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
```

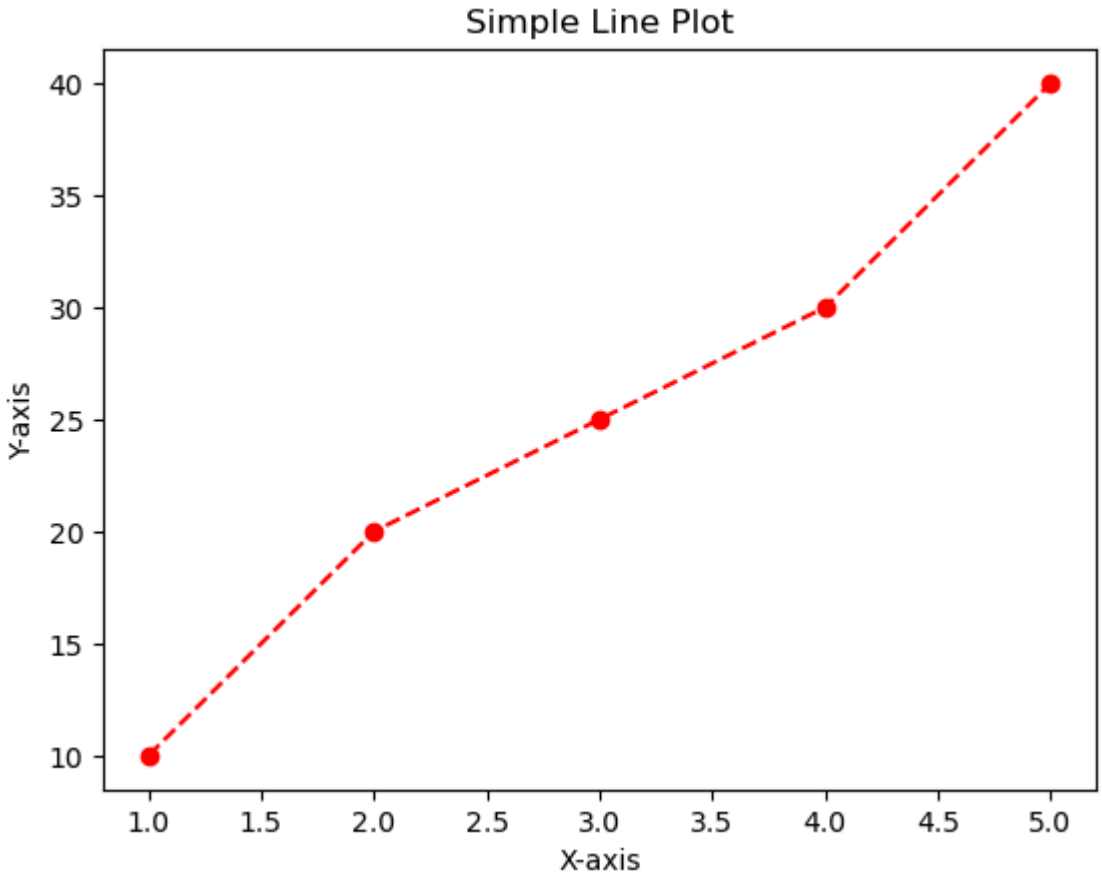
In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

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.

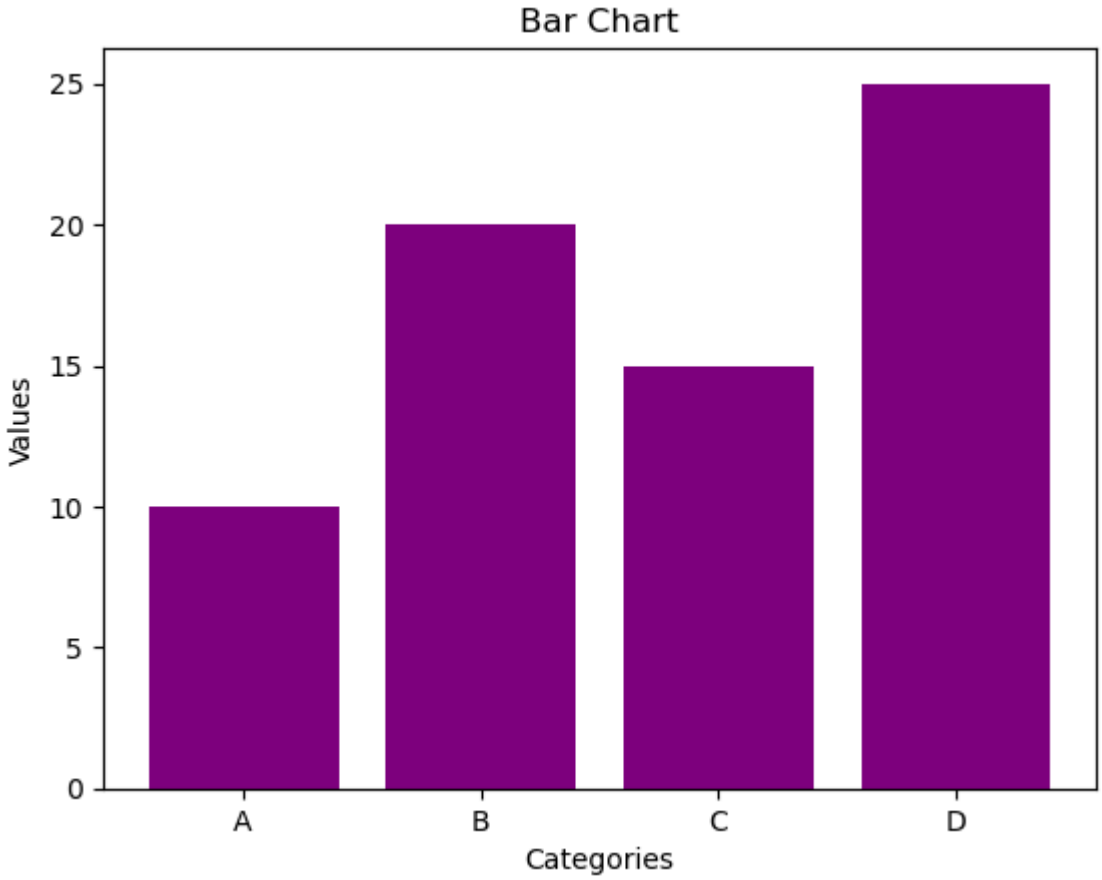
In [4]:
```
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y, marker='o', linestyle='--', color='r')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.show()
```



b. Bar Plot

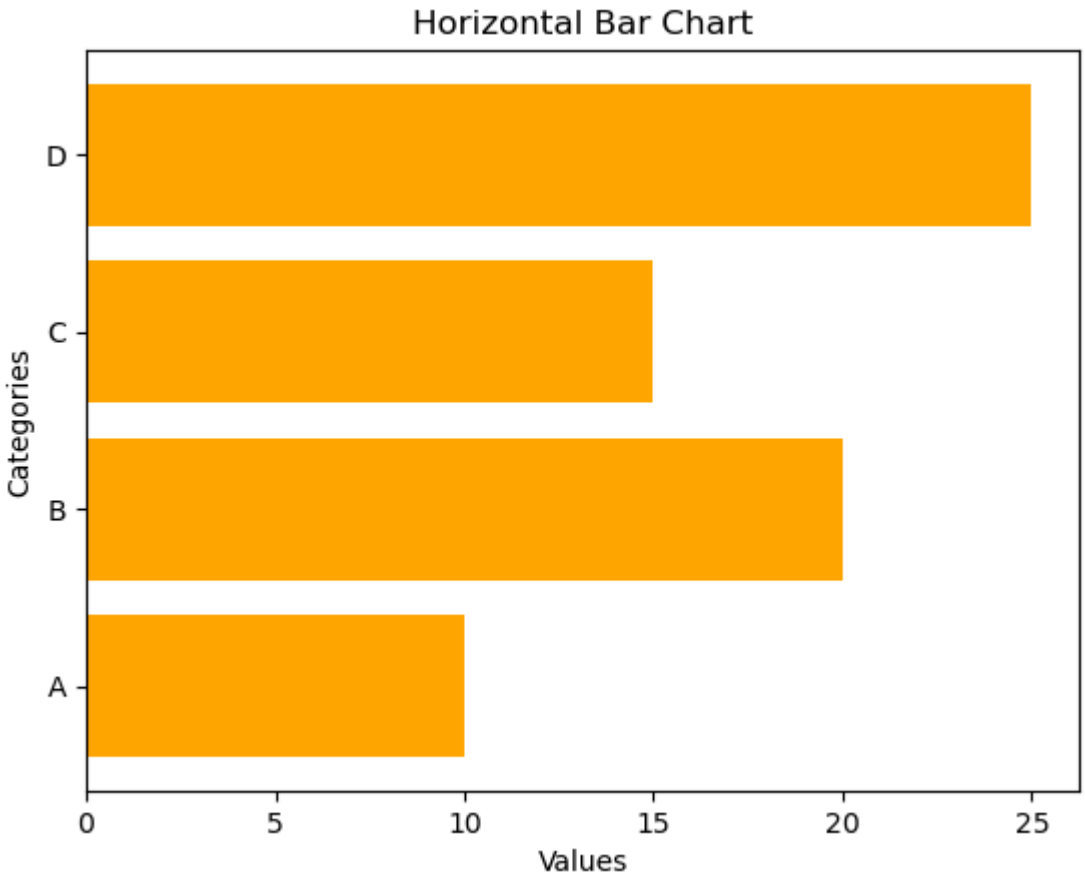
A Bar Plot is a common data visualization used to compare different categories by showing their values as rectangular bars.

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

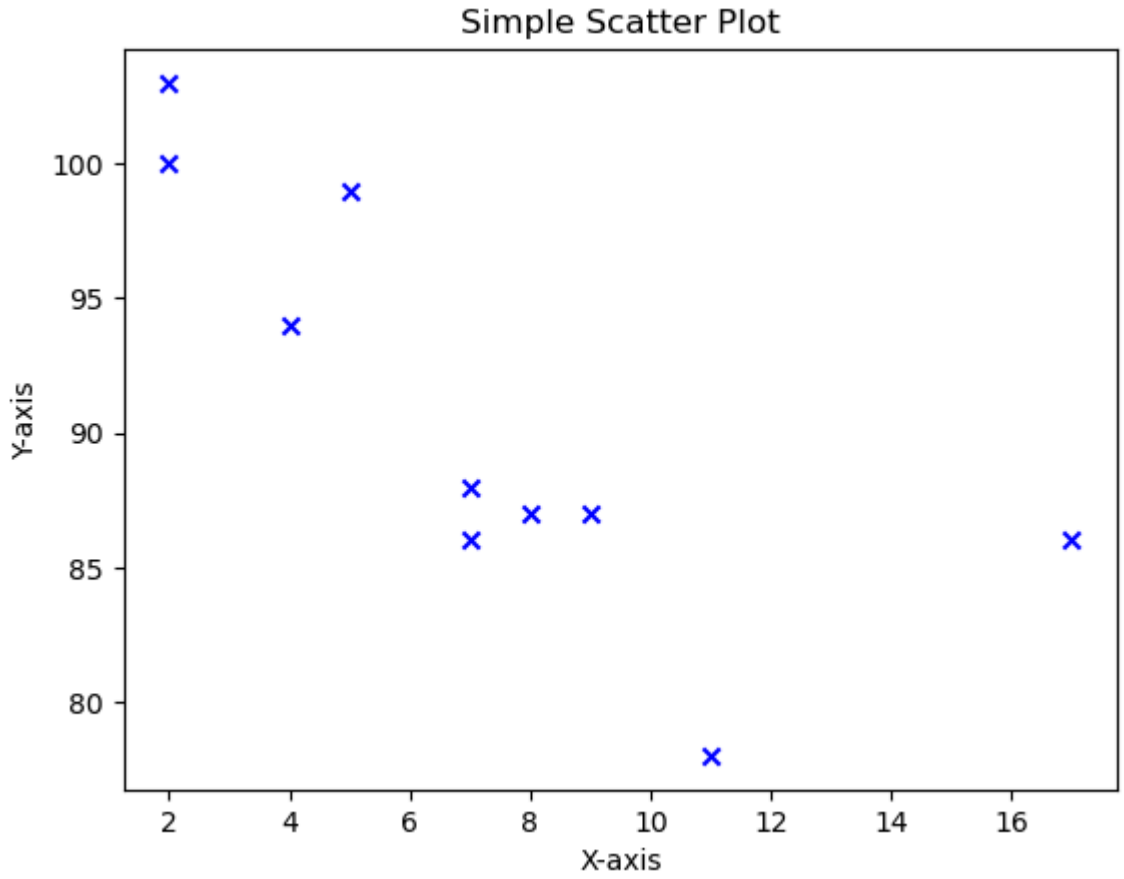
In [6]:
```
plt.barh(x, y, color='orange')
plt.xlabel("Values")
plt.ylabel("Categories")
plt.title("Horizontal Bar Chart")
plt.show()
```



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.

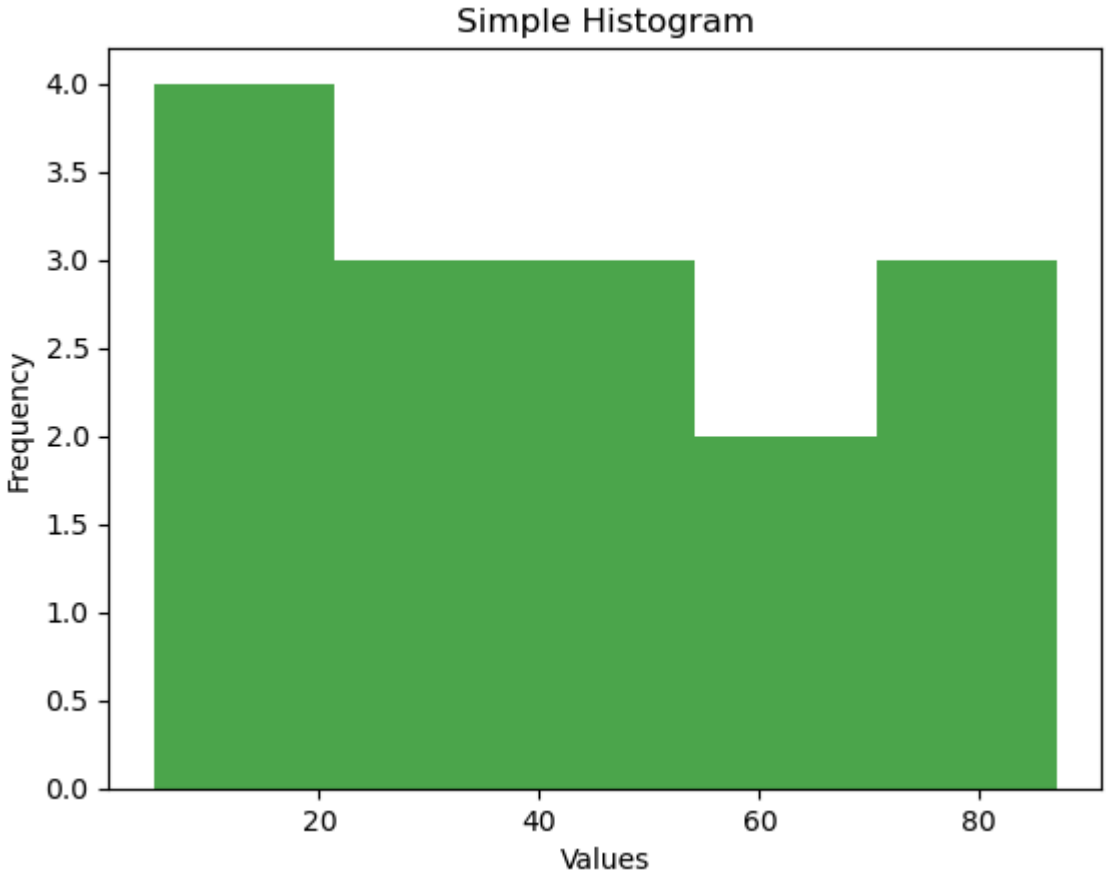
In [7]:
```
x = [5, 7, 8, 7, 2, 17, 2, 9, 4, 11]
y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78]

