Introduction to Python Libraries

What are Python Libraries?

- Python libraries are modules or packages that contain pre-written code and functions.
- They are created to extend Python's core functionalities and make complex tasks easier.
- Libraries are designed to perform specific tasks and are created by Python developers and the community.

Why Do We Use Libraries in Python?

- **Code Reusability:** Libraries provide pre-written code, saving time and effort in coding.
- **Enhanced Functionality:** Libraries extend Python's capabilities for various tasks.
- **Community Support:** Python libraries are open-source and have strong developer communities, leading to continuous updates and improvements.
- **Faster Development:** Leveraging existing code from libraries expedites application development.

Overview of Commonly Used Python Libraries

1. **Pandas:**
   - Powerful library for data manipulation and analysis.
   - Provides data structures like DataFrames for efficient data handling.
2. **NumPy:**
   - Fundamental library for numerical computations.
   - Supports large, multi-dimensional arrays and matrices.
3. **Seaborn and Matplotlib:**
   - Data visualization libraries.
   - Seaborn provides attractive statistical plots.
   - Matplotlib offers extensive customization options.
4. **Plotly:**
   - Versatile library for creating interactive visualizations and dashboards.

How to Install and Import Libraries in Python

- Installing Python libraries using package managers like **pip** or conda.
- Importing libraries into Python scripts to access their functionalities.

A step-by-step guide on how to install and import libraries in Python:

1. **Installing Libraries:**

   To install Python libraries, you can use the package manager called **pip**, which is included with Python by default. Open your terminal or command prompt and use the following command to install a library:

```
pip install library_name
```

Replace library_name with the name of the library you want to install. For example, to install the pandas library, you can use:

```
pip install pandas
```

2. Importing Libraries:

Once the library is installed, you need to import it into your Python code to use its functionality. You can do this using the import statement.

For example, to import the pandas library, you can use:

```
import pandas
```

You can also give the library a shorter alias to make it easier to refer to in your code. The common convention is to use import library_name as alias.

For example:

```
import pandas as pd
```

Using Library Functions

After importing the library, you can use its functions and classes in your code. For example, if you imported pandas as pd, you can use **pd.read_csv()** to read data from a CSV file, or **pd.DataFrame()** to create a new DataFrame.

```
import pandas as pd

# Read data from a CSV file

df = pd.read_csv('data.csv')



# Create a new DataFrame

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}

df = pd.DataFrame(data)
```

DataFrames

- DataFrames are a fundamental data structure in **Pandas**, a popular data manipulation and analysis library for Python.
- A DataFrame is a **two-dimensional**, **size-mutable**, and **heterogeneous** tabular data structure with **labeled axes (rows and columns)**.
- It's similar to a spreadsheet or SQL table and is used for data cleaning, exploration, and analysis.

How to work with DataFrames in Python using Pandas

1. Import Pandas

```
import pandas as pd
```

2. Creating a DataFrame

   You can create a DataFrame from various data sources, such as a Python dictionary, a NumPy array, or by reading data from files (e.g., CSV, Excel, SQL databases).

```
# Creating a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)
```

How to work with DataFrames in Python using Pandas

3. Viewing the DataFrame

   To see the contents of the DataFrame, you can use the head() method to display the first few rows.

   ```
   print(df.head())
   ```

4. Accessing Data

   You can access specific columns or rows of the DataFrame using indexing and slicing:

   ```
   # Accessing a single column
   ages = df['Age']

   # Slicing rows
   row = df.iloc[0]  # Get the first row
   ```

How to work with DataFrames in Python using Pandas

5. Data Manipulation

   You can perform various data manipulation operations on the DataFrame, such as filtering, sorting, adding new columns, and more.

   ```
   # Filtering
   adults = df[df['Age'] >= 18]
   # Adding a new column
   df['Gender'] = ['Female', 'Male', 'Male']
   ```

6. Statistical Analysis

   Pandas provides built-in functions for basic statistical analysis of the data in the DataFrame.

   ```
   # Summary statistics
   summary = df.describe()
   # Grouping and aggregating data
   grouped = df.groupby('Gender')['Age'].mean()
   ```

7. Exporting Data

   You can save the DataFrame to various file formats, such as CSV or Excel, using Pandas functions like to_csv() and to_excel().

   ```
   df.to_csv('my_data.csv', index=False)
   ```

Conclusion

- Python libraries are essential tools that extend Python's capabilities and simplify complex tasks.
- In the following sessions, we will explore each of these libraries in-depth and learn how to utilize them for problem-solving and data analysis.
- Let's dive into the world of Python libraries and discover the power they bring to our programming journey.

