1.What is NumPy, and why is it widely used in Python?

ANS.NumPy (Numerical Python) is a library for working with arrays and mathematical operations in Python. It is widely used due to its:

Key Features

1. Multi-dimensional arrays: Efficiently store and manipulate large datasets.
2. Vectorized operations: Perform operations on entire arrays, reducing loop needs.
3. High-performance computations: Optimized C code underlies NumPy's Python interface.

Advantages

1. Speed: Faster than Python's built-in data structures for numerical computations.
2. Efficient memory use: Compact storage of large datasets.
3. Convenient syntax: Intuitive array operations.

Common Use Cases

1. Scientific computing: Linear algebra, numerical analysis, signal processing.
2. Data analysis: Data science, statistics, machine learning.
3. Data visualization: Integration with Matplotlib, Pandas.

Integration with Other Libraries

1. Pandas (data manipulation and analysis)
2. Matplotlib and Seaborn (data visualization)
3. SciPy (scientific computing)

Benefits

1. Simplifies numerical computations
2. Improves performance
3. Enhances productivity

Example Usage

import numpy as np

Create an array

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

Basic operations

print(np.sum(arr)) # Sum of elements print(np.mean(arr)) # Mean of elements

Array multiplication

arr2 = np.array([2, 2, 2, 2, 2]) print(arr * arr2) # Element-wise multiplication

2.How does broadcasting work in NumPy?

ANS.Broadcasting in NumPy allows arrays with different shapes and sizes to be operated on element-wise. It aligns arrays by adding dimensions, replicating elements, or performing scalar operations.

Key Broadcasting Rules

1. Scalar broadcasting: Scalars (single values) are broadcast to match array shapes.
2. Dimension alignment: Arrays with mismatched dimensions are aligned by adding dimensions (length 1) on the left.

Broadcasting Examples

Scalar Broadcasting

import numpy as np

arr = np.array([1, 2, 3]) result = arr + 2 # Broadcast scalar 2 print(result) # [3, 4, 5]

Array Broadcasting

arr1 = np.array([1, 2, 3]) arr2 = np.array([4]) # or arr2 = np.array([[4]]) result = arr1 + arr2 # Broadcast arr2 to match arr1's shape print(result) # [5, 6, 7]

2D Array Broadcasting

arr1 = np.array([[1, 2], [3, 4]]) arr2 = np.array([10, 20]) # Broadcast arr2 to match arr1's columns result = arr1 + arr2 print(result)

[[11, 22],

[13, 24]]

Broadcasting with np.newaxis or None

Insert a new dimension.

arr = np.array([1, 2, 3]) arr_2d = arr[np.newaxis, :] # or arr[None, :] print(arr_2d.shape) # (1, 3)

Real-World Applications

1. Machine learning: Broadcasting enables efficient computations in neural networks.
2. Scientific simulations: Broadcasting simplifies complex calculations.

Best Practices

1. Understand array shapes and dimensions.
2. Use np.newaxis or None to insert new dimensions.
3. Leverage broadcasting for efficient computations.

Consult NumPy documentation for more broadcasting details and examples.

3.What is a Pandas DataFrame?

ANS.A Pandas DataFrame is a two-dimensional, table-like data structure with labeled columns and rows, similar to an Excel spreadsheet or SQL table. It's a core data structure in Pandas, a popular Python library for data manipulation and analysis.

Key Features

1. Labeled axes: Rows and columns have labels, enabling intuitive data access and manipulation.
2. Flexible data types: Columns can store different data types (e.g., integers, strings, dates).
3. Missing data handling: Supports missing data representation and manipulation.

Common Operations

1. Filtering: Select subsets of data using conditional statements.
2. Grouping: Aggregate data by one or more columns.
3. Sorting: Reorder data by one or more columns.
4. Merging: Combine DataFrames based on common columns.
5. Data cleaning: Handle missing data, remove duplicates.

DataFrames are useful for

1. Data analysis and visualization
2. Data cleaning and preprocessing
3. Data transformation and aggregation
4. Data science and machine learning

Example

import pandas as pd

Create a sample DataFrame

data = {'Name': ['John', 'Anna', 'Peter'], 'Age': [28, 24, 35], 'Country': ['USA', 'UK', 'Australia']} df = pd.DataFrame(data)

print(df)

Output:

```
 Name  Age    Country
```

0 John 28 USA 1 Anna 24 UK 2 Peter 35 Australia

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

ANS.The groupby() method in Pandas is used to split a DataFrame into groups based on some criteria, perform operations on each group, and then combine the results. It's a powerful tool for data analysis and aggregation.

Basic Syntax

df.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False)

Key Parameters

1. *by*: Column(s) or index level(s) to group by.

