Data Toolkit Assignment

Course: Data Science with Generative AI (May 2025)

Anoop Verma (anoopmuz@gmail.com)

Q(1) What is NumPy, and why is it widely used in Python?

Ans:- NumPy (Numerical Python) is a fundamental Python library for scientific computing, particularly for numerical operations. It provides a powerful N-dimensional array object (ndarray) and tools for working with these arrays efficiently.

- Multidimensional Arrays (ndarray)
  NumPy's core feature is the ndarray, which allows efficient storage and manipulation of large datasets.
  Supports vectors (1D), matrices (2D), and higher-dimensional arrays.
- Fast Numerical Computations
  Optimized with C and Fortran under the hood, making it much faster than native Python lists.
  Enables vectorized operations (applying operations to entire arrays without loops).
- Broad Mathematical Functions
  Includes functions for linear algebra, Fourier transforms, random number generation, and more.
  Supports broadcasting, allowing operations on arrays of different shapes.
- Interoperability
  Works seamlessly with other scientific libraries like Pandas, SciPy, Matplotlib, and Scikit-learn.
  Can interface with GPU-accelerated libraries (e.g., CuPy) for high-performance computing.
- Memory Efficiency
  Uses contiguous memory blocks, reducing overhead compared to Python lists.

Used for

- Performance: Much faster than pure Python for numerical computations.
- Ease of Use: Simplifies complex mathematical operations with concise syntax.
- Foundation for Data Science: Essential for machine learning (e.g., TensorFlow, PyTorch) and data analysis (Pandas).
- Large Ecosystem: Integrates well with other scientific and AI/ML libraries.

Q(2) How does broadcasting work in NumPy?

Ans:- Broadcasting is a powerful mechanism that allows NumPy to perform arithmetic operations on arrays of different shapes efficiently, without explicitly expanding them in memory. It follows a set of rules to automatically align dimensions for element-wise operations.

- Shape Alignment: NumPy compares array shapes from right to left (trailing dimensions). Two dimensions are compatible if: They are equal, or One of them is 1 (can be stretched to match the other).

- Size Expansion: If one array has a smaller dimension, NumPy virtually stretches it (without copying data) to match the larger array.
- Error if Incompatible: If shapes cannot be aligned, NumPy raises a ValueError.

Q(3) What is a Pandas DataFrame?

Ans:- A Pandas DataFrame is a two-dimensional, tabular data structure (similar to a spreadsheet or SQL table) that stores data in rows and columns. It is the most commonly used object in the Pandas library for data manipulation and analysis in Python.

- Column Types (Heterogeneous Data): Each column can hold a different data type (int, float, string, datetime, etc.).
- Labeled Axes: Rows are indexed (default: 0, 1, 2,..., but can be customized).
  Columns have names (like in a dictionary).
- Size Mutable: Columns can be added/deleted, and DataFrames can be resized.
- Powerful Data Operations: Supports filtering, grouping, merging, reshaping, and handling missing data. Integrates well with NumPy, Matplotlib, and Scikit-learn.

Q(4) Explain the use of the groupby() method in Pandas.

Ans:- The groupby() method in Pandas is a powerful tool for splitting data into groups, applying functions (like aggregation, transformation, or filtering), and combining results. It is essential for summarizing, analyzing, and transforming structured data efficiently.

- Split: Divides data into groups based on a key (column or condition).
- Apply: Runs a function (e.g., sum(), mean(), custom logic) on each group.
- Combine: Merges results into a new DataFrame or Series.

Q(5) Why is Seaborn preferred for statistical visualizations?

Ans:- Seaborn is a Python data visualization library built on Matplotlib that simplifies the creation of statistically meaningful and beautoful, clean and pleasing plots. It is widely preferred for statistical visualizations because:

- Designed for Statistical Analysis
- Simplified Syntax for Complex Plots
- Beautiful Default Styles & Themes
- Advanced Statistical Plots
- Built-in Datasets & Color Palettes
- Integration with Pandas & NumPy

Q(6) What are the differences between NumPy arrays and Python lists?

Ans:- NumPy arrays and Python lists are both used to store collections of data, but they serve different purposes and have distinct performance characteristics.

- Performance & Speed

  NumPy Arrays =>

  Built in C (optimized for numerical operations).
  10-100x faster than Python lists for mathematical computations.

Supports vectorized operations (no loops needed).

Python Lists =>

Slower for numerical computations due to Python's dynamic typing.
Require loops (for or while) for element-wise operations.

- Memory Efficiency

  NumPy Arrays =>

  Store data in contiguous memory blocks (less overhead).
  Fixed homogeneous data type (e.g., all int32 or float64).
  More memory-efficient for large datasets.

  Python Lists

  Store pointers to objects (higher memory overhead).
  Can hold mixed data types (e.g., [1, "hello", 3.14]).
  Less efficient for numerical data.

- Functionality & Operations
- Flexibility
- Code Readability

Q(7) What is a heatmap, and when should it be used?

Ans:- A heatmap is a 2D graphical representation of data where individual values are represented as colors. It uses a color gradient (e.g., blue → red) to visualize the magnitude of values in a matrix, making patterns, correlations, and variations easily interpretable.

Uses

- Correlation Analysis
- Missing Data Visualization
- Clustering & Pattern Detection
- Time-Series Trends
- Confusion Matrices (ML Evaluation)

Q(8) What does the term "vectorized operation" mean in NumPy?

Ans:- Vectorized operations refer to performing element-wise computations on entire arrays without explicit loops, using NumPy's optimized C-based backend for speed. Instead of processing data one element at a time (as in Python lists), NumPy applies operations to the whole array simultaneously.

Q(9) How does Matplotlib differ from Plotly?

Ans:- Both Matplotlib and Plotly are popular Python visualization libraries, but they serve different purposes and are good in different scenarios.

