Lab Program 1: Performing a linear search on a given dataset
Version 1:

```
# Function to perform linear search
def linear_search(data, target):
    # Traverse through all elements in the list
    for index, value in enumerate(data):
        # If the target is found, return the index
        if value == target:
            return index  # Returning the index where the target is found
    return -1  # Return -1 if the target is not found

# Example usage
dataset = [5, 2, 9, 1, 5, 6]
target_value = 9

# Call the linear_search function
result = linear_search(dataset, target_value)

if result != -1:
    print(f"Element {target_value} found at index {result}.")
else:
    print(f"Element {target_value} not found in the dataset.")
```

Version 2:

```
# Function to perform linear search
def linear_search(data, target):
    # Traverse through all elements in the list
    for index, value in enumerate(data):
        # If the target is found, return the index
        if value == target:
            return index  # Returning the index where the target is found
    return -1  # Return -1 if the target is not found

# Take dataset as input from the user
dataset = list(map(int, input("Enter the dataset (numbers separated by spaces): ").split()))

# Take target value as input from the user
target_value = int(input("Enter the target value to search: "))

# Call the linear_search function
result = linear_search(dataset, target_value)
```

```
if result != -1:
    print(f"Element {target_value} found at index {result}.")
else:
    print(f"Element {target_value} not found in the dataset.")
```

"""

Lab Program 2: insert an element into a sorted list.

"""

```
def insert_into_sorted_list(sorted_list, element):
    # Traverse the list to find the correct position for insertion
    position = 0
    for i in range(len(sorted_list)):
        if sorted_list[i] >= element:
            position = i
            break
    else:
        # If the element is greater than all elements in the list
        position = len(sorted_list)

    # Insert the element at the correct position
    sorted_list = sorted_list[:position] + [element] + sorted_list[position:]
    return sorted_list

# Example usage
sorted_list = [1, 3, 4, 7, 9]
element = 5
updated_list = insert_into_sorted_list(sorted_list, element)
print("Updated sorted list:", updated_list)
```

```python
"""
Lab Program 3: Object-oriented programming to demonstrate encapsulation, overloading and inheritance.
"""
# Base class demonstrating encapsulation
class Animal:
    def __init__(self, name, age):
        self.__name = name  # private variable
        self.__age = age    # private variable

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    def sound(self):  # To be overridden by subclasses
        pass

# Derived class demonstrating inheritance
class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed  # public variable

    # Overriding the sound method
    def sound(self):
        return "Bark"

# Demonstrating method overloading using default arguments
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

# Creating objects and demonstrating the concepts
def main():
    # Encapsulation
    animal = Animal("Elephant", 10)
```

```python
    print(f"Animal Name: {animal.get_name()}, Age: {animal.get_age()}")
    animal.set_name("Lion")
    animal.set_age(5)
    print(f"Updated Animal Name: {animal.get_name()}, Age: {animal.get_age()}")

    # Inheritance
    dog = Dog("Tony", 3, "Golden Retriever")
    print(f"Dog Name: {dog.get_name()}, Age: {dog.get_age()}, Breed: {dog.breed}")
    print(f"Dog Sound: {dog.sound()}")

    # Overloading
    calc = Calculator()
    print(f"Addition (2 args): {calc.add(5, 10)}")
    print(f"Addition (3 args): {calc.add(5, 10, 15)}")

if __name__ == "__main__":
    main()
```

```python
"""
4 Implement a python program to demonstrate
	a) Importing Datasets
	b) Cleaning the data
	c) Data frame manipulation using Pandas
"""

import pandas as pd

# a) Importing Datasets
# Reading data from a CSV file
df = pd.read_csv('/content/vehicles_sample.csv')

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

# b) Cleaning the Data
# Handling missing values
# Filling missing values with median for Year and mean for Price
df['Year'].fillna(df['Year'].median(), inplace=True)
df['Price'].fillna(df['Price'].mean(), inplace=True)

# Removing duplicates
df.drop_duplicates(inplace=True)

print("\nDataFrame after cleaning:")
print(df)

# c) Data Frame Manipulation
# Selecting specific columns
df_selected = df[['Make', 'Model', 'Year']]

print("\nSelected Columns (Make, Model, Year):")
print(df_selected)

# Filtering data (e.g., vehicles with Price > 34000)
df_filtered = df[df['Price'] > 34000]

print("\nFiltered Data (Price > 34000):")
print(df_filtered)

# Aggregating data (e.g., average price by make)
df_grouped = df.groupby('Make')['Price'].mean().reset_index()
```

```
print("\nAverage Price by Make:")
print(df_grouped)

# Adding a new column (e.g., Price after 5% discount)
df['Price_After_Discount'] = df['Price'] * 0.95

print("\nDataFrame with Price After Discount:")
print(df)

# Sorting data by Year
df_sorted = df.sort_values(by='Year')

print("\nDataFrame sorted by Year:")
print(df_sorted)

# Saving the cleaned DataFrame to a new CSV file
df.to_csv('/content/cleaned_vehicles.csv', index=False)
```