Common Use Cases

1. Aggregation: Calculate sum, mean, count, max, min, etc., for each group.
2. Filtering: Filter groups based on conditions.
3. Transformation: Apply custom functions to each group.

Example

import pandas as pd

Sample data

data = {'City': ['New York', 'New York', 'Los Angeles', 'Los Angeles', 'Chicago'], 'Sales': [100, 200, 50, 75, 150]} df = pd.DataFrame(data)

Group by City and calculate total Sales

grouped = df.groupby('City')['Sales'].sum()

print(grouped)

Output:

City Chicago 150 Los Angeles 125 New York 300 Name: Sales, dtype: int64

5.Why is Seaborn preferred for statistical visualizations?

ANS.Seaborn is a popular Python data visualization library built on top of Matplotlib. It's preferred for statistical visualizations due to its:

Key Features

1. High-level abstractions: Easy-to-use functions for complex statistical plots.
2. Aesthetic defaults: Attractive, publication-quality visualizations.
3. Integration with Pandas: Seamless data manipulation and visualization.

Statistical Visualization Capabilities

1. Distribution plots: Visualize univariate and bivariate distributions.
2. Regression plots: Display linear regression models.
3. Heatmaps: Illustrate correlation matrices.

Advantages

1. Simplified visualization: Easy-to-use functions reduce code complexity.
2. Customization: Fine-tune plot appearance.

Common Use Cases

1. Exploratory data analysis
2. Statistical modeling

Example

import seaborn as sns import matplotlib.pyplot as plt

Load example dataset

tips = sns.load_dataset("tips")

Create a regression plot

sns.regplot(x="total_bill", y="tip", data=tips)

Display the plot

plt.show()

6.What are the differences between NumPy arrays and Python lists?

ANS. NumPy arrays and Python lists are both data structures used for storing collections of values. However, they have distinct differences:

NumPy Arrays

1. Homogeneous: Elements must be of the same data type.
2. Multi-dimensional: Supports n-dimensional arrays.
3. Fixed-size: Resizable, but inefficient.
4. Compact memory layout: Efficient memory usage.

Python Lists

1. Heterogeneous: Elements can be of different data types.

2. One-dimensional: Can be nested for multi-dimensional structures.
3. Dynamic size: Efficiently resizable.

Key differences

1. Performance: NumPy arrays are significantly faster for numerical computations.
2. Memory usage: NumPy arrays are more memory-efficient.

Use cases

1. NumPy arrays: Scientific computing, numerical analysis, machine learning.
2. Python lists: General-purpose programming, data structures, algorithms.

Conversion

1. numpy.array() converts a list to an array.
2. list() converts an array to a list.

Consult NumPy documentation for more details.

7.What is a heatmap, and when should it be used?

ANS. A heatmap is a graphical representation of data where values are displayed as colors. It's a powerful visualization tool for:

Types of Heatmaps

1. Correlation heatmaps: Show relationships between variables (e.g., correlation matrices).
2. Density heatmaps: Visualize data density (e.g., geographic distribution).

When to Use Heatmaps

1. Data exploration: Identify patterns, correlations, and outliers.
2. Relationship visualization: Illustrate complex relationships.

Best Practices

1. Choose appropriate color schemes: Diverging or sequential colors for clarity.
2. Use clear labels and titles: Facilitate understanding.

Tools for Creating Heatmaps

1. Seaborn (Python)
2. Matplotlib (Python)
3. Plotly (Python)

Consult visualization libraries' documentation for implementation details.

8.What does the term “vectorized operation” mean in NumPy?

ANS. In NumPy, a vectorized operation is an operation performed on entire arrays (vectors) simultaneously, without explicit loops. This approach enables efficient, concise, and readable code.

Characteristics of Vectorized Operations

1. Element-wise operations: Perform operations on corresponding elements of arrays.
2. Broadcasting: Automatically align arrays with different shapes.

Benefits

1. Faster execution: Vectorized operations are optimized for performance.
2. Concise code: Reduce explicit looping.

Examples of Vectorized Operations

1. Basic arithmetic: a + b, a * b
2. Comparison: a > b
3. Aggregate functions: np.sum(a), `np.mean(a)

Real-World Applications

1. Scientific computing
2. Data analysis and visualization

Best Practices

1. Use NumPy's built-in functions.
2. Leverage broadcasting for flexible operations.

9.A How does Matplotlib differ from Plotly?

ANS. Matplotlib and Plotly are popular Python data visualization libraries with distinct strengths:

Matplotlib

1. Maturity: Established, widely-used library.
2. 2D plotting: Excellent for traditional plots (line, scatter, bar, histogram).
3. Customization: Fine-grained control over plot elements.
4. Integration: Seamless with NumPy, Pandas, and SciPy.