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).

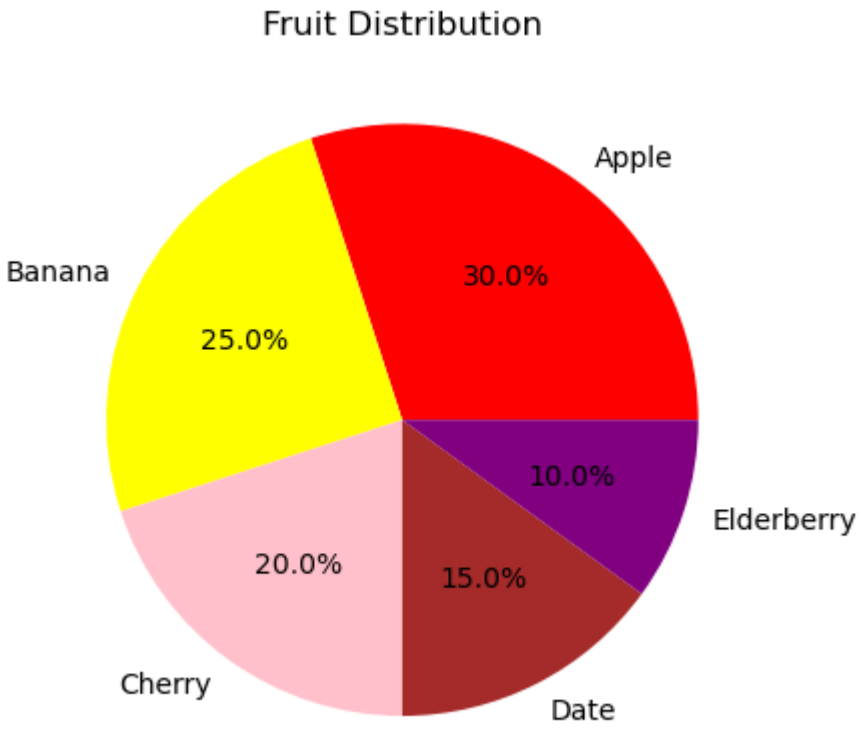
In [8]:
```
data = [22, 87, 5, 43, 56, 73, 55, 54, 11, 20, 51, 5, 79, 31, 27]
plt.hist(data, bins=5, color='green', alpha=0.7)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Simple Histogram")
plt.show()
```



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()
```



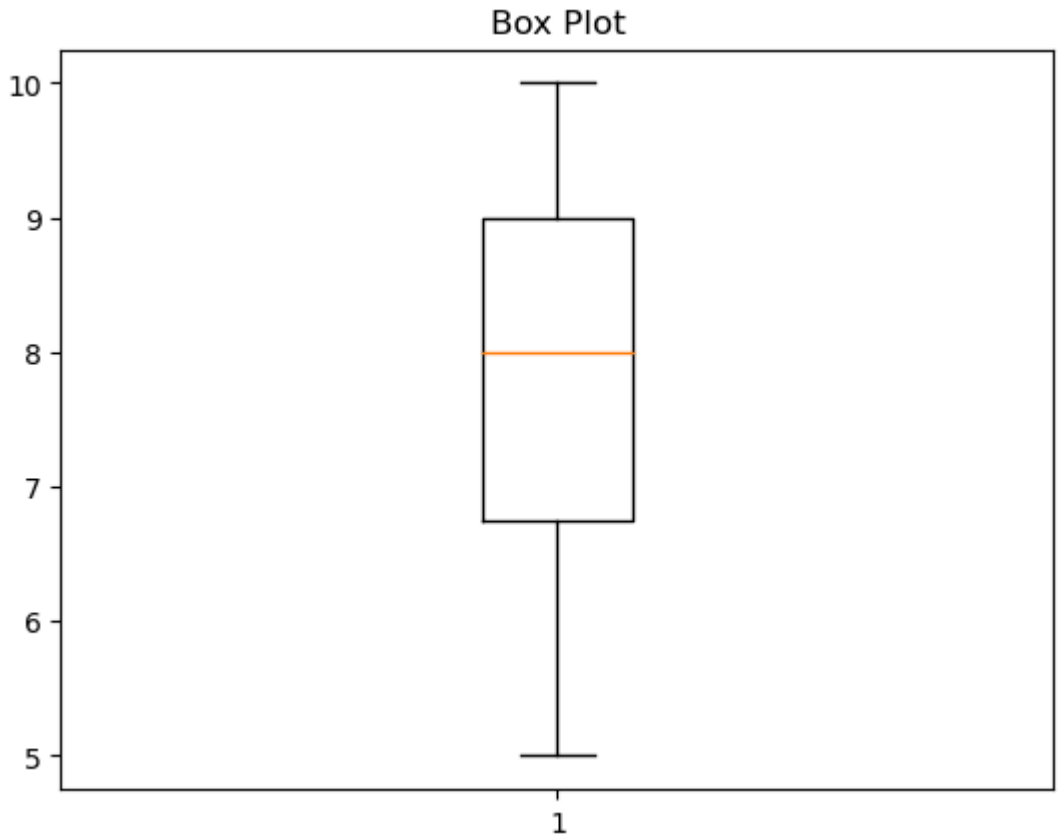
f. Box Plot

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.

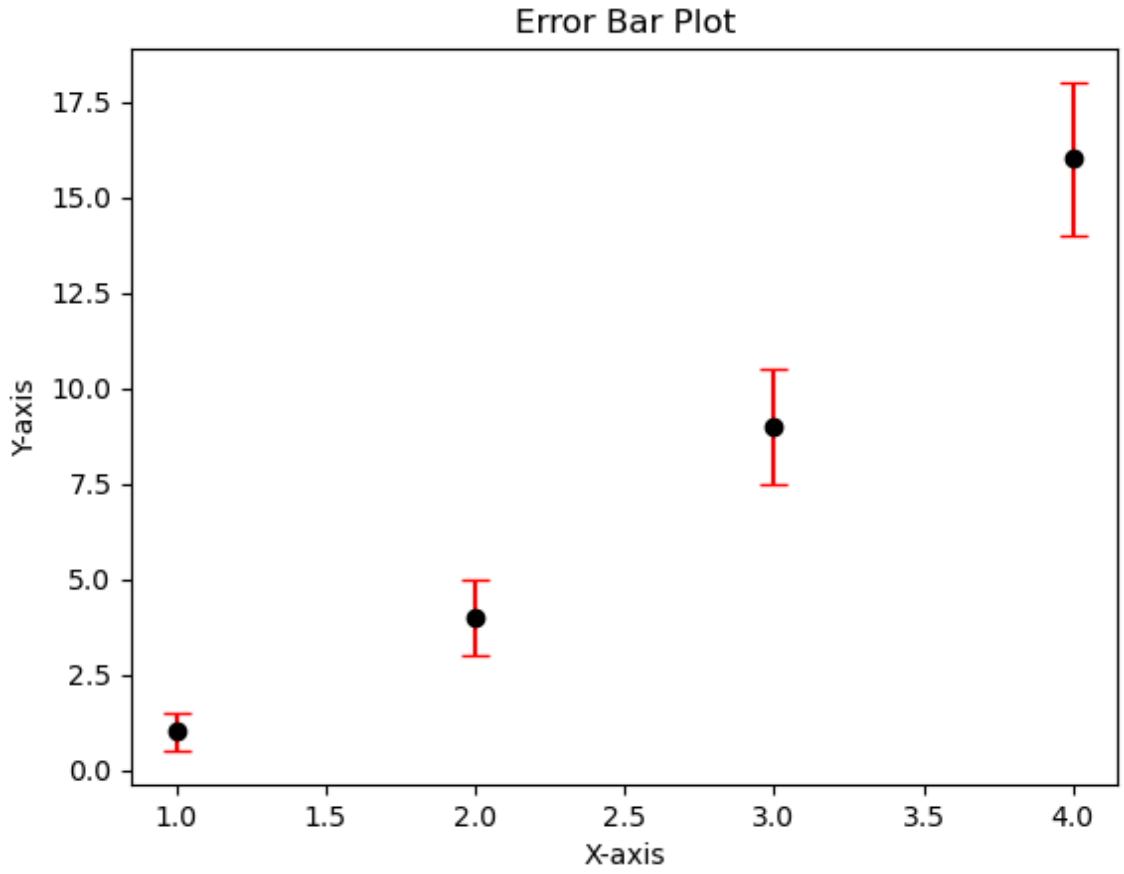
In [10]:
```
data = [7, 8, 9, 5, 6, 9, 8, 9, 10, 6, 7, 8]

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.

