally (horizontal bar plot). The length or width of the bars represents the magnitude or quantity of the data being plotted.

Line plots are suitable for various types of data, including:

1. Categorical Data:
    - Bar plots are commonly used to visualize categorical data, such as different categories or groups. The bars represent the categories, and the height or length of the bars represents the count, frequency, or any other aggregated value associated with each category.

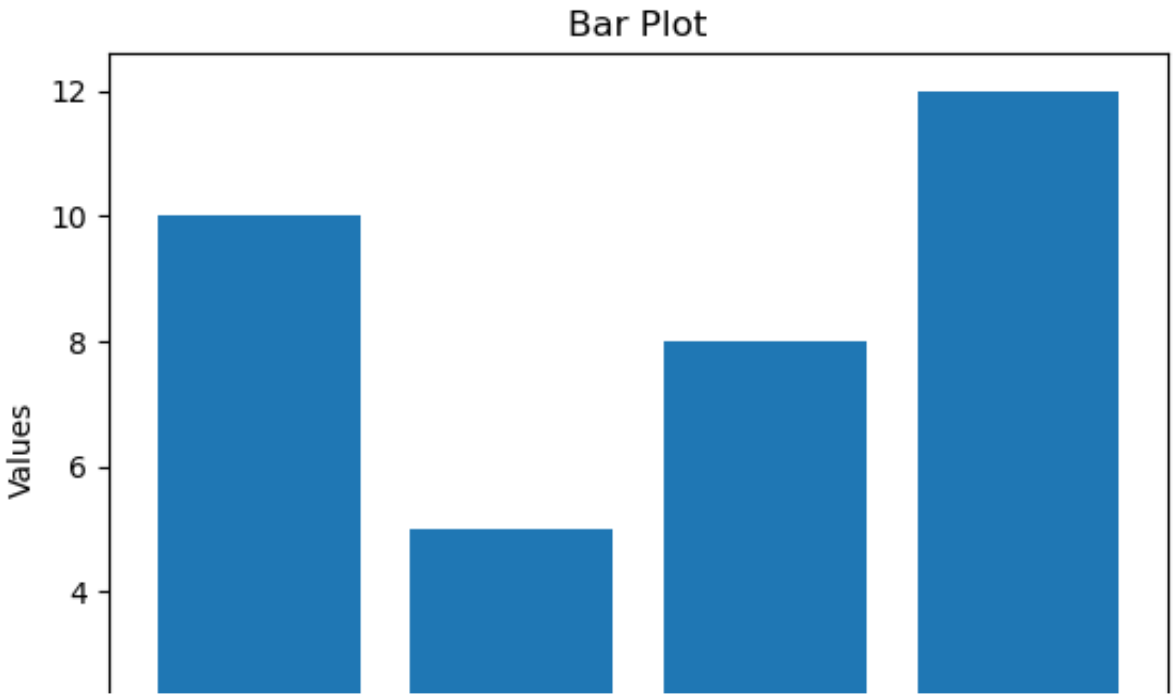
2. Numerical Data:
    - Bar plots can also be used to represent numerical data that has been grouped into categories or intervals. The bars then represent the categories or intervals, and the height or length of the bars corresponds to the aggregated value of the numerical data within each category or interval.

3. Histograms:
    - A special case of bar plots is histograms, which are used to represent the distribution of continuous numerical data. The bars in a histogram represent intervals or bins of values, and the height of each bar represents the frequency or count of data points falling within that interval.

4. Time Series Data:
    - Bar plots can be used to visualize time series data, where the categories represent different time points or periods, and the height or length of the bars represents the values or metrics associated with each time point.

Example 1:

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

x = ['HP', 'Lenovo', 'Dell', 'Apple']
y = [10, 5, 8, 12]
plt.bar(x, y)
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Plot")
plt.show()
```



4. Horizontal Bar Plot (`barh()`):

A horizontal bar plot, also known as a barh plot, is a type of visualization in Matplotlib where bars are drawn horizontally instead of vertically. It is useful for comparing the magnitude of different categories or variables.

In a horizontal bar plot, the y-axis represents the categories or variables, while the length of the bars represents the magnitude or value associated with each category. The longer the bar, the higher the value.

The data used to plot a horizontal bar plot can be categorical or numerical.

1. Categorical Data: The y-axis can represent different categories, such as countries, cities, or product names. The length of the bars can represent a count, frequency, or any other metric associated with each category.

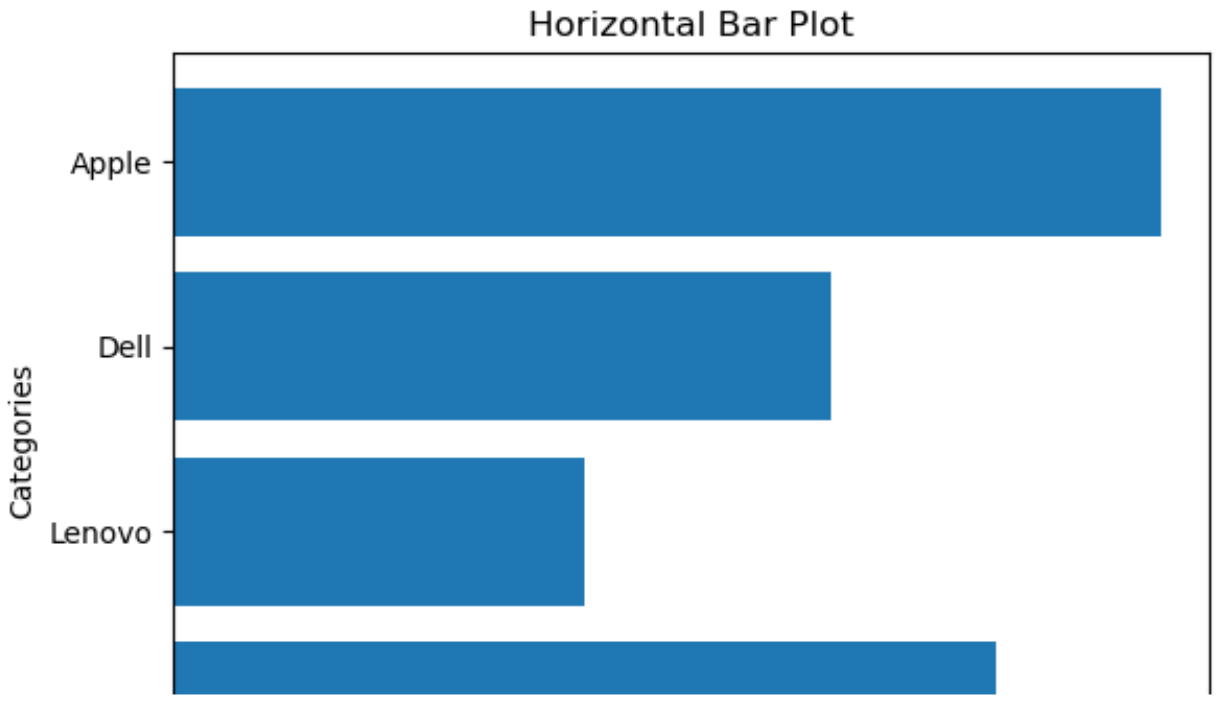
2. Numerical Data: The y-axis can represent continuous numerical values, such as years, temperatures, or sales figures. The length of the bars can represent the magnitude or value associated with each numerical value.

3. Grouped Data: Multiple bars can be plotted for each category or variable, representing different sub-categories or sub-variables. This is useful for comparing multiple metrics within each category.

Horizontal bar plots are particularly effective when you have long category names or when you want to emphasize the differences between categories by having the bars extend horizontally.

Example 1:

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

y = ['HP', 'Lenovo', 'Dell', 'Apple']
x = [10, 5, 8, 12]
plt.barh(y, x)
plt.xlabel("Values")
plt.ylabel("Categories")
plt.title("Horizontal Bar Plot")
plt.show()
```



5. Histogram Plot (`hist()`):

Histogram is a graphical representation that displays the distribution of a dataset using rectangular bars. It provides insights into the underlying frequency or count of values within specified bins or intervals. The x-axis represents the range of values, and the y-axis represents the frequency or count of values falling within each bin.

The histogram plot in Matplotlib is created using the `hist()` function. It takes a single dataset or multiple datasets as input and automatically calculates the bin edges and frequencies to construct the histogram.

Matplotlib histogram plot is particularly useful when working with numerical data, allowing you to visualize the distribution, identify outliers, and understand the overall shape of the dataset. It is commonly used in various fields such as data analysis, statistics, and machine learning.

You can use different kinds of data to plot a histogram using Matplotlib, including:

1. Numeric Data: Any numerical dataset can be used to create a histogram, such as stock prices, exam scores, temperature readings, or heights of individuals.

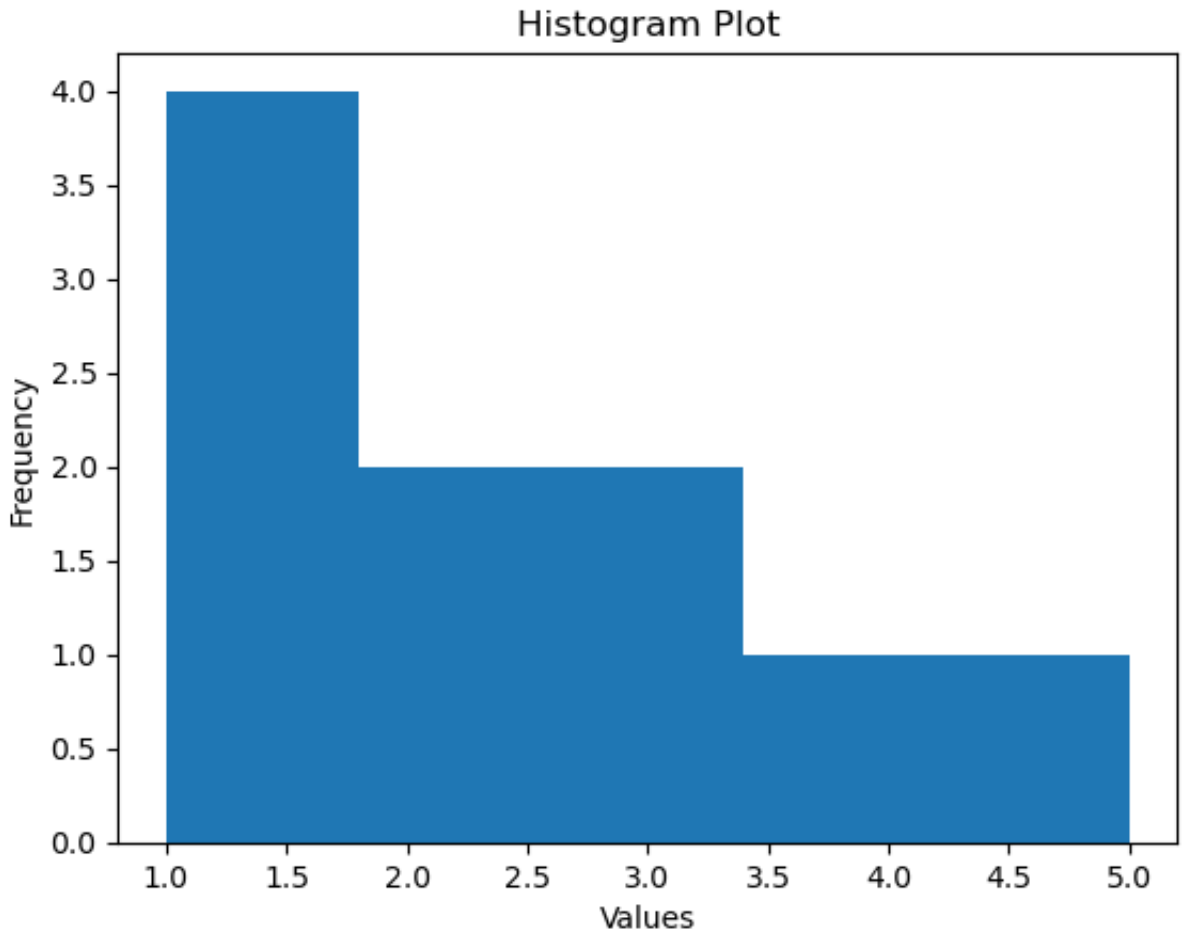
2. Continuous Data: Histograms are commonly used to represent continuous data, where the values lie within a range of real numbers.

3. Discrete Data: Histograms can also be used to represent discrete data, such as the number of people in different age groups or the frequency of certain events occurring.

Matplotlib provides various options to customize the appearance of the histogram, such as adjusting the number of bins, setting the range of values, adding labels and titles, changing bar colors, and more. These options allow you to tailor the plot to effectively communicate the characteristics of your data.