Plotly

1. Interactive visualizations: Zoom, hover, and click interactions.
2. 3D plotting: Supports 3D plots and animations.
3. Web-based: Ideal for web applications and dashboards.

Key differences

1. Interactivity: Plotly excels in interactive visualizations.
2. Dimensionality: Plotly supports 3D plots; Matplotlib focuses on 2D.

Choose Matplotlib for

1. Publication-quality 2D plots
2. Custom, precise layouts

Choose Plotly for

1. Interactive, web-based visualizations
2. 3D plots and animations

10.What is the significance of hierarchical indexing in Pandas?

ANS. Hierarchical indexing (MultiIndex) in Pandas enables efficient data manipulation and analysis by:

Key Benefits

1. Data organization: Structure data with multiple levels of indexing.
2. Easy selection: Access specific data subsets using hierarchical indexing.

Use Cases

1. Time series analysis: Date and time hierarchies.
2. Panel data: Cross-sectional and time-series data.

Key Operations

1. Selecting data: loc and iloc with MultiIndex.
2. Grouping: Group by one or more index levels.

Best Practices

1. Use meaningful index names for clarity.
2. Leverage Pandas' built-in MultiIndex functions.

11.What is the role of Seaborn’s pairplot() function?

ANS.Seaborn's pairplot() function creates a matrix of pairwise relationships between variables in a dataset, displaying:

Key Features

1. Scatterplots: Pairwise relationships between variables.
2. Diagonal plots: Distribution plots (histograms or density plots) for each variable.

Effective Use Cases

1. Exploratory data analysis (EDA): Visualize correlations and relationships.
2. Data quality checks: Identify outliers and patterns.

Customization Options

1. Hue parameter: Color points by category.
2. Palette: Customize color schemes.

Integration

1. Seamlessly works with Pandas DataFrames.

Consult Seaborn documentation for detailed examples and customization options.

1. What is the purpose of the describe() function in Pandas?

ANS.The describe() function in Pandas provides a concise summary of a DataFrame's central tendency, dispersion, and shape, including:

Summary Statistics

1. Count: Number of non-missing values.
2. Mean: Average value.
3. Standard Deviation (std): Measure of variability.
4. Minimum (min): Smallest value.
5. 25% (25%): First quartile (Q1).
6. 50% (50%): Median (second quartile, Q2).
7. 75% (75%): Third quartile (Q3).
8. Maximum (max): Largest value.

Use Cases

1. Exploratory Data Analysis (EDA): Quick overview of data distribution.
2. Data quality checks: Identify outliers, missing values.

Options

1. Include/exclude data types: include or exclude parameters.
2. Percentiles: Customize percentile calculations.

Example

import pandas as pd

Sample data

data = {'Age': [25, 30, 28, 35, 22], 'Height': [170, 180, 175, 190, 165]} df = pd.DataFrame(data)

Generate summary statistics

summary = df.describe()

print(summary)

13.Why is handling missing data important in Pandas?

ANS.Handling missing data is crucial in Pandas because it:

Importance of Handling Missing Data

1. Ensures accuracy: Missing values can skew analysis and modeling results.
2. Prevents errors: Many operations fail or produce unexpected results with missing data.

Common Methods for Handling Missing Data

1. Drop missing values: df.dropna()
2. Fill missing values: df.fillna(), supporting various strategies (e.g., mean, median, forward fill)

Best Practices

1. Inspect data: Identify missing values with df.isnull().sum().
2. Choose appropriate strategies: Consider data context and analysis goals.

14.What are the benefits of using Plotly for data visualization?

ANS.Plotly offers numerous benefits for data visualization:

Key Benefits

1. Interactive Visualizations: Enhance exploration and presentation with zoom, hover, and click interactions.
2. Customization: Fine-tune layouts, colors, and styles.
3. 3D and Animated Plots: Create engaging, dynamic visualizations.

Use Cases

1. Web Applications: Embed interactive plots in dashboards and websites.
2. Presentations: Create engaging, interactive stories.

Advantages

1. Cross-platform compatibility: Seamless integration with Python, R, and other languages.

Popular Plotly Features

1. Dash: Build web applications with interactive Plotly visualizations.

15.How does NumPy handle multidimensional arrays?

ANS.NumPy efficiently handles multidimensional arrays through its powerful N-dimensional array (ndarray) object, offering:

Key Features

1. Multi-dimensional support: Arrays with any number of dimensions.
2. Homogeneous data type: Elements must be of the same data type.
3. Vectorized operations: Perform operations on entire arrays.

Key Operations

1. Indexing and slicing: Access and manipulate array elements.
2. Array reshaping: Change array dimensions with reshape().
3. Array concatenation: Combine arrays with concatenate().