Matplotlib is static, used for publication quality plots, while Plotly is dynamic and interactive, usedful for web-based applications.

Q(10) What is the significance of hierarchical indexing in Pandas?

Ans:- Hierarchical indexing (or MultiIndex) in Pandas allows you to store and manipulate high-dimensional data in a 2D DataFrame by creating multiple levels of row or column labels. This is especially useful for:

- Representing Complex Data Relationships
- Efficient Data Aggregation
- Advanced Slicing and Selection
- Memory Efficiency

Q(11) What is the role of Seaborn's pairplot() function?

Ans:- pairplot() is a high-level Seaborn function that automatically visualizes pairwise relationships in a dataset. It creates a grid of scatter plots (for numerical variables) and histograms/KDE plots (for single-variable distributions), making it ideal for exploratory data analysis (EDA).

Q(12) What is the purpose of the describe() function in Pandas?

Ans:- The describe() function in Pandas provides a quick statistical summary of numerical and categorical columns in a DataFrame. It is essential for:

- Exploratory Data Analysis (EDA) – Get an instant overview of your data's distribution.
- Identifying Key Statistics – Mean, median, spread, and outliers.
- Data Quality Checks – Spot missing values or anomalies.

Q(13) Why is handling missing data important in Pandas?

Ans:- Missing data (represented as NaN, None, or NaT in Pandas) can skew analysis, break models, and lead to incorrect conclusions. Proper handling is necessary for:

- Accurate Analysis & Statistics
- Model Reliability
- Preventing Bias
- Maintaining Data Structure

Q(14) What are the benefits of using Plotly for data visualization?

Ans:- Plotly is a powerful Python library for creating interactive, publication-quality visualizations that stand out for their flexibility and ease of use. The benefits are following:

- Rich Interactivity
- Wide Range of Chart Types
- Seamless Integration with Dash
- Easy Customization
- Collaboration & Export Options
- Performance with Large Datasets
- Cross-Language Support

Q(15) How does NumPy handle multidimensional arrays?

Ans:- NumPy's ndarray (N-dimensional array) is the core data structure for handling multidimensional numerical data efficiently.

- Memory Layout & Strides: Contiguous Memory Blocks: NumPy arrays store data in a single, contiguous block of memory (unlike Python lists, which store pointers to scattered objects).
- Shape & Reshaping: Shaping -> Tuple representing the size of each dimension (e.g., (3, 4) for 3 rows × 4 columns). Reshaping -> Changes array dimensions without altering data (requires total size to match).
- Broadcasting Rules: NumPy applies operations element-wise across arrays of different shapes by automatically expanding dimensions.
- Vectorized Operations: Avoids Python loops by applying operations entire arrays at once (optimized in C). Supports universal functions (ufuncs) like np.sin(), np.exp().
- Indexing & Slicing: Basic Slicing -> Returns views (not copies) for memory efficiency. Fancy Indexing -> Uses integer/boolean arrays (creates copies).
- Reduction Operations: Aggregations along axes (axis=0 for columns, axis=1 for rows)
- Stacking & Splitting: Concatenation -> np.vstack(), np.hstack(). Splitting -> np.split(), np.vsplit().
- Performance Advantages: Speed -> 10–100x faster than Python lists for numerical operations. Memory -> Compact storage (e.g., int32 uses 4 bytes/element vs. Python's 24+ bytes).

Q(16) What is the role of Bokeh in data visualization?

Ans:- Bokeh is a Python library designed for creating interactive, web-based visualizations that can scale to large datasets. It excels in building dynamic dashboards and applications, particularly for modern browsers.

Q(17) Explain the difference between apply() and map() in Pandas.

Ans:- In Pandas, both apply() and map() are used for transforming data, but they serve different purposes and operate at different levels.

map()
Purpose: Used primarily for element-wise transformations on a Series (a single column).
Input: Works on a pd.Series.
Maps each element in the Series to another value using a dictionary, function, or another Series.
Commonly used for replacing values (similar to a lookup table).
apply()
Purpose: More flexible, used for applying a function along an axis (rows or columns) of a DataFrame or element-wise on a Series.
Input: Works on both pd.Series and pd.DataFrame.
When used on a Series, it behaves similarly to map() but can handle more complex operations.
When used on a DataFrame, it can apply a function row-wise (axis=1) or column-wise (axis=0).

Q(18) What are some advanced features of NumPy?

Ans:- NumPy is a foundational library for numerical computing in Python, and it offers many advanced features that go beyond basic array operations.

- Advanced Indexing: Boolean and Fancy indexing
- Broadcasting: Perform operations on arrays of different shapes without explicit loops.

- Structured Arrays (Custom Data Types): Define arrays with heterogeneous data types (like a DataFrame).
- Universal Functions (ufunc): Fast, vectorized operations (e.g., np.sin, np.exp). Custom ufuncs can be created:
- Linear Algebra (numpy.linalg): Matrix operations (dot, inv, eig), solving linear systems, SVD, etc.
- Memory Efficiency (Views & Copies): Views (no copy) vs. Copies (new memory allocation).
- Masked Arrays (numpy.ma): Handle missing/invalid data without modifying the original array.
- Interoperability (C/Fortran Integration): C-Types (ctypes) & numpy.ctypeslib for low-level integration. np.f2py to call Fortran code from Python.
- Advanced Random Number Generation: Custom distributions, parallel RNGs, and seeding.
- Performance Optimization: np.einsum: Einstein summation for complex tensor operations. numba Integration: JIT-compile NumPy code for speed.
- Interpolation & Polynomials: np.polyfit / np.polyval: Fit polynomials to data. np.interp: Linear interpolation.
- Vectorized String Operations (np.char): Perform string operations on arrays.
- Custom Array Subclassing Extend NumPy arrays with custom methods.

Q(19) How does Pandas simplify time series analysis?

Ans:- Pandas is exceptionally powerful for time series analysis, offering built-in tools that simplify handling, manipulating, and analyzing temporal data.