Example 1:

In [5]:
```
import matplotlib.pyplot as plt

data = [1, 2, 1, 3, 3, 1, 4, 2, 1, 5]
plt.hist(data, bins=5)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Histogram Plot")
plt.show()
```



6. Box Plot (`boxplot()`):

A box plot, also known as a box-and-whisker plot, is a graphical representation of the distribution of a dataset. It displays a summary of the minimum, first quartile, median, third quartile, and maximum values of a dataset, as well as any potential outliers.

In Matplotlib, you can create a box plot using the `boxplot()` function. The data kind typically used to plot a box plot in Matplotlib can be numeric or categorical. Let's explore both scenarios:

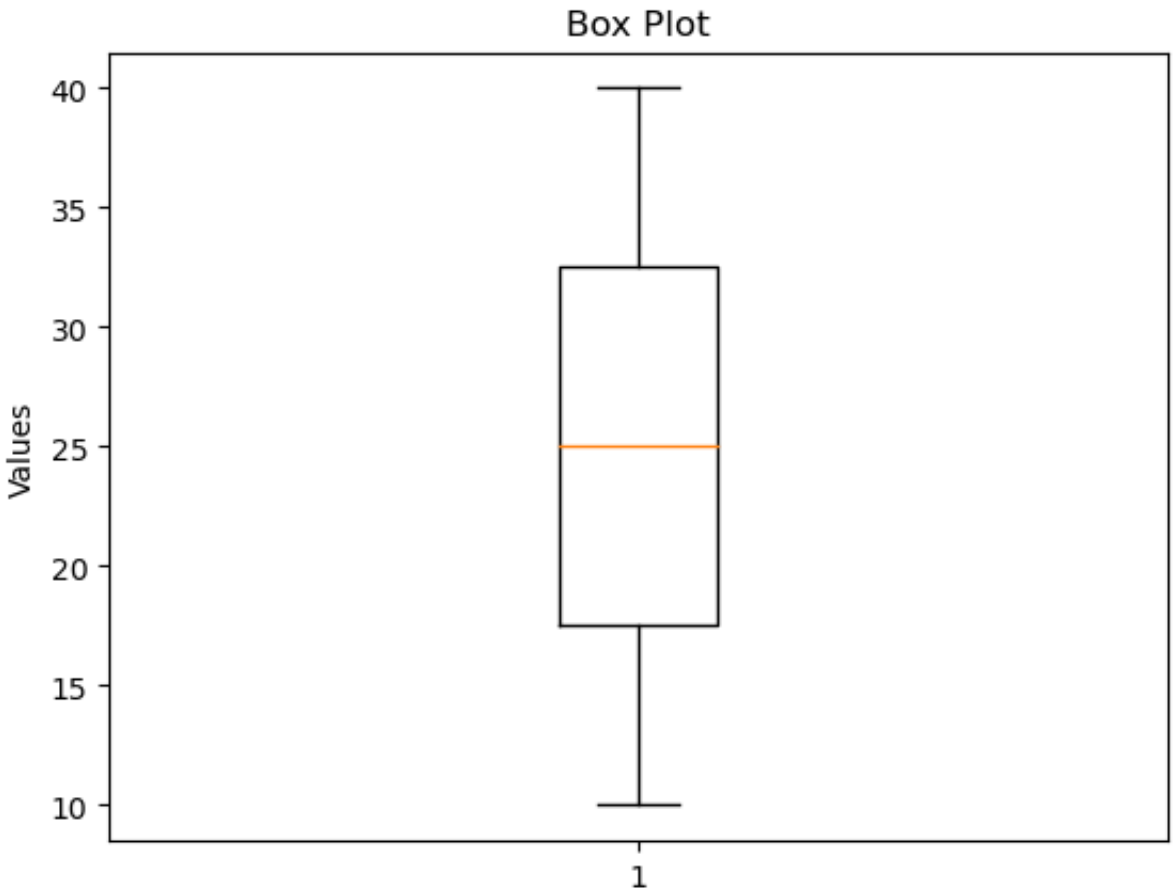
1. Numeric Data: If you have a numeric dataset, such as a list or an array, you can use the `boxplot()` function to create a box plot. Matplotlib will automatically calculate the statistical summary of the data and generate the corresponding visual representation. Numeric data could include continuous variables like temperature, sales figures, or test scores.

2. Categorical Data: Box plots can also be used to represent the distribution of a categorical dataset. In this case, the data is grouped into categories, and the box plot shows the distribution of values within each category. Categorical data could include things like different product categories, days of the week, or survey responses.

Whether you're working with numeric or categorical data, box plots provide a concise summary of the dataset's distribution, including the median, quartiles, and any potential outliers. They are particularly useful for comparing distributions or identifying variations between different groups or categories.

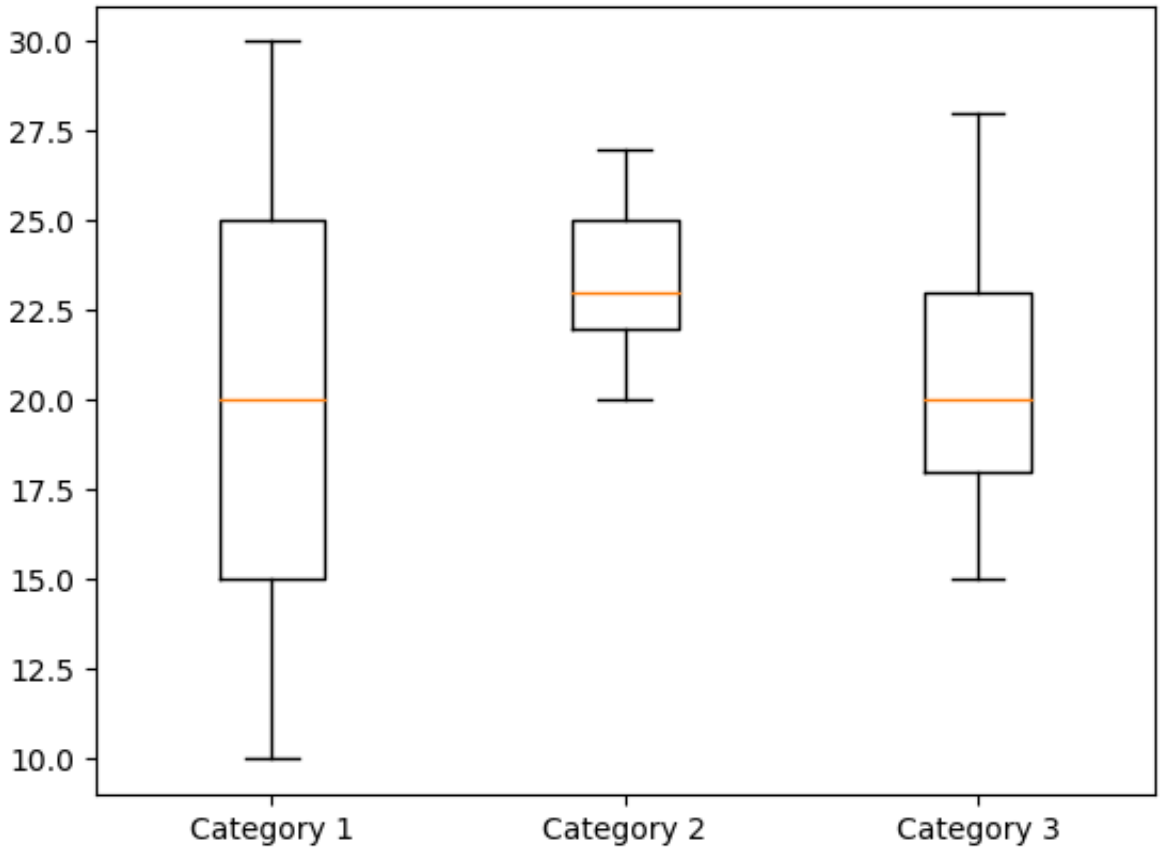
Example 1:

In [6]:
```
# Numerical data Example:
import matplotlib.pyplot as plt

data = [10, 15, 20, 25, 30, 35, 40]
plt.boxplot(data)
plt.ylabel("Values")
plt.title("Box Plot")
plt.show()
```



Example 2:

In [7]:
```
# Categorical Data Example.
import matplotlib.pyplot as plt

data = {
       'Category 1': [10, 15, 20, 25, 30],
       'Category 2': [20, 22, 23, 25, 27],
       'Category 3': [15, 18, 20, 23, 28]
}

plt.boxplot(data.values(), labels=data.keys())
plt.show()
```



Example 2:

In [8]:
```
import matplotlib.pyplot as plt
```

7. Violin Plot (`violinplot()`):

A violin plot is a type of data visualization in Matplotlib that combines elements of a box plot and a kernel density plot. It displays the distribution of a continuous variable or numeric data across different categories or groups.

In a violin plot, each category or group is represented by a "violin" shape, which consists of a rotated kernel density plot mirrored on either side. The width of the violin represents the density or frequency of the data at different values, while the inner boxplot displays additional summary statistics such as the median, quartiles, and possible outliers.

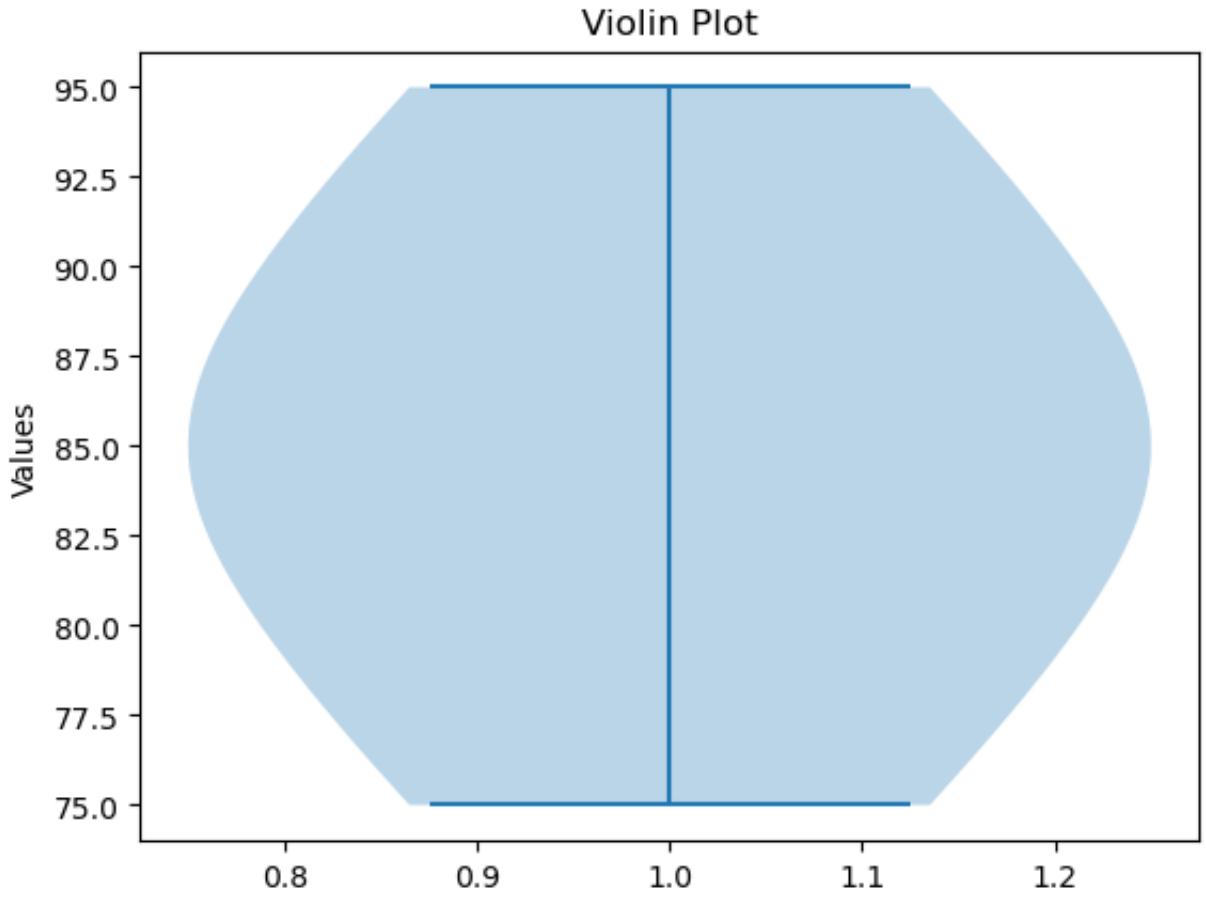
A violin plot is useful when you want to compare the distribution of a continuous variable across different categories or groups, such as comparing the distribution of exam scores among different classes or the distribution of income across different job titles.