Benefits

1. Efficient memory usage: Compact storage.
2. Fast computations: Optimized C code underlies NumPy.

Example

import numpy as np

Create a 2D array (matrix)

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

Perform element-wise multiplication

result = arr * 2 print(result)

Real-World Applications

1. Scientific computing (linear algebra, numerical analysis)
2. Machine learning (neural networks, data processing)
3. Data analysis and visualization

Best Practices

1. Understand array shapes and indexing.
2. Leverage vectorized operations for performance.
3. Use NumPy's built-in functions for efficient computations.

16.What is the role of Bokeh in data visualization?

ANS.Bokeh is a popular Python library for interactive data visualization, enabling creation of web-based plots, charts, and dashboards. Its key roles include:

Key Features

1. Interactive visualizations: Hover, zoom, pan, and click interactions.
2. Web-based plots: Embed visualizations in web applications and notebooks.
3. Customizable: Control layout, colors, fonts, and styles.
4. High-performance: Efficient rendering for large datasets.

Use Cases

1. Web applications: Integrate interactive visualizations into dashboards.
2. Data exploration: Interactive visualizations for data analysis.
3. Presentations: Create engaging, web-based stories.
4. Real-time data visualization: Stream data into interactive plots.

Advantages

1. Ease of use: Simple, intuitive API.

2. Customization: Fine-grained control over plot elements.
3. Integration: Seamless integration with Pandas, NumPy, and Jupyter Notebooks.

Example

from bokeh.plotting import figure, show import numpy as np

x = np.linspace(0, 10, 100) y = np.sin(x)

p = figure(title="Simple Line Plot") p.line(x, y)

show(p)

Best Practices

1. Use Bokeh's high-level interfaces (e.g., figure) for simplicity.
2. Customize plots with Bokeh's low-level components (e.g., glyphs).
3. Leverage Bokeh's integration with Jupyter Notebooks for interactive visualizations.

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

ANS.In Pandas, apply() and map() are two powerful functions for element-wise operations. While they share similarities, there are key differences:

Key differences

1. Flexibility: apply() allows for more complex operations, including custom functions and lambda functions. map() is optimized for simple, element-wise mappings.
2. Performance: map() is generally faster for simple operations due to its optimized implementation.
3. Return types: apply() can return various data types, including Series, DataFrames, or scalars. map() typically returns a Series.

Use cases

1. *apply()*: Complex operations, data transformations, or aggregations.
2. *map()*: Simple, element-wise mappings (e.g., replacing values, converting data types).

Example

import pandas as pd

Create a sample Series

s = pd.Series([1, 2, 3, 4, 5])

Using map() for simple operation

result_map = s.map(lambda x: x ** 2) print(result_map)

Using apply() for complex operation

result_apply = s.apply(lambda x: x ** 2 if x > 3 else x) print(result_apply)

18.What are some advanced features of NumPy?

ANS.NumPy offers several advanced features for efficient numerical computations:

Advanced Features

1. Broadcasting: Automatically aligns arrays for element-wise operations, enabling flexible computations.
2. Vectorized operations: Performs operations on entire arrays, reducing the need for loops.
3. Linear Algebra functions: Optimized implementations for matrix multiplication, eigenvalue decomposition, and more.
4. Random number generation: High-quality random number generators for simulations and statistical analysis.
5. Advanced indexing: Flexible indexing and slicing using arrays, slices, and boolean masks.

Computational Functions

1. Universal Functions (ufuncs): Element-wise operations (e.g., np.sin, np.exp) with support for broadcasting.
2. Reducing functions: Aggregate operations (e.g., np.sum, np.mean, np.median).
3. Cumulative functions: Cumulative sum, product, and other operations.

Matrix and Linear Algebra Operations

1. Matrix multiplication: np.matmul or @ operator.
2. Eigenvalue decomposition: np.linalg.eig.
3. Singular Value Decomposition (SVD): np.linalg.svd.
4. Determinants: np.linalg.det.

Statistical Functions

1. Mean, median, mode: np.mean, np.median, np.mode.
2. Standard deviation, variance: np.std, np.var.
3. Correlation coefficient: np.corrcoef.

Example

import numpy as np

Vectorized operation

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

Broadcasting

arr1 = np.array([[1, 2], [3, 4]]) arr2 = np.array([5, 6]) result = arr1 + arr2[:, np.newaxis] print(result)

Linear Algebra

mat = np.array([[1, 2], [3, 4]]) eigenvalues, eigenvectors = np.linalg.eig(mat) print(eigenvalues)

19.How does Pandas simplify time series analysis?

ANS.Pandas simplifies time series analysis through its robust data structures and functions, offering:

Key Features

1. Date and time support: Efficient datetime indexing and manipulation.
2. Time series data structures: Series (1D labeled array) and DataFrame (2D labeled data structure).