Steps to Execute the Program in Google Colab

- **Download the dataset** vehicles_sample.csv
- Open Google Colab:
    - Go to Google Colab.
- Create a New Notebook:
    - Click on File -> New Notebook.
- Upload the CSV File:
    - In the left sidebar, click on the Files tab.
    - Click on the Upload button to upload your vehicles_sample.csv file.
- Write the Python Code given above
- Run the Code:
    - Click on the play button next to the code cell to execute the code.
- Download the Cleaned CSV File:

  - After the code runs, the cleaned DataFrame will be saved as cleaned_vehicles.csv in the /content/ directory.

  To download the cleaned CSV file, you can use the following code in a new code cell:

  from google.colab import files
  files.download('/content/cleaned_vehicles.csv')

Explanation of the Code

- Importing Datasets:
    - The dataset is read from a CSV file named vehicles_sample.csv using pd.read_csv.

- Cleaning the Data:
    - Handle missing values in the Year column using fillna with the median and in the Price column using fillna with the mean.
    - Remove duplicates using drop_duplicates.
- Data Frame Manipulation:
    - Select specific columns using column indexing.
    - Filter data for vehicles with a price greater than 34000.
    - Aggregate data to calculate the average price by make using groupby.
    - Add a new column to calculate the price after a 5% discount.
    - Sort data by year using sort_values.
    - Save the cleaned and manipulated DataFrame to a new CSV file named cleaned_vehicles.csv using to_csv.

“””

5) Implement a python program to demonstrate the following using numpy

a) Array manipulation, Searching, Sorting and splitting.

b) broadcasting and Plotting numpy arrays

“””

```
import numpy as np
import matplotlib.pyplot as plt

# a) Array Manipulation, Searching, Sorting, and Splitting

# Creating a 2D array
array = np.arange(1, 17).reshape(4, 4)
print("Original 2D Array:\n", array)

# Reshaping the array to 2x8
reshaped_array = array.reshape(2, 8)
print("\nReshaped Array (2x8):\n", reshaped_array)

# Flattening the array
flattened_array = array.flatten()
print("\nFlattened Array:\n", flattened_array)

# Slicing the array
sliced_array = array[1:3, 1:3]  # Extracting a 2x2 sub-array
print("\nSliced Array (2x2):\n", sliced_array)

# Sorting the flattened array
sorted_array = np.sort(flattened_array)
print("\nSorted Flattened Array:\n", sorted_array)

# Splitting the array into 2 sub-arrays
split_arrays = np.array_split(array, 2)
print("\nSplit Arrays:")
for i, split_array in enumerate(split_arrays, start=1):
    print(f"Array {i}:\n{split_array}")

# b) Broadcasting and Plotting NumPy Arrays

# Creating arrays for broadcasting
x = np.array([[1], [2], [3]])
y = np.array([10, 20, 30])
print(x)
print(y)
```

```
# Broadcasting: Adding x and y
broadcasted_result = x + y
print("\nBroadcasted Result (x + y):\n", broadcasted_result)

# Plotting NumPy Arrays
# Creating a sine wave using numpy
t = np.linspace(0, 2 * np.pi, 100)
sin_wave = np.sin(t)
print(sin_wave)

# Plotting the sine wave
plt.plot(t, sin_wave)
plt.title('Sine Wave')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
```

Explanation of the Code

a) Array Manipulation, Searching, Sorting, and Splitting

- Array Creation: A 4x4 array is created using np.arange and reshape.
- Reshaping: The array is reshaped to 2x8 using the reshape method.
- Flattening: The array is flattened into a 1D array using flatten.
- Slicing: A 2x2 sub-array is sliced from the original array.
- Sorting: The flattened array is sorted using np.sort.
- Splitting: The array is split into two sub-arrays using np.array_split.

b) Broadcasting and Plotting NumPy Arrays

- Broadcasting: Arrays x and y are broadcasted and added together.
- Plotting: A sine wave is generated using np.sin and plotted using Matplotlib.

"""

6) Implement a python program to demonstrate Data visualization with various Types of Graphs using numpy.

