Install Libraries

In [1]:
```
!pip install matplotlib seaborn
```

```
Requirement already satisfied: matplotlib in c:\users\mites\anaconda3\lib
\site-packages (3.7.2)
Requirement already satisfied: seaborn in c:\users\mites\anaconda3\lib\sit
e-packages (0.12.2)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\mites\anaconda
3\lib\site-packages (from matplotlib) (1.0.5)
Requirement already satisfied: cycler>=0.10 in c:\users\mites\anaconda3\li
b\site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\mites\anacond
a3\lib\site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\mites\anacond
a3\lib\site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: numpy>=1.20 in c:\users\mites\anaconda3\lib
\site-packages (from matplotlib) (1.24.3)
Requirement already satisfied: packaging>=20.0 in c:\users\mites\anaconda3
\lib\site-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=6.2.0 in c:\users\mites\anaconda3\l
ib\site-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in c:\users\mites\ana
conda3\lib\site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\mites\anac
onda3\lib\site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: pandas>=0.25 in c:\users\mites\anaconda3\li
b\site-packages (from seaborn) (2.0.3)
Requirement already satisfied: pytz>=2020.1 in c:\users\mites\anaconda3\li
b\site-packages (from pandas>=0.25->seaborn) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\mites\anaconda3
\lib\site-packages (from pandas>=0.25->seaborn) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\mites\anaconda3\lib\si
te-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

Import Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
```

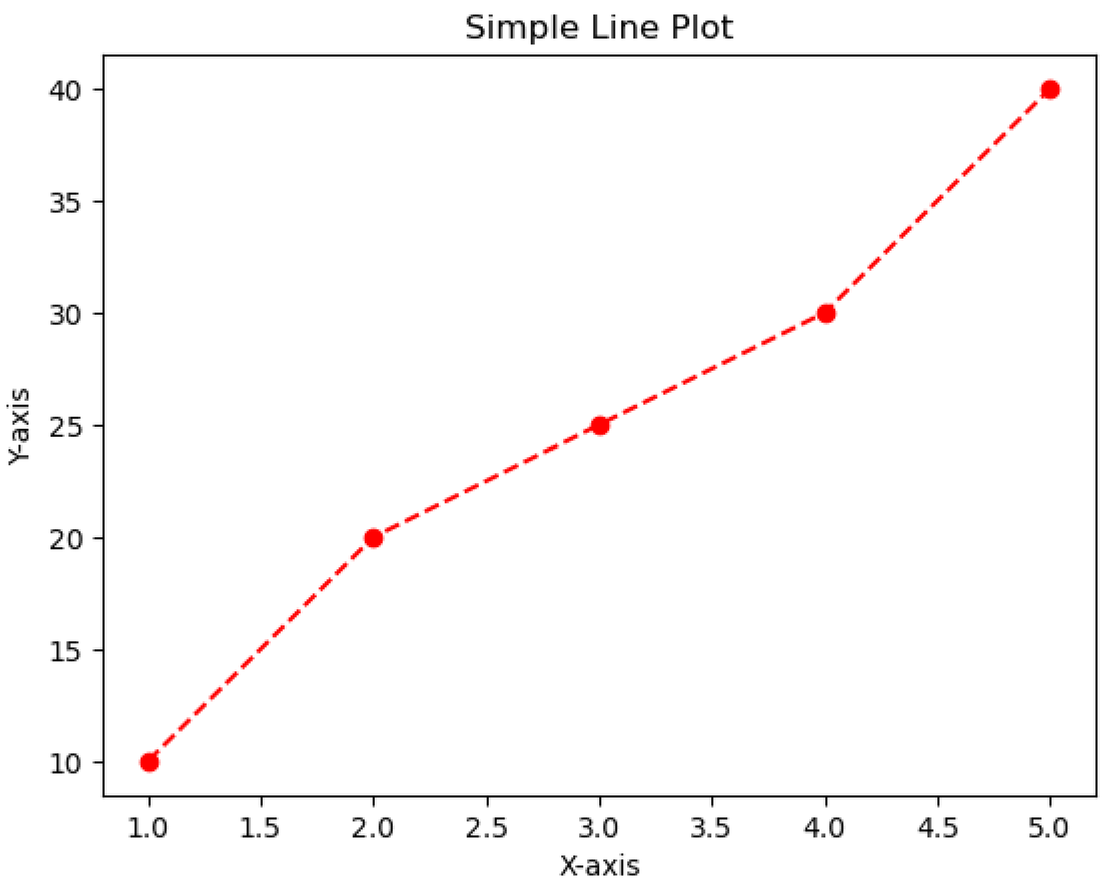
In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

Matplotlib Plots

a. Line Plot

A Line Plot is used to visualize trends over time or continuous data. It helps in identifying patterns, relationships, and fluctuations in data.

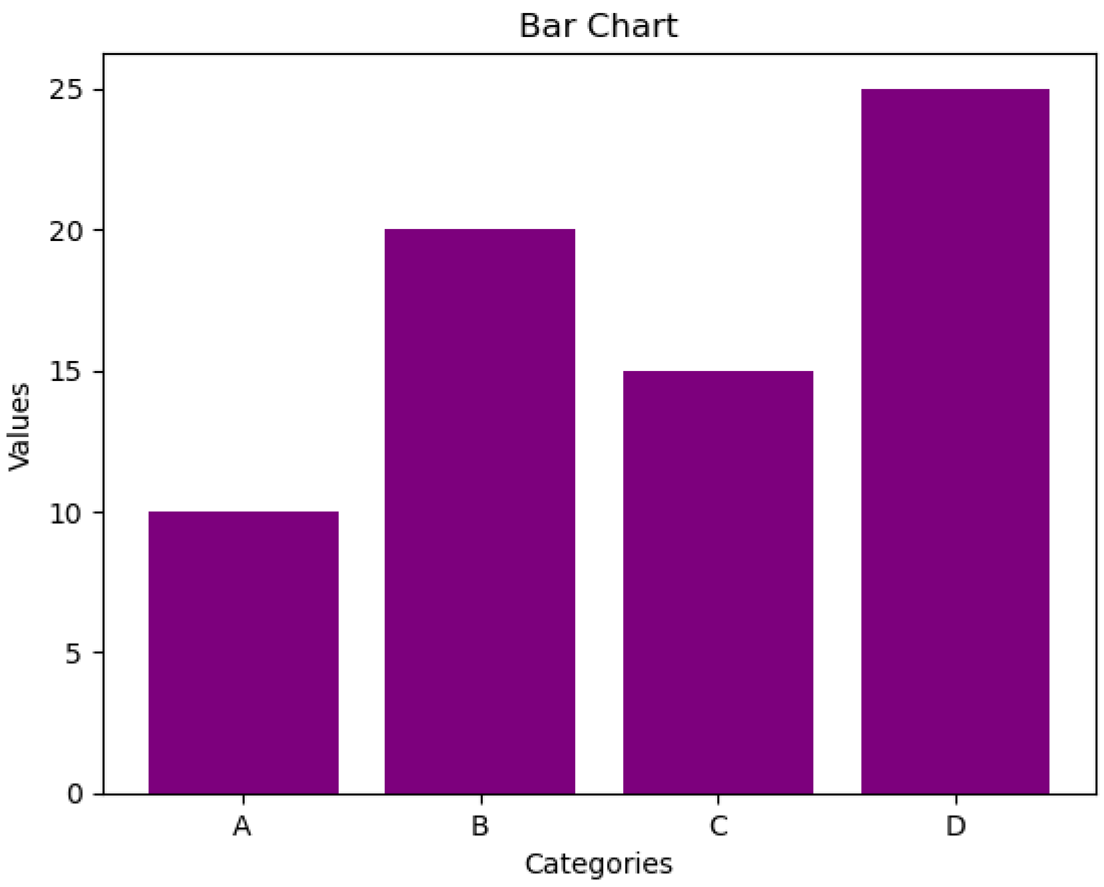
In [4]:
```
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y, marker='o', linestyle='--', color='r')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.show()
```



b. Bar Plot

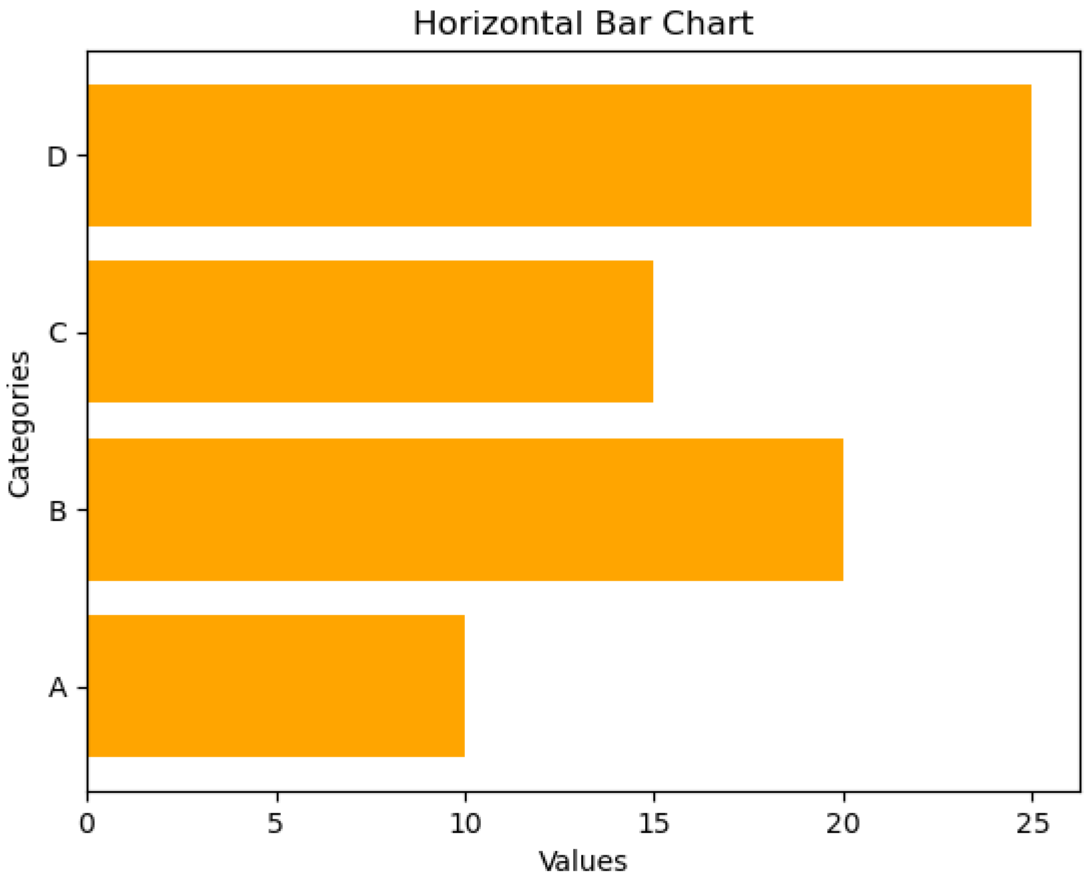
A Bar Plot is a common data visualization used to compare different categories by showing their values as rectangular bars.