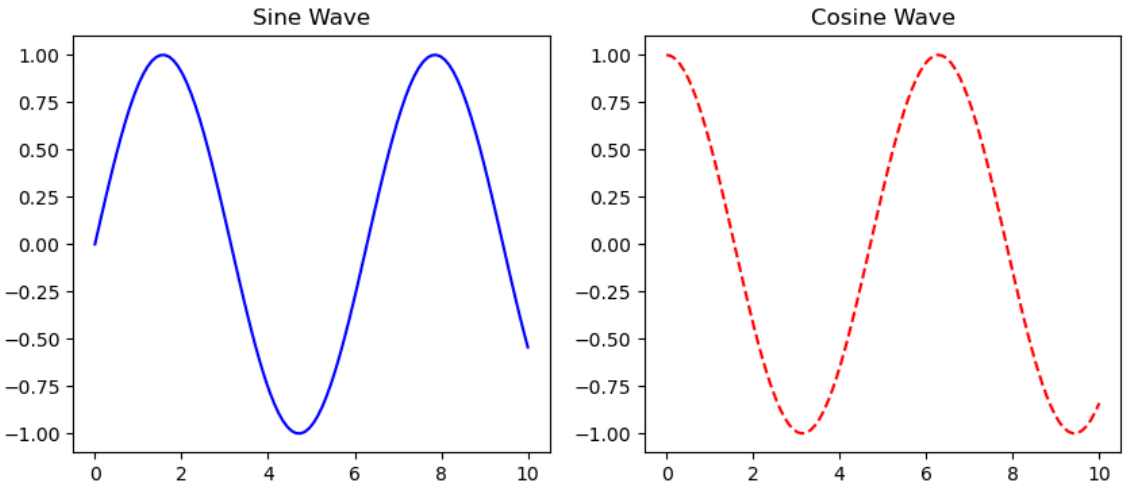
In [11]:
```
x = np.arange(1, 5)
y = x ** 2
errors = [0.5, 1, 1.5, 2]

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()
```



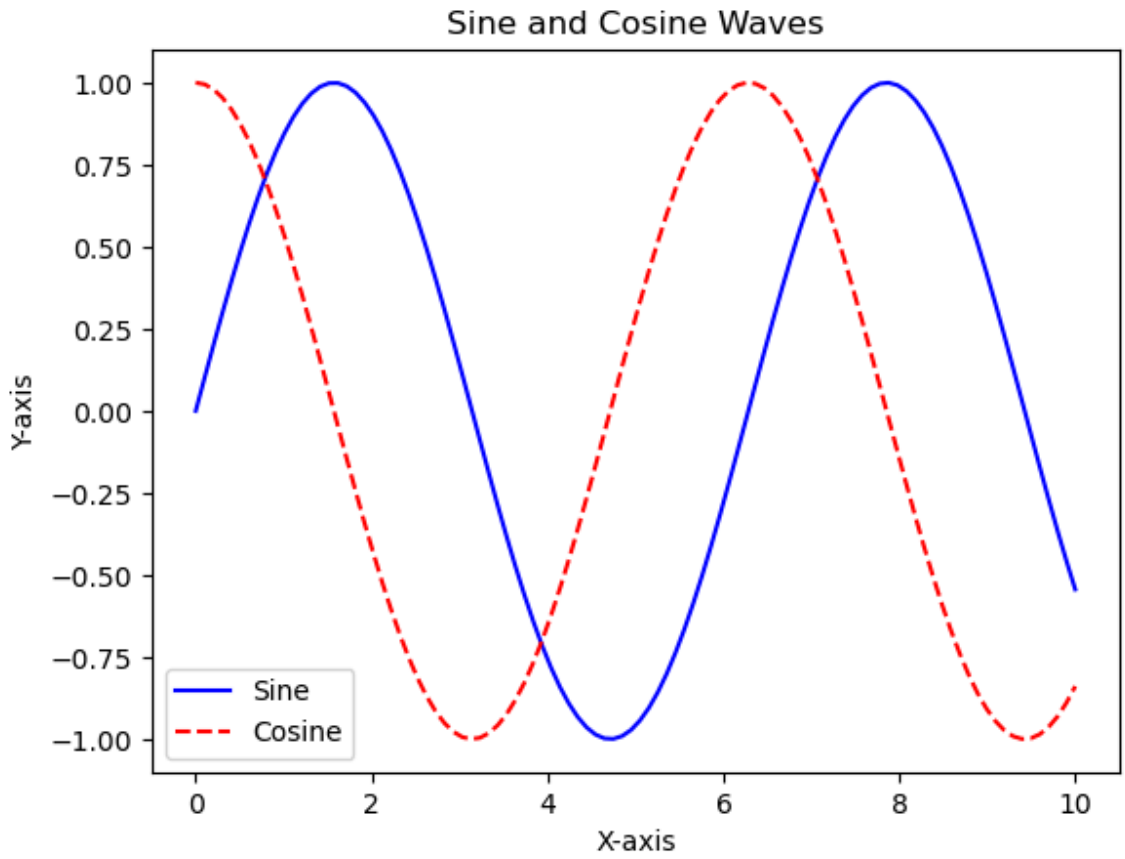
h. Subplots (Multiple Plots in One Figure)

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()
```



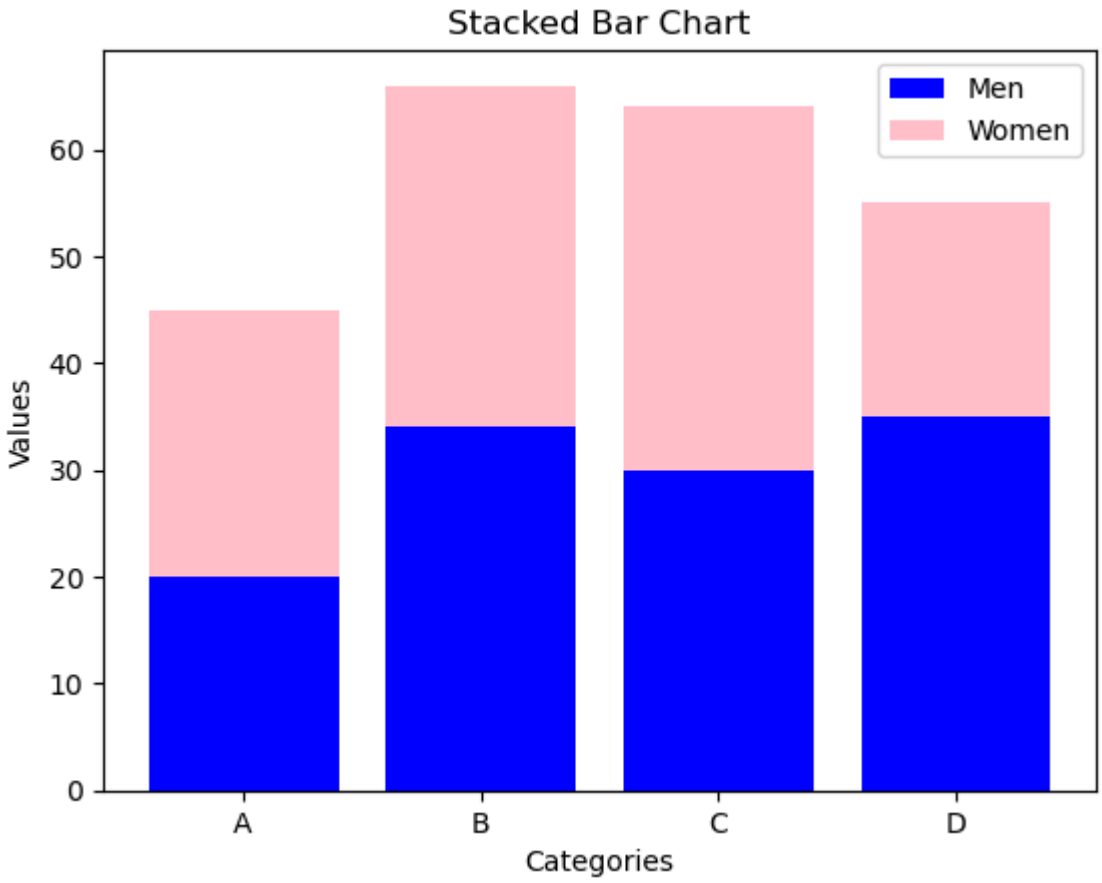
In [7]:
```
# Create the plot
plt.plot(x, y1, color='b', label='Sine')
plt.plot(x, y2, color='r', linestyle='--', label='Cosine')

# 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()
```



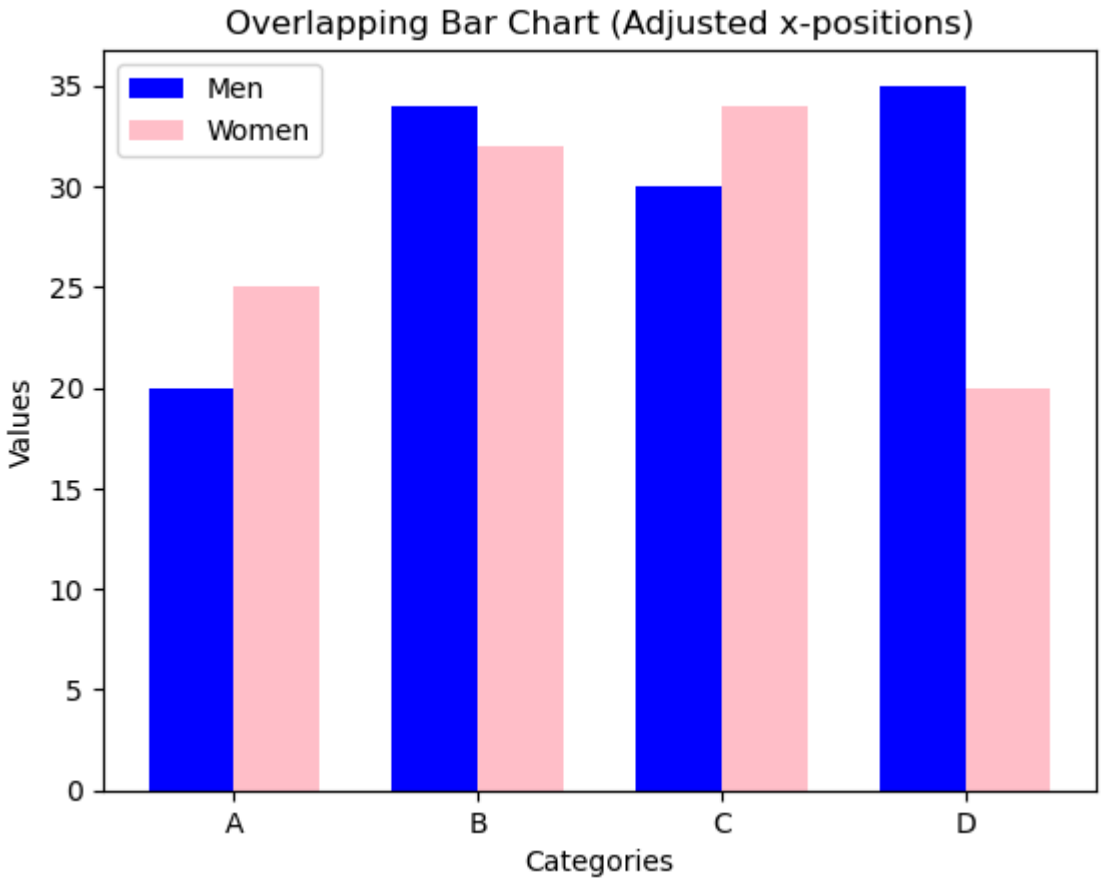
i. Stacked Bar Chart

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]:
```
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()
```



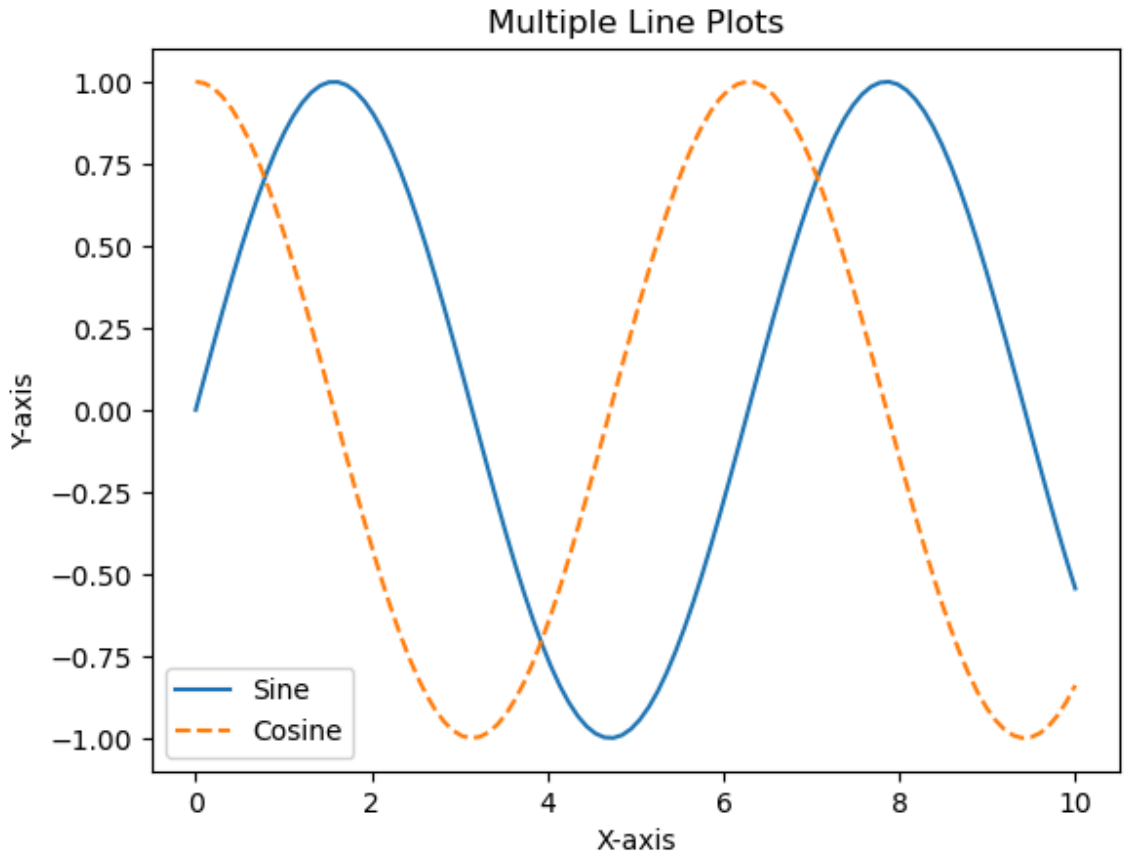
In [15]:
```
x = np.arange(len(categories))  # the label locations
width = 0.35  # the width of the bars