"""

```
import numpy as np
import matplotlib.pyplot as plt

# Generating sample data using NumPy
np.random.seed(42)  # For reproducibility

# Line plot data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Scatter plot data
x_scatter = np.random.rand(100)
y_scatter = np.random.rand(100)

# Bar chart data
categories = ['A', 'B', 'C', 'D', 'E']
values = np.random.randint(5, 20, size=5)

# Histogram data
data = np.random.randn(1000)

# Pie chart data
labels = ['Python', 'Java', 'C++', 'JavaScript']
sizes = [35, 25, 20, 20]

# Creating the subplots
plt.figure(figsize=(14, 10))

# Line plot
plt.subplot(2, 3, 1)
plt.plot(x, y, color='blue', linestyle='-', linewidth=2)
plt.title('Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)

# Scatter plot
plt.subplot(2, 3, 2)
plt.scatter(x_scatter, y_scatter, color='red', alpha=0.5)
plt.title('Scatter Plot')
```

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)

# Bar chart
plt.subplot(2, 3, 3)
plt.bar(categories, values, color='green')
plt.title('Bar Chart')
plt.xlabel('Categories')
plt.ylabel('Values')

# Histogram
plt.subplot(2, 3, 4)
plt.hist(data, bins=20, color='purple', edgecolor='black')
plt.title('Histogram')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Pie chart
plt.subplot(2, 3, 5)
plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=['skyblue', 'orange', 'lightgreen', 'pink'])
plt.title('Pie Chart')

# Display all plots
plt.tight_layout()
plt.show()
```

Explanation of the Code

- Data Generation:
  - Line Plot Data: A sine wave is generated using np.sin and np.linspace.
  - Scatter Plot Data: Random x and y coordinates are generated using np.random.rand.
  - Bar Chart Data: Random integer values for categories are generated using np.random.randint.
  - Histogram Data: Normally distributed random data is generated using np.random.randn.
  - Pie Chart Data: Proportions for different categories (languages) are predefined.
- Subplots:
  - The plt.subplot function is used to create a grid of subplots in a single figure.
  - Each subplot displays a different type of graph: line plot, scatter plot, bar chart, histogram, and pie chart.
- Plotting:
  - Line Plot: A simple sine wave with grid lines.
  - Scatter Plot: Scatter points with random positions.

- Bar Chart: Categories plotted against random values.
- Histogram: Distribution of data in bins.
- Pie Chart: Proportions of different categories.

- Displaying the Plots:
  - The plt.tight_layout() function adjusts the spacing between subplots to prevent overlap.
  - The plt.show() function displays all the plots together.

“””

7) Write a Python program that creates a m X n integer array and print its attributes using matplotlib.

“””

```
import numpy as np
import matplotlib.pyplot as plt

def create_and_visualize_matrix(m, n):
    # Create a random m x n integer array
    matrix = np.random.randint(0, 100, size=(m, n))

    # Create a figure and axis
    fig, ax = plt.subplots(figsize=(10, 8))

    # Display the matrix as an image
    im = ax.imshow(matrix, cmap='viridis')

    # Add a colorbar
    cbar = ax.figure.colorbar(im, ax=ax)
    cbar.ax.set_ylabel("Value", rotation=-90, va="bottom")

    # Set title and labels
    ax.set_title(f"Visualization of {m}x{n} Integer Matrix")
    ax.set_xlabel("Column Index")
    ax.set_ylabel("Row Index")

    # Add text annotations in each cell
    for i in range(m):
        for j in range(n):
            text = ax.text(j, i, matrix[i, j], ha="center", va="center", color="w")

    # Display the plot
    plt.tight_layout()
    plt.show()

    # Print matrix attributes
    print(f"Matrix shape: {matrix.shape}")
    print(f"Matrix size: {matrix.size}")
    print(f"Matrix data type: {matrix.dtype}")
    print(f"Matrix dimensions: {matrix.ndim}")
    print(f"Minimum value: {matrix.min()}")
    print(f"Maximum value: {matrix.max()}")
    print(f"Mean value: {matrix.mean():.2f}")
```