Simplifications

1. Automatic date parsing: Easy conversion of date strings to datetime format.
2. Resampling: Flexible aggregation and interpolation with resample().
3. Rolling and expanding operations: Efficient calculations with rolling() and expanding().

Common Use Cases

1. Financial analysis: Handle stock prices, returns, and trading data.
2. Sensor data analysis: Process time-stamped sensor readings.

Example

import pandas as pd

Sample time series data

data = {'Value': [10, 20, 30, 40, 50]} index = pd.date_range('2022-01-01', periods=5) ts = pd.Series(data['Value'], index=index)

Resample monthly data to quarterly

quarterly_ts = ts.resample('Q').mean() print(quarterly_ts)

20.What is the role of a pivot table in Pandas?

ANS.In Pandas, a pivot table is a data summarization tool used to:

Key Features

1. Data aggregation: Group and aggregate data by one or more columns.
2. Data transformation: Reshape data from long to wide format.

Common Use Cases

1. Data analysis: Summarize and explore large datasets.
2. Data visualization: Prepare data for plotting.

Creating Pivot Tables

1. pd.pivot_table(): Specify values, index, columns, and aggfunc.

Benefits

1. Improved data insights: Easily identify trends and patterns.
2. Efficient data analysis: Streamline data summarization.

21.Why is NumPy’s array slicing faster than Python’s list slicing?

ANS.NumPy's array slicing is faster than Python's list slicing due to several reasons:

Architectural Differences

1. Memory Layout: NumPy arrays store elements contiguously in memory, enabling efficient slicing.
2. Homogeneous Data Type: NumPy arrays have a uniform data type, reducing overhead.

Performance Optimizations

1. Vectorized Operations: NumPy's slicing leverages optimized C code.
2. Cache Locality: Contiguous memory layout minimizes cache misses.
3. Avoiding Python Overhead: NumPy bypasses Python's dynamic typing and object overhead.

Implementation Advantages

1. View-based Slicing: NumPy creates views into the original array, avoiding data copying.
2. Strided Memory Access: NumPy's slicing uses efficient strided memory access patterns.

Benchmark Comparison

import numpy as np import timeit

NumPy array

arr = np.arange(1000000)

Python list

lst = list(range(1000000))

Slicing benchmarks

numpy_slice = timeit.timeit(lambda: arr[::10], number=100) python_slice = timeit.timeit(lambda: lst[::10], number=100)

print(f"NumPy slicing: {numpy_slice:.6f} sec") print(f"Python slicing: {python_slice:.6f} sec")

This benchmark demonstrates NumPy's slicing performance advantage.

22.What are some common use cases for Seaborn?

ANS.Seaborn is a Python data visualization library built on top of Matplotlib, offering attractive and informative statistical graphics. Common use cases include:

Statistical Graphics

1. Visualizing distributions: distplot for histogram and kernel density estimation.
2. Scatterplot matrices: pairplot for pairwise relationships.
3. Regression plots: regplot for linear regression visualization.
4. Heatmaps: heatmap for correlation matrices or categorical data.

Exploratory Data Analysis (EDA)

1. Data summary: boxplot, violinplot for distribution comparison.
2. Correlation analysis: heatmap for correlation matrix visualization.

Informative and Attractive Visualizations

1. Bar plots: barplot for categorical data comparison.
2. Count plots: countplot for categorical data visualization.

Advanced Use Cases

1. Custom themes: Create publication-quality plots with customizable themes.
2. Integration with Matplotlib: Combine Seaborn's high-level functions with Matplotlib's low-level control.

Real-World Applications

1. Data science and machine learning
2. Statistical analysis and research
3. Business intelligence and data visualization
4. Education and academic research

Example

import seaborn as sns import matplotlib.pyplot as plt

tips = sns.load_dataset("tips") sns.scatterplot(x="total_bill", y="tip", data=tips) plt.show()

```
        Practical
```

1.A How do you create a 2D NumPy array and calculate the sum of each row?

sol.To create a 2D NumPy array and calculate the sum of each row:

Creating a 2D NumPy Array

import numpy as np

Method 1: Using np.array()

array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) print(array_2d)

Method 2: Using np.zeros() or np.ones()

array_2d = np.ones((3, 3), dtype=int) print(array_2d)

Method 3: Using np.random.rand()

array_2d = np.random.rand(3, 3) print(array_2d)

Calculating the Sum of Each Row

Using np.sum() with axis=1

row_sums = np.sum(array_2d, axis=1) print(row_sums)

Alternative method: Using np.ndarray.sum() with axis=1

row_sums = array_2d.sum(axis=1) print(row_sums)

In the code above:

1. We first import the NumPy library.
2. We create a 2D NumPy array using np.array(), np.zeros(), np.ones(), or np.random.rand().