In [5]:
```
x = ['A', 'B', 'C', 'D']
y = [10, 20, 15, 25]
plt.bar(x, y, color='purple')
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Chart")
plt.show()
```



Horizontal Bar Plot

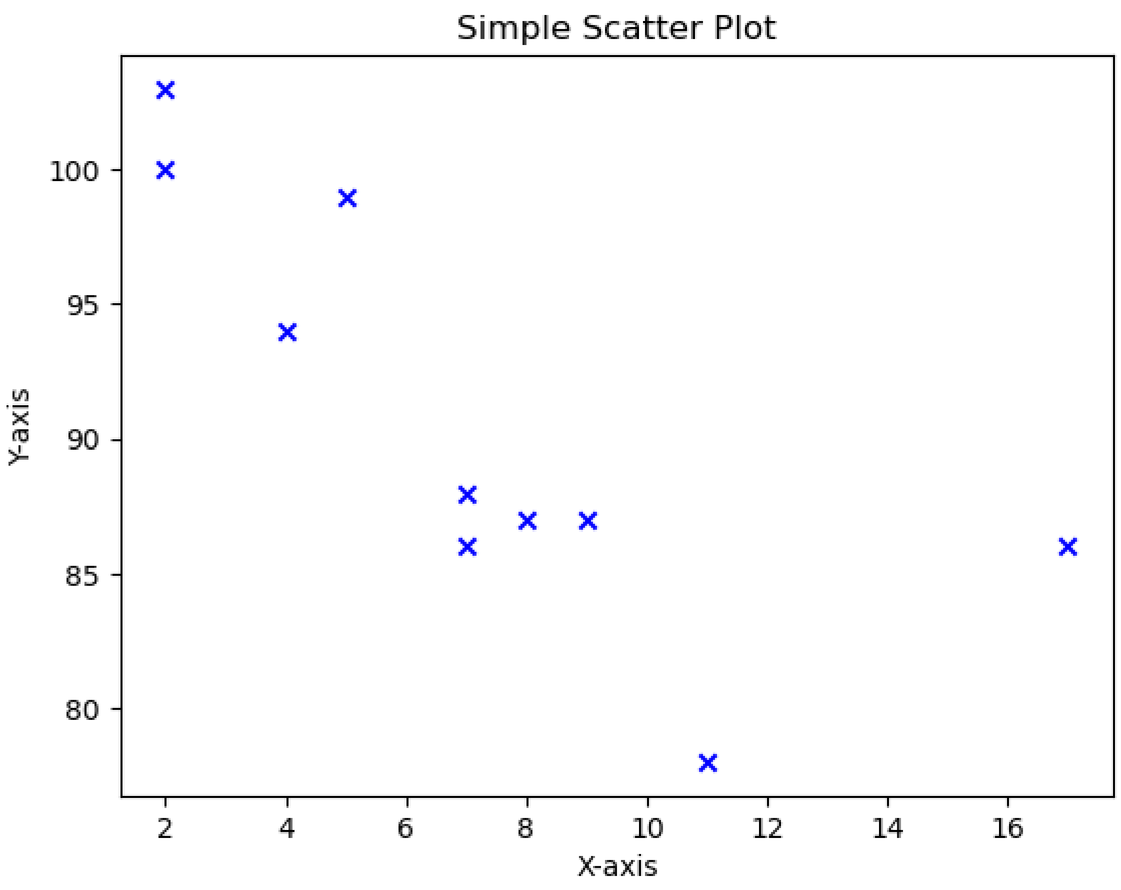
In [6]:
```
plt.barh(x, y, color='orange')
plt.xlabel("Values")
plt.ylabel("Categories")
plt.title("Horizontal Bar Chart")
plt.show()
```



c. Scatter Plot

A scatter plot is a type of data visualization that uses dots to represent the values of two different numeric variables. It’s commonly used to observe and show relationships between two continuous variables.

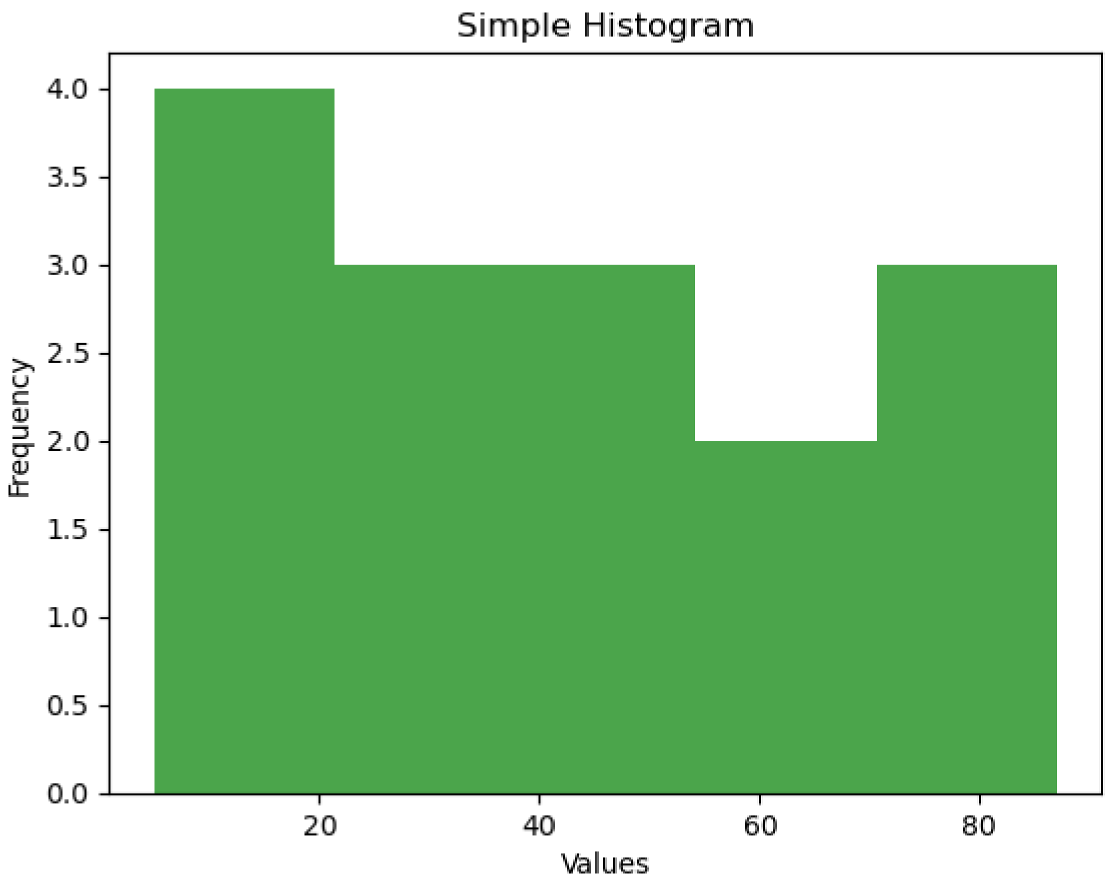
In [7]:
```
x = [5, 7, 8, 7, 2, 17, 2, 9, 4, 11]
y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78]

plt.scatter(x, y, color='blue', marker='x')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Scatter Plot")
plt.show()
```



d. Histogram

A histogram is a graphical representation of the distribution of a dataset. It consists of bars that represent the frequency of values within specified ranges (bins).

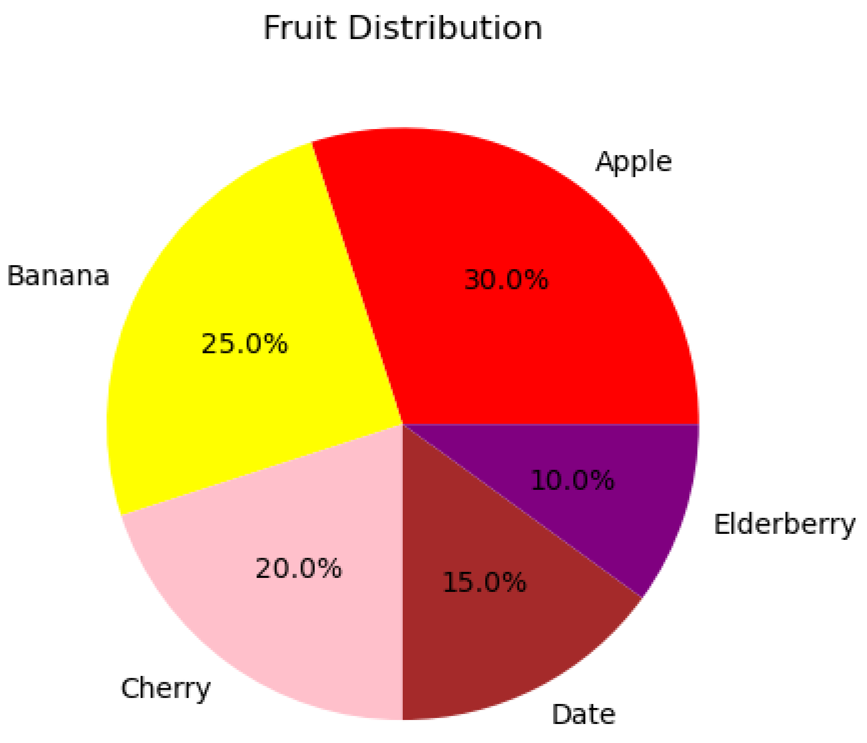
In [8]:
```
data = [22, 87, 5, 43, 56, 73, 55, 54, 11, 20, 51, 5, 79, 31, 27]
plt.hist(data, bins=5, color='green', alpha=0.7)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Simple Histogram")
plt.show()
```



e. Pie Chart

A pie chart is a circular statistical graphic divided into slices to illustrate proportions. Each slice represents a category’s percentage of the total.