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.

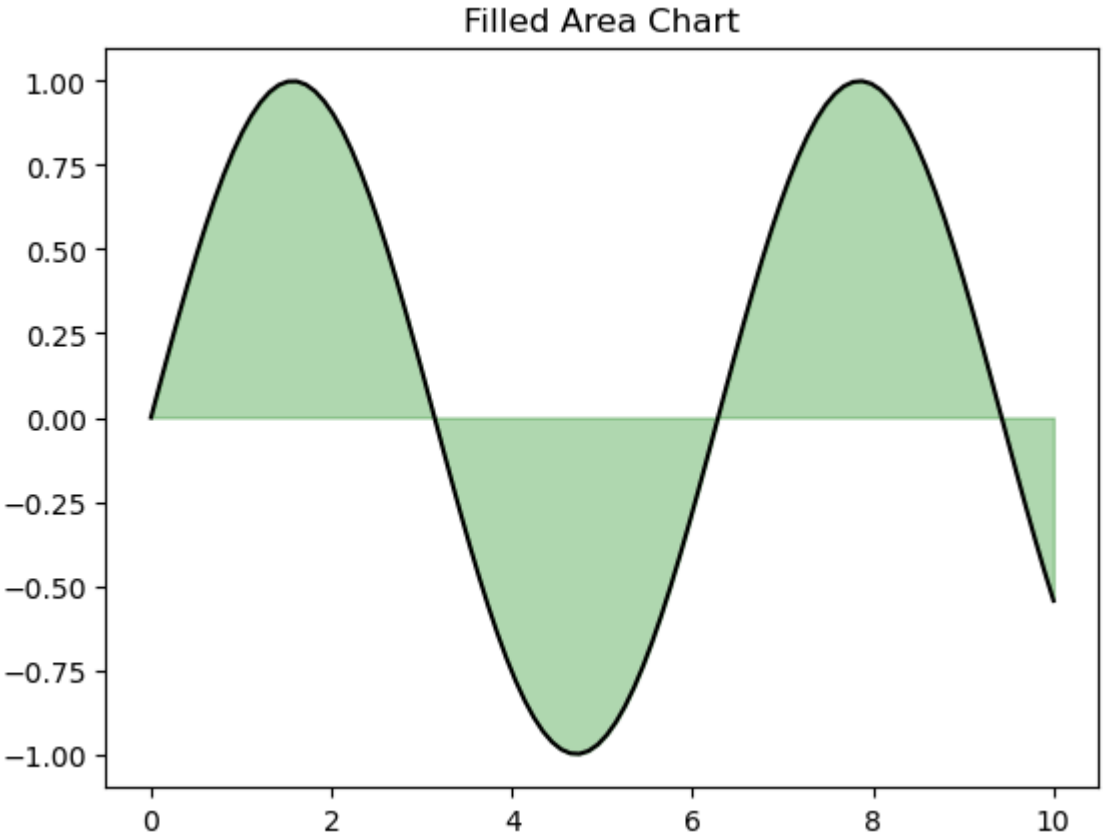
In [14]:
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

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.

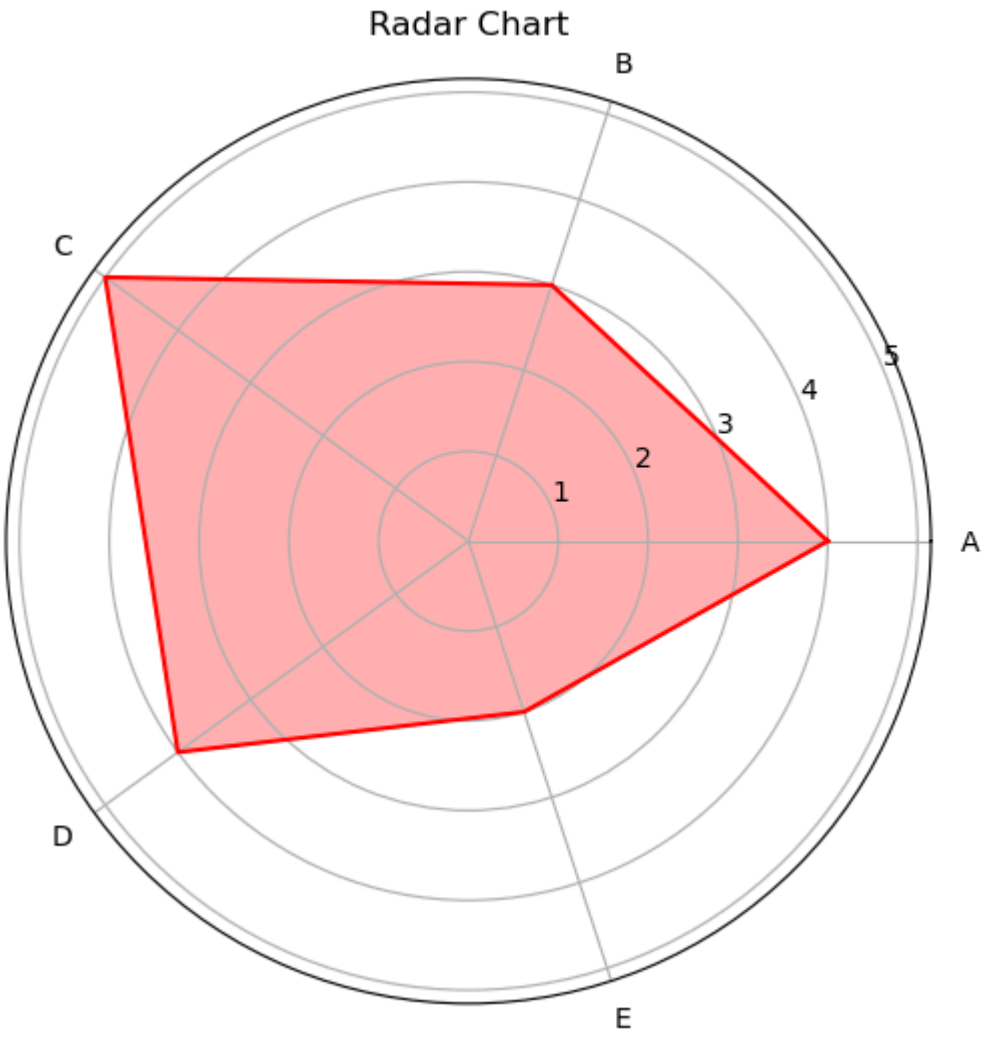
- plt.subplots(figsize=(6,6), subplot_kw=dict(polar=True)): Creates a subplot with a polar projection (polar=True). figsize sets the size of the figure.
- fig: The figure object.
- ax: The axes object, which we'll use to draw on.