- Native Time Series Data Structures
  DatetimeIndex: Pandas automatically recognizes datetime strings (e.g., "2023-01-01") and converts them to a DatetimeIndex, enabling fast time-based operations.
- Flexible Time-Based Indexing
  Partial String Indexing: Select data using partial datetime strings (e.g., year, month).
  Slicing with Timestamps
- Resampling and Frequency Conversion
  resample(): Aggregate or downsample data (e.g., daily → monthly).
  Upsampling with Interpolation
- Shifting and Lagging
- Rolling and Expanding Window Operations
- Handling Time Zones
- Periods (Fixed-Frequency Intervals)
- Offset Aliases for Flexible Date Ranges
- Handling Missing Data in Time Series
- Time-Aware Joins and Merging
- Built-in Time Series Visualization
- Holiday Calendars & Custom Business Days
- Seasonality Decomposition

Q(20) What is the role of a pivot table in Pandas?

Ans:- A pivot table in Pandas is a powerful tool for summarizing, aggregating, and analyzing complex datasets by reorganizing and reshaping data into a more readable format. It helps uncover patterns, trends, and relationships, similar to pivot tables in Excel but with more flexibility and programmatic control.

Q(21) Why is NumPy’s array slicing faster than Python’s list slicing?

Ans:- NumPy's array slicing is significantly faster than Python's list slicing due to fundamental differences in how the two handle data in the memory

NumPy Arrays:

Store homogeneous data types (e.g., all integers or floats) in a contiguous block of memory.

Slicing creates a view (not a copy) of the original array, sharing the same memory buffer. This avoids data duplication.

Python Lists:

Store heterogeneous objects (e.g., integers, strings) as references to scattered memory locations.

Slicing always creates a new list (shallow copy), requiring memory allocation and iteration.

Q(22) What are some common use cases for Seaborn?

Ans:- Seaborn is a powerful Python data visualization library built on Matplotlib, designed to make statistical graphics more intuitive and aesthetically pleasing. Here are some of its most common use cases:

- Statistical Relationship Visualization: Explore correlations or trends between numerical variables.
- Distribution Plots: Understand the distribution of a single variable or compare distributions across categories.
- Categorical Data Visualization: Compare values across discrete categories.
- Heatmaps and Clustered Matrices: Visualize matrix-like data (e.g., correlations, confusion matrices).
- Time Series Trends: Plot temporal patterns.
- Pairwise Relationships: Compare all numerical variables in a dataset at once.- Faceting (Multi-Plot Grids): Create small multiples for subsets of data.
- Styling and Themes: Improve aesthetics without extra code.

```
# Q(1) How do you create a 2D NumPy array and calculate the sum of each row?

import numpy as np

# Create a 2D NumPy array (3x3 matrix)
arr = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Calculate the sum of each row
row_sums = np.sum(arr, axis=1)

# Display the original array and row sums
print("Original 2D Array:")
print(arr)
print("\nSum of each row:")
print(row_sums)
```

```
Original 2D Array:
[[1 2 3]
 [4 5 6]
```

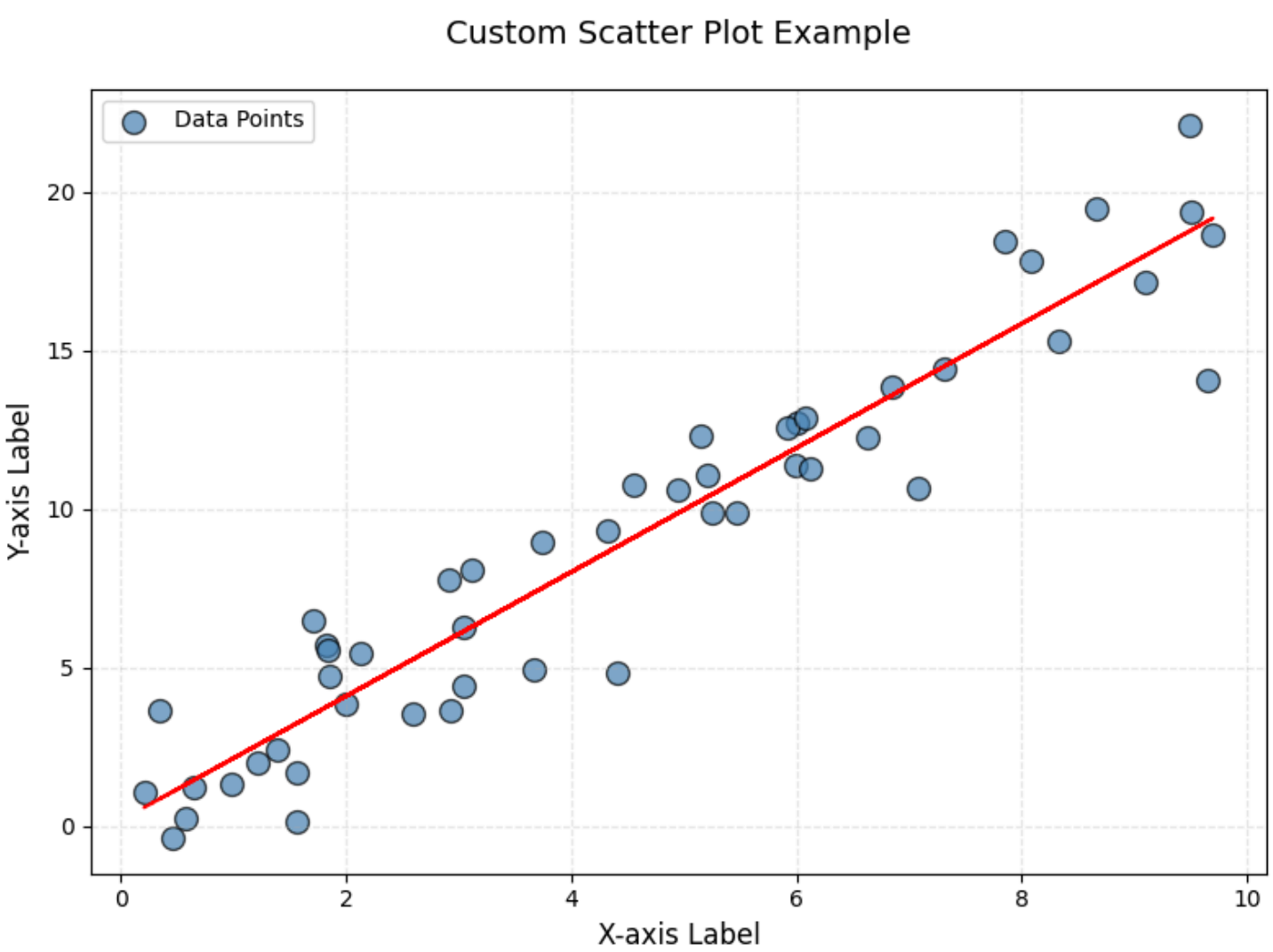
```
 [7 8 9]]

Sum of each row:
[ 6 15 24]
```

```
# Q(2) Write a Pandas script to find the mean of a specific column in a
# DataFrame?

import pandas as pd

# Create a sample DataFrame
data = {
    'Name': ['Anoop', 'Bablu', 'Verma'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)

# Calculate mean of the 'Salary' column
salary_mean = df['Salary'].mean()

print(f"Mean salary: Rs. {salary_mean:,.2f}")
```

```
Mean salary: Rs. 60,000.00
```