In [9]:
```
sizes = [30, 25, 20, 15, 10]
labels = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
colors = ['red', 'yellow', 'pink', 'brown', 'purple']

plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors)
plt.title("Fruit Distribution")
plt.show()
```



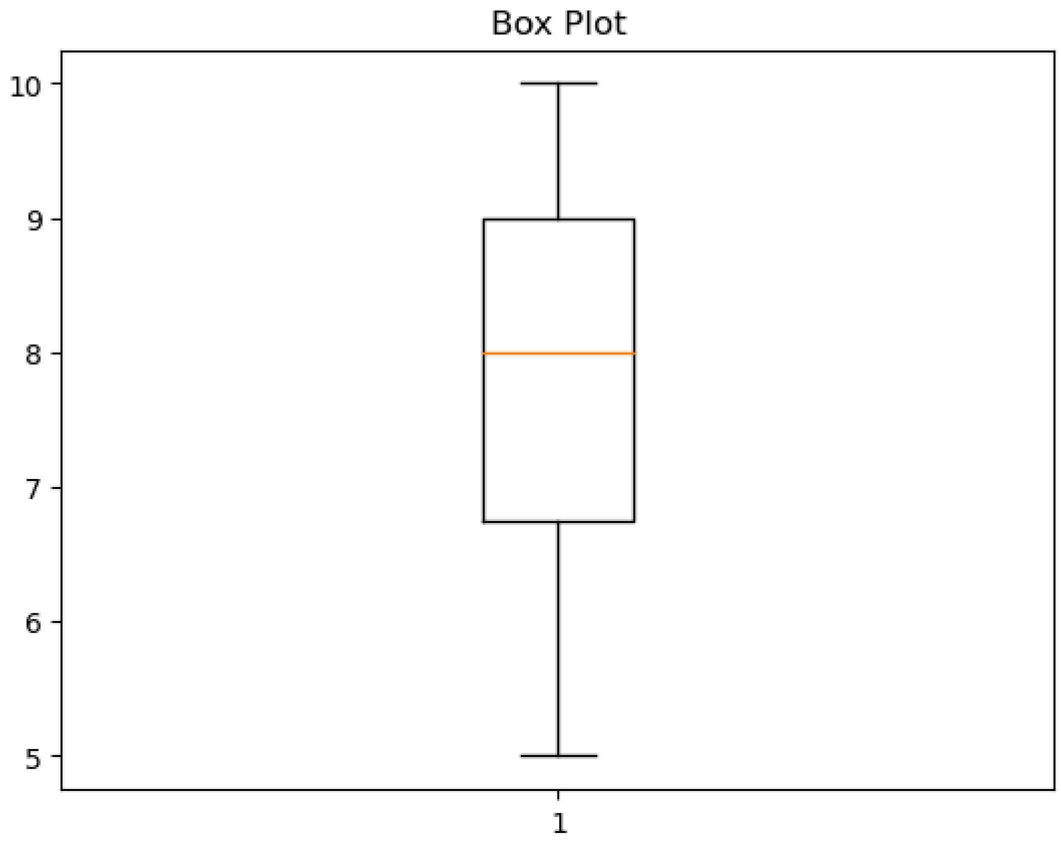
f. Box Plot

A Box Plot (also called a Box-and-Whisker Plot) is a statistical visualization that represents the distribution of a dataset through five key summary statistics.

```
Minimum – The smallest value in the dataset (excluding outliers).
First Quartile (Q1) – The 25th percentile (lower quartile), where
25% of the data falls below this value.
Median (Q2) – The 50th percentile, dividing the dataset into two e
qual halves.
Third Quartile (Q3) – The 75th percentile (upper quartile), where
75% of the data falls below this value.
Maximum – The largest value in the dataset (excluding outliers).
```

It is particularly useful for detecting outliers, understanding spread, and comparing distributions across categories.

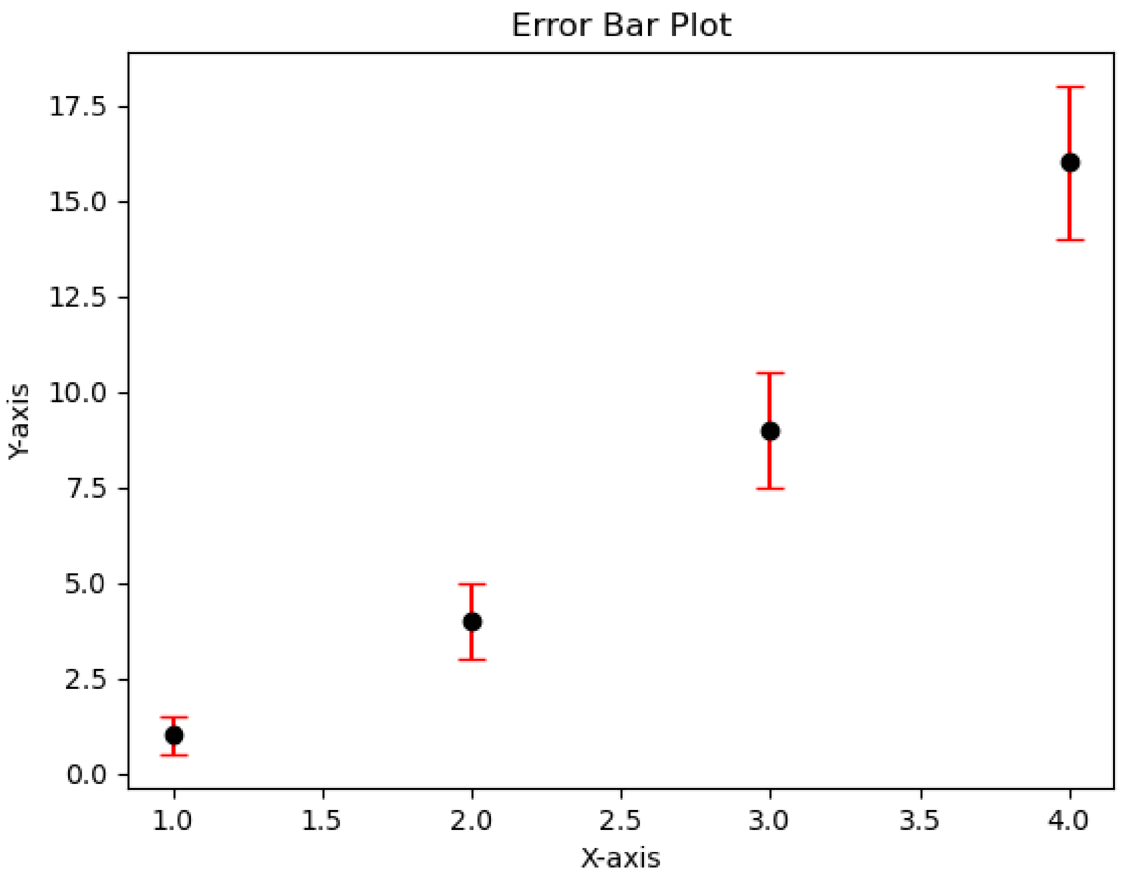
In [10]:
```
data = [7, 8, 9, 5, 6, 9, 8, 9, 10, 6, 7, 8]

plt.boxplot(data)
plt.title("Box Plot")
plt.show()
```



g. Error Bar Plot

An Error Bar Plot is a type of plot used to represent variability or uncertainty in data. It shows error margins around data points, indicating confidence intervals, standard deviation, or measurement errors.

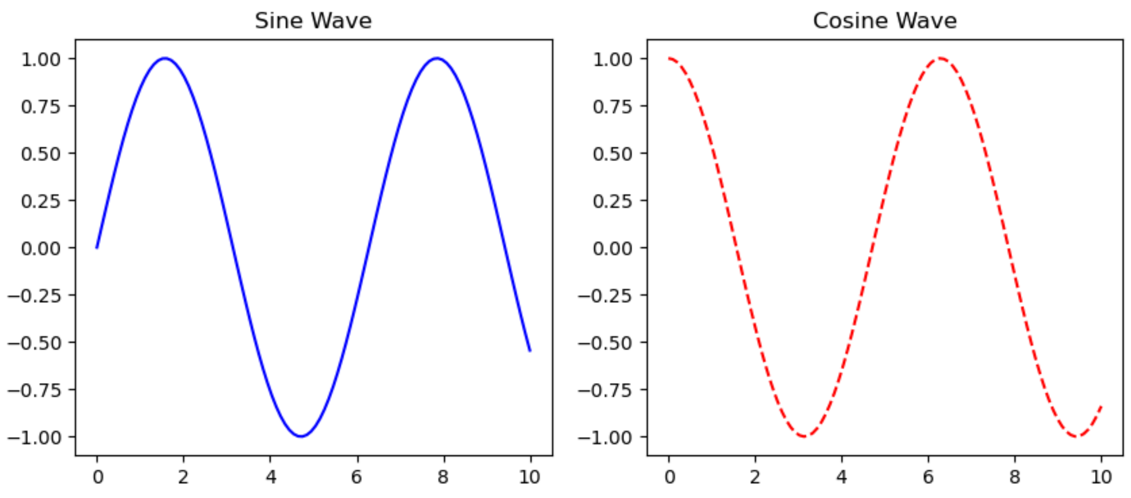
In [11]:
```
x = np.arange(1, 5)
y = x ** 2
errors = [0.5, 1, 1.5, 2]

plt.errorbar(x, y, yerr=errors, fmt='o', color='black', ecolor='red', capsi
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Error Bar Plot")
plt.show()
```



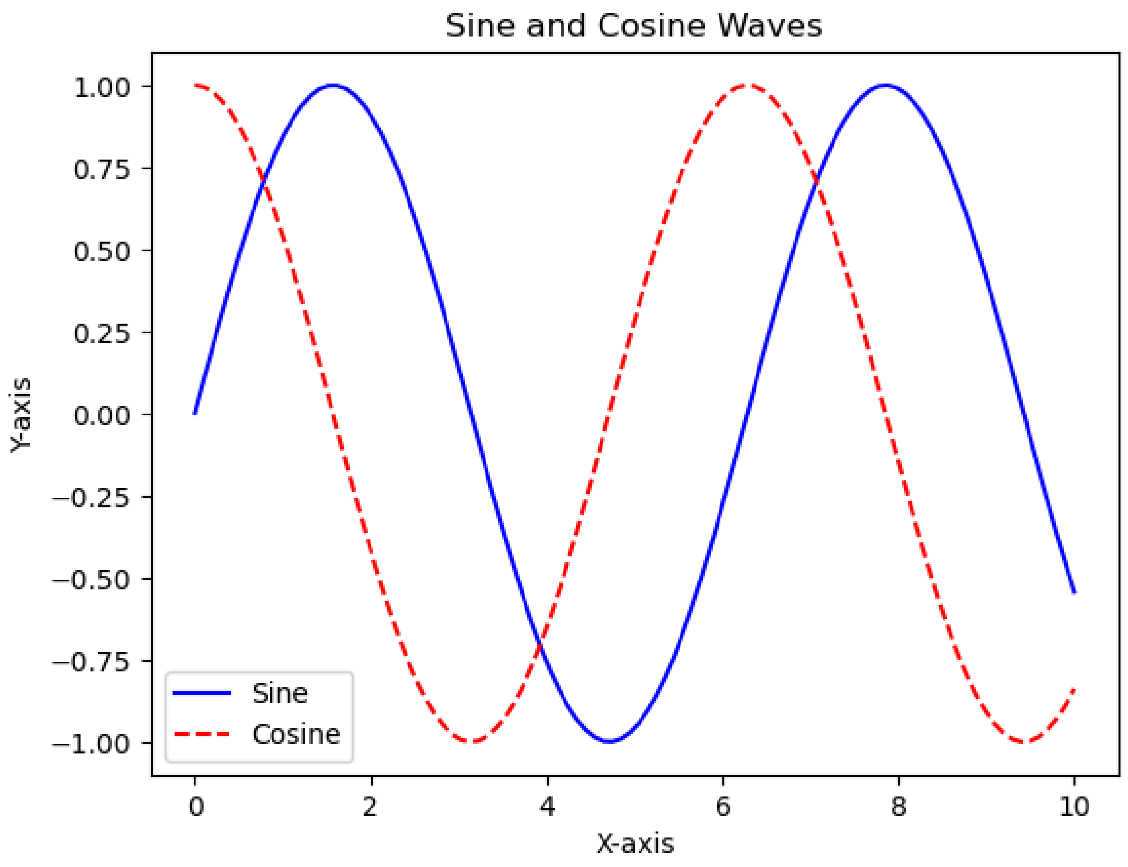
h. Subplots (Multiple Plots in One Figure)

Subplots allow you to display multiple plots within a single figure in Matplotlib. This is useful when comparing multiple visualizations side by side, analyzing trends across different datasets, or presenting different perspectives of the same data.