Data Manipulation and Cleaning with Pandas

Introduction to Pandas

- Pandas is a powerful Python library for data manipulation and analysis.
- It provides data structures and functions needed to efficiently work with structured data.

Why Use Pandas?

- Simplifies data manipulation tasks, making them faster and more intuitive.
- Works well with tabular and structured data.
- Integrates seamlessly with other libraries like NumPy and Matplotlib.

Loading Data from Different Sources

- Pandas supports various file formats, including CSV, Excel, SQL databases, and more.
- Use **pd.read_csv()**, **pd.read_excel()**, or **pd.read_sql()** to load data.

Basic Data Manipulation Operations

- Selecting data using indexing and column names.
- Filtering data based on conditions.
- Transforming data using functions and methods.

Basic Data Manipulation Operations…

Example:

```
import pandas as pd
# Create a DataFrame
data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [28, 24, 22],
        'Gender': ['Male', 'Female', 'Male']}
df = pd.DataFrame(data)

# Selecting specific columns
names = df['Name']
ages = df['Age']

# Filtering data based on conditions
males = df[df['Gender'] == 'Male']

# Transforming data
df['Age'] = df['Age'] + 5
```

Handling Missing Data

- Identify missing values in a dataset.
- Use functions like **isnull()** and **notnull()** to detect missing data.
- Handle missing data using **fillna()** or **dropna()** functions.

Handling Missing Data…

Example:

```
import pandas as pd

# Create a DataFrame with missing values
data = {'A': [1, 2, None, 4],
        'B': [5, None, 7, 8]}
df = pd.DataFrame(data)

# Detect missing values
missing = df.isnull()

# Fill missing values with 0
df.fillna(0, inplace=True)
```

Handling Duplicates

- Identify and remove duplicate rows using **duplicated()** and **drop_duplicates()** functions.

Handling Duplicates

Example:

```
import pandas as pd

# Create a DataFrame with duplicate rows
data = {'A': [1, 2, 2, 3],
        'B': ['X', 'Y', 'Y', 'Z']}
df = pd.DataFrame(data)

# Identify duplicate rows
duplicates = df.duplicated()

# Drop duplicate rows
df.drop_duplicates(inplace=True)
```

Data Cleaning Techniques with Pandas

- Replace values using **replace()** method.
- Rename columns with **rename()** method.
- Handling outliers and extreme values.

Data Cleaning Techniques with Pandas

Example:

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['John Doe', 'Alice Smith', 'Bob Johnson'],
        'Age': ['28', '24', '22']}
df = pd.DataFrame(data)

# Standardize string format
df['Name'] = df['Name'].str.title()

# Convert data types
df['Age'] = df['Age'].astype(int)
```

Data Cleaning Techniques with Pandas

Example:

```
import pandas as pd

# Create a DataFrame with outliers
data = {'Score': [95, 78, 101, 80]}
df = pd.DataFrame(data)

# Replace outlier with NaN
df['Score'] = df['Score'].replace(101, pd.NA)

# Rename column
df.rename(columns={'Score': 'Exam_Score'}, inplace=True)
```

Summary

- Pandas is a powerful library for data manipulation and cleaning in Python.
- It simplifies common data tasks, making them more efficient and intuitive.
- Handle missing data, duplicates, and apply various data cleaning techniques with ease.

Exercise

- Practice loading data from a CSV file using Pandas.
- Perform basic data manipulation and cleaning operations on the dataset.

Data Visualization with Matplotlib and Seaborn

Introduction to Matplotlib and Seaborn Libraries

- Matplotlib and Seaborn are popular Python libraries used for data visualization.
- Matplotlib provides a wide range of plotting options, while Seaborn offers a high-level interface to create attractive statistical graphics.
- Together, they form a powerful toolkit for data visualization in Python.

Creating Basic Plots

We can use Matplotlib to create various basic plots, such as:

- Line plots to visualize trends over time.
- Scatter plots to explore relationships between two variables.
- Bar plots to compare categorical data.
- Histograms to understand the distribution of numerical data.

Import the libraries

```
import matplotlib.pyplot as plt

import seaborn as sns
```

Example: Creating Basic Plots

```
# Line Plot using Matplotlib
plt.plot([1, 2, 3, 4], [10, 20, 25, 30])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.show()

# Scatter Plot using Seaborn
sns.scatterplot(x=[1, 2, 3, 4], y=[10, 20, 25, 30])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```

Customizing Plot Appearance

- Matplotlib allows us to customize the appearance of our plots to make them more informative and visually appealing.
- We can add labels to axes, titles, legends, and colors to make the plots more expressive and understandable.