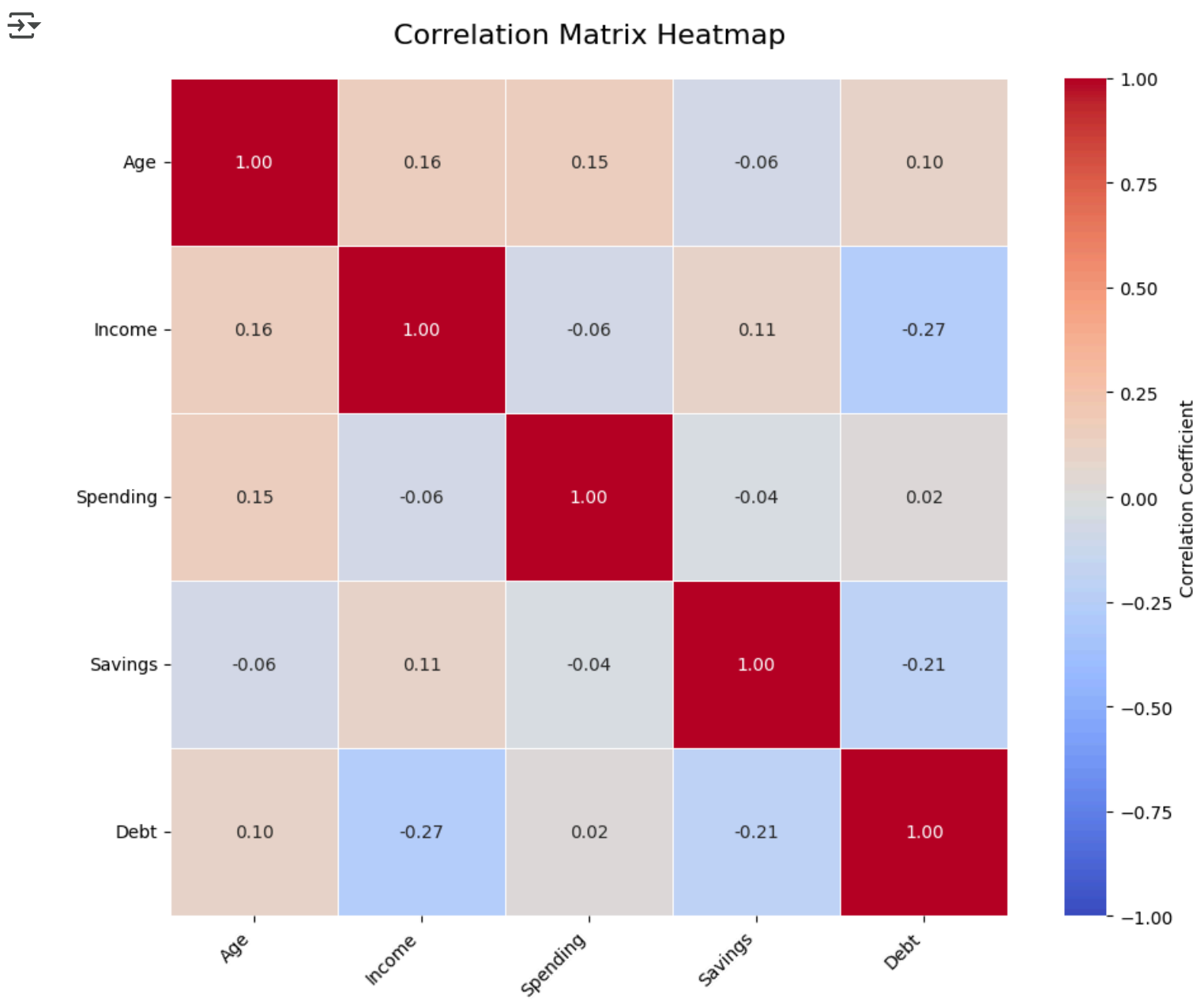
```
# Q(3) Create a scatter plot using Matplotlib.

import matplotlib.pyplot as plt
import numpy as np

# 1. Generate sample data
np.random.seed(42)  # For reproducibility
x = np.random.rand(50) * 10  # 50 random x-values (0-10)
y = 2 * x + np.random.randn(50) * 2  # Linear relationship with noise

# 2. Create the scatter plot
plt.figure(figsize=(8, 6))  # Set figure size

scatter = plt.scatter(
    x,
    y,
    c='steelblue',          # Point color
    s=100,                  # Point size
    alpha=0.7,              # Transparency (0-1)
    edgecolor='black',      # Border color
    linewidth=1,            # Border thickness
    marker='o',             # Shape ('o', 's', '^', etc.)
    label='Data Points'
)

# 3. Customize the plot
plt.title('Custom Scatter Plot Example', fontsize=14, pad=20)
plt.xlabel('X-axis Label', fontsize=12)
plt.ylabel('Y-axis Label', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.3)
plt.legend(fontsize=10)

# 4. Add trendline (optional)
```

```
z = np.polyfit(x, y, 1)
p = np.poly1d(z)
plt.plot(x, p(x), 'r--', label='Trendline')

# 5. Save and display
plt.tight_layout()  # Prevent label cutoff
plt.savefig('scatter_plot.png', dpi=300)  # Save high-res image
plt.show()
```




```
# Q(4) How do you calculate the correlation matrix using Seaborn and
# visualize it with a heatmap?

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# 1. Create a sample DataFrame with numerical data
np.random.seed(42)
data = {
    'Age': np.random.randint(20, 60, 100),
    'Income': np.random.normal(50000, 15000, 100),
    'Spending': np.random.normal(800, 200, 100),
    'Savings': np.random.normal(10000, 3000, 100),
    'Debt': np.random.normal(5000, 2000, 100)
```

```
}
df = pd.DataFrame(data)

# 2. Calculate the correlation matrix
corr_matrix = df.corr()

# 3. Create the heatmap visualization
plt.figure(figsize=(10, 8))

# Customize the heatmap
heatmap = sns.heatmap(
    corr_matrix,
    annot=True,           # Show correlation values in cells
    fmt=".2f",            # Format to 2 decimal places
    cmap='coolwarm',      # Color palette
    vmin=-1, vmax=1,      # Fix color scale from -1 to 1
    linewidths=0.5,       # Add lines between cells
    square=True,          # Keep cells square
    cbar_kws={'label': 'Correlation Coefficient'}  # Color bar label
)

# 4. Enhance the plot
plt.title('Correlation Matrix Heatmap', pad=20, fontsize=16)
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels
plt.yticks(rotation=0)

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