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()
```

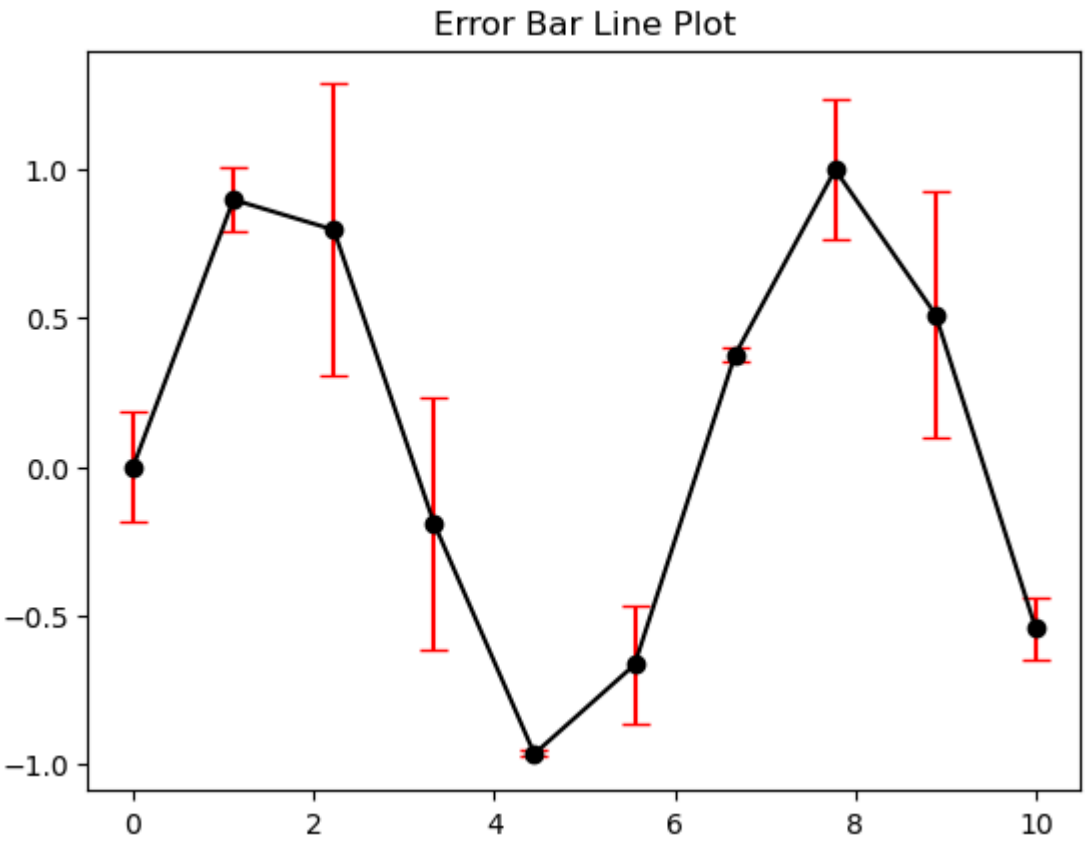


m. Error Bars in Line Plot

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.

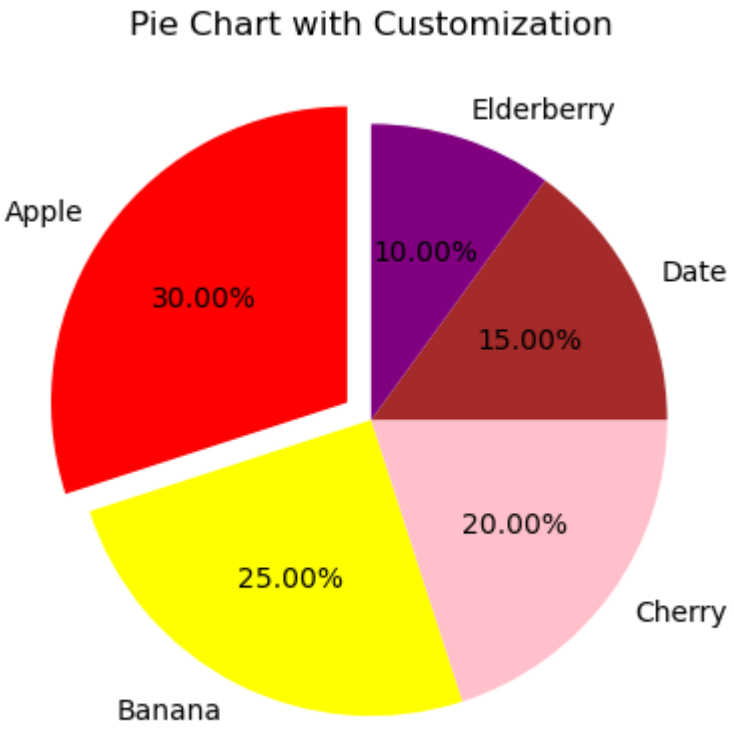
In [16]:
```
errors = np.random.rand(10) * 0.5
```

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

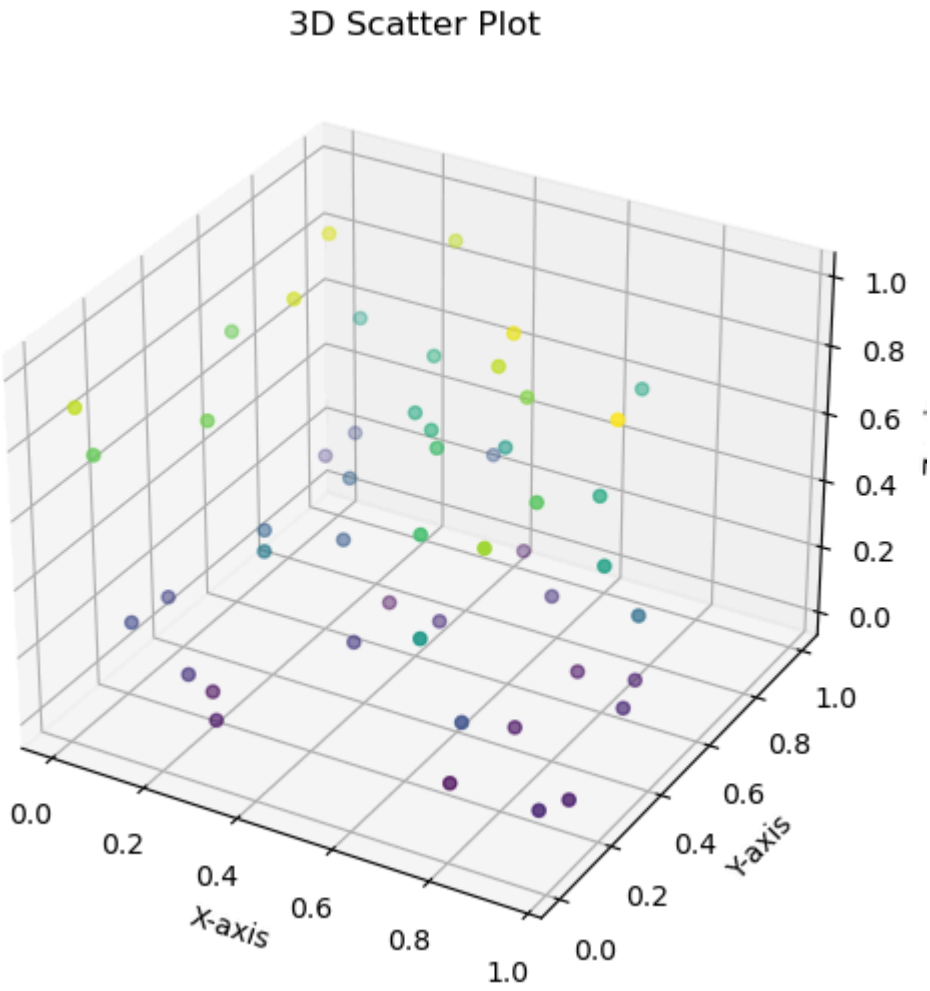
1. fig = plt.figure(figsize=(8, 6)): This line creates the figure object, which is the overall canvas for the plot. figsize=(8, 6) sets the size of the figure to 8 inches wide and 6 inches tall.
2. ax = fig.add_subplot(111, projection='3d'): This adds a subplot to the figure. 111 is a compact way of 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()
```

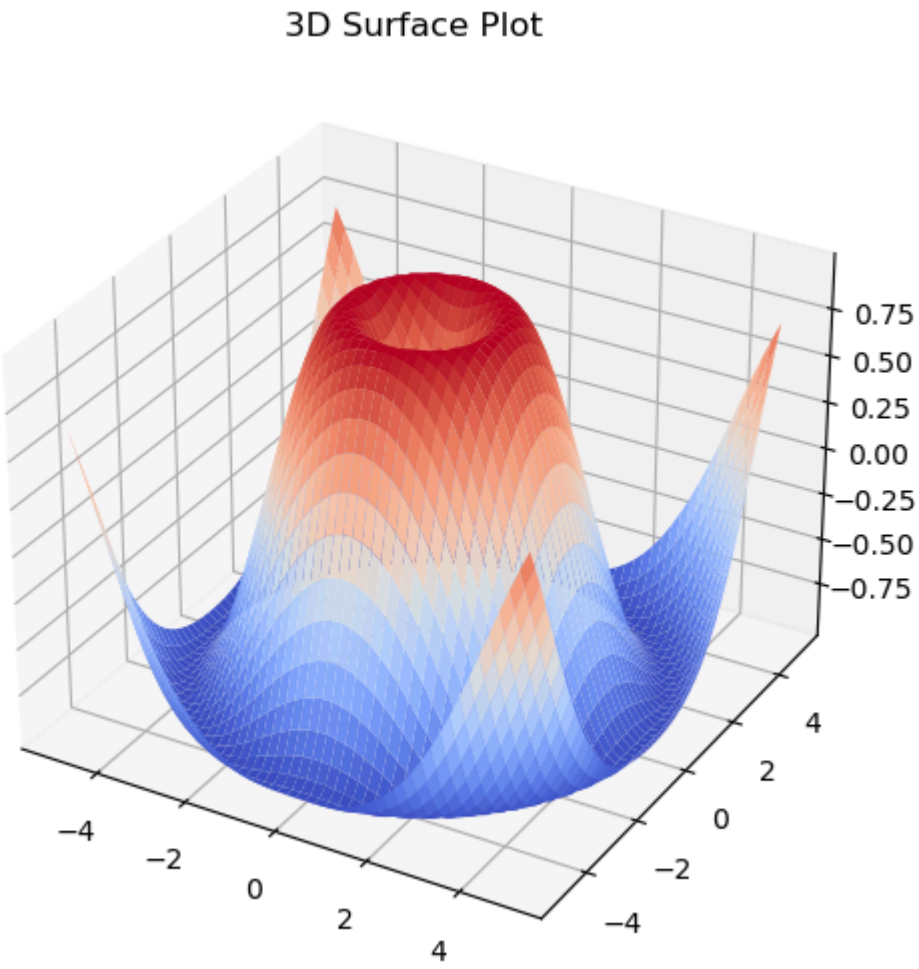


p. 3D Surface Plot

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

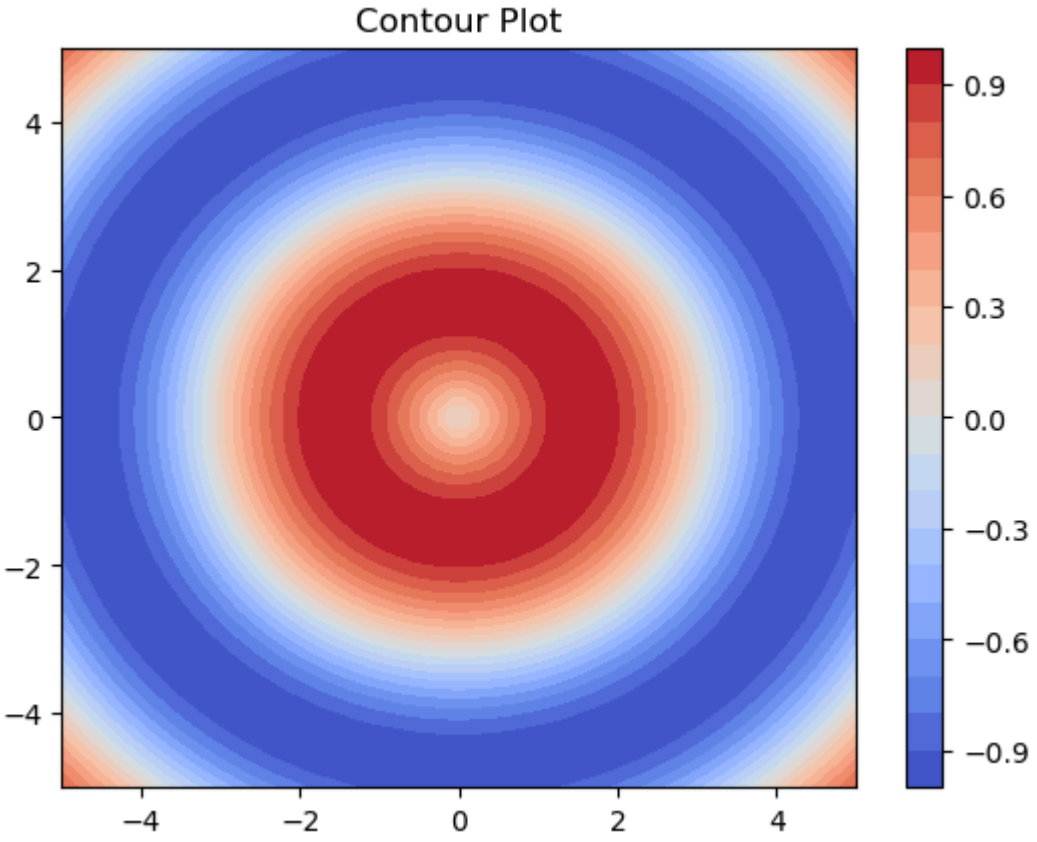
A contour plot is a graphical representation of 3D data in two dimensions using contour lines. Each contour line represents a constant value (similar to elevation maps), allowing visualization of gradients and patterns in data.

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]:
```
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()
```



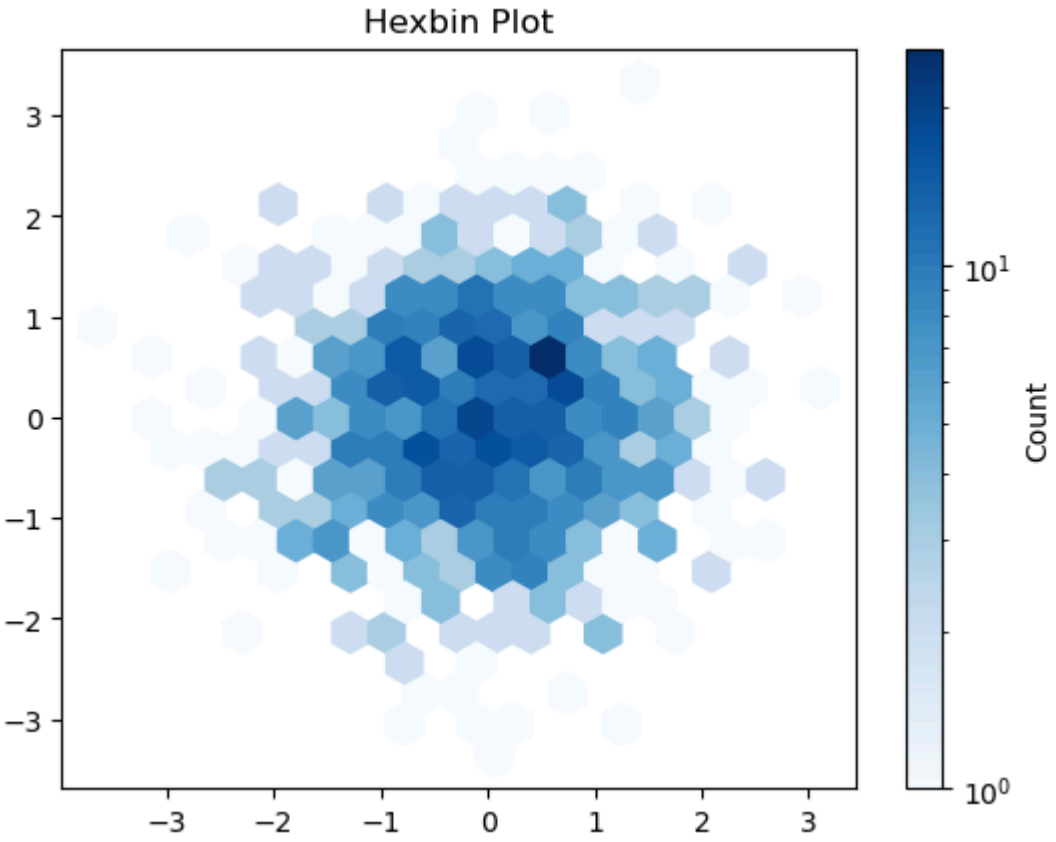
r. Hexbin Plot (Density Visualization)

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.