In [6]:
```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

axes[0].plot(x, y1, color='b', label='Sine')
axes[0].set_title("Sine Wave")

axes[1].plot(x, y2, color='r', linestyle='--', label='Cosine')
axes[1].set_title("Cosine Wave")

plt.show()
```



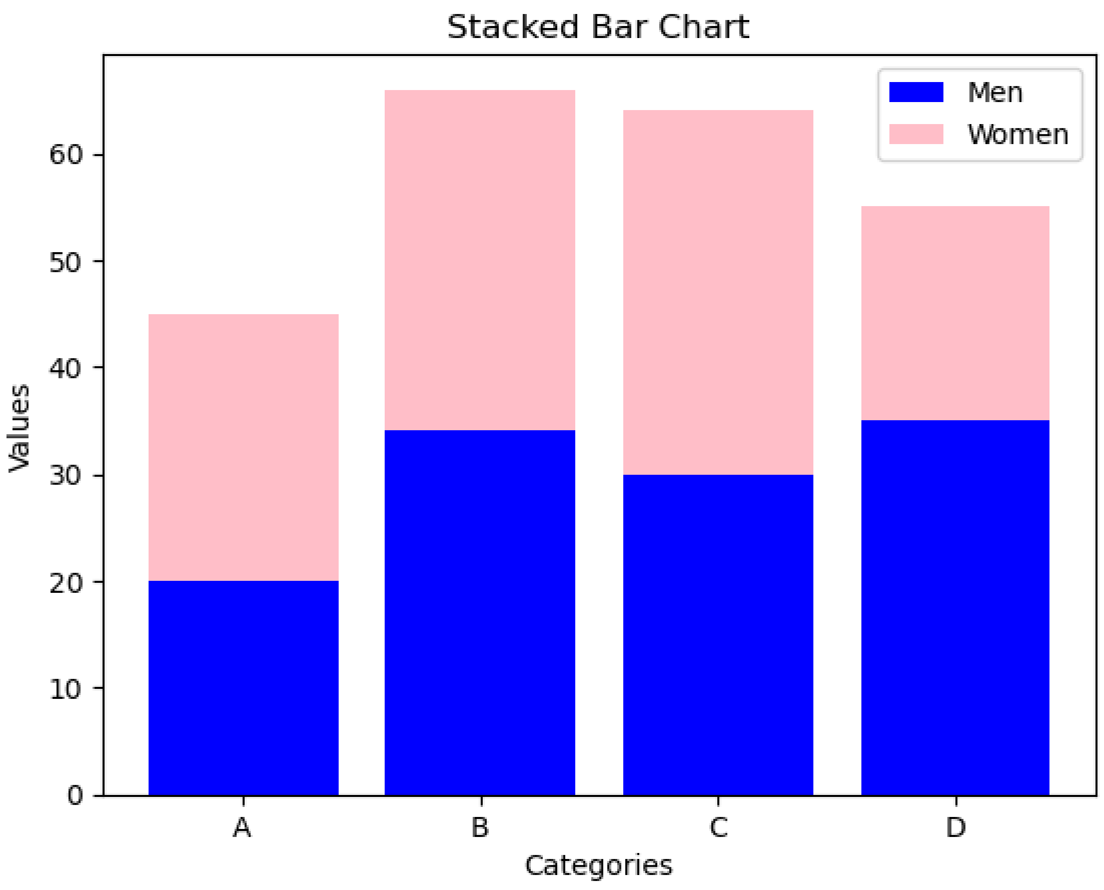
In [7]:
```python
# Create the plot
plt.plot(x, y1, color='b', label='Sine')
plt.plot(x, y2, color='r', linestyle='--', label='Cosine')

# Add labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Sine and Cosine Waves")

# Add a legend
plt.legend()

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



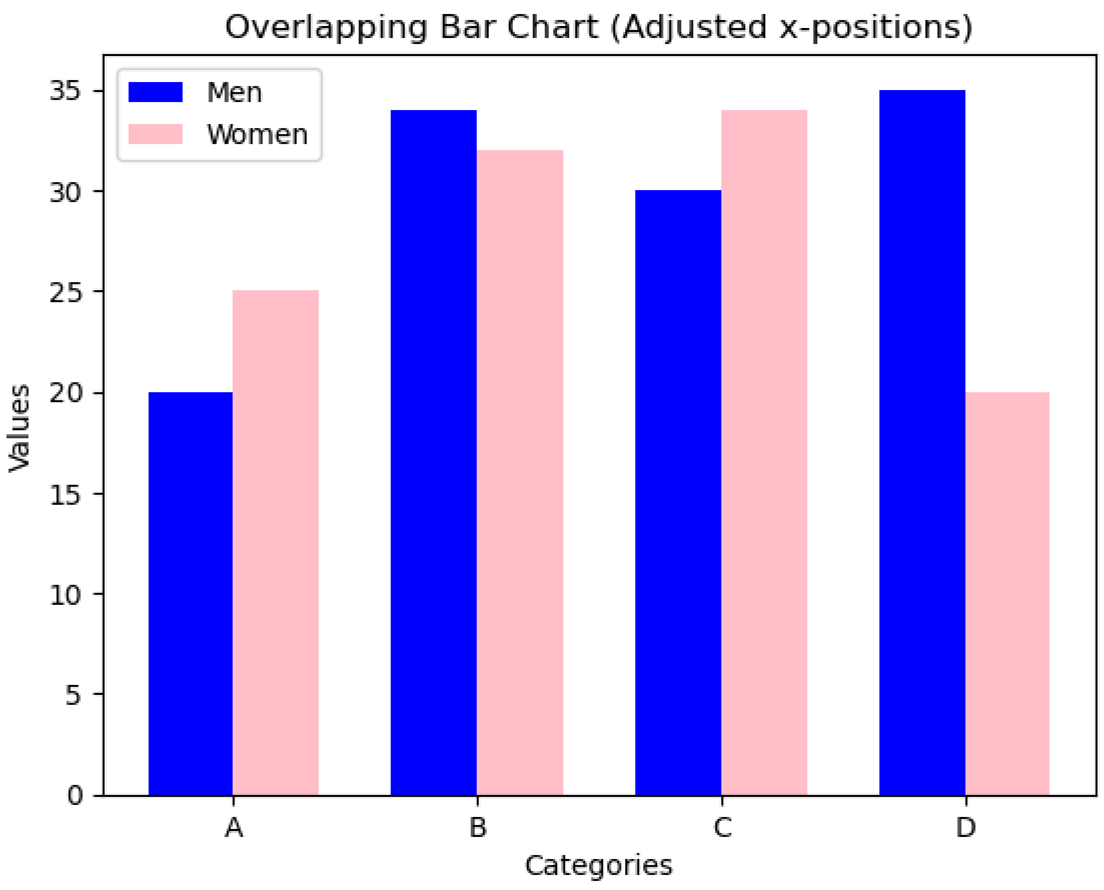
i. Stacked Bar Chart

A Stacked Bar Chart is a type of bar chart used to compare parts of a whole across different categories. It represents data using rectangular bars that are stacked on top of each other, showing the contribution of each segment to the total.

In [9]:
```python
categories = ['A', 'B', 'C', 'D']
men = [20, 34, 30, 35]
women = [25, 32, 34, 20]

plt.bar(categories, men, label='Men', color='blue')
plt.bar(categories, women, bottom=men, label='Women', color='pink')

plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Stacked Bar Chart")
plt.legend()
plt.show()
```



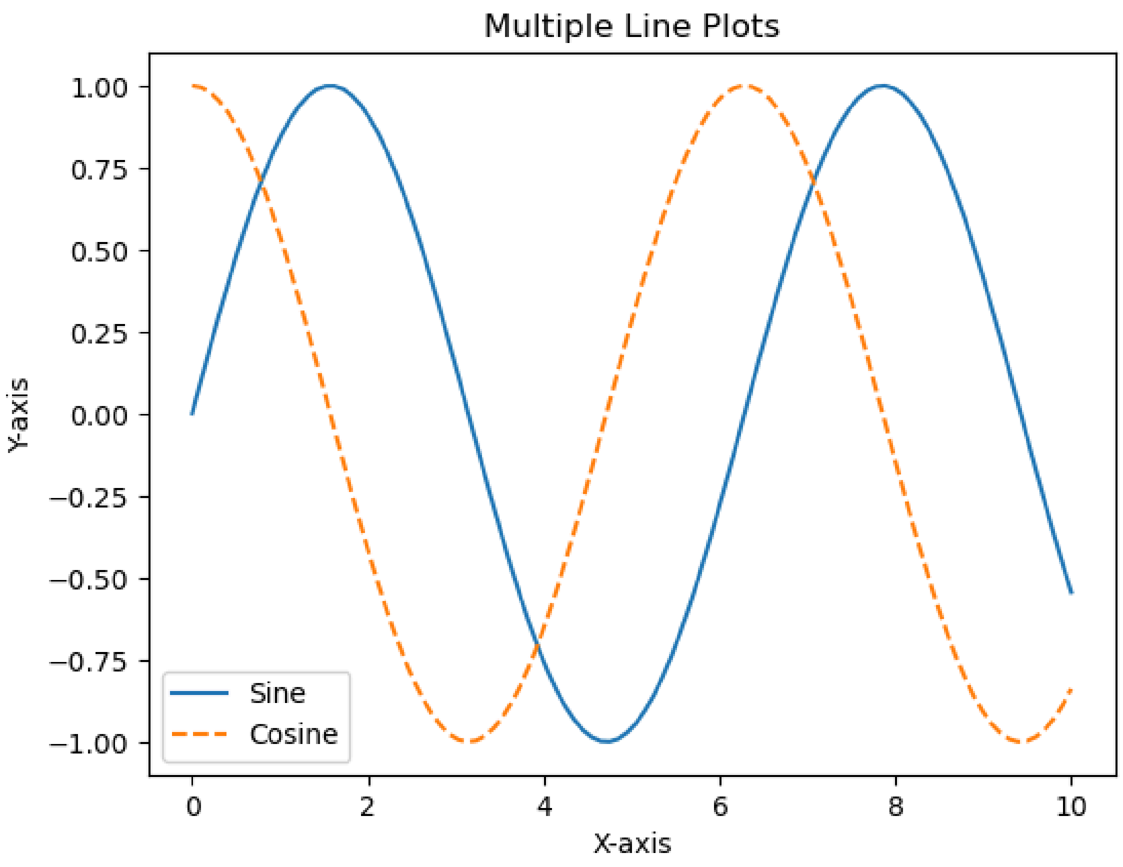
In [15]:
```
x = np.arange(len(categories))  # the label locations
width = 0.35  # the width of the bars