```
# Q(5) Generate a bar plot using Plotly.

import plotly.express as px
import pandas as pd

# Sample data
data = {
    'Category': ['Electronics', 'Clothing', 'Groceries', 'Furniture', 'Books'],
    'Sales': [45000, 32000, 28000, 19000, 12000],
    'Profit': [9000, 8000, 5000, 3500, 2000]
}
df = pd.DataFrame(data)

# Create interactive bar plot
fig = px.bar(
```

```
    df,
    x='Category',                          # X-axis column
    y='Sales',                            # Y-axis column
    color='Profit',                       # Color bars by profit
    color_continuous_scale='Viridis',     # Color scale
    title='Sales by Category with Profit Highlighting',
    labels={'Sales': 'Total Sales (Rs.)', 'Profit': 'Profit (Rs.)'},  # Axis labels
    text='Sales',                         # Show values on bars
    hover_data={'Profit': ':.0f'},        # Format hover tooltip
    height=600                            # Figure height
)

# Customize layout
fig.update_layout(
    title_font_size=20,
    title_x=0.5,                          # Center title
    xaxis_title='Product Category',
    yaxis_title='Total Sales (Rs.)',
    hovermode='x unified',                # Show all data on hover
    bargap=0.3,                           # Gap between bars
    template='plotly_white'               # Clean theme
)

# Format bar text
fig.update_traces(
    texttemplate='%{text:Rs.,.0f}',       # Format numbers with Rs. and commas
    textposition='outside',               # Place text above bars
    marker_line_color='rgb(0,0,0)',       # Black bar borders
    marker_line_width=1.5
)

# Add annotations
fig.add_annotation(
    text="Electronics has highest sales",
    x='Electronics', y=45000,
    showarrow=True,
    arrowhead=1,
    ax=0, ay=-40
)

# Show the plot
fig.show()

# Save as HTML (optional)
fig.write_html("interactive_bar_chart.html")
```


```
# Q(6) Create a DataFrame and add a new column based on an existing column.

import pandas as pd
import numpy as np

# 1. Create a sample DataFrame
data = {
    'Product': ['Laptop', 'Phone', 'Tablet', 'Monitor', 'Keyboard'],
    'Price': [1200, 800, 400, 300, 80],
    'Units_Sold': [15, 32, 45, 12, 60]
}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

# 2. Add new columns using different methods

# Method 1: Simple arithmetic operation (Total Revenue)
df['Revenue'] = df['Price'] * df['Units_Sold']

# Method 2: Apply a function (Price category)
df['Price_Category'] = df['Price'].apply(
    lambda x: 'Premium' if x > 1000 else ('Mid-Range' if x > 500 else 'Budget')
)
```

```python
# Method 3: Using np.where (Discount flag)
df['Has_Discount'] = np.where(df['Price'] < 500, 'Yes', 'No')

# Method 4: Complex calculation (Weighted score)
df['Score'] = (df['Price'] * 0.3) + (df['Units_Sold'] * 0.7)

# Method 5: Using assign() (Chaining method)
df = df.assign(
    Unit_Profit = lambda x: x['Price'] * 0.2,  # 20% profit
    Inventory_Value = lambda x: x['Price'] * x['Units_Sold']
)

print("\nDataFrame with new columns:")
print(df)

# 3. Save to CSV (optional)
df.to_csv('enhanced_product_data.csv', index=False)
```

```
Original DataFrame:
    Product  Price  Units_Sold
0    Laptop   1200          15
1     Phone    800          32
2    Tablet    400          45
3   Monitor    300          12
4  Keyboard     80          60