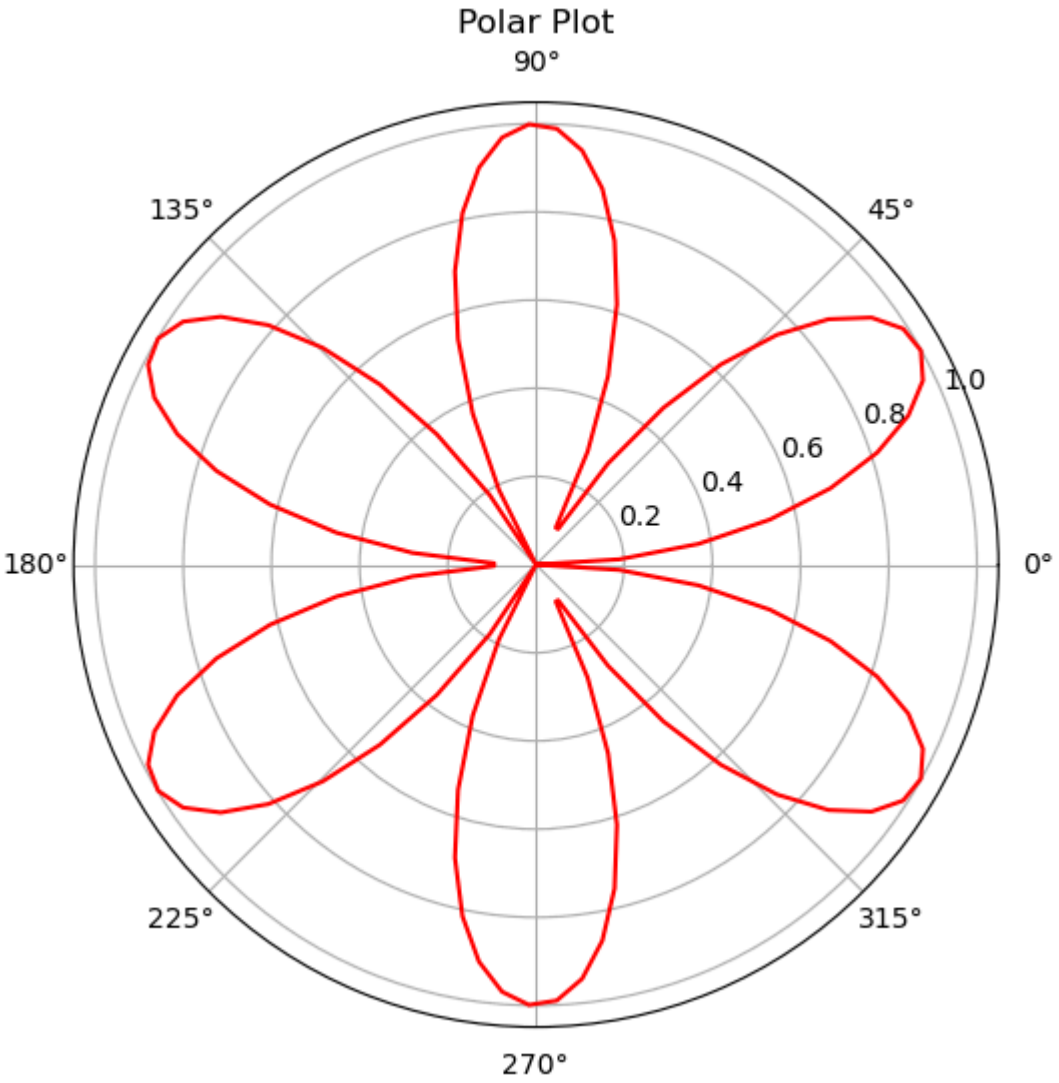
    - plt.hexbin(x, y): This function creates the hexbin plot using the x and y coordinates.
    - gridsize=30: This parameter controls the resolution of the hexagonal grid. A smaller gridsize means finer hexagons (more detail), and a larger gridsize means coarser hexagons (less detail). Think of it as dividing 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()
```



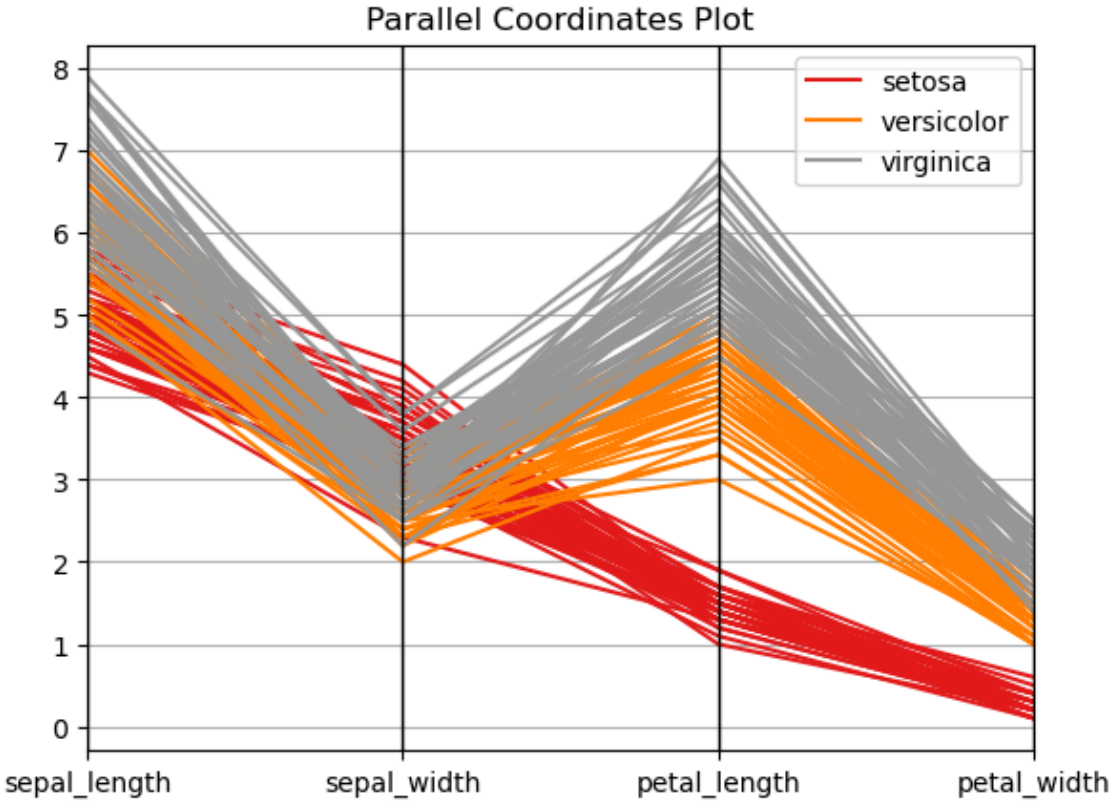
s. Polar Plot

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]:
```
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)

Import Library

In [1]:
```
import numpy as np
```

Creating Arrays

In [2]:
```
# 1D Array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# 2D Array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

```
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
```

Using Built-in Functions

In [3]:
```
# Array of zeros
zeros = np.zeros((2, 3))
print(zeros)

# Array of ones
ones = np.ones((3, 3))
print(ones)

# Array with a range of numbers
range_arr = np.arange(0, 10, 2)
print(range_arr)

# Linearly spaced array
lin_space = np.linspace(0, 1, 5)
print(lin_space)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

Basic Operations

In [4]:
```
arr = np.array([1, 2, 3, 4, 5])

# Addition
print(arr + 5)

# Multiplication
print(arr * 2)

# Exponentiation
print(arr ** 2)
```

```
[ 6  7  8  9 10]
[ 2  4  6  8 10]
[ 1  4  9 16 25]
```

Mathematical Functions

In [5]:
```
# Sine and Cosine

angles = np.array([0, np.pi/2, np.pi])
print(np.sin(angles))
print(np.cos(angles))

#  sin(π)≈0, but due to floating-point precision, it shows a very small value close to zero (1.224647e-16)
```

```
[0.0000000e+00 1.0000000e+00 1.2246468e-16]
[ 1.000000e+00  6.123234e-17 -1.000000e+00]
```

In [6]:
```
# Square root and Exponential

nums = np.array([1, 4, 9, 16])
print(np.sqrt(nums))
print(np.exp(nums))
```

```
[1. 2. 3. 4.]
[2.71828183e+00 5.45981500e+01 8.10308393e+03 8.88611052e+06]
```

Array Indexing and Slicing

In [7]:
```
#Indexing

arr = np.array([10, 20, 30, 40, 50])

# Accessing elements
print(arr[0])  # First element
print(arr[-1]) # Last element
```

```
10
50
```

In [8]:
```
# Slicing

print(arr[1:4])   # Elements from index 1 to 3
print(arr[:3])    # First three elements
print(arr[::2])   # Every second element
```

```
[20 30 40]
[10 20 30]
[10 30 50]
```

In [9]:
```
# 2D Array Indexing

arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing elements
print(arr2[1, 2])  # Row 2, Column 3

# Slicing
print(arr2[0:2, 1:3])  # Sub-matrix
```

```
6
[[2 3]
 [5 6]]
```

Broadcasting

In [10]:
```
#Broadcasting

arr = np.array([1, 2, 3])
print(arr + 5)  # Adds 5 to each element
```

```
[6 7 8]
```

Vectorized Operations

In [11]:
```
# Vectorized Operations

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise multiplication
print(a * b)
```

```
[ 4 10 18]
```

Conditional Statements

In [12]:
```
# Conditional Statements

arr = np.array([10, 20, 30, 40, 50])

# Boolean indexing
print(arr[arr > 30])  # Elements greater than 30
```

```
[40 50]
```

Array Manipulation

In [13]:
```
# Reshape

arr = np.arange(1, 7)
reshaped_arr = arr.reshape((2, 3))
print(arr)
print(reshaped_arr)
```

```
[1 2 3 4 5 6]
[[1 2 3]
 [4 5 6]]
```

In [14]:
```
# Flattening

arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
flattened = arr2.flatten()
print(flattened)
```

```
[[1 2 3]
 [4 5 6]]
[1 2 3 4 5 6]
```

In [15]:
```
# Transpose

arr3 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr3)
transposed = arr3.T
print(transposed)
```

```
[[1 2 3]
 [4 5 6]]
[[1 4]
 [2 5]
 [3 6]]
```

Aggregations and Statistics

In [16]:
```
# Basic Statistical Functions

arr = np.array([1, 2, 3, 4, 5])

# Sum and Mean
print(np.sum(arr))
print(np.mean(arr))

# Standard Deviation and Variance
print(np.std(arr))
print(np.var(arr))
```

```
15
3.0
1.4142135623730951
2.0
```

In [17]:
```
# Min and Max

print(np.min(arr))
print(np.max(arr))
```

```
1
5
```

In [18]:
```
# Axis-wise Operations

arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# Sum across columns
print(np.sum(arr2, axis=0))

# Sum across rows
print(np.sum(arr2, axis=1))
```

```
[5 7 9]
[ 6 15]
```

Random Numbers and Distributions

In [19]:
```
# Random integers
rand_ints = np.random.randint(0, 10, size=(2, 3))
print(rand_ints)

# Random floats between 0 and 1
rand_floats = np.random.rand(2, 3)
print(rand_floats)

# Normal Distribution
norm_dist = np.random.randn(2, 3)
print(norm_dist)
```

```
[[2 4 3]
 [0 3 8]]
[[0.98279547 0.25050344 0.2776913 ]
 [0.61946616 0.88325081 0.10157034]]
[[-0.28791975 -0.66302971 -0.62931675]
 [-1.78201822  1.083215   -0.37892021]]
```