3. We calculate the sum of each row using np.sum() or the sum() method of the NumPy array, specifying axis=1 to indicate row-wise summation.

Example Output

For the array [[1, 2, 3], [4, 5, 6], [7, 8, 9]], the output will be:

[6 15 24]

2.Write a Pandas script to find the mean of a specific column in a DataFrameA.

sol.Here's a simple Pandas script to calculate the mean of a specific column in a DataFrame:

Import Pandas library

import pandas as pd

Create a sample DataFrame

data = { 'Name': ['John', 'Anna', 'Peter', 'Linda'], 'Age': [28, 24, 35, 32], 'Score': [90, 85, 95, 92] } df = pd.DataFrame(data)

Specify the column name

column_name = 'Score'

Calculate the mean of the specified column

mean_value = df[column_name].mean()

print(f"Mean of {column_name}: {mean_value}")

Alternative Methods

Using the mean() function directly on the column

mean_value = df['Score'].mean()

Using the describe() function for summary statistics

summary_stats = df['Score'].describe() print(summary_stats)

Tips

1. Ensure the specified column exists in the DataFrame.
2. Handle missing values using df.fillna() or df.dropna() before calculating the mean.

3. For grouped data, use df.groupby() followed by mean().

3.Create a scatter plot using Matplotlib.

sol.Here's an example of creating a scatter plot using Matplotlib:

import matplotlib.pyplot as plt import numpy as np

Sample data

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

Create scatter plot

plt.scatter(x, y)

Set title and labels

plt.title('Scatter Plot Example') plt.xlabel('X-axis') plt.ylabel('Y-axis')

Display grid

plt.grid(True)

Show plot

plt.show()

Customizing the Scatter Plot

Color and Marker

- c: Specify color (e.g., 'red', '#FF0000', or RGB tuple).
- marker: Choose marker style (e.g., 'o', '^', 's').

Size and Transparency

- s: Set marker size.
- alpha: Adjust transparency (0 = fully transparent, 1 = opaque).

Example with Customizations

import matplotlib.pyplot as plt import numpy as np

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

plt.scatter(x, y, c='blue', marker='^', s=100, alpha=0.7)

plt.title('Customized Scatter Plot') plt.xlabel('X-axis') plt.ylabel('Y-axis') plt.grid(True) plt.show()

4.How do you calculate the correlation matrix using Seaborn and visualize it with a heatmap?

sol.Here's how to calculate the correlation matrix using Pandas and visualize it with a heatmap using Seaborn:

Calculating Correlation Matrix and Visualizing with Heatmap

Import libraries

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

Load sample dataset (e.g., Iris)

from sklearn.datasets import load_iris iris = load_iris() df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

Calculate correlation matrix

corr_matrix = df.corr()

Create heatmap

plt.figure(figsize=(8, 6)) sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', square=True)

Set title

plt.title('Correlation Matrix Heatmap')

Show plot

plt.show()

Explanation

1. Import necessary libraries: Seaborn, Matplotlib, Pandas, and NumPy.
2. Load a sample dataset: Iris dataset from Scikit-learn.

3. Calculate the correlation matrix: Use Pandas' corr() function.
4. Create a heatmap: Seaborn's heatmap() function with annotation and colormap.

Customization Options

1. Colormap: Choose from Seaborn's colormaps (e.g., 'coolwarm', 'viridis', 'Blues').
2. Annotation: Display correlation values with annot=True.
3. Square: Ensure heatmap is square with square=True.
4. Figure size: Adjust plot size with figsize.

5.Generate a bar plot using Plotly.

sol.Here's an example of generating a bar plot using Plotly:

Basic Bar Plot

import plotly.express as px

Sample data

data = { 'Category': ['A', 'B', 'C', 'D'], 'Value': [10, 20, 15, 30] }

Create bar plot

fig = px.bar(data, x='Category', y='Value', title='Bar Plot Example')

Show plot

fig.show()

Customized Bar Plot

import plotly.express as px

Sample data

data = { 'Category': ['A', 'B', 'C', 'D'], 'Value1': [10, 20, 15, 30], 'Value2': [25, 15, 20, 10] }

Create bar plot with multiple bars

fig = px.bar(data, x='Category', y=['Value1', 'Value2'], barmode='group', title='Customized Bar Plot')

Show plot

fig.show()

Key Features

1. Interactive: Zoom, hover, and click interactions.
2. Customizable: Colors, fonts, layout, and more.
3. Multiple bar modes: Grouped, stacked, or overlaid bars.

Tips

1. Use px.bar() for simple bar plots.
2. Customize appearance with color, pattern, and text_auto.
3. Use barmode parameter for grouped, stacked, or overlaid bars.