DataFrame with new columns:
    Product  Price  Units_Sold  Revenue Price_Category Has_Discount  Score  \
0    Laptop   1200          15    18000        Premium           No  370.5
1     Phone    800          32    25600      Mid-Range           No  262.4
2    Tablet    400          45    18000         Budget          Yes  151.5
3   Monitor    300          12     3600         Budget          Yes   98.4
4  Keyboard     80          60     4800         Budget          Yes   66.0

   Unit_Profit  Inventory_Value
0        240.0            18000
1        160.0            25600
2         80.0            18000
3         60.0             3600
4         16.0             4800
```

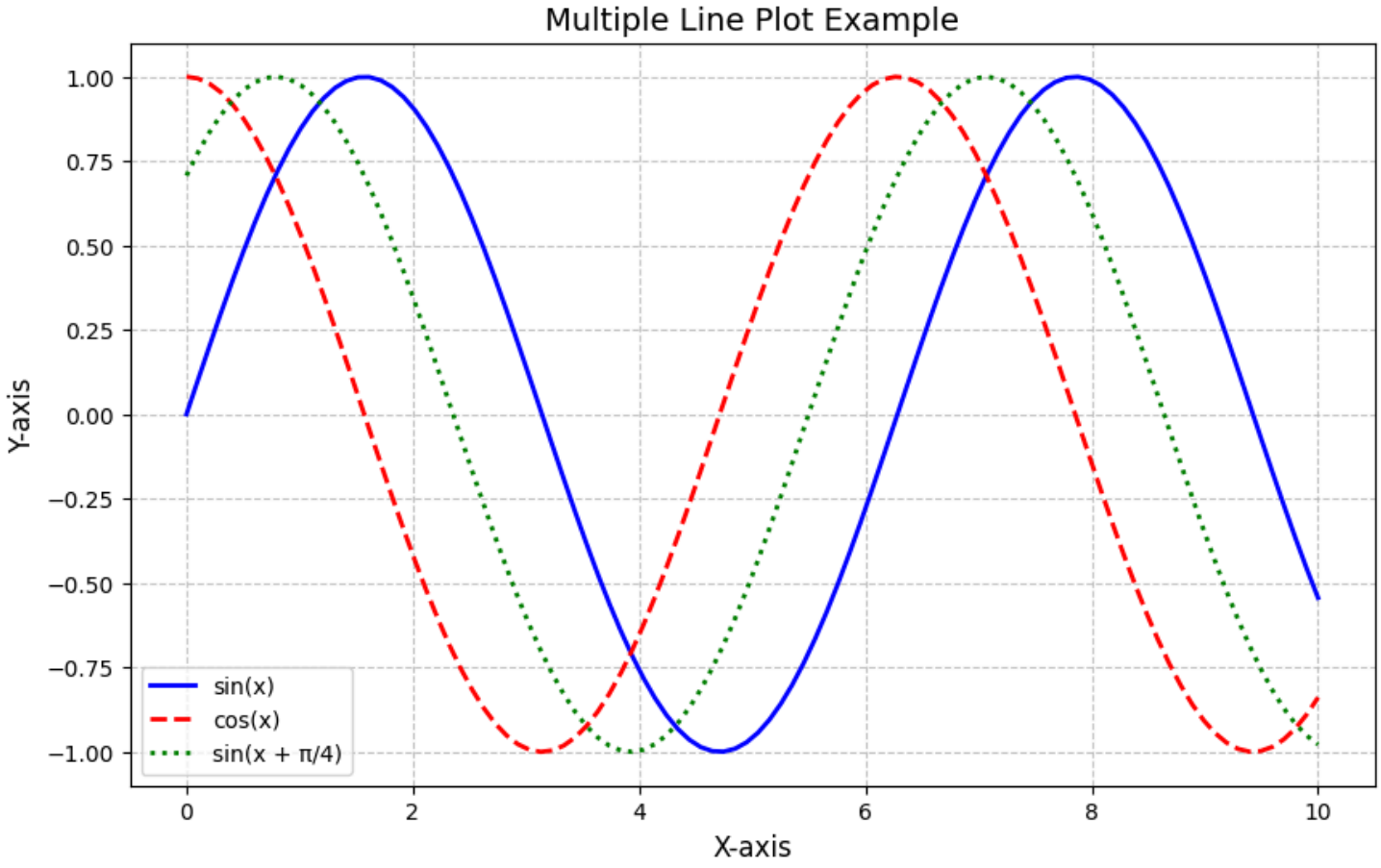
```python
# Q(7) Write a program to perform element-wise multiplication of two NumPy
# arrays.

import numpy as np

def elementwise_multiplication(arr1, arr2):
    """
    Performs element-wise multiplication of two NumPy arrays.

    Parameters:
    arr1 (numpy.ndarray): First input array
    arr2 (numpy.ndarray): Second input array

    Returns:
    numpy.ndarray: Result of element-wise multiplication
    """
    # Check if arrays have the same shape
    if arr1.shape != arr2.shape:
```

```
        raise ValueError("Input arrays must have the same shape for element-wise multiplication")

    # Perform element-wise multiplication
    return arr1 * arr2

# Example usage
if __name__ == "__main__":
    # Create two sample arrays
    array1 = np.array([[1, 2, 3], [4, 5, 6]])
    array2 = np.array([[7, 8, 9], [10, 11, 12]])

    print("Array 1:")
    print(array1)
    print("\nArray 2:")
    print(array2)