```
# Example usage
create_and_visualize_matrix(5, 7)
```

“””

8) Write a Python program to demonstrate the generation of

- **Linear regression models.**
- **Logistic regression models.**

“””

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score, confusion_matrix

def linear_regression_demo():
    print("Linear Regression Demo")
    print("----------------------")

    # Generate sample data
    np.random.seed(0)
    X = np.random.rand(100, 1) * 10
    y = 2 * X + 1 + np.random.randn(100, 1)

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

    # Create and train the model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate the model
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"Coefficients: {model.coef_[0][0]:.4f}")
    print(f"Intercept: {model.intercept_[0]:.4f}")
    print(f"Mean squared error: {mse:.4f}")
    print(f"R-squared score: {r2:.4f}")

    # Visualize the results
    plt.figure(figsize=(10, 6))
    plt.scatter(X_test, y_test, color='blue', label='Actual data')
    plt.plot(X_test, y_pred, color='red', label='Regression line')
    plt.title('Linear Regression')
```

```
    plt.xlabel('X')
    plt.ylabel('y')
    plt.legend()
    plt.show()

def logistic_regression_demo():
    print("\nLogistic Regression Demo")
    print("------------------------")

    # Generate sample data
    np.random.seed(0)
    X = np.random.randn(100, 2)
    y = (X[:, 0] + X[:, 1] > 0).astype(int)

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

    # Create and train the model
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate the model
    accuracy = accuracy_score(y_test, y_pred)
    conf_matrix = confusion_matrix(y_test, y_pred)

    print(f"Coefficients: {model.coef_[0]}")
    print(f"Intercept: {model.intercept_[0]:.4f}")
    print(f"Accuracy: {accuracy:.4f}")
    print("Confusion Matrix:")
    print(conf_matrix)

    # Visualize the results
    plt.figure(figsize=(10, 6))
    plt.scatter(X_test[y_test == 0][:, 0], X_test[y_test == 0][:, 1], label='Class 0', marker='o')
    plt.scatter(X_test[y_test == 1][:, 0], X_test[y_test == 1][:, 1], label='Class 1', marker='s')

    # Plot decision boundary
    x1_min, x1_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
    x2_min, x2_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.02),
                           np.arange(x2_min, x2_max, 0.02))
```

```
    Z = model.predict(np.c_[xx1.ravel(), xx2.ravel()])
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3)

    plt.title('Logistic Regression')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    linear_regression_demo()
    logistic_regression_demo()
```

“””

9) Write a Python program to demonstrate Time series analysis with Pandas.

“””

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Generate a date range
date_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')

# Create a sample time series DataFrame
np.random.seed(42)  # For reproducibility
data = np.random.randn(len(date_rng)) * 10 + 50  # Random data points around 50
df = pd.DataFrame(data, columns=['value'], index=date_rng)

# Display the first few rows of the DataFrame
print("Sample Time Series Data:\n", df.head())

# Plot the time series
plt.figure(figsize=(10, 6))
plt.plot(df.index, df['value'], label='Daily Data')
plt.title('Daily Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()

# Resample the time series data to monthly frequency and calculate the mean
df_monthly = df.resample('ME').mean()

# Display the resampled data
print("\nMonthly Resampled Data:\n", df_monthly)

# Plot the resampled time series
plt.figure(figsize=(10, 6))
plt.plot(df_monthly.index, df_monthly['value'], label='Monthly Average', color='orange')
plt.title('Monthly Average Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

```
# Compute and plot a rolling window (moving average)
df['rolling_mean'] = df['value'].rolling(window=30).mean()

plt.figure(figsize=(10, 6))
plt.plot(df.index, df['value'], label='Daily Data')
plt.plot(df.index, df['rolling_mean'], label='30-Day Rolling Mean', color='red')
plt.title('Time Series with 30-Day Rolling Mean')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

“”””

10) Write a Python program to demonstrate Data Visualization using Seaborn.

“”””

```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Load the iris dataset
iris = sns.load_dataset('iris')

# Display the first few rows of the dataset
print("Iris Dataset Sample:\n", iris.head())

# Set the theme for Seaborn
sns.set_theme(style="whitegrid")

# 1. Scatter plot with Seaborn
plt.figure(figsize=(8, 6))
sns.scatterplot(x='sepal_length', y='sepal_width', hue='species', data=iris)
plt.title('Sepal Length vs Sepal Width')
plt.show()

# 2. Pair plot to visualize pairwise relationships in the dataset
plt.figure(figsize=(8, 6))
sns.pairplot(iris, hue='species', diag_kind='kde')
plt.suptitle('Pair Plot of Iris Dataset', y=1.02)
plt.show()

# 3. Boxplot to visualize the distribution of sepal length across species
plt.figure(figsize=(8, 6))
sns.boxplot(x='species', y='sepal_length', data=iris)
plt.title('Box Plot of Sepal Length by Species')
plt.show()

# 4. Violin plot for distribution and comparison
plt.figure(figsize=(8, 6))
sns.violinplot(x='species', y='petal_length', data=iris)
plt.title('Violin Plot of Petal Length by Species')
plt.show()

# 5. Heatmap for the correlation matrix
plt.figure(figsize=(8, 6))
correlation_matrix = iris.drop('species', axis=1).corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
```

```
plt.title('Heatmap of Feature Correlations')
plt.show()
```

https://mite-students.contineo.in/parentsodd