To create a violin plot in Matplotlib, you can use the `violinplot()` function. The data used to plot a violin plot can be in various forms:

1. A single numeric array: You can pass a single numeric array to `violinplot()` to create a single violin plot.
2. Multiple numeric arrays: If you have multiple numeric arrays corresponding to different categories or groups, you can pass them as a list of arrays to create multiple violins.
3. Pandas DataFrame: You can also use a Pandas DataFrame to create a violin plot, where each column represents a different category or group.

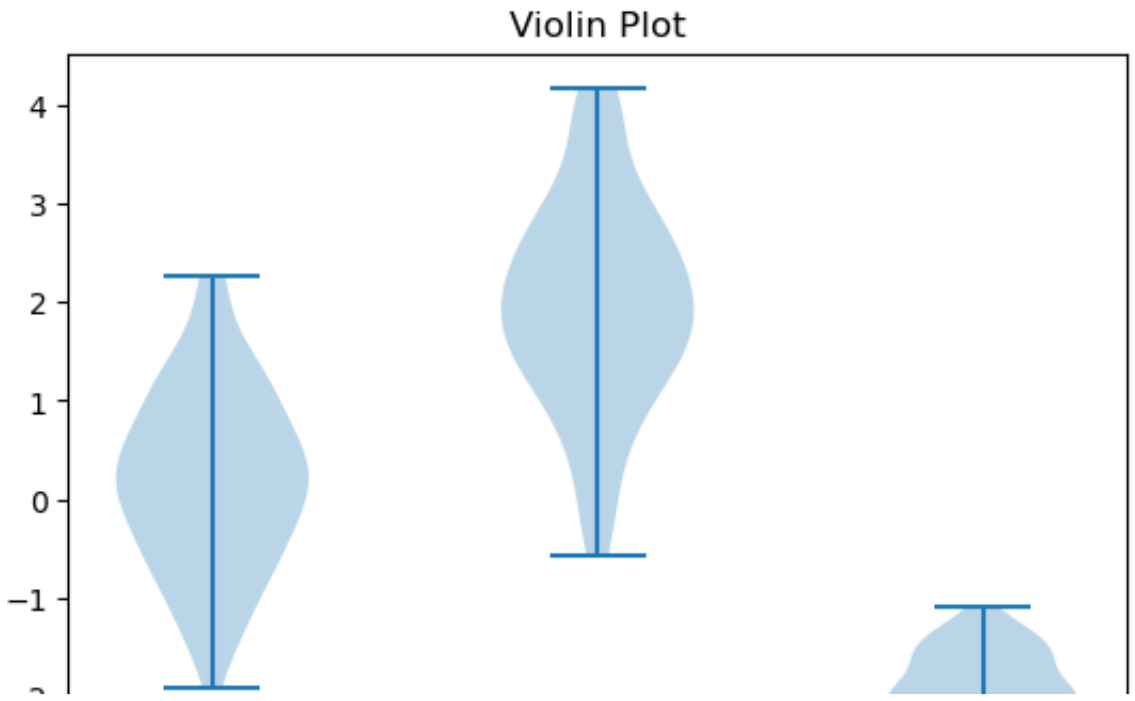
Example 1:

In [9]:
```
import matplotlib.pyplot as plt

data = [80, 85, 90, 75, 95, 85]
plt.violinplot(data)
plt.ylabel("Values")
plt.title("Violin Plot")
plt.show()
```



Example 2:

In [10]:
```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data
data1 = np.random.normal(0, 1, 100)
data2 = np.random.normal(2, 1, 100)
data3 = np.random.normal(-2, 0.5, 100)

# Create a figure and axis
fig, ax = plt.subplots()

# Plot a violin plot
ax.violinplot([data1, data2, data3])

# Set the labels for the x-axis
ax.set_xticks([1, 2, 3])
ax.set_xticklabels(['Group 1', 'Group 2', 'Group 3'])

# Set the title of the plot
ax.set_title('Violin Plot')

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



8. Pie Chart (`pie()`):

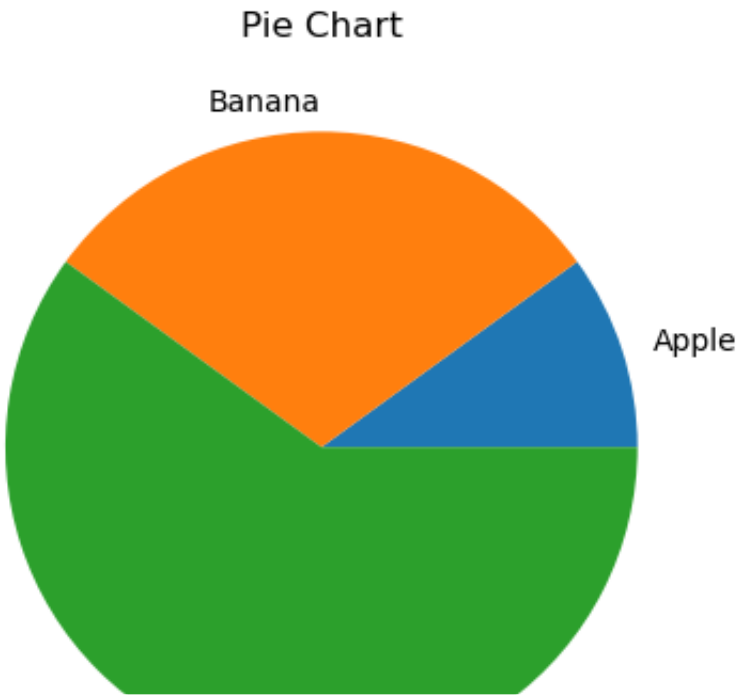
Matplotlib's `pie` plot is a type of chart used to display data in a circular format, resembling a pie. It is commonly used to represent data proportions or percentages in relation to a whole. The pie chart divides a circle into slices, where each slice represents a category or data point, and the size of each slice corresponds to the proportion or percentage it represents.

The `pie` plot in Matplotlib accepts data in the form of an array or a list, where each element represents the value associated with a specific category. Matplotlib automatically calculates the angles and sizes of the slices based on the provided values.

It's important to note that pie charts are most effective when visualizing data with a limited number of categories or data points. They are particularly useful for illustrating relative proportions or composition, but can become less effective when dealing with large amounts of data or categories with similar values.

Example 1:

In [11]:
```
import matplotlib.pyplot as plt

sizes = [10, 30, 60]
labels = ['Apple', 'Banana', 'Cherry']
plt.pie(sizes, labels=labels)
plt.title("Pie Chart")
plt.show()
```



9. Donut Chart:

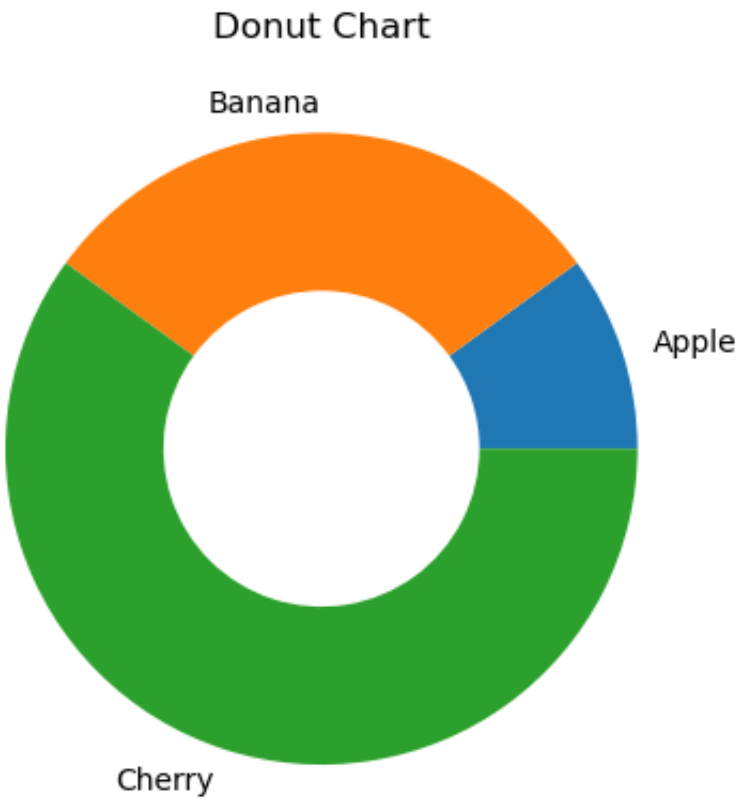
Matplotlib does not have a specific plot type called "donut plot" built-in, but it is possible to create a donut-like chart using pie charts and additional customization. A donut plot is essentially a pie chart with a hole in the center.

To create a donut plot using Matplotlib, you can use the `pie()` function and customize it by adjusting the parameters.

When it comes to the data kind that can be used for a donut plot, it is typically used to represent parts of a whole, just like a regular pie chart. Each data value represents the proportion or percentage of a category relative to the whole. The sum of all data values should be 100 or a proportional representation of the total.

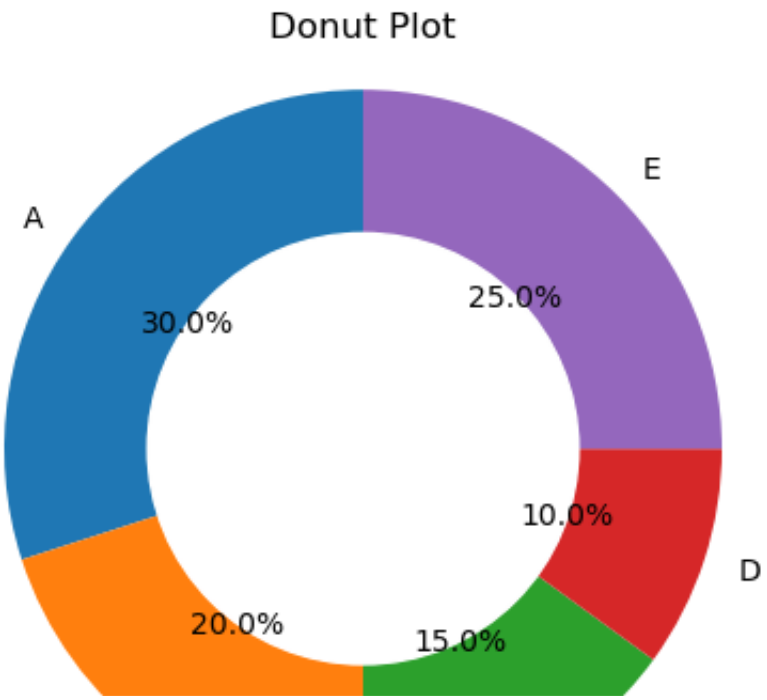
Example 1:

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

sizes = [10, 30, 60]
labels = ['Apple', 'Banana', 'Cherry']
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, wedgeprops=dict(width=0.5))
ax.set_title("Donut Chart")
plt.show()
```



Example 2:

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

# Data
sizes = [30, 20, 15, 10, 25]
labels = ['A', 'B', 'C', 'D', 'E']

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the pie chart
ax.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90, wedgepr

# Add a circle in the center to create a donut-like effect
circle = plt.Circle((0, 0), 0.6, color='white')
ax.add_artist(circle)

# Equal aspect ratio ensures a circular shape
ax.axis('equal')

# Set a title
ax.set_title('Donut Plot')

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



10. Area Plot (`fill_between()`):

Matplotlib's area plot, also known as a stacked area plot, is a type of plot that displays the cumulative contribution of different categories over a continuous variable (e.g., time). It represents the data as a series of stacked polygons, where the height of each polygon at a given x-value corresponds to the cumulative value of the categories up to that point.

To create an area plot in Matplotlib, you can use the `fill_between()` function. This function fills the area between two curves, allowing you to stack multiple areas to create the stacked area plot effect. Area plots are particularly useful when you want to visualize the composition or distribution of different categories and their contributions to the whole. They are commonly used to represent data such as:

1. Time series data: Area plots can show the cumulative values of different categories over time, allowing you to observe how the contribution of each category changes over the time period.