    # Perform element-wise multiplication
    result = elementwise_multiplication(array1, array2)

    print("\nElement-wise multiplication result:")
    print(result)
```

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

Array 2:
[[ 7  8  9]
 [10 11 12]]

Element-wise multiplication result:
[[ 7 16 27]
 [40 55 72]]
```

```
# Q(8) Create a line plot with multiple lines using Matplotlib.

import matplotlib.pyplot as plt
import numpy as np

# Create sample data
x = np.linspace(0, 10, 100)  # X-axis values (0 to 10 with 100 points)
y1 = np.sin(x)                # First line: sine wave
y2 = np.cos(x)                # Second line: cosine wave
y3 = np.sin(x + np.pi/4)      # Third line: phase-shifted sine wave

# Create the plot
plt.figure(figsize=(10, 6))    # Set figure size

# Plot each line with different styles and labels
plt.plot(x, y1, label='sin(x)', color='blue', linewidth=2)
plt.plot(x, y2, label='cos(x)', color='red', linestyle='--', linewidth=2)
plt.plot(x, y3, label='sin(x + π/4)', color='green', linestyle=':', linewidth=2)

# Add title and labels
plt.title('Multiple Line Plot Example', fontsize=14)
plt.xlabel('X-axis', fontsize=12)
plt.ylabel('Y-axis', fontsize=12)

# Add legend
```

```
plt.legend(fontsize=10)

# Add grid
plt.grid(True, linestyle='--', alpha=0.7)

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



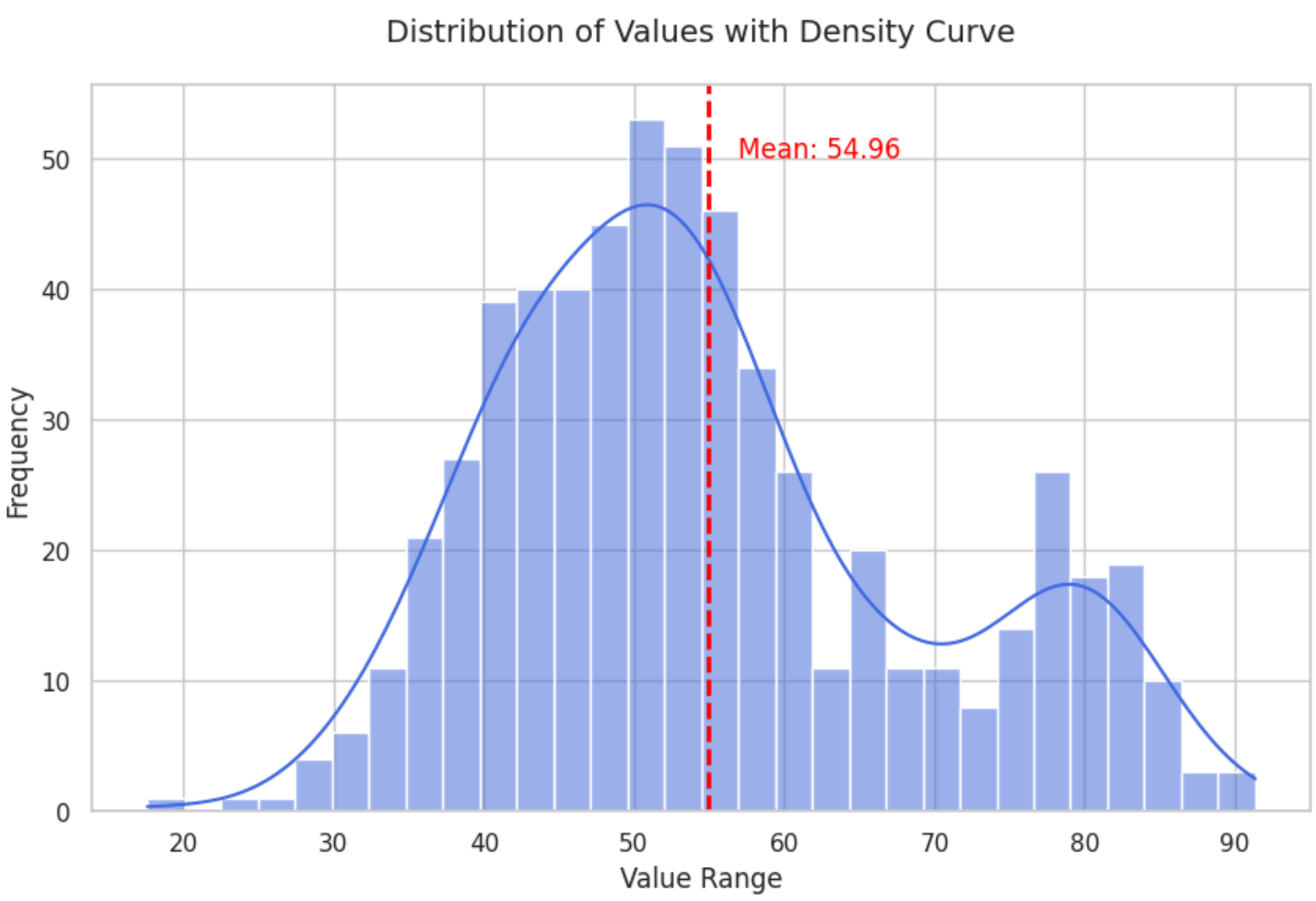

```
# Q(9) Generate a Pandas DataFrame and filter rows where a column value is
# greater than a threshold.

import pandas as pd
import numpy as np

## Create a sample DataFrame
# Set random seed for reproducibility
np.random.seed(42)

# Create DataFrame with random data
df = pd.DataFrame({
    'Product': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
    'Price': np.random.randint(10, 100, size=10),
    'Quantity': np.random.randint(1, 20, size=10),
    'Rating': np.round(np.random.uniform(1, 5, size=10), 1)
})

print("Original DataFrame:")
print(df)
```

```
print("\n" + "="*50 + "\n")

## Filter rows where Price > threshold (50 in this case)
price_threshold = 50
filtered_df = df[df['Price'] > price_threshold]

print(f"Rows where Price > {price_threshold}:")
print(filtered_df)
print("\n" + "="*50 + "\n")