Demonstrate how to add labels, titles, and legends to the plots.

```
# Customizing Bar Plot using Matplotlib

plt.bar(['A', 'B', 'C'], [50, 30, 40], color=['red', 'blue', 'green'])

plt.xlabel('Categories')

plt.ylabel('Values')

plt.title('Customized Bar Plot')

plt.show()
```

Creating Subplots

- Subplots allow us to combine multiple plots into a single figure, making it easier to compare different visualizations side by side.
- We can use Matplotlib to create subplots and organize our visualizations effectively.

The concept of subplots is beneficial in the following ways:

1. **Visual Comparison:** Subplots enable us to compare multiple datasets visually in a compact and organized manner. By placing related plots together, we can quickly identify similarities and differences, making it easier to draw insights from the data.
2. **Contextual Understanding:** Subplots help provide context for each visualization by presenting them within the same frame of reference. This context is crucial when analyzing different aspects of the same dataset or comparing data from different time periods or categories.

3.Conservation of Space: Instead of creating separate figures for each plot, subplots conserve space, allowing us to fit multiple plots into one figure. This is especially useful when we want to create a comprehensive visual representation of various aspects of the data.

4.Presentation of Related Data: Subplots are particularly valuable when displaying related data. For instance, we can compare the performance of different models, visualize the effect of different parameters on a system, or observe trends across different regions.

6.Facilitating Communication: Subplots make it easier to communicate complex information effectively. By arranging visualizations in an organized manner, we can present a cohesive story to the audience, making it simpler for them to understand the insights from the data.

Example: Create Subplot

```
# Creating Subplots using Matplotlib
plt.subplot(2, 1, 1)
plt.plot([1, 2, 3, 4], [10, 20, 25, 30])
plt.title('Subplot 1')

plt.subplot(2, 1, 2)
plt.plot([1, 2, 3, 4], [5, 15, 20, 25])
plt.title('Subplot 2')

plt.tight_layout()
plt.show()
```

Using Seaborn for Advanced Data Visualization

- Seaborn builds on Matplotlib and provides a high-level interface to create complex and informative statistical graphics.
- It simplifies the process of creating advanced visualizations, such as box plots, violin plots, and pair plots.

# Creating Box Plot using Seaborn

```
sns.boxplot(x='category', y='value', data=data)

plt.xlabel('Categories')

plt.ylabel('Values')

plt.title('Box Plot')

plt.show()
```

Summary

- In this lecture, we covered the fundamentals of data visualization using Matplotlib and Seaborn.
- We learned how to create basic plots, customize their appearance, and combine multiple plots using subplots.
- We also explored how Seaborn can be used for advanced data visualization.
- With these powerful tools at your disposal, you can create engaging and insightful visualizations to gain valuable insights from your data.

Data Visualization with Plotly

Introduction to Plotly library for interactive data visualization

- Interactive and visually appealing plots
- Diverse chart types for various data types
- Enhanced data exploration capabilities

Creating interactive plots

- **Scatter plots:** Visualize relationships between two variables
- **Line plots:** Show trends over time or continuous data
- **Bar plots:** Compare different categories or groups

Adding interactive elements

- **Hover tooltips:** Additional information on data points
- **Zoom and pan:** Focus on specific regions for detailed analysis
- **Interactive legends:** Toggle visibility of different series

Creating dashboards with Plotly

- Combine multiple interactive plots in one layout
- Comprehensive view of data for effective communication
- Facilitate data exploration and analysis

Practical Examples

1. **Example 1 -** Scatter Plot: We will create an interactive scatter plot to visualize the relationship between two variables, such as age and income, for a dataset of individuals.

2. **Example 2 -** Line Plot: We will demonstrate how to create an interactive line plot to visualize the trend in stock prices over a specific period.

3. **Example 3** - Bar Plot: We will use an interactive bar plot to compare sales performance across different regions or products.

Exercise

Key Takeaways

- Plotly is a versatile library for interactive data visualization in Python.
- It allows us to create various types of plots and customize them for effective data representation.
- The interactive elements make data exploration and communication more engaging.
- Dashboards provide a comprehensive view of data and facilitate data-driven decision-making.

Linear Regression with NumPy

- Introduction to NumPy library for numerical computation.
- Understanding NumPy arrays and their operations.

NumPy for Numerical Computation

- NumPy is a powerful library in Python for numerical computations.
- It provides support for large, multi-dimensional arrays and matrices.
- NumPy also offers a wide range of mathematical functions for working with arrays.

NumPy Arrays

- NumPy arrays are similar to Python lists but more efficient and convenient for mathematical operations.
- Arrays can have multiple dimensions (1D, 2D, 3D, etc.).
- Access elements using indexing similar to lists.