plt.bar(x - width/2, men, width, label='Men', color='blue')
plt.bar(x + width/2, women, width, label='Women', color='pink')

plt.xticks(x, categories)  # Ensure x-axis labels are correct
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Overlapping Bar Chart (Adjusted x-positions)")
plt.legend()
plt.show()
```



j. Multiple Line Plots

A Multiple Line Plot is a type of line chart that displays two or more lines on the same axes to compare trends or relationships between different datasets over a common variable, typically time or categories.

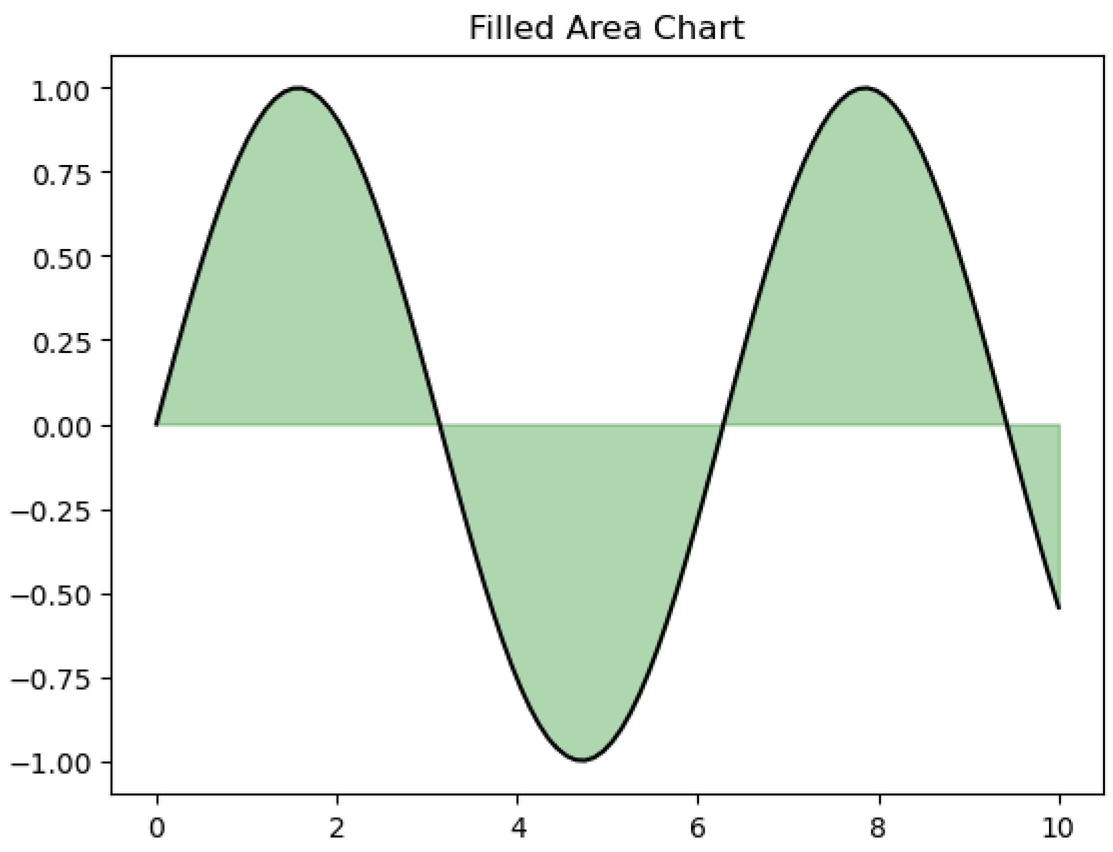
In [14]:
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, label="Sine", linestyle='-')
plt.plot(x, y2, label="Cosine", linestyle='--')

plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Multiple Line Plots")
plt.legend()
plt.show()
```



k. Filled Area Chart

A Filled Area Chart is a variation of a line chart where the area between the line and the x-axis (or another reference line) is filled with color to emphasize magnitude or trends over time.

In [15]:
```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.fill_between(x, y, alpha=0.3, color='green')
plt.plot(x, y, color='black')
plt.title("Filled Area Chart")
plt.show()
```



I. Radar Chart (Spider Plot)

A Radar Chart (also called a Spider Chart, Web Chart, or Polar Chart) is a graphical method for displaying multivariate data in a two-dimensional chart with three or more quantitative variables represented on axes starting from the same central point.

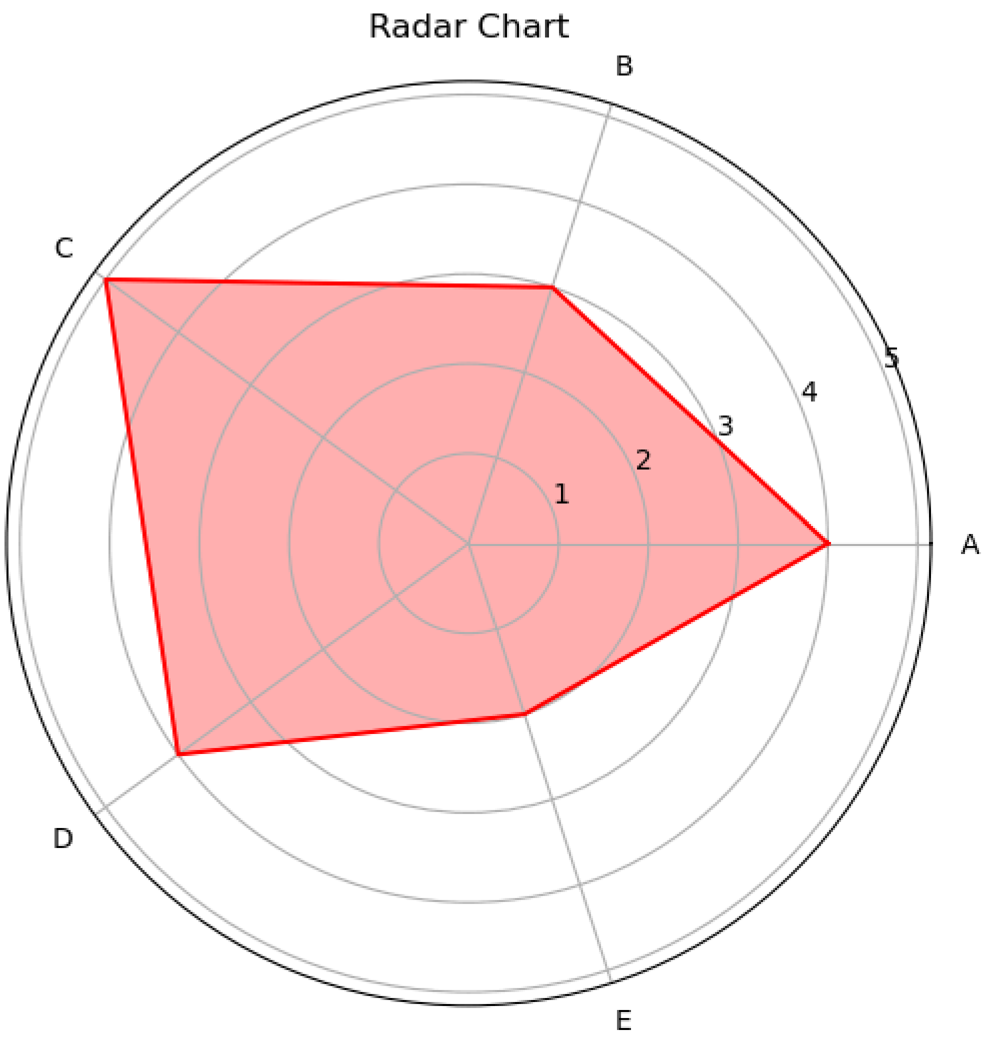
1. labels = ['A', 'B', 'C', 'D', 'E']: This defines the labels for the axes of the radar chart. Each label represents a different category or attribute.
2. values = [4, 3, 5, 4, 2]: This defines the values associated with each category. These values determine how far each point is from the center of the radar chart.
3. angles = np.linspace(0, 2 * np.pi, len(labels), endpoint=False).tolist(): This calculates the angles at which each axis should be placed.
    - np.linspace(0, 2 * np.pi, len(labels), endpoint=False): Creates evenly spaced numbers between 0 and 2π (a full circle) based on the number of labels. endpoint=False is important so that the last angle does not coincide with the first.
    - .tolist(): Converts the NumPy array to a Python list.
4. values += values[:1] and angles += angles[:1]: These lines are crucial for closing the radar chart. They append the first value and angle to the end of the values and angles lists. This connects the last point back to the first, creating the closed polygon shape.
5. fig, ax = plt.subplots(figsize=(6,6), subplot_kw=dict(polar=True)): This creates the figure and the axes for the radar chart.

- plt.subplots(figsize=(6,6), subplot_kw=dict(polar=True)): Creates a subplot with a polar projection (polar=True). figsize sets the size of the figure.
- fig: The figure object.
- ax: The axes object, which we'll use to draw on.

6. ax.fill(angles, values, color='red', alpha=0.3): This fills the area enclosed by the radar chart.

   - ax.fill(): Fills a polygon.
   - angles: The angles of the vertices.
   - values: The radial distances of the vertices.
   - color='red': The fill color.
   - alpha=0.3: The transparency of the fill.
7. ax.plot(angles, values, color='red'): This draws the lines connecting the data points. This creates the outline of the radar chart.
8. ax.set_xticks(angles[:-1]): Sets the positions of the x-axis ticks (the radial lines). angles[:-1] is used to exclude the last angle, as it is a duplicate for closing the shape.
9. ax.set_xticklabels(labels): Sets the labels for the x-axis ticks.