Linear Algebra with NumPy

In [20]:
```
from numpy import linalg

# Matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication
print(np.dot(A, B))

# Determinant
print(linalg.det(A))

# Inverse
print(linalg.inv(A))

# Eigenvalues and Eigenvectors
eigenvals, eigenvecs = linalg.eig(A)
print("Eigenvalues:", eigenvals)
print("Eigenvectors:", eigenvecs)
```

```
[[19 22]
 [43 50]]
-2.0000000000000004
[[-2.   1. ]
 [ 1.5 -0.5]]
Eigenvalues: [-0.37228132  5.37228132]
Eigenvectors: [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

In [ ]:

Import Library

In [1]:
```
import pandas as pd
```

Import Data

In [2]:
```
cars = pd.read_csv('cars_data.csv')
```

Inital Insights

In [3]:
```
cars.shape #no of rows and columns
```

Out[3]:
```
(428, 15)
```

In [4]:
```
rows = cars.shape[0]
columns = cars.shape[1]
```

In [5]:
```
print("Rows : {}\nColumn : {}".format(rows,columns))
```

```
Rows : 428
Column : 15
```

In [6]:
```
cars.head()
```

Out[6]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsepower
0	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6.0	265.0
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4.0	200.0
2	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4.0	200.0
3	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6.0	270.0
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6.0	225.0

In [7]:
```
cars.tail()
```

Out[7]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horse
423	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5.0	
424	Volvo	C70 HPT convertible 2dr	Sedan	Europe	Front	$42,565	$40,083	2.3	5.0	
425	Volvo	S80 T6 4dr	Sedan	Europe	Front	$45,210	$42,573	2.9	6.0	
426	Volvo	V40	Wagon	Europe	Front	$26,135	$24,641	1.9	4.0	
427	Volvo	XC70	Wagon	Europe	All	$35,145	$33,112	2.5	5.0	

In [8]:
```
cars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 428 entries, 0 to 427
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Make         428 non-null    object 
 1   Model        428 non-null    object 
 2   Type         428 non-null    object 
 3   Origin       428 non-null    object 
 4   DriveTrain   428 non-null    object 
 5   MSRP         428 non-null    object 
 6   Invoice      428 non-null    object 
 7   EngineSize   428 non-null    float64
 8   Cylinders    426 non-null    float64
 9   Horsepower   426 non-null    float64
 10  MPG_City     428 non-null    int64  
 11  MPG_Highway  428 non-null    int64  
 12  Weight       428 non-null    int64  
 13  Wheelbase    428 non-null    int64  
 14  Length       428 non-null    int64  
dtypes: float64(3), int64(5), object(7)
memory usage: 50.3+ KB
```

In [9]:
```
cars.columns #name of columns
```

Out[9]:
```
Index(['Make', 'Model', 'Type', 'Origin', 'DriveTrain', 'MSRP', 'Invoice',
       'EngineSize', 'Cylinders', 'Horsepower', 'MPG_City', 'MPG_Highway',
       'Weight', 'Wheelbase', 'Length'],
      dtype='object')
```

In [10]:
```
cars.dtypes #data types
```

Out[10]:
```
Make            object
Model           object
Type            object
Origin          object
DriveTrain      object
MSRP            object
Invoice         object
EngineSize     float64
Cylinders      float64
Horsepower     float64
MPG_City         int64
MPG_Highway      int64
Weight           int64
Wheelbase        int64
Length           int64
dtype: object
```

In [11]:
```
cars.nunique() #no of unique values
```

Out[11]:
```
Make            38
Model          425
Type             6
Origin           3
DriveTrain       3
MSRP           410
Invoice        425
EngineSize      43
Cylinders        7
Horsepower     110
MPG_City        28
MPG_Highway     33
Weight         348
Wheelbase       40
Length          67
dtype: int64
```

In [12]:
```
cars.describe()
```

Out[12]:

	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	Weight	Wheelbase
count	428.000000	426.000000	426.000000	428.000000	428.000000	428.000000	428.000000
mean	3.196729	5.807512	215.854460	20.060748	26.843458	3577.953271	108.154206
std	1.108595	1.558443	72.003218	5.238218	5.741201	758.983215	8.311813
min	1.300000	3.000000	73.000000	10.000000	12.000000	1850.000000	89.000000
25%	2.375000	4.000000	165.000000	17.000000	24.000000	3104.000000	103.000000
50%	3.000000	6.000000	210.000000	19.000000	26.000000	3474.500000	107.000000
75%	3.900000	6.000000	255.000000	21.250000	29.000000	3977.750000	112.000000
max	8.300000	12.000000	500.000000	60.000000	66.000000	7190.000000	144.000000

In [13]:
```
cars.isna().sum()
```

Out[13]:
```
Make           0
Model          0
Type           0
Origin         0
DriveTrain     0
MSRP           0
Invoice        0
EngineSize     0
Cylinders      2
Horsepower     2
MPG_City       0
MPG_Highway    0
Weight         0
Wheelbase      0
Length         0
dtype: int64
```

Selecting & Filtering Data

In [14]:
```
cars.iloc[0:3,0:5] #iloc[index_rows,index_column]
```

Out[14]:

	Make	Model	Type	Origin	DriveTrain
0	Acura	MDX	SUV	Asia	All
1	Acura	RSX Type S 2dr	Sedan	Asia	Front
2	Acura	TSX 4dr	Sedan	Asia	Front

In [15]:
```
cars.loc[0:5,'Make':'MSRP'] #loc[values_rows,name_column]
```

Out[15]:

	Make	Model	Type	Origin	DriveTrain	MSRP
0	Acura	MDX	SUV	Asia	All	$36,945
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820
2	Acura	TSX 4dr	Sedan	Asia	Front	$26,990
3	Acura	TL 4dr	Sedan	Asia	Front	$33,195
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755
5	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100

In [16]:
```
cars[cars['MPG_City'] > 30] # Filter rows based on conditions
```

Out[16]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders
149	Honda	Civic Hybrid 4dr manual (gas/electric)	Hybrid	Asia	Front	$20,140	$18,451	1.4	4.0
150	Honda	Insight 2dr (gas/electric)	Hybrid	Asia	Front	$19,110	$17,911	2.0	3.0
154	Honda	Civic DX 2dr	Sedan	Asia	Front	$13,270	$12,175	1.7	4.0
155	Honda	Civic HX 2dr	Sedan	Asia	Front	$14,170	$12,996	1.7	4.0
156	Honda	Civic LX 4dr	Sedan	Asia	Front	$15,850	$14,531	1.7	4.0
159	Honda	Civic EX 4dr	Sedan	Asia	Front	$17,750	$16,265	1.7	4.0
352	Scion	xA 4dr hatch	Sedan	Asia	Front	$12,965	$12,340	1.5	4.0
353	Scion	xB	Wagon	Asia	Front	$14,165	$13,480	1.5	4.0
373	Toyota	Prius 4dr (gas/electric)	Hybrid	Asia	Front	$20,510	$18,926	1.5	4.0
379	Toyota	Corolla CE 4dr	Sedan	Asia	Front	$14,085	$13,065	1.8	4.0
380	Toyota	Corolla S 4dr	Sedan	Asia	Front	$15,030	$13,650	1.8	4.0
381	Toyota	Corolla LE 4dr	Sedan	Asia	Front	$15,295	$13,889	1.8	4.0
382	Toyota	Echo 2dr manual	Sedan	Asia	Front	$10,760	$10,144	1.5	4.0
383	Toyota	Echo 2dr auto	Sedan	Asia	Front	$11,560	$10,896	1.5	4.0
384	Toyota	Echo 4dr	Sedan	Asia	Front	$11,290	$10,642	1.5	4.0
404	Volkswagen	Jetta GLS TDI 4dr	Sedan	Europe	Front	$21,055	$19,638	1.9	4.0

In [17]:
```
cars[(cars['MPG_City'] > 30) & (cars['MPG_City'] < 32)]
```

Out[17]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsepow
353	Scion	xB	Wagon	Asia	Front	$14,165	$13,480	1.5	4.0	108

In [18]:
```
cars[cars['Type'] == 'Sedan'] # Filter rows with string match
```

Out[18]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Hors
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4.0	
2	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4.0	
3	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6.0	
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6.0	
5	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41,100	3.5	6.0	
...	...	...	...	...	...	...	...	...	...	
421	Volvo	S80 2.9 4dr	Sedan	Europe	Front	$37,730	$35,542	2.9	6.0	
422	Volvo	S80 2.5T 4dr	Sedan	Europe	All	$37,885	$35,688	2.5	5.0	
423	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5.0	
424	Volvo	C70 HPT convertible 2dr	Sedan	Europe	Front	$42,565	$40,083	2.3	5.0	
425	Volvo	S80 T6 4dr	Sedan	Europe	Front	$45,210	$42,573	2.9	6.0	

262 rows × 15 columns

In [19]:
```
cars[cars['Type'].str.contains('Sedan')]
```

Out[19]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Hors
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4.0	
2	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4.0	
3	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6.0	
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6.0	
5	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41,100	3.5	6.0	
...	...	...	...	...	...	...	...	...	...	
421	Volvo	S80 2.9 4dr	Sedan	Europe	Front	$37,730	$35,542	2.9	6.0	
422	Volvo	S80 2.5T 4dr	Sedan	Europe	All	$37,885	$35,688	2.5	5.0	
423	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5.0	
424	Volvo	C70 HPT convertible 2dr	Sedan	Europe	Front	$42,565	$40,083	2.3	5.0	
425	Volvo	S80 T6 4dr	Sedan	Europe	Front	$45,210	$42,573	2.9	6.0	

262 rows × 15 columns

Drop NaN values

In [20]:
```
cars = cars.dropna(subset=['Horsepower']).reset_index(drop=True)
```

In [21]:
```
cars.isna().sum()
```

Out[21]:
```
Make          0
Model         0
Type          0
Origin        0
DriveTrain    0
MSRP          0
Invoice       0
EngineSize    0
Cylinders     2
Horsepower    0
MPG_City      0
MPG_Highway   0
Weight        0
Wheelbase     0
Length        0
dtype: int64
```

In [22]:
```
cars.tail()
```

Out[22]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horse
421	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5.0	
422	Volvo	C70 HPT convertible 2dr	Sedan	Europe	Front	$42,565	$40,083	2.3	5.0	
423	Volvo	S80 T6 4dr	Sedan	Europe	Front	$45,210	$42,573	2.9	6.0	
424	Volvo	V40	Wagon	Europe	Front	$26,135	$24,641	1.9	4.0	
425	Volvo	XC70	Wagon	Europe	All	$35,145	$33,112	2.5	5.0	

Fill NaN values in 'Cylinder' Column

In [23]:
```
cars[cars.Cylinders.isna()]
```

Out[23]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsep
245	Mazda	RX-8 4dr automatic	Sports	Asia	Rear	$25,700	$23,794	1.3	NaN	
246	Mazda	RX-8 4dr manual	Sports	Asia	Rear	$27,200	$25,179	1.3	NaN	

In [24]:
```
cars[cars.Make=='Mazda'].describe()
```

Out[24]:

	EngineSize	Cylinders	Horsepower	MPG_City	MPG_Highway	Weight	Wheelbase	
count	11.000000	9.000000	11.000000	11.000000	11.000000	11.000000	11.000000	11
mean	2.190909	4.444444	169.727273	21.454545	27.272727	2980.909091	105.090909	177
std	0.770006	0.881917	34.931622	3.587858	4.244783	435.576504	10.290331	13
min	1.300000	4.000000	130.000000	15.000000	19.000000	2387.000000	89.000000	156
25%	1.800000	4.000000	142.500000	18.000000	25.000000	2729.000000	103.500000	173
50%	2.000000	4.000000	160.000000	23.000000	28.000000	3029.000000	105.000000	178