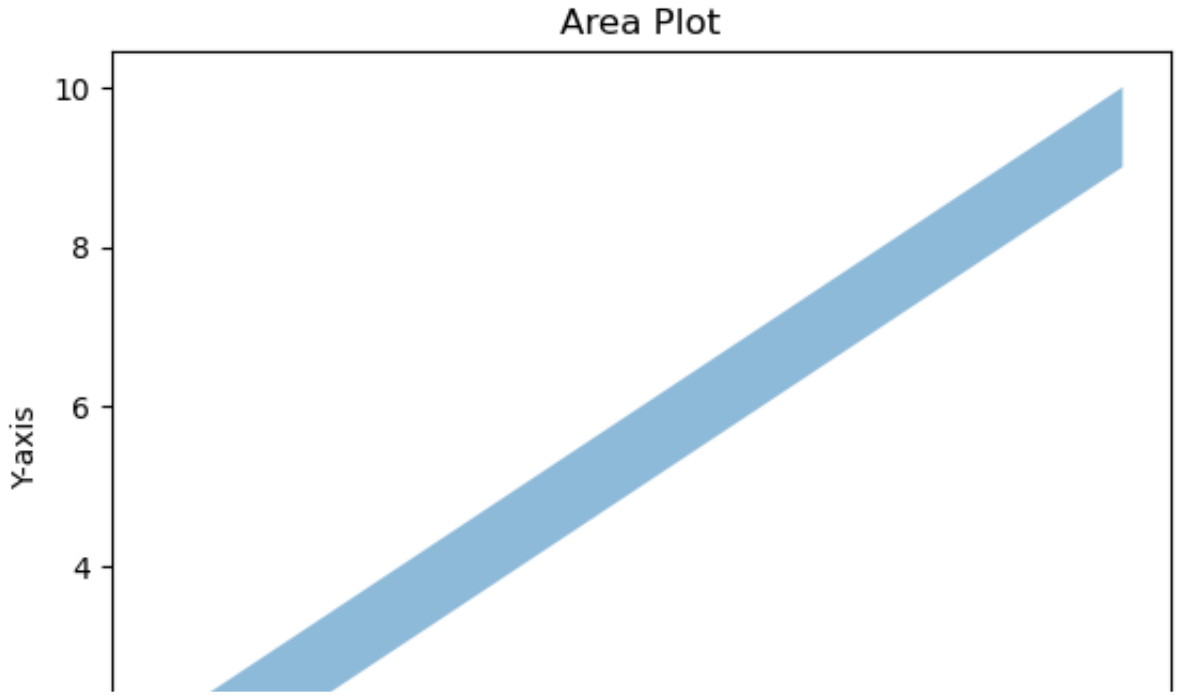
2. Proportional data: If you have data that represents proportions or percentages, an area plot can illustrate the relative contributions of different categories to the total.

3. Stacked data: Area plots are useful for displaying the cumulative values of stacked data, such as the cumulative revenue generated by different product categories.

4. Hierarchical data: If you have hierarchical data, an area plot can help visualize the nested contributions of different categories within each level of the hierarchy.

Example 1:

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

x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 3, 5, 7, 9]
plt.fill_between(x, y1, y2, alpha=0.5)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Area Plot")
plt.show()
```



11. Stack Plot (`stackplot()`):

Matplotlib's stack plot, also known as a stacked area plot, is a type of plot that displays multiple datasets stacked on top of one another to showcase their cumulative contribution to the whole. It is particularly useful for illustrating the composition or distribution of different categories over time or any other continuous variable.

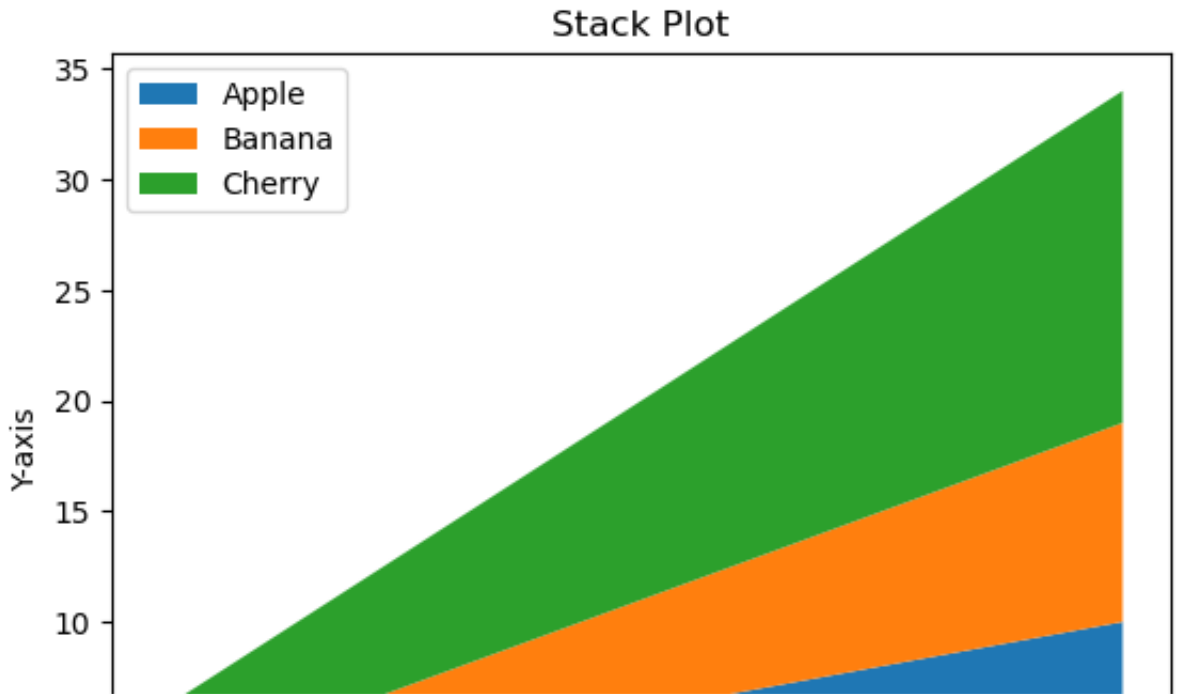
In a stack plot, each dataset is represented as a colored area, and the cumulative effect of all the datasets creates the final stacked plot. The y-axis represents the magnitude or proportion of the data, while the x-axis represents the variable on which the stacking is based (e.g., time, years, or any other continuous variable). To create a stack plot using Matplotlib, you can utilize the `stackplot()` function. The function takes the x-axis values and a sequence of y-axis values for each dataset to be stacked. You can specify the colors, labels, and other customization options to enhance the plot's readability and visual appeal.

A stack plot is commonly used when you want to compare the parts of a whole or track the changes in the composition of multiple categories over time. Some examples of data suitable for stack plots include:

1. Stock market data: Displaying the cumulative contribution of different stocks to a portfolio's overall value.
2. Population demographics: Illustrating the distribution of different age groups or ethnicities within a population.
3. Sales data: Tracking the sales performance of different products or product categories over time.
4. Energy consumption: Showing the breakdown of energy sources (e.g., coal, natural gas, renewable energy) in the overall energy usage.
5. Financial data: Visualizing the contributions of various expense categories to the total expenses over time.

Example 1:

In [15]:
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9], [3, 6, 9, 12, 15]]
labels = ['Apple', 'Banana', 'Cherry']
plt.stackplot(x, y, labels=labels)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Stack Plot")
plt.legend(loc='upper left')
plt.show()
```



12. Stem Plot (`stem()`):

The matplotlib stem plot is a type of plot used to visualize data with discrete values, where each data point is represented as a marker (usually a vertical line or a marker shape) rising from a baseline. It is commonly used to display values that occur at specific time points or discrete intervals.

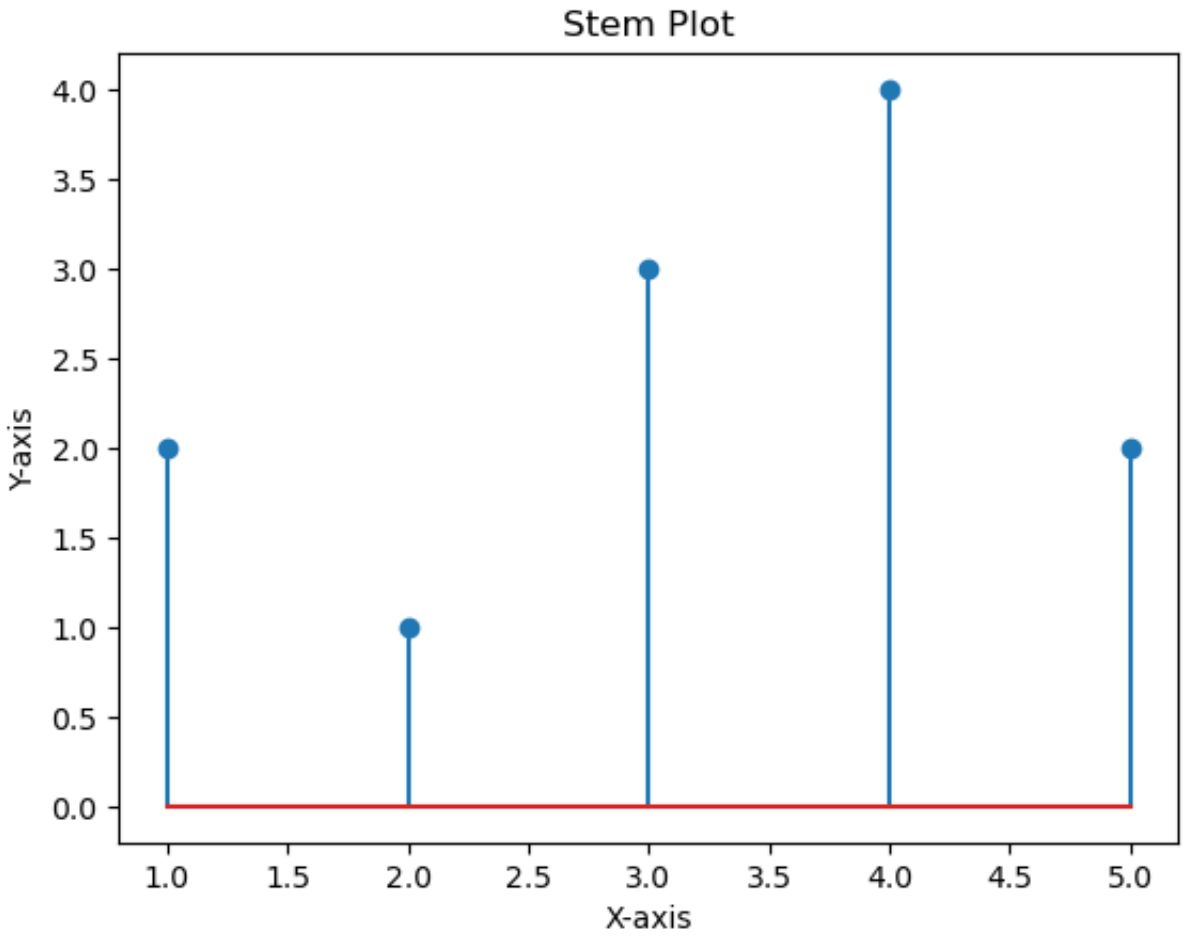
The stem plot is useful when you want to emphasize the exact values of the data points and their positions relative to the baseline. It is often used in signal processing, control systems, and other fields where discrete data representation is crucial. Stem plot can be a powerful tool for visualizing discrete data and patterns, especially when you need to emphasize individual data points. It provides a clear representation of the data values and their relationships to the baseline.

You can use various data kinds to plot a matplotlib stem plot, including:

1. Discrete values: Use a sequence of discrete values as the y-values to represent data occurring at specific points.
2. Time series: Use timestamps as the x-values to represent data occurring at different time points.
3. Categorical data: Use categorical labels as the x-values to represent data belonging to different categories.