In [16]:
```
labels = ['A', 'B', 'C', 'D', 'E']
values = [4, 3, 5, 4, 2]

angles = np.linspace(0, 2 * np.pi, len(labels), endpoint=False).tolist()
values += values[:1]
angles += angles[:1]

fig, ax = plt.subplots(figsize=(6,6), subplot_kw=dict(polar=True))
ax.fill(angles, values, color='red', alpha=0.3)
ax.plot(angles, values, color='red')
ax.set_xticks(angles[:-1])
ax.set_xticklabels(labels)
plt.title("Radar Chart")
plt.show()
```

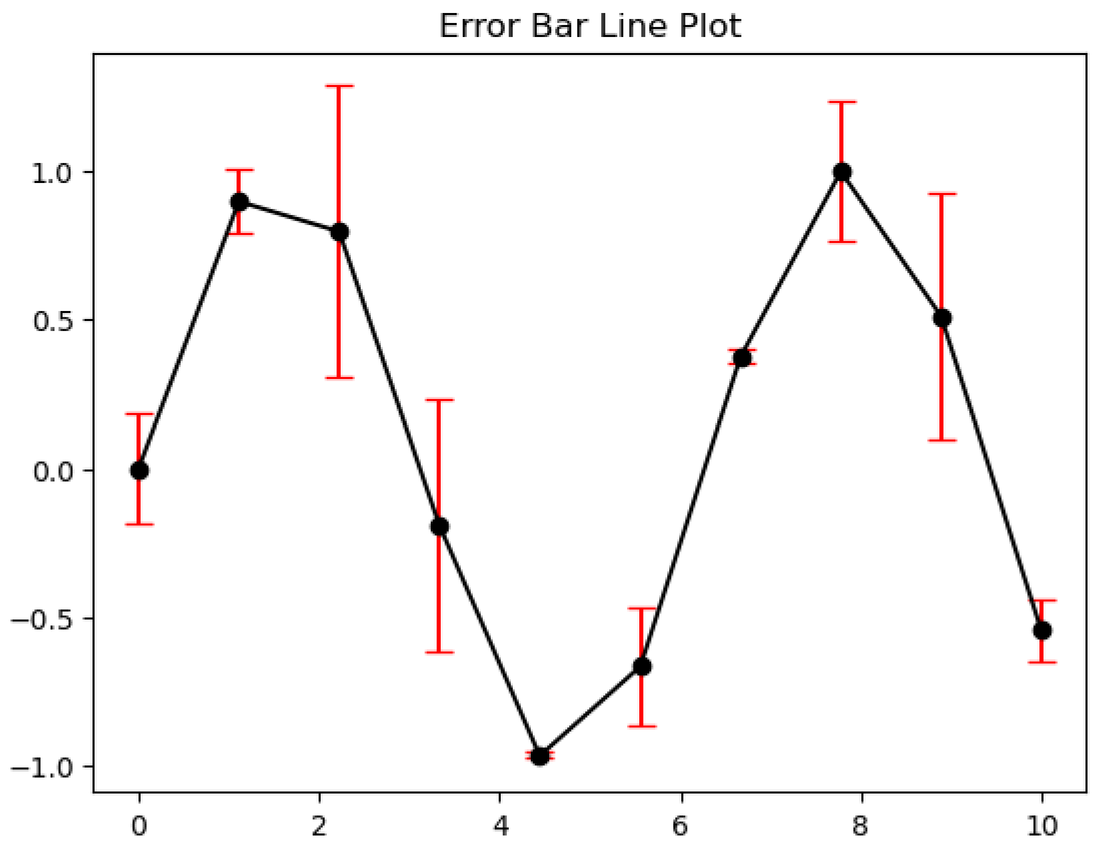


m. Error Bars in Line Plot

Error bars represent uncertainty or variation in data points. They visually indicate the possible range of values based on standard deviation, confidence intervals, or measurement errors.

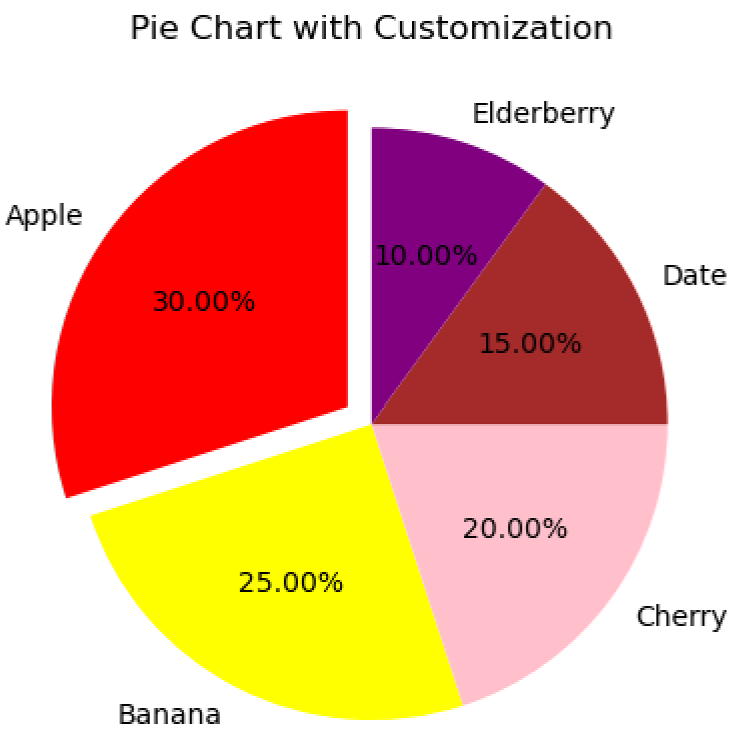
In [16]:
```
errors = np.random.rand(10) * 0.5
```

In [6]:
```
x = np.linspace(0, 10, 10)
y = np.sin(x)
errors = np.random.rand(10) * 0.5

plt.errorbar(x, y, yerr=errors, fmt='o-', color='black', ecolor='red', caps
plt.title("Error Bar Line Plot")
plt.show()
```



n. Pie Chart with Customizations

In [9]:
```
sizes = [30, 25, 20, 15, 10]
labels = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
colors = ['red', 'yellow', 'pink', 'brown', 'purple']

plt.pie(sizes, labels=labels, autopct='%1.2f%%', colors=colors, startangle=
plt.title("Pie Chart with Customization")
plt.show()
```



o. 3D Scatter Plot

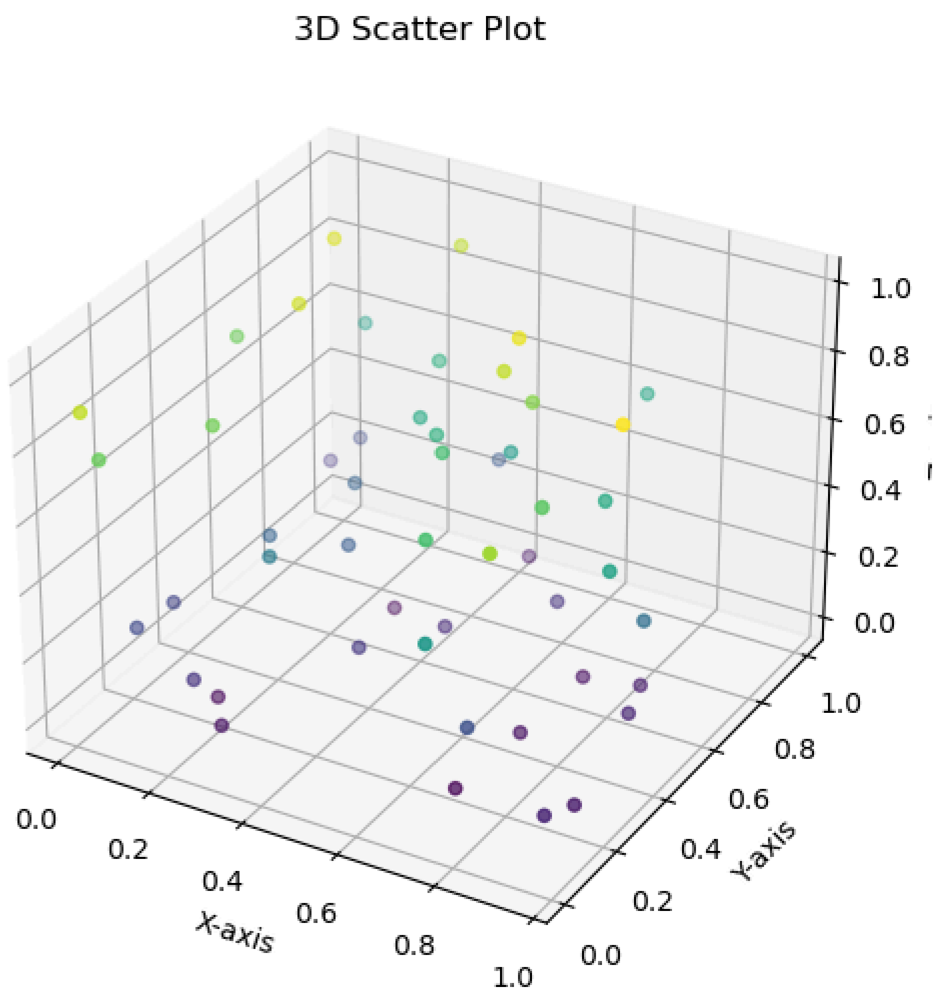
1. fig = plt.figure(figsize=(8, 6)): This line creates the figure object, which is the overall canvas for the plot. figsize=(8, 6) sets the size of the figure to 8 inches wide and 6 inches tall.
2. ax = fig.add_subplot(111, projection='3d'): This adds a subplot to the figure. 111 is a compact way of saying "1 row, 1 column, first plot". projection='3d' is crucial; it tells Matplotlib that this subplot should be a 3D plot. The ax object is what you'll use to manipulate the plot itself.
3. x = np.random.rand(50), y = np.random.rand(50), z = np.random.rand(50): These lines generate 50 random numbers between 0 and 1 for each of the x, y, and z coordinates. np.random.rand(50) uses NumPy to create an array of 50 random floats.
4. ax.scatter(x, y, z, c=z, cmap='viridis'): This is the core of the scatter plot. ax.scatter(x, y, z) plots the points using the generated x, y, and z coordinates. c=z sets the color of each point based on its z-value. cmap='viridis' uses the 'viridis' colormap, which is a perceptually uniform colormap that's good for visualizing data. Other colormaps like 'plasma', 'magma', 'inferno', or 'coolwarm' could also be used.

   - Positioning: The x, y, and z arguments together determine the 3D location of each point. For example, the first point will be at coordinates (x[0], y[0], z[0]), the second at (x[1], y[1], z[1]), and so on.

- Coloring: The c=z argument tells Matplotlib to use the z-value of each point to determine its color. Points with higher z-values will have different colors than points with lower z-values. The cmap='viridis' part specifies the colormap to use for this mapping (viridis is a color scale that goes from purple to yellow).

5. ax.set_xlabel("X-axis"), ax.set_ylabel("Y-axis"), ax.set_zlabel("Z-axis"): These lines set the labels for the x, y, and z axes, making the plot more informative.
6. plt.title("3D Scatter Plot"): This sets the title of the plot.