75%	2.300000	4.000000	198.500000	24.000000	30.000000	3072.000000	109.000000	187
max	4.000000	6.000000	238.000000	26.000000	34.000000	3812.000000	126.000000	203

In [25]:
```
cars[cars.Make=='Mazda'].Cylinders.unique()
```

Out[25]:
```
array([ 4.,  6., nan])
```

In [26]:
```
cars.loc[cars['Make'] == 'Mazda', ['Cylinders']]
```

Out[26]:

	Cylinders
238	4.0
239	4.0
240	4.0
241	4.0
242	6.0
243	4.0
244	4.0
245	NaN
246	NaN
247	4.0
248	6.0

In [27]:
```
cars.loc[cars['Make'] == 'Mazda', ['Cylinders']] = cars.loc[cars['Make'] == 'Mazda'
```

In [28]:
```
cars[cars.Make=='Mazda'].isna().sum()
```

Out[28]:
```
Make          0
Model         0
Type          0
Origin        0
DriveTrain    0
MSRP          0
Invoice       0
EngineSize    0
Cylinders     0
Horsepower    0
MPG_City      0
MPG_Highway   0
Weight        0
Wheelbase     0
Length        0
dtype: int64
```

In [29]:
```
cars[cars.Make=='Mazda'].Cylinders.unique()
```

Out[29]:
```
array([4., 6.])
```

Raplace Values

In [30]:
```
cars['MSRP'] = cars['MSRP'].str.replace('$','')  # Remove $ and , in MSRP columns
cars['MSRP'] = cars['MSRP'].str.replace(',','')
```

In [31]:
```
cars['Invoice'] = cars['Invoice'].str.replace('$','')  # Remove $ and , in Invoice
cars['Invoice'] = cars['Invoice'].str.replace(',','')
```

In [32]:
```
cars.head()
```

Out[32]:

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Horsepower	M
0	Acura	MDX	SUV	Asia	All	36945	33337	3.5	6.0	265.0	
1	Acura	RSX Type S 2dr	Sedan	Asia	Front	23820	21761	2.0	4.0	200.0	
2	Acura	TSX 4dr	Sedan	Asia	Front	26990	24647	2.4	4.0	200.0	
3	Acura	TL 4dr	Sedan	Asia	Front	33195	30299	3.2	6.0	270.0	
4	Acura	3.5 RL 4dr	Sedan	Asia	Front	43755	39014	3.5	6.0	225.0	

Change Data Type

In [33]:
```
cars['MSRP'] = cars['MSRP'].astype('int') # Change data type into integer
cars['Invoice'] = cars['Invoice'].astype('int')
```

In [34]:
```
cars.dtypes
```

Out[34]:
```
Make            object
Model           object
Type            object
Origin          object
DriveTrain      object
MSRP             int32
Invoice          int32
EngineSize     float64
Cylinders      float64
Horsepower     float64
MPG_City         int64
MPG_Highway      int64
Weight           int64
Wheelbase        int64
Length           int64
dtype: object
```

Modifying Data

In [35]:
```
cars.rename(columns={'EngineSize': 'Engine_Size'}, inplace=True) # Rename columns
```

In [36]:
```
cars.columns
```

Out[36]:
```
Index(['Make', 'Model', 'Type', 'Origin', 'DriveTrain', 'MSRP', 'Invoice',
       'Engine_Size', 'Cylinders', 'Horsepower', 'MPG_City', 'MPG_Highway',
       'Weight', 'Wheelbase', 'Length'],
      dtype='object')
```

In [37]:
```
cars.drop(columns=['Model'], inplace=True) # Remove columns
```

In [38]:
```
cars.columns
```

Out[38]:
```
Index(['Make', 'Type', 'Origin', 'DriveTrain', 'MSRP', 'Invoice',
       'Engine_Size', 'Cylinders', 'Horsepower', 'MPG_City', 'MPG_Highway',
       'Weight', 'Wheelbase', 'Length'],
      dtype='object')
```

In [39]:
```
cars.sort_values(by='MSRP', ascending=True) # Sort DataFrame
```

Out[39]:

	Make	Type	Origin	DriveTrain	MSRP	Invoice	Engine_Size	Cylinders	Horsepower
204	Kia	Sedan	Asia	Front	10280	9875	1.6	4.0	104.0
166	Hyundai	Sedan	Asia	Front	10539	10107	1.6	4.0	103.0
380	Toyota	Sedan	Asia	Front	10760	10144	1.5	4.0	108.0
343	Saturn	Sedan	USA	Front	10995	10319	2.2	4.0	140.0
205	Kia	Sedan	Asia	Front	11155	10705	1.6	4.0	104.0
...	...	...	...	...	...	...	...	...	...
259	Mercedes-Benz	Sedan	Europe	Rear	94820	88324	5.0	8.0	302.0
268	Mercedes-Benz	Sports	Europe	Rear	121770	113388	5.5	8.0	493.0
269	Mercedes-Benz	Sports	Europe	Rear	126670	117854	5.5	12.0	493.0
260	Mercedes-Benz	Sedan	Europe	Rear	128420	119600	5.5	12.0	493.0
332	Porsche	Sports	Europe	Rear	192465	173560	3.6	6.0	477.0

426 rows × 14 columns

Aggregation & Grouping

In [40]:
```
cars.groupby(['Origin','Type'])['MSRP'].sum()
```

Out[40]:
```
Origin  Type
Asia    Hybrid      59760
        SUV        739225
        Sedan     2139813
        Sports     552681
        Truck      163069
        Wagon      254581
Europe  SUV        483460
        Sedan     3271745
        Sports    1655970
        Wagon      454215
USA     SUV        864730
        Sedan     2307495
        Sports     407315
        Truck      435524
        Wagon      156420
Name: MSRP, dtype: int32
```

In [41]:
```
cars.groupby(['Type']).agg({'MSRP': 'sum', 'MPG_City': 'mean'}) # Multiple aggregat
```

Out[41]:

Type	MSRP	MPG_City
Hybrid	59760	55.000000
SUV	2087415	16.100000
Sedan	7719053	21.092308
Sports	2615966	18.408163
Truck	598593	16.500000
Wagon	865216	21.100000

Applying Functions

In [42]:
```
cars['MSRP']
```

Out[42]:
```
0      36945
1      23820
2      26990
3      33195
4      43755
       ...
421    40565
422    42565
423    45210
424    26135
425    35145
Name: MSRP, Length: 426, dtype: int32
```

In [43]:
```
cars['MSRP'].apply(lambda x: x*2) # Apply function to a column
```

Out[43]:
```
0      73890
1      47640
2      53980
3      66390
4      87510
       ...
421    81130
422    85130
423    90420
424    52270
425    70290
Name: MSRP, Length: 426, dtype: int64
```

In [44]:
```
cars['Type'].map({'Sedan': 1}) # Map specific values in a column
```

Out[44]:
```
0      NaN
1      1.0
2      1.0
3      1.0
4      1.0
      ...
421    1.0
422    1.0
423    1.0
424    NaN
425    NaN
Name: Type, Length: 426, dtype: float64
```

In [45]:
```
cars.apply(lambda row: row['MSRP'] - row['Invoice'], axis=1)
```

Out[45]:
```
0      3608
1      2059
2      2343
3      2896
4      4741
       ...
421    2362
422    2482
423    2637
424    1494
425    2033
Length: 426, dtype: int64
```

Merging & Joining

In [46]:
```
cars_row1 = pd.read_csv('cars_data_row1.csv')
```

In [47]:
```
cars_row1.shape
```

Out[47]:
```
(199, 15)
```

In [48]:
```
cars_row2 = pd.read_csv('cars_data_row2.csv')
```

In [49]:
```
cars_row2.shape
```

Out[49]:
```
(229, 15)
```

In [50]:
```
cars_row = pd.concat([cars_row1, cars_row2], axis=0) # Concatenate along rows (stac
```

In [51]:
```
cars_row.shape
```

Out[51]:
```
(428, 15)
```

In [52]:
```
cars_col1 = pd.read_csv('cars_data_col1.csv')
```

In [53]:
```
cars_col1.shape
```

Out[53]:
```
(428, 7)
```

In [54]:
```
cars_col2 = pd.read_csv('cars_data_col2.csv')
```

In [55]:
```
cars_col2.shape
```

Out[55]:
```
(428, 8)
```

In [56]:
```
cars_col = pd.concat([cars_col1, cars_col2], axis=1) # Concatenate along columns
```

In [57]:
```
cars_col.shape
```

Out[57]:
```
(428, 15)
```

In [58]:
```
cars_data_merge1 = pd.read_csv('cars_data_merge1.csv')
```

In [59]:
```
cars_data_merge2 = pd.read_csv('cars_data_merge2.csv')
```

In [60]:
```
cars_data_merge = cars_data_merge1.merge(cars_data_merge2, on='id', how='inner') #
```

In [61]:
```
cars_data_merge
```

Out[61]:

	id	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders
0	1	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6.0
1	2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4.0
2	3	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4.0
3	4	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6.0
4	5	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6.0
...	...	...	...	...	...	...	...	...	...	...
423	424	Volvo	C70 LPT convertible 2dr	Sedan	Europe	Front	$40,565	$38,203	2.4	5.0
424	425	Volvo	C70 HPT convertible 2dr	Sedan	Europe	Front	$42,565	$40,083	2.3	5.0
425	426	Volvo	S80 T6 4dr	Sedan	Europe	Front	$45,210	$42,573	2.9	6.0
426	427	Volvo	V40	Wagon	Europe	Front	$26,135	$24,641	1.9	4.0
427	428	Volvo	XC70	Wagon	Europe	All	$35,145	$33,112	2.5	5.0

428 rows × 16 columns

Pivoting & Reshaping Data

In [62]:
```
cars.pivot(columns='Origin', values='MSRP') #Pivot table
```

Out[62]:

Origin	Asia	Europe	USA
0	36945.0	NaN	NaN
1	23820.0	NaN	NaN
2	26990.0	NaN	NaN
3	33195.0	NaN	NaN
4	43755.0	NaN	NaN
...	...	...	...
421	NaN	40565.0	NaN
422	NaN	42565.0	NaN
423	NaN	45210.0	NaN
424	NaN	26135.0	NaN
425	NaN	35145.0	NaN

426 rows × 3 columns

In [63]:
```
cars.T # Transpose
```

Out[63]:

	0	1	2	3	4	5	6	7	8	9	...	
Make	Acura	Acura	Acura	Acura	Acura	Acura	Acura	Audi	Audi	Audi	...	V
Type	SUV	Sedan	Sedan	Sedan	Sedan	Sedan	Sports	Sedan	Sedan	Sedan	...	Se
Origin	Asia	Asia	Asia	Asia	Asia	Asia	Asia	Europe	Europe	Europe	...	Eu
DriveTrain	All	Front	Front	Front	Front	Front	Rear	Front	Front	Front	...	
MSRP	36945	23820	26990	33195	43755	46100	89765	25940	35940	31840	...	3
Invoice	33337	21761	24647	30299	39014	41100	79978	23508	32506	28846	...	2
Engine_Size	3.5	2.0	2.4	3.2	3.5	3.5	3.2	1.8	1.8	3.0	...	
Cylinders	6.0	4.0	4.0	6.0	6.0	6.0	6.0	4.0	4.0	6.0	...	
Horsepower	265.0	200.0	200.0	270.0	225.0	225.0	290.0	170.0	170.0	220.0	...	2
MPG_City	17	24	22	20	18	18	17	22	23	20	...	
MPG_Highway	23	31	29	28	24	24	24	31	30	28	...	
Weight	4451	2778	3230	3575	3880	3893	3153	3252	3638	3462	...	3
Wheelbase	106	101	105	108	115	115	100	104	105	104	...	
Length	189	172	183	186	197	197	174	179	180	179	...	

14 rows × 426 columns

Creating Dataframe

In [64]:
```
cols = ['Name','Gender','DOB','Mobile_no','Email']
rows = [['Mitesh','Male','24-09-1985','9828555308','mitesh.parishkar@gmail.com']]
data = pd.DataFrame(data=rows,columns=cols)
```

In [65]:
```
data
```

Out[65]:

	Name	Gender	DOB	Mobile_no	Email
0	Mitesh	Male	24-09-1985	9828555308	mitesh.parishkar@gmail.com

In [66]:
```
data.dtypes
```

Out[66]:
```
Name         object
Gender       object
DOB          object
Mobile_no    object
Email        object
dtype: object
```

Working with Dates

In [67]:
```
import warnings
```

In [68]:
```
warnings.filterwarnings('ignore')
```

In [69]:
```
data['DOB'] = pd.to_datetime(data['DOB']) # Convert column to datetime
```

In [70]:
```
data.dtypes
```

Out[70]:
```
Name                object
Gender              object
DOB         datetime64[ns]
Mobile_no           object
Email               object
dtype: object
```

In [71]:
```
data['DOB'].dt.year # Extract year
```

Out[71]:
```
0    1985
Name: DOB, dtype: int32
```

In [72]:
```
data['DOB'].dt.month # Extract month
```

Out[72]:
```
0    9
Name: DOB, dtype: int32
```

In [73]:
```
data['DOB'].dt.day # Extract day
```

Out[73]:
```
0    24
Name: DOB, dtype: int32
```

Exporting Processed Data

In [74]:
```
data.to_csv('data.csv', index=False)
```

In [ ]:
```

```