Example 1:

In [16]:
```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [2, 1, 3, 4, 2]

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the stem plot
ax.stem(x, y)

# Customize the plot
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Stem Plot')

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



13. Quiver Plot (`quiver()`):

Matplotlib's Quiver plot is a type of plot used to represent vector fields. It is particularly useful for visualizing physical quantities such as velocity or force fields. In a Quiver plot, arrows are used to represent vectors at specific points in a coordinate system.

To create a Quiver plot using Matplotlib, you need to provide the coordinates of the points and the corresponding vector components. Typically, the data used for a Quiver plot consists of four arrays:

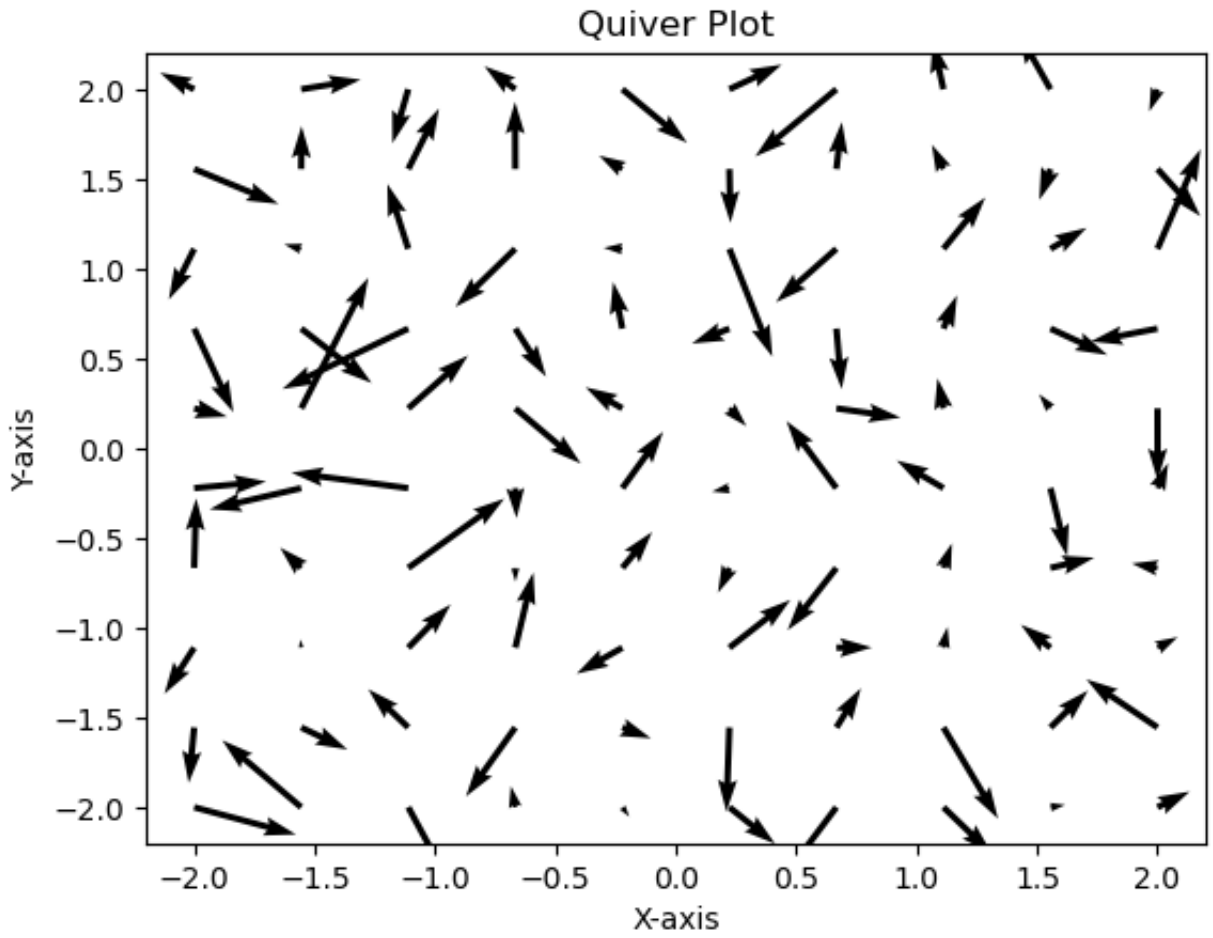
1. X-coordinates: An array containing the x-coordinates of the points.
2. Y-coordinates: An array containing the y-coordinates of the points.
3. U-components: An array containing the x-components of the vectors at each point.
4. V-components: An array containing the y-components of the vectors at each point.

The lengths of the X, Y, U, and V arrays should be equal in this case.

Example 1:

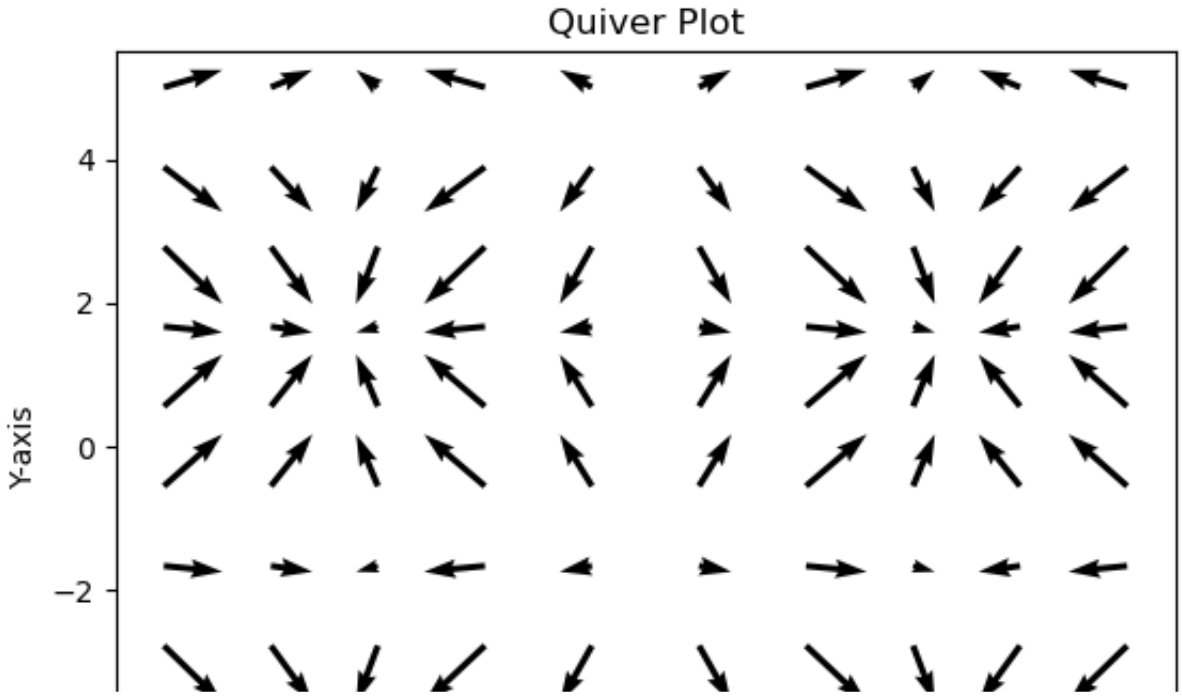
In [17]:

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

# Generate random data
x = np.linspace(-2, 2, 10)
y = np.linspace(-2, 2, 10)
X, Y = np.meshgrid(x, y)
U = np.random.randn(10, 10)
V = np.random.randn(10, 10)

# Create the plot
fig, ax = plt.subplots()
ax.quiver(X, Y, U, V)

# Customize the plot
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Quiver Plot')

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



Example 2:

In [18]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 10)
y = np.linspace(-5, 5, 10)
X, Y = np.meshgrid(x, y)
U = np.sin(X)
V = np.cos(Y)
plt.quiver(X, Y, U, V)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Quiver Plot")
plt.show()
```



14. Polar Plot (`polar()`):

Polar plot is a type of plot that represents data in a polar coordinate system. In a polar plot, data points are specified by their angle and distance from the origin (r, θ) instead of their x and y coordinates. It is particularly useful for visualizing data with cyclic patterns or directional relationships.

The polar plot in Matplotlib allows you to create various types of polar plots, such as line plots, scatter plots, bar plots, and more. It provides a circular grid with radial lines and concentric circles to represent the polar coordinate system.

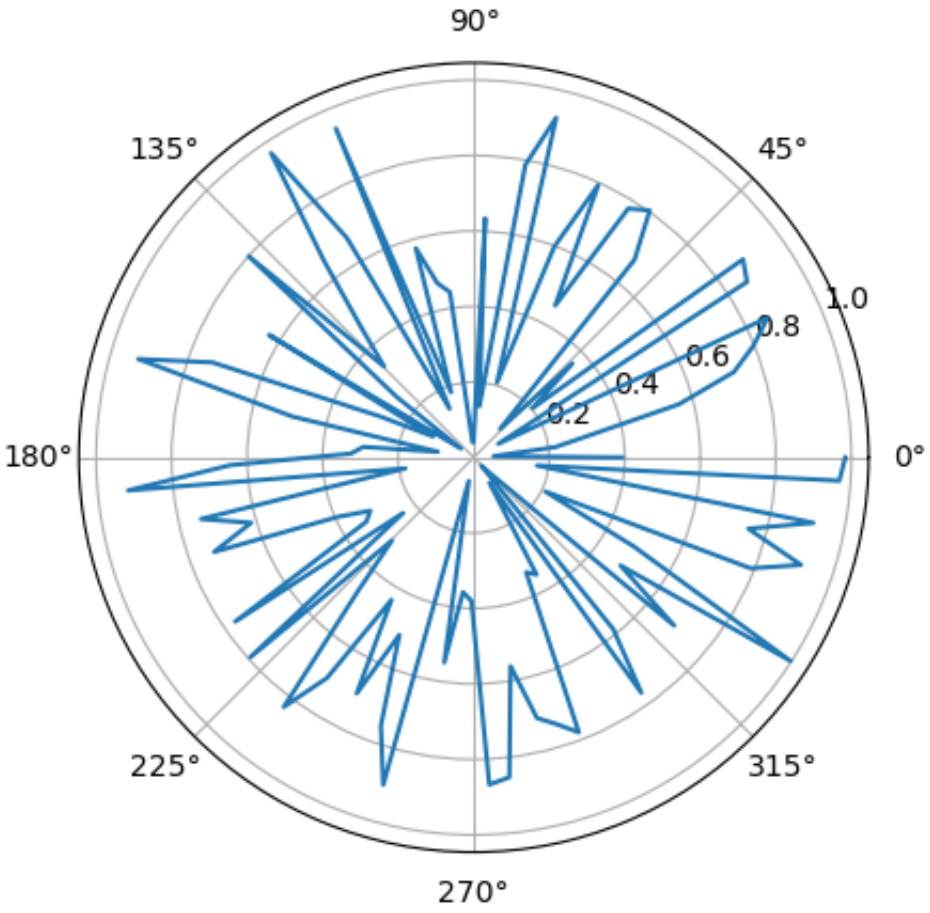
The data used to plot a polar plot typically consists of two arrays or lists:

1. Array/List of Angles (θ):
    - This represents the angles at which the data points are located.
    - The angles are usually specified in radians, ranging from 0 to 2π (or 0 to 360 degrees).
    - Each angle corresponds to a data point or a category on the polar plot.

2. Array/List of Distances (r):
    - This represents the distance from the origin (0) to the data points.
    - The distances can be positive values, indicating the length from the origin, or negative values for inward pointing.

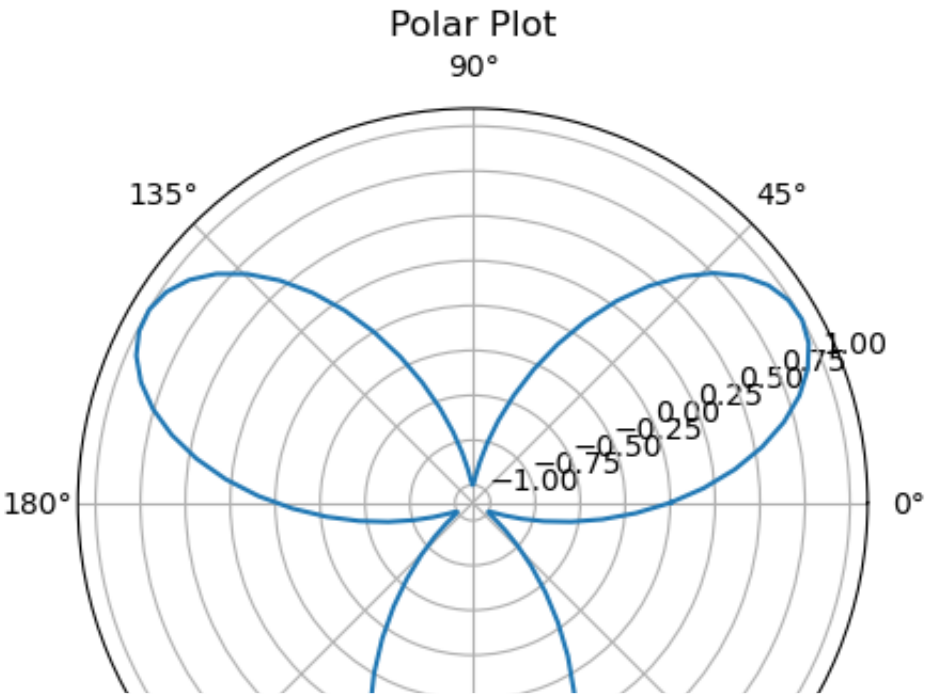
Example 1:

In [19]:
```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data
angles = np.linspace(0, 2*np.pi, 100)
distances = np.random.rand(100)

# Create a polar plot
plt.polar(angles, distances)

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



Example 2:

In [20]:
```
import matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 100)
r = np.sin(3*theta)
plt.polar(theta, r)
plt.title("Polar Plot")
plt.show()
```



15. 3D Plot (`plot_surface()`):

Matplotlib is a popular Python library for creating visualizations and plots. It provides a wide range of functions and methods to generate different types of plots, including 3D plots using the `plot_surface()` function.

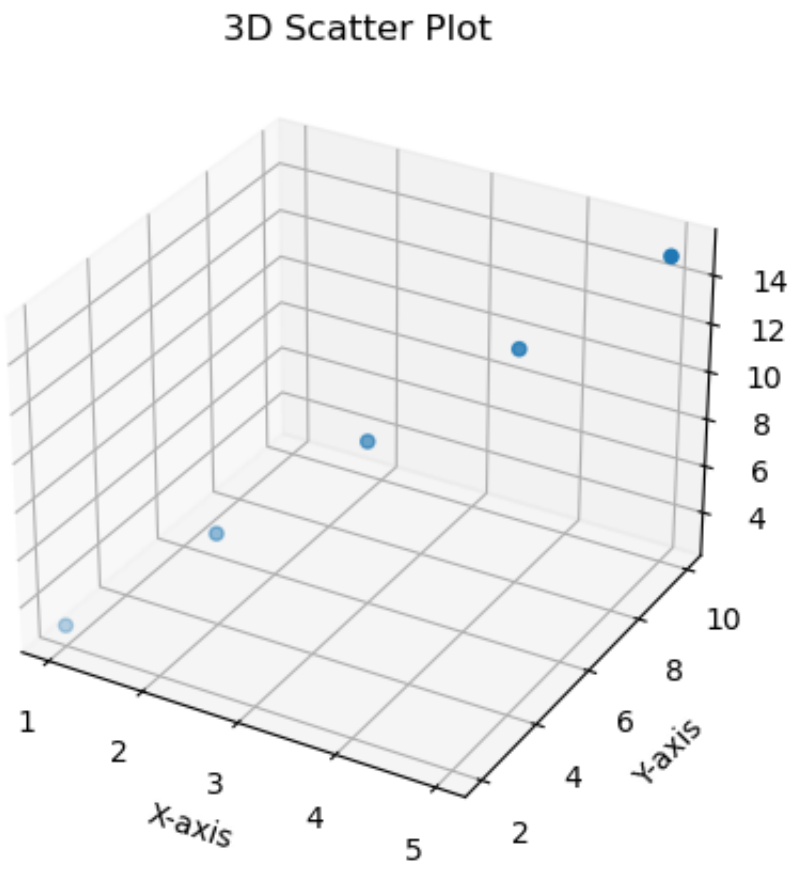
The `plot_surface()` function in Matplotlib is specifically designed for creating 3D surface plots. It visualizes the relationship between three continuous variables, typically represented by x, y, and z coordinates. The resulting plot displays a surface or mesh that represents the interaction between the three variables.

You can use various kinds of data to create 3D plots with `plot_surface()`, such as mathematical functions, scientific data, or simulation results. As long as you have three-dimensional data (x, y, and z), you can visualize it using `plot_surface()` and explore the relationship between the variables in a three-dimensional space.

With Matplotlib's 3D plotting functionality, you can create various types of 3D plots, including surface plots, wireframe plots, scatter plots, and bar plots. These plots provide a way to represent data points that have three dimensions: x, y, and z.

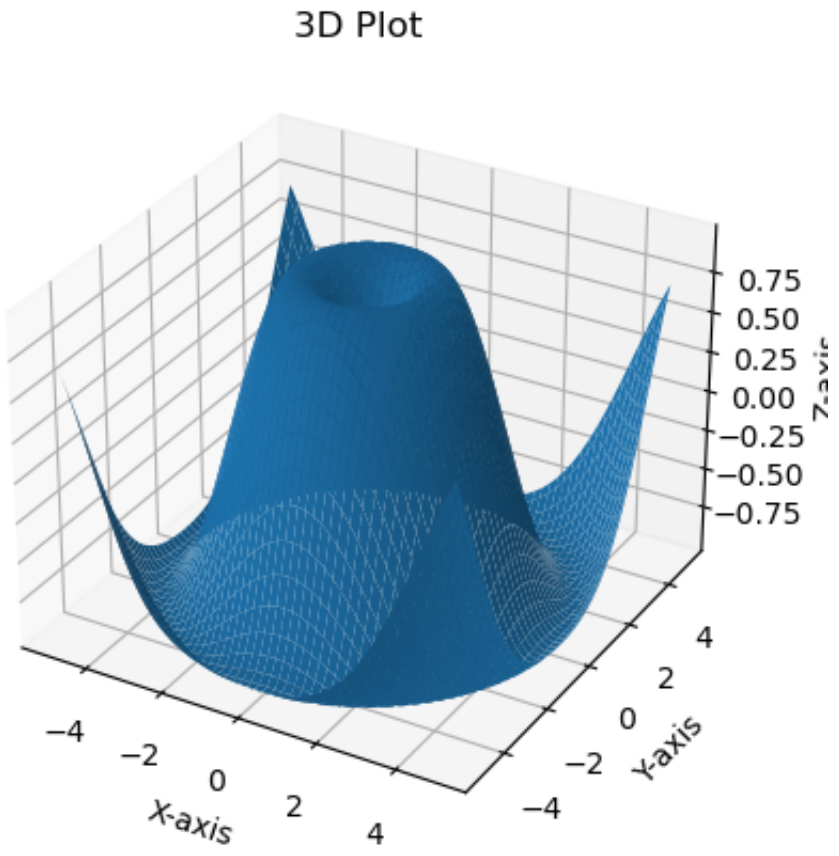
Example 1:

In [21]:
```
# scatter plots

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
z = [3, 6, 9, 12, 15]

# Create a figure and axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Create the 3D scatter plot
ax.scatter(x, y, z)

# Set labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
ax.set_title('3D Scatter Plot')

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



Example 2:

In [22]:
```
# surface plot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

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


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))
ax.plot_surface(X, Y, Z)
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
ax.set_zlabel("Z-axis")
ax.set_title("3D Plot")
plt.show()
```



16. Contour Plot (`contour()`):

A Matplotlib contour plot is a 2D visualization that represents the variation of a continuous variable over a two-dimensional space. It uses contour lines to display the levels or values of the variable on a grid. Contour plots are particularly useful for visualizing data with a smooth, continuous distribution, such as elevation, temperature, or population density.

`contour()` : This function creates a contour plot with contour lines representing the levels of the variable. The colors of the lines can be customized based on the data values.

Contour plots can be created using various kinds of data, such as:

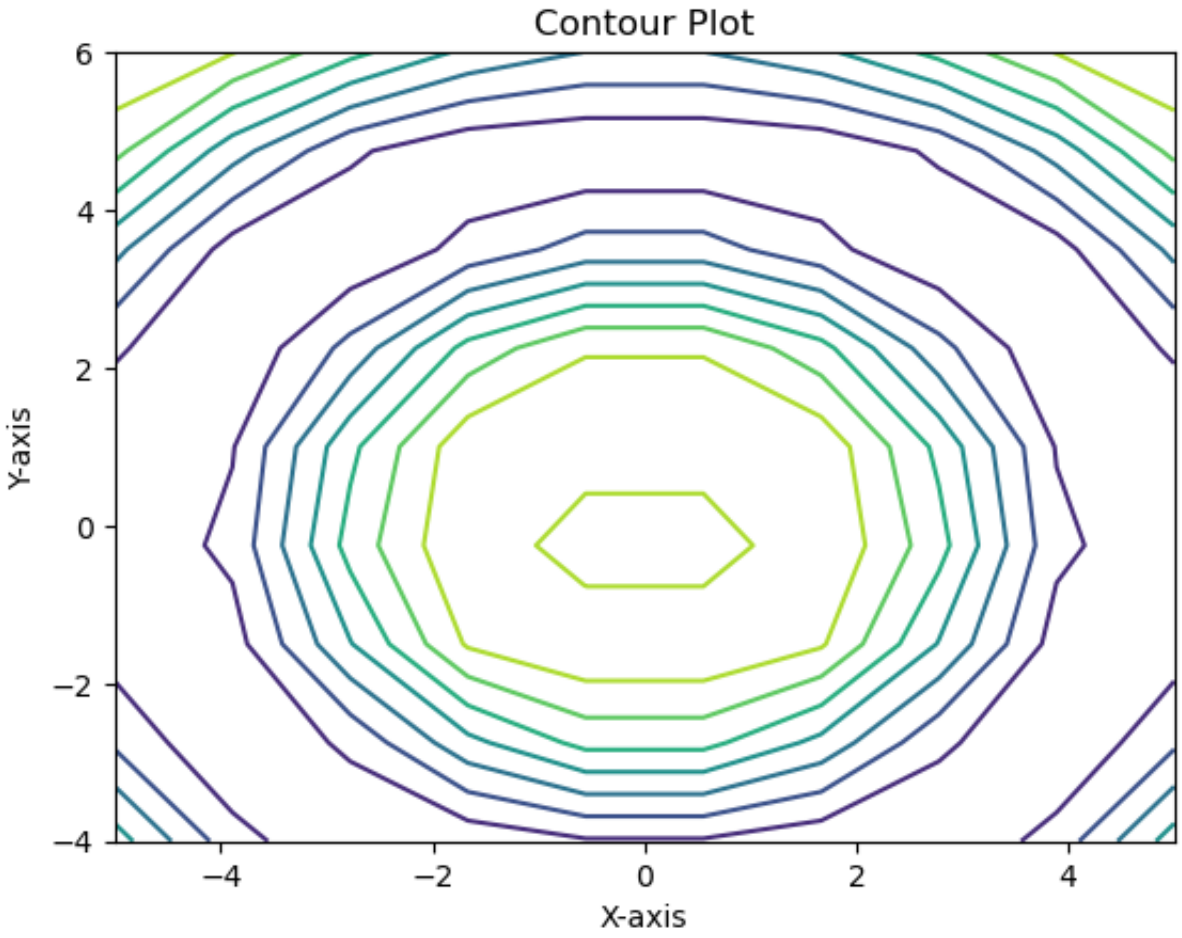
1. Gridded Data: Contour plots are commonly used to visualize data on a regular grid, such as geographic data represented by latitude and longitude coordinates. For example, you can create a contour plot to visualize temperature variations across a map.

2. Function Evaluations: Contour plots can also be used to visualize the output of a mathematical function. By evaluating the function at different points on a grid, you can create a contour plot that represents the function's values.

3. Experimental Data: If you have experimental data collected at specific points in a two-dimensional space, you can interpolate the data onto a grid and create a contour plot to visualize the continuous distribution of the variable.