In [17]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

x = np.random.rand(50)
y = np.random.rand(50)
z = np.random.rand(50)

ax.scatter(x, y, z, c=z, cmap='viridis')
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
ax.set_zlabel("Z-axis")
plt.title("3D Scatter Plot")
plt.show()
```

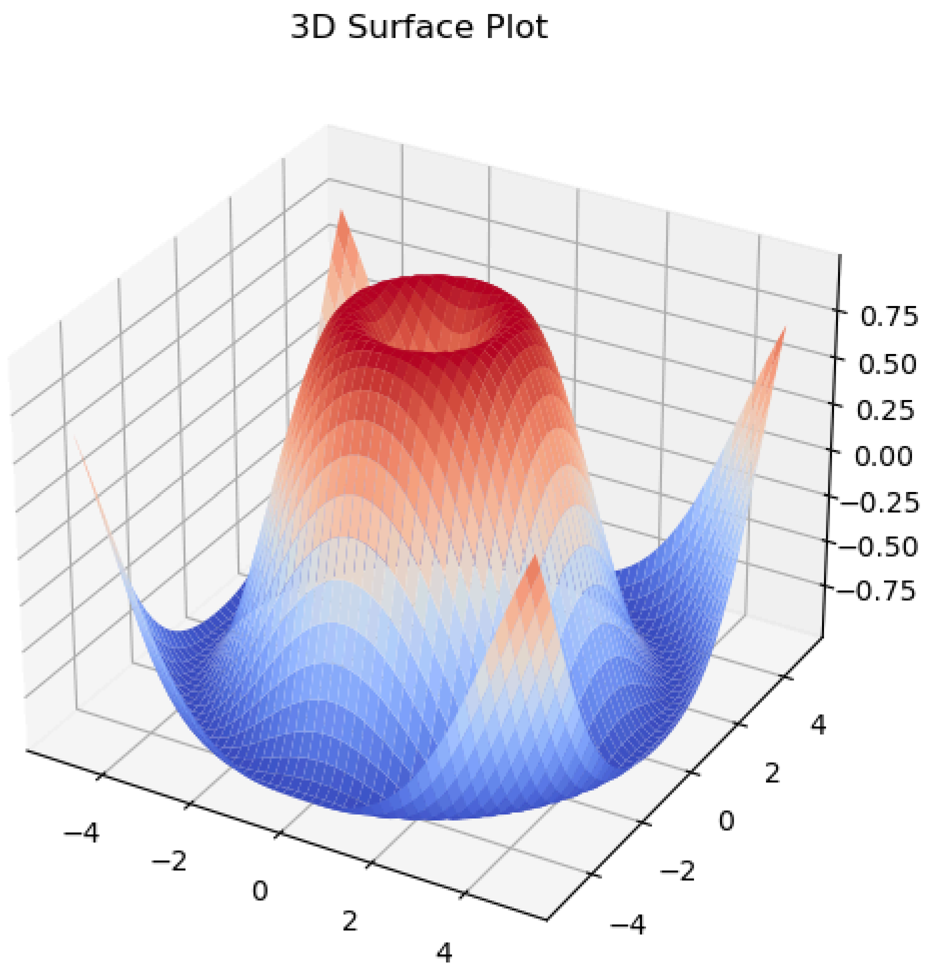


p. 3D Surface Plot

A 3D Surface Plot is a three-dimensional visualization that represents continuous data as a surface in a 3D space. It is primarily used to study relationships between three numerical variables (X, Y, and Z).

1. x = np.linspace(-5, 5, 50) and y = np.linspace(-5, 5, 50): These lines create two arrays, x and y, using np.linspace(). np.linspace(-5, 5, 50) generates 50 evenly spaced numbers between -5 and 5 (inclusive). These arrays will define the x and y coordinates of the grid on which the surface will be plotted.
2. X, Y = np.meshgrid(x, y): np.meshgrid() is a crucial function here. It takes the 1D arrays x and y and creates two 2D arrays, X and Y. These X and Y arrays represent all the possible combinations of x and y values. Think of it as creating a grid of points. X will contain the x-coordinate for each point on the grid, and Y will contain the corresponding y-coordinate.
3. Z = np.sin(np.sqrt(X**2 + Y**2)): This line calculates the z-value for each point on the grid. It's defining a function z = sin(sqrt(x^2 + y^2)). NumPy's ability to work with arrays element-wise makes this calculation very efficient. X**2 and Y**2 square each element of the X and Y arrays, respectively. np.sqrt() calculates the square root, and np.sin() calculates the sine. The result, Z, is a 2D array of z-values, one for each point on the grid. This Z array defines the height of the surface at each (X, Y) point.
4. fig = plt.figure(figsize=(8, 6)) and ax = fig.add_subplot(111, projection='3d'): These are the same as in the scatter plot example. They create the figure and the 3D subplot where the surface will be drawn.
5. ax.plot_surface(X, Y, Z, cmap='coolwarm', edgecolor='none'): This is the core of the surface plot. ax.plot_surface() takes the X, Y, and Z arrays and creates the 3D surface. cmap='coolwarm' sets the colormap to 'coolwarm', which ranges from blue to red. edgecolor='none' removes the black lines that are sometimes drawn around the faces of the surface, making it look smoother.
6. ax.set_title("3D Surface Plot") and plt.show(): These lines set the title of the plot and display it, respectively.

In [19]:
```
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, Z, cmap='coolwarm', edgecolor='none')
ax.set_title("3D Surface Plot")
plt.show()
```



q. Contour Plot

A contour plot is a graphical representation of 3D data in two dimensions using contour lines. Each contour line represents a constant value (similar to elevation maps), allowing visualization of gradients and patterns in data.

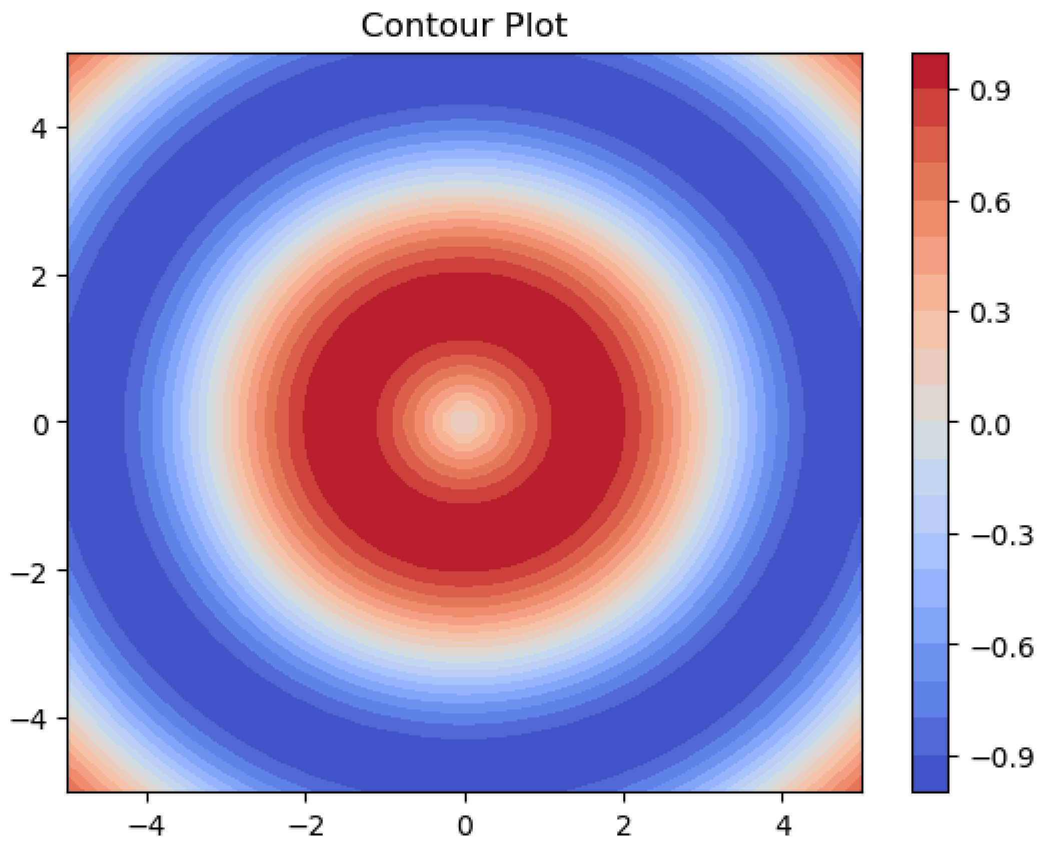
1. x = np.linspace(-5, 5, 50), y = np.linspace(-5, 5, 50), X, Y = np.meshgrid(x, y), and Z = np.sin(np.sqrt(X**2 + Y**2)): These lines are exactly the same as in the 3D surface plot code. They create the x and y coordinates and calculate the z-values using the function z = sin(sqrt(x^2 + y^2)).
2. plt.contourf(X, Y, Z, levels=20, cmap='coolwarm'): This is the core of the contour plot.

    - plt.contourf(): This function creates a filled contour plot. The 'f' stands for filled. If you used plt.contour() instead, you would get only the contour lines, without the filled colors.

- X, Y, and Z: These are the same arrays calculated earlier, providing the x, y, and z values.
- levels=20: This specifies the number of contour levels (or the number of filled regions). The plot will divide the range of z-values into 20 intervals, and each interval will be represented by a different color. You can also provide a sequence of specific z-values for the levels instead of the number of levels.
- cmap='coolwarm': This sets the colormap, just like in the 3D plot. It determines the colors used to fill the regions between the contour lines.

3. plt.colorbar(): This adds a colorbar to the plot. The colorbar shows the correspondence between the colors used in the contour plot and the z-values. It helps you understand what range of z-values each color represents.
4. plt.title("Contour Plot") and plt.show(): These lines set the title and display the plot.

In [21]:
```python
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