## Filter using multiple conditions (Price > 50 AND Rating > 3.5)
filtered_multiple = df[(df['Price'] > 50) & (df['Rating'] > 3.5)]

print("Rows where Price > 50 AND Rating > 3.5:")
print(filtered_multiple)
```

```
Original DataFrame:
  Product  Price  Quantity  Rating
0       A     61         4     1.1
1       B     24         8     3.1
2       C     81         3     2.6
3       D     70         2     1.2
4       E     30        12     4.9
5       F     92         6     1.9
6       G     96         2     1.4
7       H     84         1     3.5
8       I     84        12     2.5
9       J     97        12     4.9

==================================================

Rows where Price > 50:
  Product  Price  Quantity  Rating
0       A     61         4     1.1
2       C     81         3     2.6
3       D     70         2     1.2
5       F     92         6     1.9
6       G     96         2     1.4
7       H     84         1     3.5
8       I     84        12     2.5
9       J     97        12     4.9

==================================================

Rows where Price > 50 AND Rating > 3.5:
  Product  Price  Quantity  Rating
9       J     97        12     4.9
```

```
# Q(10) Create a histogram using Seaborn to visualize a distribution.

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Set style for better aesthetics
sns.set(style="whitegrid")

# Generate sample data (normally distributed with some outliers)
np.random.seed(42)
data = np.concatenate([np.random.normal(50, 10, 500),
                       np.random.normal(80, 5, 100)])
```

```
# Create DataFrame
df = pd.DataFrame({'Values': data})

# Create histogram with density curve
plt.figure(figsize=(10, 6))
hist_plot = sns.histplot(data=df, x='Values',
                         bins=30,
                         kde=True,  # Adds kernel density estimate
                         color='royalblue',
                         edgecolor='white',
                         linewidth=1)

# Customize the plot
plt.title('Distribution of Values with Density Curve', fontsize=14, pad=20)
plt.xlabel('Value Range', fontsize=12)
plt.ylabel('Frequency', fontsize=12)

# Add vertical line at mean
mean_value = df['Values'].mean()
plt.axvline(mean_value, color='red', linestyle='--', linewidth=2)
plt.text(mean_value+2, plt.ylim()[1]*0.9, f'Mean: {mean_value:.2f}', color='red')

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



```
# Q(11) Perform matrix multiplication using NumPy.

import numpy as np

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

# Matrix multiplication
result = A @ B

print("Result using @ operator:")
print(result)
```

```
Result using @ operator:
[[19 22]
 [43 50]]
```

```
# Method 2
result = np.matmul(A, B)

print("\nResult using np.matmul():")
print(result)
```

```
Result using np.matmul():
[[19 22]
 [43 50]]
```

```
# Method 3
result = np.dot(A, B)

print("\nResult using np.dot():")
print(result)
```

```
Result using np.dot():
[[19 22]
 [43 50]]
```

```
# Matrix multiplication
# Create 3x3 matrices
X = np.random.randint(1, 10, size=(3, 3))
Y = np.random.randint(1, 10, size=(3, 3))

# Matrix multiplication
result = X @ Y

print("\n3x3 Matrix Multiplication Example:")
print("Matrix X:\n", X)
print("\nMatrix Y:\n", Y)
print("\nProduct X @ Y:\n", result)
```

```
3x3 Matrix Multiplication Example:
Matrix X:
 [[4 2 3]
 [3 4 4]
```

```
 [5 4 8]]

Matrix Y:
 [[6 6 8]
 [7 8 2]
 [9 1 8]]

Product X @ Y:
 [[ 65  43  60]
 [ 82  54  64]
 [130  70 112]]
```

```
# Q(12) Use Pandas to load a CSV file and display its first 5 rows.

import pandas as pd
import seaborn as sns

# Load the 'tips' dataset that comes with Seaborn
tips_df = sns.load_dataset('tips')

# Save it temporarily as a CSV file (for demonstration)
tips_df.to_csv('tips.csv', index=False)

# Now load it back with Pandas
df = pd.read_csv('tips.csv')

# Display the first 5 rows
print("\nFirst 5 rows of the tips dataset:")
print(df.head())
```

```
First 5 rows of the tips dataset:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

```
# General code , CSV file not found
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.read_csv('your_file.csv')  # Replace 'your_file.csv' with your actual file path

# Display the first 5 rows
print("First 5 rows of the DataFrame:")
print(df.head())
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipython-input-22-4075807972.py in <cell line: 0>()
      3
      4 # Load the CSV file into a DataFrame
----> 5 df = pd.read_csv('your_file.csv')  # Replace 'your_file.csv' with your actual file
path
      6
      7 # Display the first 5 rows
```

```
# Q(13) Create a 3D scatter plot using Plotly.

import plotly.express as px
import pandas as pd
import numpy as np

# Create sample data
np.random.seed(42)
df = pd.DataFrame({
    'X': np.random.normal(0, 1, 100),
    'Y': np.random.normal(0, 1, 100),
    'Z': np.random.normal(0, 1, 100),
    'Category': np.random.choice(['A', 'B', 'C'], 100),
    'Size': np.random.uniform(5, 20, 100)
})

# Create 3D scatter plot
fig = px.scatter_3d(
    df,
    x='X',
    y='Y',
    z='Z',
    color='Category',  # Different colors for categories
    size='Size',       # Different marker sizes
    opacity=0.8,       # Slightly transparent markers
    title='Interactive 3D Scatter Plot',
    labels={'X': 'X Axis', 'Y': 'Y Axis', 'Z': 'Z Axis'},
    width=800,
    height=600
)

# Customize layout
fig.update_layout(
    scene=dict(
        xaxis_title='X Value',
        yaxis_title='Y Value',
        zaxis_title='Z Value',
        camera=dict(eye=dict(x=1.5, y=1.5, z=0.5))  # Adjust initial view
    ),
    margin=dict(l=0, r=0, b=0, t=30)
)

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