In [23]:
```
## Example 1:
```

In [24]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 10, 50)
y = np.linspace(-4, 6, 9, 60)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
plt.contour(X, Y, Z)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Contour Plot")
plt.show()
```



17. Contourf Plot (`contourf()`):

Matplotlib contour plots are used to visualize 3D data on a 2D plane by representing the data as contour lines or filled contour regions. Contour lines are curves that connect points of equal value in a dataset, while filled contour regions are areas bounded by contour lines.

The `contour()` function in Matplotlib creates contour lines, and the `contourf()` function creates filled contour plots. These functions take in the same set of arguments and are used to plot different types of data.

The `contour()` function can be used with various types of data, such as:

1. 2D Arrays: Contour lines can be plotted for evenly spaced x and y values with corresponding z values representing the contour heights.

2. Coordinate Grids: Contour lines can be plotted for irregularly spaced x and y coordinate grids with corresponding z values representing the contour heights.1. 2D Arrays: Contour lines can be plotted for evenly spaced x and y values with corresponding z values representing the contour heights.

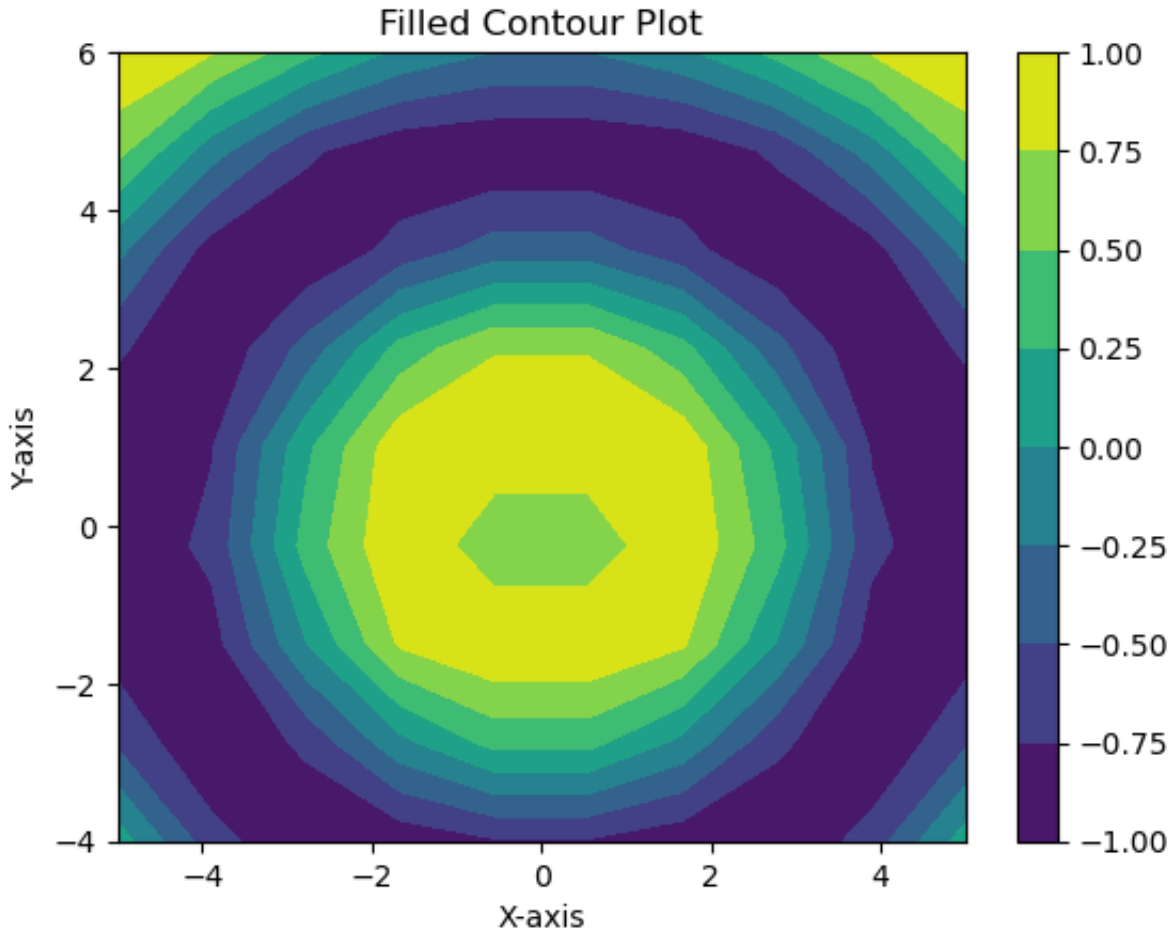
3. Triangulations: Contour lines can be plotted for unstructured triangular grids with corresponding z values representing the contour heights.

The `contourf()` function, on the other hand, is specifically used to create filled contour plots. It can be used with the same types of data as `contour()`, such as 2D arrays, coordinate grids, and triangulations.

The data used to create contour or filled contour plots typically represents continuous variables, such as temperature, elevation, or any other variable that can be measured across a 2D space.

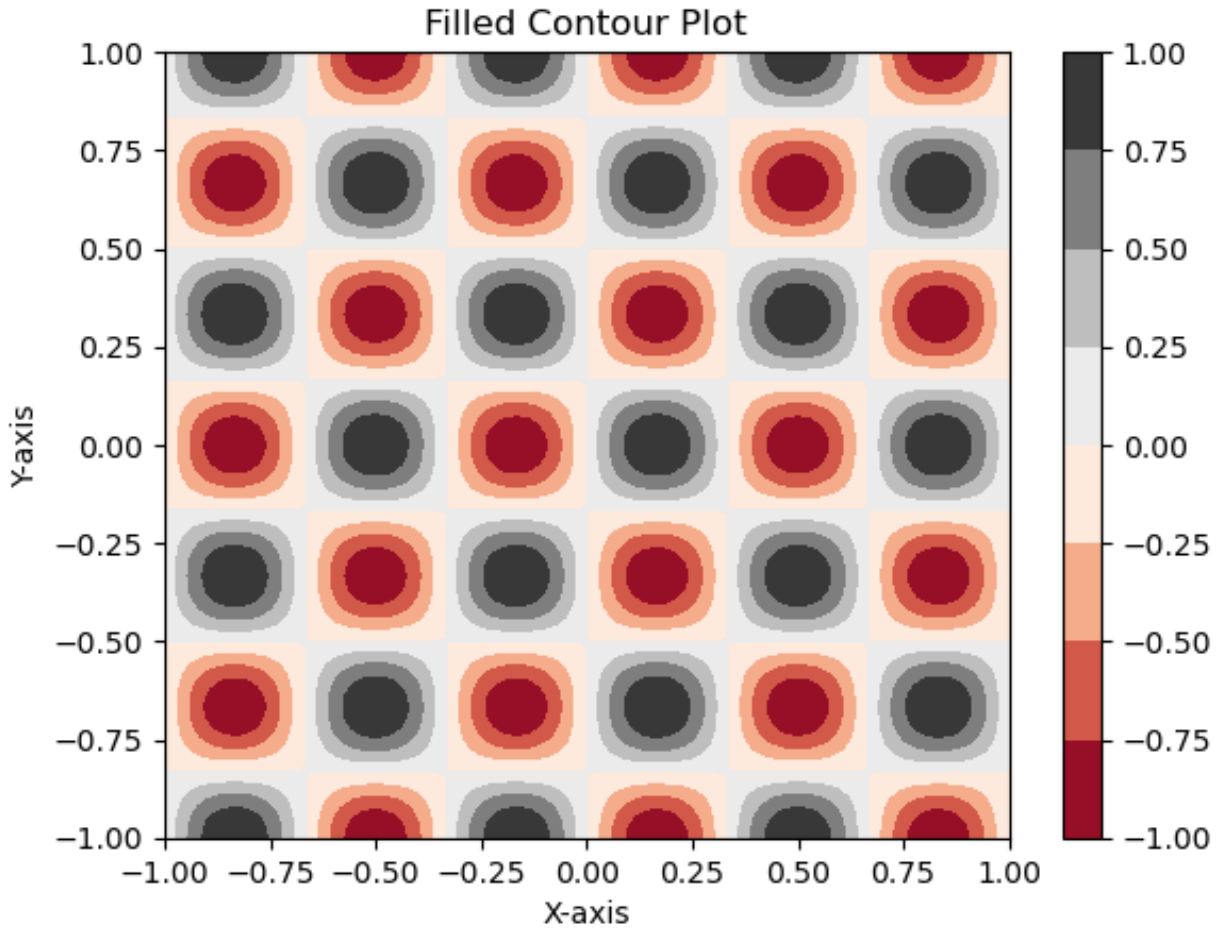
Example 1:

In [25]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 10, 50)
y = np.linspace(-4, 6, 9, 60)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
plt.contourf(X, Y, Z)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Filled Contour Plot")
plt.colorbar()
plt.show()
```



In [26]:

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

# Create a 2D array representing z values
x = np.linspace(-1, 1, 100)
y = np.linspace(-1, 1, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(3 * np.pi * X) * np.cos(3 * np.pi * Y)

# Create a filled contour plot
plt.contourf(X, Y, Z, cmap='RdGy')

# Add color bar for reference
plt.colorbar()

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Filled Contour Plot')

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



18. Hexbin Plot (`hexbin()`):

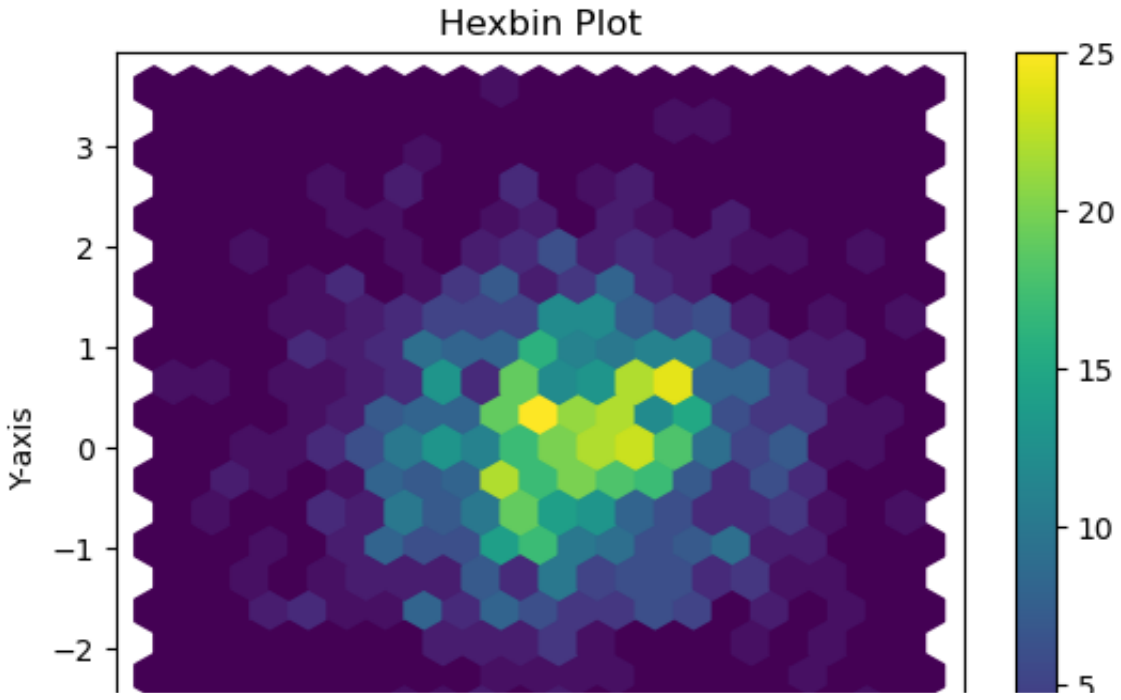
The hexbin plot in Matplotlib is a type of 2D histogram that represents the distribution of data using hexagonal bins. It is particularly useful when you have a large dataset and want to visualize the density of points in a scatter plot-like format. The hexbin plot divides the data space into hexagonal bins and assigns a color to each bin based on the number of points falling within that bin. The color intensity represents the density of points in that region, where darker colors indicate higher point density.

The hexbin plot is suitable for visualizing continuous data, particularly when you have a large number of data points and want to understand their distribution and density patterns. It is often used in fields such as data analysis, exploratory data visualization, and spatial data analysis.

The hexbin plot is an effective way to visualize the density and distribution of data points, making it useful for tasks such as identifying clusters, patterns, or trends in your dataset.