6.Create a DataFrame and add a new column based on an existing column.

sol.Here's how to create a Pandas DataFrame and add a new column based on an existing column:

Creating a DataFrame and Adding a New Column

import pandas as pd

Create a sample DataFrame

data = { 'Name': ['John', 'Anna', 'Peter', 'Linda'], 'Age': [28, 24, 35, 32] } df = pd.DataFrame(data)

Print the original DataFrame

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

Add a new column 'Adult' based on 'Age'

df['Adult'] = df['Age'].apply(lambda x: 'Yes' if x >= 18 else 'No')

Print the updated DataFrame

print("\nUpdated DataFrame:") print(df)

Explanation

1. Import the Pandas library.

2. Create a sample DataFrame with columns 'Name' and 'Age'.
3. Use the apply() function to create a new column 'Adult' based on the values in 'Age'.
4. The lambda function checks if the age is 18 or older and returns 'Yes' or 'No'.

Alternative Methods

Using np.where()

import numpy as np

df['Adult'] = np.where(df['Age'] >= 18, 'Yes', 'No')

Using Boolean Masking

df['Adult'] = ('Yes' if age >= 18 else 'No' for age in df['Age'])

Using Pandas' Vectorized Operations

df['Adult'] = (df['Age'] >= 18).map({True: 'Yes', False: 'No'})

7.Write a program to perform element-wise multiplication of two NumPy arraysA.

sol.Here's a simple program to perform element-wise multiplication of two NumPy arrays:

import numpy as np

Define two NumPy arrays

array1 = np.array([1, 2, 3, 4, 5]) array2 = np.array([6, 7, 8, 9, 10])

Perform element-wise multiplication

result = np.multiply(array1, array2)

Alternative method using *

result_alt = array1 * array2

print("Array 1:", array1) print("Array 2:", array2) print("Element-wise Multiplication Result:", result) print("Alternative Result:", result_alt)

Output

Array 1: [1 2 3 4 5] Array 2: [6 7 8 9 10] Element-wise Multiplication Result: [6 14 24 36 50] Alternative Result: [6 14 24 36 50]

Explanation

1. Import NumPy: import numpy as np

2. Create two NumPy arrays: np.array()
3. Element-wise multiplication: np.multiply() or * operator

Key Benefits

1. Efficient computation: NumPy operations are optimized for performance.
2. Concise syntax: Element-wise multiplication is easily expressed.

Real-World Applications

1. Scientific computing: Array operations are crucial in numerical simulations.
2. Machine learning: Element-wise multiplication is used in neural network computations.
3. Data analysis: Array operations facilitate data manipulation and transformation.

8.A Create a line plot with multiple lines using Matplotlib.

sol.Here's an example of creating a line plot with multiple lines using Matplotlib:

import matplotlib.pyplot as plt import numpy as np

Data for multiple lines

x = np.linspace(0, 10, 100)

y1 = np.sin(x) y2 = np.cos(x) y3 = np.sin(2*x)

Create line plot

plt.plot(x, y1, label='sin(x)', color='blue') plt.plot(x, y2, label='cos(x)', color='red', linestyle='--')
plt.plot(x, y3, label='sin(2x)', color='green', linewidth=2)

Set title and labels

plt.title('Multiple Line Plot Example') plt.xlabel('x') plt.ylabel('y')

Legend and grid

plt.legend() plt.grid(True)

Show plot

plt.show()

Customization Options

1. Line styles: Solid ('-'), dashed ('--'), dash-dot ('-.'), dotted (':').
2. Colors: Specify color names, hex codes, or RGB tuples.
3. Line widths: Adjust with linewidth parameter.
4. Markers: Add markers with marker parameter.

Real-World Applications

1. Time series analysis: Visualize multiple time series data.
2. Comparative studies: Plot multiple datasets for comparison.

9.Generate a Pandas DataFrame and filter rows where a column value is greater than a thresholdA.

sol.Here's how to generate a Pandas DataFrame and filter rows where a column value is greater than a threshold:

Filtering a DataFrame

import pandas as pd

Generate sample data

data = { 'Name': ['John', 'Anna', 'Peter', 'Linda', 'Phil'], 'Age': [28, 24, 35, 32, 40], 'Score': [90, 85, 95, 92, 98] }

Create DataFrame

df = pd.DataFrame(data)

Print original DataFrame

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

Filter rows where Age > 30

filtered_df = df[df['Age'] > 30]

Print filtered DataFrame

print("\nFiltered DataFrame (Age > 30):") print(filtered_df)

Explanation

1. Import Pandas: import pandas as pd
2. Create sample data: Dictionary with column names and values.
3. Create DataFrame: pd.DataFrame()
4. Filter rows: Use boolean masking (df['Age'] > 30) to select rows.