NumPy Array Operations

- NumPy allows element-wise operations on arrays.
- Mathematical operations (addition, subtraction, multiplication, etc.) can be performed on arrays.
- Broadcasting allows operations on arrays of different shapes.

Implementing Linear Regression with NumPy

- Linear regression is a statistical method to model the relationship between a dependent variable and one or more independent variables.
- We can implement linear regression using NumPy to find the best-fit line that describes the relationship between the variables.

Linear Regression Equation

- The equation of a simple linear regression can be written as:
- $y = mx + c$
- where y is the dependent variable, x is the independent variable, m is the slope, and c is the intercept.

Example: Linear Regression with NumPy

- Suppose we have data on the number of hours studied and exam scores of students.
- We want to find the linear regression model to predict the scores based on the number of hours studied.

NumPy for Linear Regression

- NumPy provides functions for linear algebra, making it easy to implement linear regression.
- We can use the numpy.polyfit() function to find the best-fit line.

Steps to Implement Linear Regression with NumPy

- **Import the required libraries:** We need to import NumPy and any other libraries necessary for data manipulation and visualization.
- **Prepare the data**: Load the data into NumPy arrays and preprocess it if needed (e.g., handling missing values or scaling features).
- **Define the independent and dependent variables:** Separate the data into input features (independent variable) and target variable (dependent variable).

- **Calculate the coefficients:** Use NumPy's linear algebra functions to calculate the slope and intercept of the best-fit line.
- **Make predictions**: Apply the linear regression model to make predictions on new data.
- **Evaluate the model:** Assess the performance of the model using appropriate metrics such as mean squared error or R-squared

Code Example

```
import numpy as np

Sample data

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

exam_scores = np.array([65, 78, 83, 89, 92])
```

Code Example (contd.)

Finding the best-fit line

```
slope, intercept = np.polyfit(hours_studied, exam_scores, 1)
```

Predicting exam scores

```
predicted_scores = slope * hours_studied + intercept
```

Visualization

- We can visualize the data and the best-fit line using Matplotlib.

```
import matplotlib.pyplot as plt
plt.scatter(hours_studied, exam_scores, label='Data’)
plt.plot(hours_studied, predicted_scores, color='red’, label='Best-fit Line’)
plt.xlabel('Hours Studied’)
plt.ylabel('Exam Scores’)
plt.legend()
plt.show()
```

```
# Importing NumPy
import numpy as np


# Sample data
hours_studied = np.array([2, 3, 5, 7, 8, 10])
exam_scores = np.array([60, 70, 80, 85, 90, 95])


# Calculating the coefficients
m, c = np.polyfit(hours_studied, exam_scores, 1)


# Making predictions
predicted_scores = m * hours_studied + c


# Displaying the coefficients and predictions
print("Slope (m):", m)
print("Intercept (c):", c)
print("Predicted Scores:", predicted_scores)
```

Conclusion

- NumPy is a powerful library for numerical computation and array operations.
- We can use NumPy to implement linear regression and other mathematical operations efficiently.

Logistic Regression with Scikit-learn

Objective: Understand the concepts of logistic regression and how to implement it using Scikit-learn.

Introduction to Logistic Regression

- Logistic regression is a classification algorithm used for binary classification problems.
- It predicts the probability of an instance belonging to a particular class.
- Applications include medical diagnosis, credit risk analysis, spam detection, etc.

Scikit-learn for Logistic Regression

- **Scikit-learn (sklearn):** A machine learning library in Python.
- Provides tools for building and evaluating machine learning models.
- **Logistic Regression:** A class in sklearn for implementing logistic regression.

Splitting Data into Training and Testing Sets

- It's essential to split data into training and testing sets.
- Training set for model training, testing set for evaluation.
- train_test_split: A function in sklearn to split data.

Evaluating the Logistic Regression Model

- Evaluating model performance is crucial.
- Metrics like accuracy, precision, recall, F1-score, ROC curves.
- classification_report: A function in sklearn to generate a classification report.

Code Example

- Import necessary libraries and functions.
- Load dataset (features and target).
- Split data into training and testing sets.
- Initialize LogisticRegression model.
- Train the model on training data.
- Make predictions and calculate accuracy.
- Generate and display the classification report.

Code Example (Contd.)

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset
# X: Features, y: Target variable

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the model
model = LogisticRegression()
```

Code Example (Contd.)

```
# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Generate classification report
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Summary

- Logistic regression is a powerful classification algorithm.
- Scikit-learn simplifies its implementation.
- Splitting data and evaluating the model are crucial steps.
- Classification report provides detailed performance metrics.