In [27]:
```
## Example 1:
```

In [28]:
```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data
x = np.random.randn(1200)
y = np.random.randn(1200)

# Create a hexbin plot
plt.hexbin(x, y, gridsize=20)

# Customize the plot
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Hexbin Plot')

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



19. Polar Scatter Plot (`scatter()` with polar coordinates):

A polar scatter plot is a type of plot in Matplotlib that represents data points in a polar coordinate system. In this plot, each data point is defined by its radial distance from the origin and its angular position.

To create a polar scatter plot in Matplotlib, you can use the `scatter()` function and provide the data points in polar coordinates. The radial distance corresponds to the magnitude of the data point, and the angular position determines its direction or angle.

The data used to plot a polar scatter plot can vary depending on the context and application. However, some common examples of data suitable for a polar scatter plot include:

1. Wind Direction and Wind Speed:
    - The radial distance represents the wind speed, and the angular position represents the wind direction. Each data point represents a specific wind measurement at a particular location.

2. Polar Coordinate Data:
    - Data that naturally exists in polar coordinates, such as seismic data, astronomical data, or geographical data with polar coordinates like latitude and longitude.

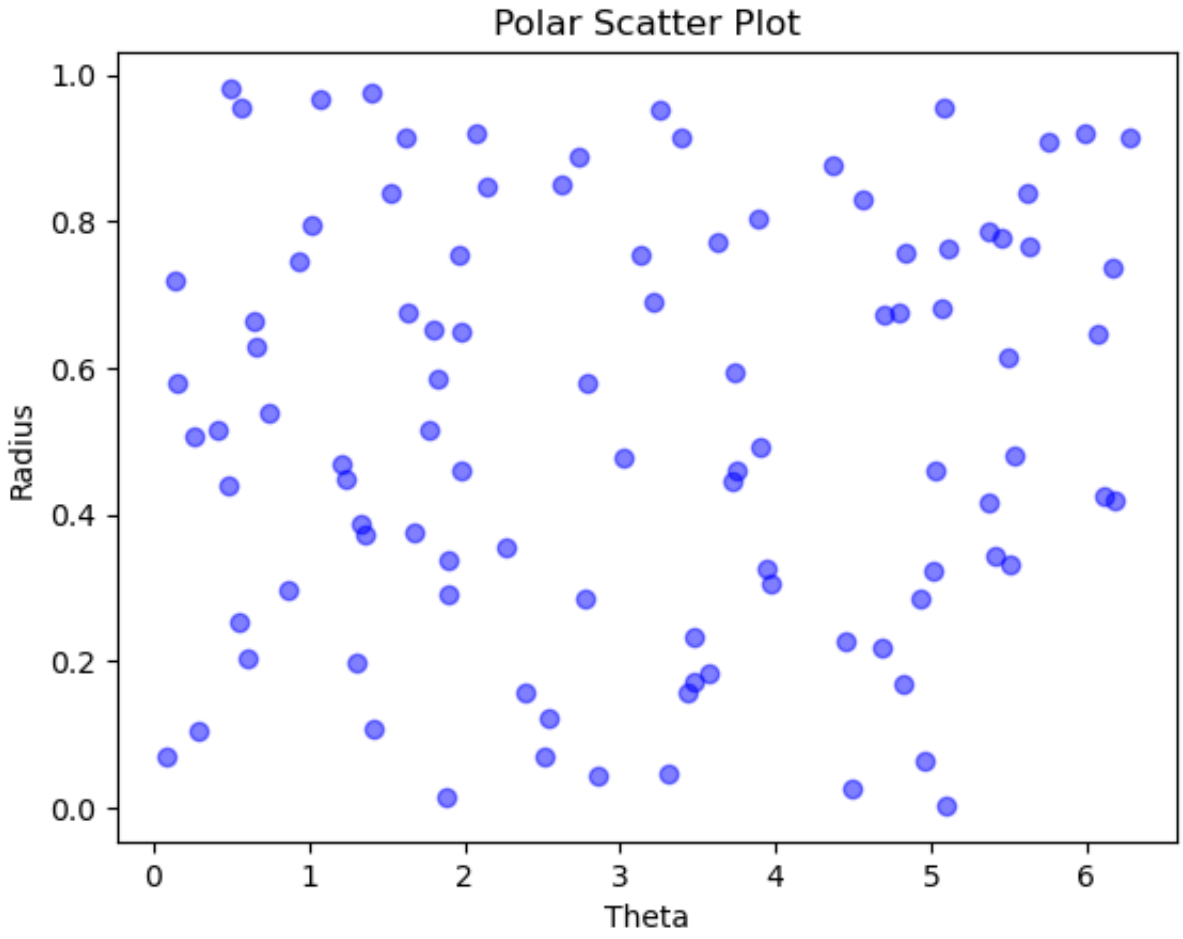
3. Circular Data:
    - Data that exhibits cyclical patterns or periodicity, such as daily temperatures, tidal data, or cyclical patterns in biological or ecological studies.

4. Angular Data:
    - Data that represents angles or directions, such as orientation measurements, compass readings, or directional patterns.

Example 1:

In [29]:

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

# Generate random data
theta = np.random.rand(100) * 2 * np.pi
r = np.random.rand(100)

# Create a polar scatter plot
plt.scatter(theta, r, c='blue', alpha=0.5)

# Set plot title and labels
plt.title('Polar Scatter Plot')
plt.xlabel('Theta')
plt.ylabel('Radius')

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

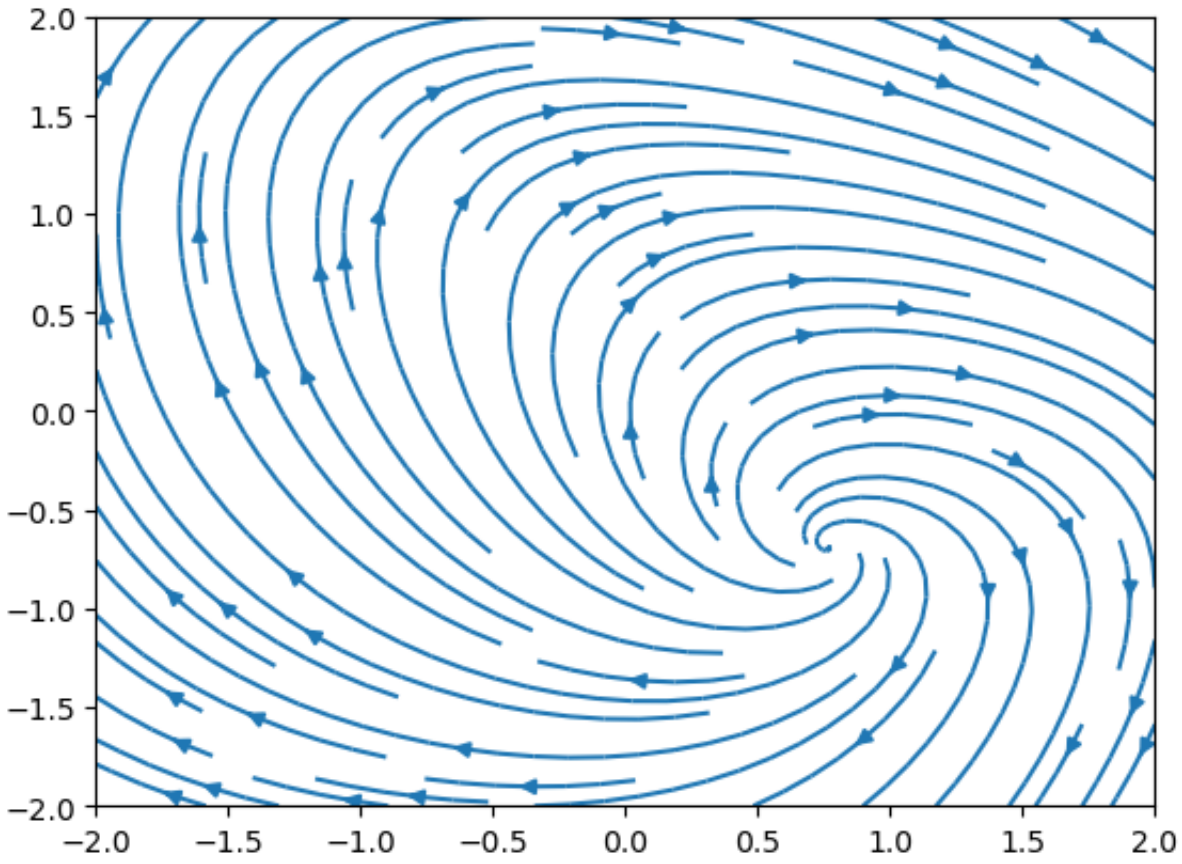


20. Streamplot (`streamplot()`):

Matplotlib's `streamplot` is a function that generates a streamline plot, also known as a streamplot. Streamline plots are used to visualize the flow of a 2D vector field. It is particularly useful for representing fluid flow patterns, such as velocity or electric fields.

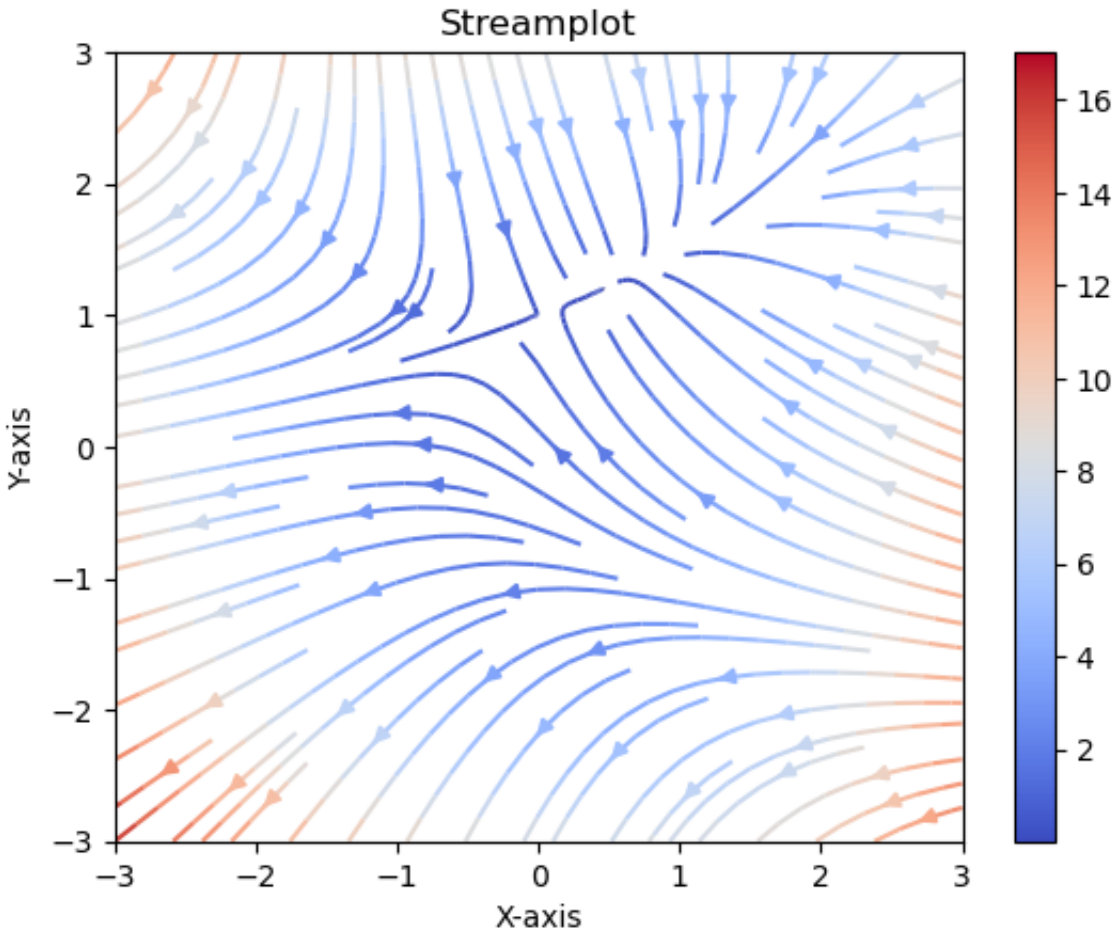
Example 1:

In [30]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)
U = np.sin(X) + Y
V = np.cos(Y) - X

plt.streamplot(X, Y, U, V)
plt.show()
```



Example 2:

In [31]:
```
import matplotlib.pyplot as plt
import numpy as np

Y, X = np.mgrid[-3:3:200j, -3:3:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U**2 + V**2)
plt.streamplot(X, Y, U, V, color=speed, cmap='coolwarm')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Streamplot")
plt.colorbar()
plt.show()
```



Thank you for following through to this point. I guess you should have a question, two or more.

Just before that;

Matplotlib offers a wide range of customization options by combining these customization options, you can create visually appealing and informative plots tailored to your data analysis needs. Some common customization options along with their corresponding methods or functions:

1. Setting Plot Title:
    - `set_title()` : Sets the title of the plot.

    Example: `ax.set_title('My Plot')`
2. Setting Axis Labels:
    - `set_xlabel()` : Sets the label for the x-axis.
    - `set_ylabel()` : Sets the label for the y-axis.

    Example: `ax.set_xlabel('X-axis')`
3. Setting Axis Limits:
    - `set_xlim()` : Sets the limits for the x-axis.
    - `set_ylim()` : Sets the limits for the y-axis. Example: `ax.set_xlim(0, 10)`
4. Changing Line Style and Color:
    - `plot()` : Additional parameters such as `linestyle` and `color` can be passed to modify the line style and color of the plot.

    Example: `ax.plot(x, y, linestyle='--', color='r')`
5. Adding Gridlines:
    - `grid()` : Displays gridlines on the plot.

    Example: `ax.grid(True)`
6. Adding Legends:
    - `legend()` : Adds a legend to the plot, specifying labels for different elements.

    Example: `ax.legend(['Line 1', 'Line 2'])`
7. Changing Marker Style:
    - `plot()` : Additional parameter `marker` can be passed to change the marker style.

    Example: `ax.plot(x, y, marker='o')`
8. Adding Annotations:
    - `annotate()` : Adds text annotations to specific points on the plot.

    Example: `ax.annotate('Max Value', xy=(3, 6), xytext=(4, 7), arrowprops=dict(arrowstyle='->'))`
9. Changing Figure Size:
    - `figure()` : Additional parameter `figsize` can be passed to specify the width and height of the figure.

    Example: `plt.figure(figsize=(8, 6))`
10. Adding Subplots:

- `subplots()` : Creates multiple subplots within the same figure.

Example: `fig, ax = plt.subplots(nrows=2, ncols=2)`

As we dive deeper there are more customization options available for Matplotlip.

Always consider the nature of your data and the insights you want to convey when deciding to use a particular function or exploring other visualization options.