Advanced Filtering

Multiple Conditions

filtered_df = df[(df['Age'] > 30) & (df['Score'] > 90)]

Using query() Method

filtered_df = df.query('Age > 30 and Score > 90')

10.Create a histogram using Seaborn to visualize a distribution.

sol.Here's an example of creating a histogram using Seaborn to visualize a distribution:

Basic Histogram

import seaborn as sns import matplotlib.pyplot as plt

Load sample dataset (e.g., Tips)

tips = sns.load_dataset("tips")

Create histogram

plt.figure(figsize=(8, 6)) sns.histplot(tips["total_bill"], bins=20, kde=True)

Set title and labels

plt.title("Distribution of Total Bill") plt.xlabel("Total Bill ($)") plt.ylabel("Frequency")

Show plot

plt.show()

Customized Histogram

import seaborn as sns import matplotlib.pyplot as plt

Load sample dataset (e.g., Tips)

tips = sns.load_dataset("tips")

Create histogram with custom bins and color

plt.figure(figsize=(8, 6)) sns.histplot(tips["total_bill"], bins=[0, 10, 20, 30, 40, 50, 60], color="skyblue", kde=True)

Set title and labels

plt.title("Distribution of Total Bill") plt.xlabel("Total Bill ($)") plt.ylabel("Frequency")

Show plot

plt.show()

Key Features

1. Distribution insight: Visualize shape, central tendency, and variability.
2. Customizable: Bin size, color, and kernel density estimation (KDE).
3. Integration: Combine with other Seaborn plots (e.g., boxplots, scatterplots).

Real-World Applications

1. Exploratory data analysis (EDA): Understand variable distributions.
2. Data visualization: Communicate insights effectively.
3. Statistical analysis: Inform hypothesis testing and modeling.

11.Perform matrix multiplication using NumPy.

sol.Here's how to perform matrix multiplication using NumPy:

Matrix Multiplication Example

import numpy as np

Define two matrices

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

Matrix multiplication using np.matmul()

C = np.matmul(A, B)

Alternative method using @ operator

D = A @ B

print("Matrix A:\n", A) print("Matrix B:\n", B) print("Matrix Product (np.matmul()):\n", C) print("Matrix Product (@ operator):\n", D)

Explanation

1. Import NumPy: import numpy as np
2. Define matrices: np.array() or np.matrix()
3. Matrix multiplication: np.matmul() or @ operator

Key Benefits

1. Efficient computation: NumPy's optimized C code.
2. Concise syntax: Easy to read and write.

Real-World Applications

1. Linear algebra: Solve systems of equations.
2. Machine learning: Neural network computations.
3. Data analysis: Data transformation and projection.

12.Use Pandas to load a CSV file and display its first 5 rowsA.

sol.Here's how to use Pandas to load a CSV file and display its first 5 rows:

Import Pandas library

import pandas as pd

Load CSV file

def load_csv(file_path): try: df = pd.read_csv(file_path) return df except FileNotFoundError: print("File not found. Please check the file path.") return None except pd.errors.EmptyDataError: print("No data in file. Please check the file contents.") return None except pd.errors.ParserError: print("Error parsing file. Please check file format.") return None

Display first 5 rows

def display_rows(df): if df is not None: print(df.head())

Example usage

file_path = 'example.csv' # Replace with your CSV file path df = load_csv(file_path) display_rows(df)

Key Steps

1. Import Pandas: import pandas as pd
2. Load CSV: pd.read_csv(file_path)
3. Display rows: df.head()

Tips

1. Ensure the CSV file is in the same directory or provide the full path.
2. Handle exceptions for file errors.
3. Customize row display using df.head(n) where n is the number of rows.

13.Create a 3D scatter plot using Plotly.

sol.Here's an example of creating a 3D scatter plot using Plotly:

import plotly.graph_objects as go import numpy as np

Sample data

np.random.seed(0) x = np.random.randn(100) y = np.random.randn(100) z = np.random.randn(100)

Create 3D scatter plot

fig = go.Figure(data=[go.Scatter3d(x=x, y=y, z=z, mode='markers')])

Customize plot

fig.update_layout(title='3D Scatter Plot Example', scene=dict(xaxis_title='X Axis', yaxis_title='Y Axis', zaxis_title='Z Axis'),)

Show plot

fig.show()

Key Features

1. Interactive visualization: Zoom, rotate, and hover interactions.
2. Customizable: Colors, markers, and layout.

Real-World Applications

1. Data visualization: Explore multivariate relationships.
2. Scientific visualization: Visualize 3D data (e.g., medical imaging).
3. Machine learning: Visualize high-dimensional data.