plt.contourf(X, Y, Z, levels=20, cmap='coolwarm')
plt.colorbar()
plt.title("Contour Plot")
plt.show()
```



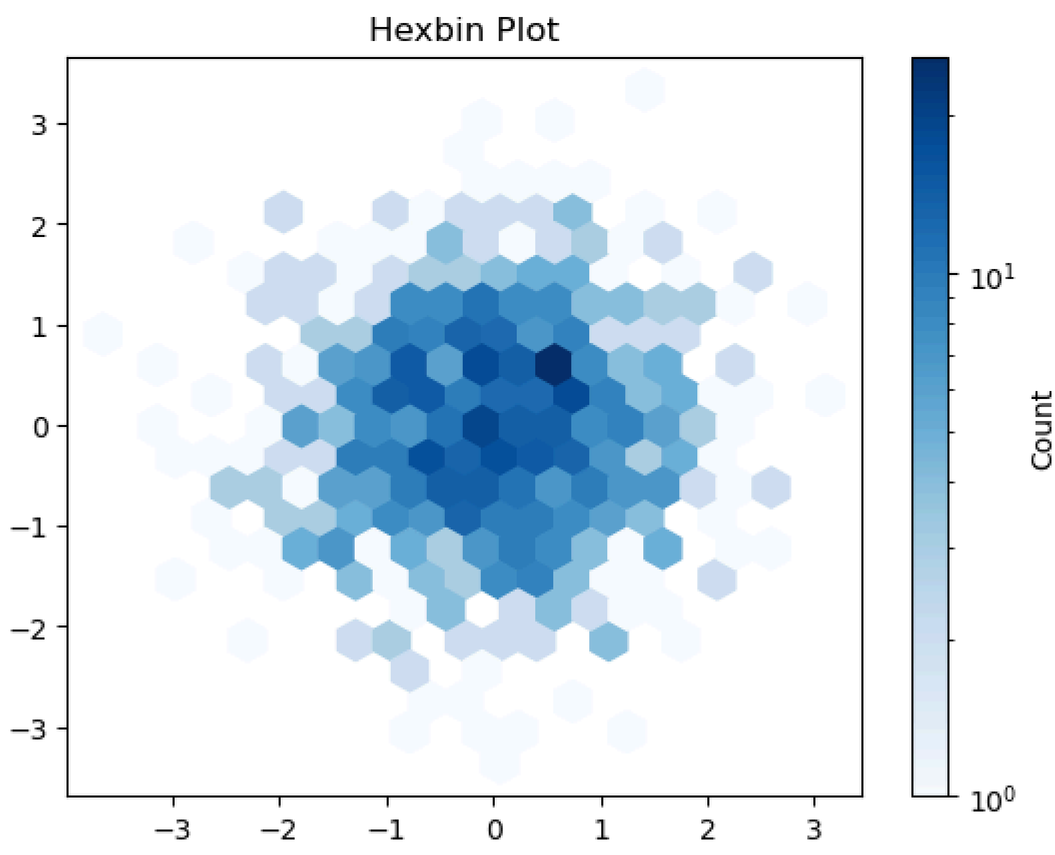
r. Hexbin Plot (Density Visualization)

1. x = np.random.randn(1000) and y = np.random.randn(1000): These lines generate 1000 random numbers from a standard normal distribution (mean 0, standard deviation 1) for both the x and y coordinates. np.random.randn() is a convenient way to do this.

2. plt.hexbin(x, y, gridsize=30, cmap='Blues', norm=LogNorm()): This is the core of the hexbin plot.

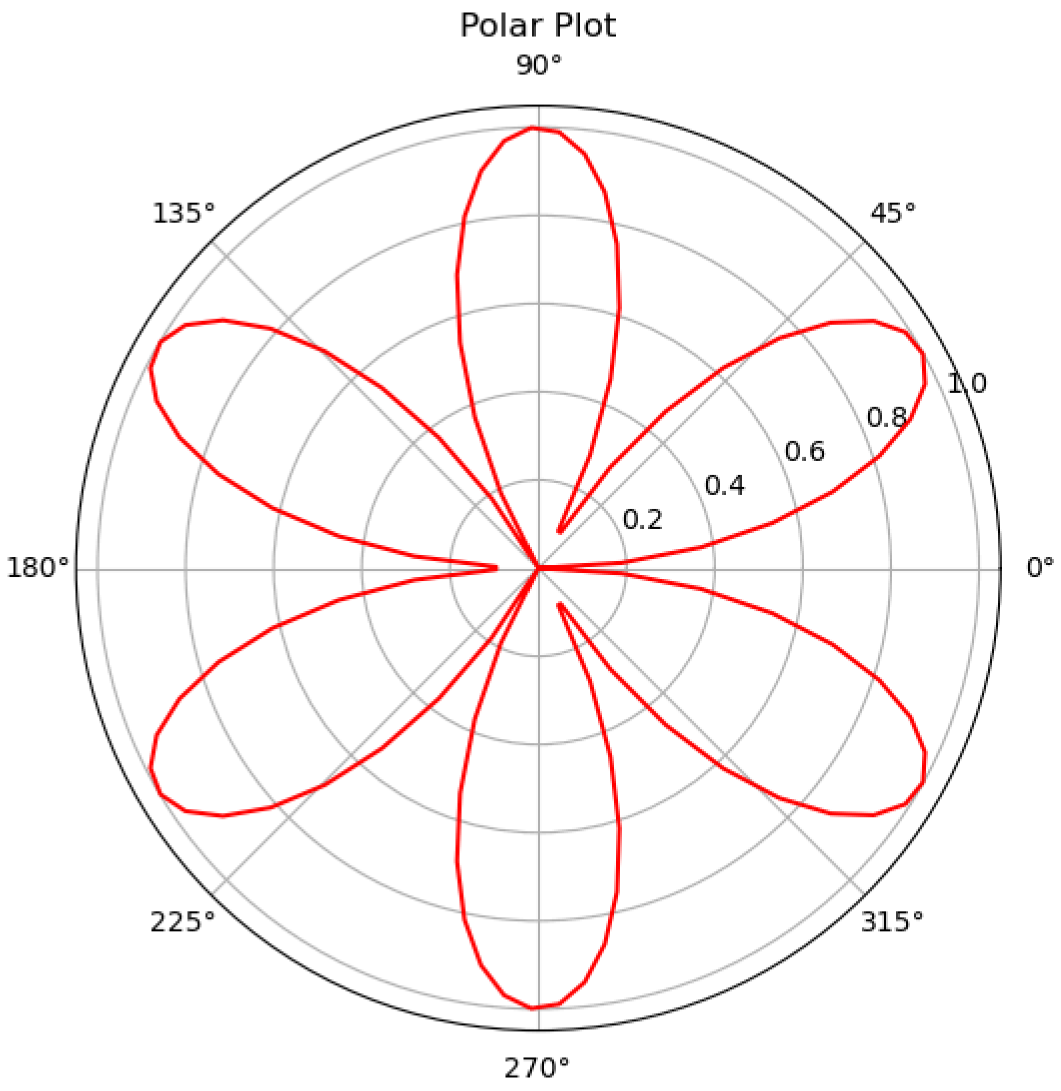
   - plt.hexbin(x, y): This function creates the hexbin plot using the x and y coordinates.
   - gridsize=30: This parameter controls the resolution of the hexagonal grid. A smaller gridsize means finer hexagons (more detail), and a larger gridsize means coarser hexagons (less detail). Think of it as dividing the plot area into a grid of hexagons.
   - cmap='Blues': This sets the colormap to 'Blues', meaning the hexagons will be colored with shades of blue. Denser regions (more points) will be a darker blue, and sparser regions will be a lighter blue.
   - norm=LogNorm(): This is important for visualizing data with a wide range of densities. LogNorm() applies a logarithmic scaling to the color mapping. This helps to prevent very dense regions from completely dominating the color scale and making it difficult to see variations in less dense areas. If you didn't use LogNorm(), the differences in color between the less dense regions might be too subtle to perceive.
3. plt.colorbar(label='Count'): This adds a colorbar to the plot. The label='Count' argument sets the label of the colorbar to "Count," indicating that the color represents the number of points within each hexagon.

In [24]:
```
x = np.random.randn(1000)
y = np.random.randn(1000)

plt.hexbin(x, y, gridsize=20, cmap='Blues', norm=LogNorm())
plt.colorbar(label='Count')
plt.title("Hexbin Plot")
plt.show()
```



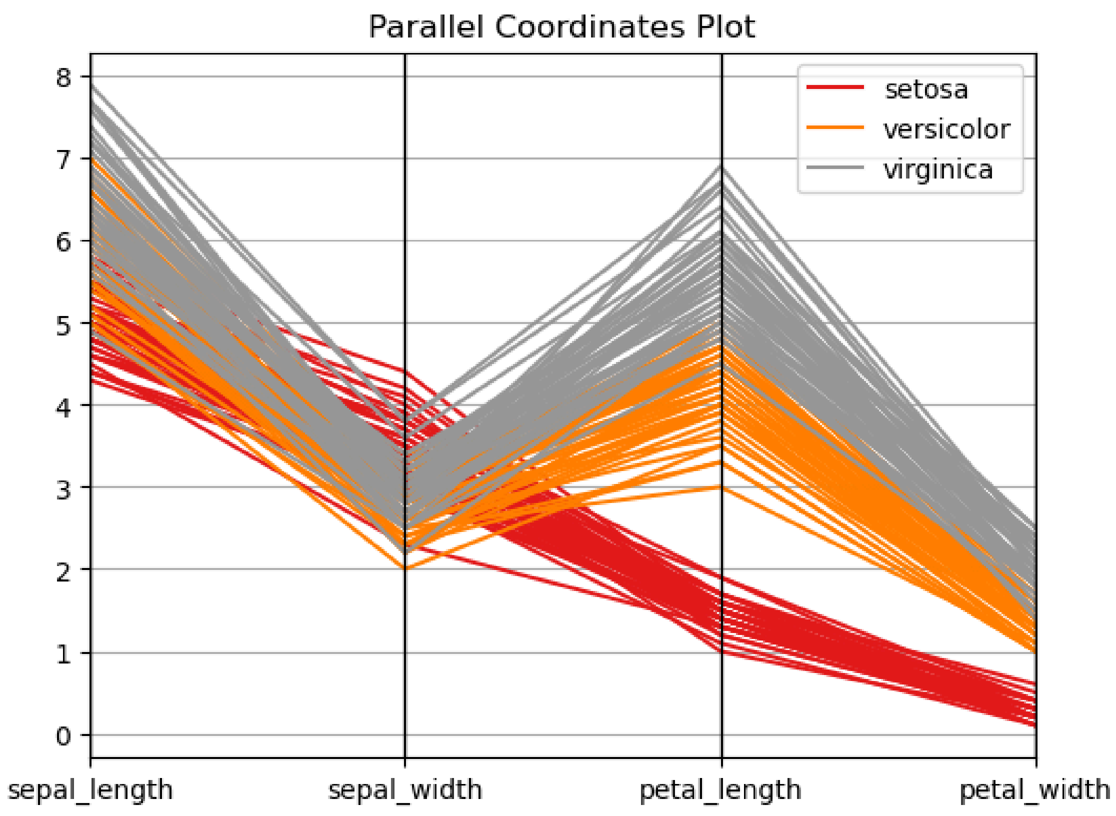
s. Polar Plot

In [23]:
```
theta = np.linspace(0, 2*np.pi, 100)
r = np.abs(np.sin(3*theta))

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='polar')

ax.plot(theta, r, color='red')
ax.set_title("Polar Plot")
plt.show()
```



t. Parallel Coordinates Plot

In [24]:
```python
from pandas.plotting import parallel_coordinates

df = sns.load_dataset("iris")
parallel_coordinates(df, class_column="species", colormap=plt.get_cmap("Set
plt.title("Parallel Coordinates Plot")
plt.show()
```



u. Interactive 3D Scatter Plot (Using Plotly)