SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 5th Sem

INTRODUCTION TO DATA SCIENCE (UCS23G03J)- Lab Manual

This document outlines the procedures and expected outcomes for each lab program in the "Introduction to Data Science" course. Each lab is structured with a Title, Aim, Procedure, Source Code, Input, and Expected Output to guide your practical learning.

Lab 1: Perform Analysis on Simple Dataset I for Data Science

- **Title:** Performing Basic Data Analysis on a Simple Dataset
- **Aim:** To understand and apply fundamental data analysis techniques, including descriptive statistics and data inspection, on a simple dataset using Python's Pandas library.
- **Procedure:**
    1. **Prepare Data:** Create a simple CSV file named `sample_data.csv` with a few columns (e.g., `Name`, `Age`, `Score`) and some sample data.
    2. **Install Libraries:** Ensure you have `pandas` installed (`pip install pandas`).
    3. **Load Dataset:** Write a Python script to load the `sample_data.csv` into a Pandas DataFrame.
    4. **Inspect Data:** Use `df.head()`, `df.info()`, and `df.describe()` to get an overview of the dataset.
    5. **Check Missing Values:** Use `df.isnull().sum()` to identify any missing values.
- **Source Code:**

```
import pandas as pd

# Create a dummy CSV file for demonstration if it doesn't exist
# In a real scenario, you would have your file ready
try:
    with open('sample_data.csv', 'w') as f:
        f.write('Name,Age,Score\n')
        f.write('Alice,24,88\n')
        f.write('Bob,27,92\n')
        f.write('Charlie,22,78\n')
        f.write('David,29,95\n')
        f.write('Eve,25,85\n')
        f.write('Frank,30,NaN\n') # Example with a missing value
except IOError:
    print("Could not create sample_data.csv. Please ensure write permissions.")

# Load the dataset
try:
    df = pd.read_csv('sample_data.csv')
    print("Dataset loaded successfully!\n")
```

```
    # Display the first few rows
    print("--- Head of the Dataset ---")
    print(df.head())
    print("\n")

    # Display basic information about the dataset
    print("--- Dataset Info ---")
    df.info()
    print("\n")

    # Display descriptive statistics
    print("--- Descriptive Statistics ---")
    print(df.describe())
    print("\n")

    # Check for missing values
    print("--- Missing Values Count ---")
    print(df.isnull().sum())
    print("\n")

except FileNotFoundError:
    print("Error: 'sample_data.csv' not found. Please create the file or check the path.")
except Exception as e:
    print(f"An error occurred: {e}")
```

- **Input:** A CSV file named `sample_data.csv` in the same directory as the Python script, with content like:

```
Name,Age,Score
Alice,24,88
Bob,27,92
Charlie,22,78
David,29,95
Eve,25,85
Frank,30,NaN
```

- **Expected Output:**

```
Dataset loaded successfully!

--- Head of the Dataset ---
    Name  Age  Score
0    Alice   24   88.0
1      Bob   27   92.0
2  Charlie   22   78.0
3    David   29   95.0
4      Eve   25   85.0

--- Dataset Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    6 non-null      object
 1   Age     6 non-null      int64
```

- 2   Score   5 non-null   float64
- dtypes: float64(1), int64(1), object(1)
- memory usage: 272.0+ bytes
-
-
- --- Descriptive Statistics ---
- Age   Score
- count   6.000000   5.000000
- mean   26.166667 87.600000
- std   2.994439   6.587868
- min   22.000000 78.000000
- 25%   24.250000 85.000000
- 50%   26.000000 88.000000
- 75%   28.500000 92.000000
- max   30.000000 95.000000
-
- --- Missing Values Count ---
- Name   0
- Age   0
- Score   1
- dtype: int64

Lab 2: Create and upload dataset for data analytics

- **Title:** Creating and Preparing a Custom Dataset for Data Analytics
- **Aim:** To gain practical experience in manually creating a structured dataset and understanding how to prepare it for subsequent data analysis tasks.
- **Procedure:**
    1. **Choose Data:** Decide on a simple theme for your dataset (e.g., student grades, product sales).
    2. **Create File:** Open a plain text editor (like Notepad, Sublime Text, VS Code) or a spreadsheet program (like Excel, Google Sheets).
    3. **Define Columns:** Determine the column headers (e.g., `StudentID`, `Subject`, `Marks`).
    4. **Enter Data:** Populate the rows with relevant data, ensuring consistency. Use commas (`,`) to separate values for CSV format.
    5. **Save as CSV:** Save the file with a `.csv` extension (e.g., `my_students.csv`). If using a spreadsheet, use "Save As" and select CSV format.
    6. **Verify Data:** Write a Python script to read and display the contents of your newly created CSV file to ensure it's correctly formatted and readable.
- **Source Code:**

```
import pandas as pd
import os

# Define the filename for the dataset
file_name = 'my_students.csv'

# --- Manual creation of the dataset (simulated for demonstration) ---
# In a real lab, the user would manually create this file.
# This block is just to ensure the file exists for the Python script to run.
data_content = """StudentID,Subject,Marks
101,Math,85
102,Science,90
103,Math,78
104,English,92
105,Science,88
"""
try:
    with open(file_name, 'w') as f:
        f.write(data_content)
    print(f"'{file_name}' created successfully for demonstration.\n")
except IOError:
    print(f"Could not create '{file_name}'. Please check permissions.")

# --- Python script to "upload" (read) and display the dataset ---
if os.path.exists(file_name):
    try:
        df = pd.read_csv(file_name)
        print(f"Dataset '{file_name}' loaded successfully!\n")
        print("--- Content of the Dataset ---")
        print(df)
        print("\n")
        print("--- Dataset Info ---")
        df.info()
    except Exception as e:
        print(f"An error occurred while reading the dataset: {e}")
else:
```

- print(f"Error: '{file_name}' not found. Please ensure you have created it.")
-

- **Input:** A CSV file named `my_students.csv` created manually, for example:
- StudentID,Subject,Marks
- 101,Math,85
- 102,Science,90
- 103,Math,78
- 104,English,92
- 105,Science,88

- **Expected Output:**
- 'my_students.csv' created successfully for demonstration.
-
- Dataset 'my_students.csv' loaded successfully!
-
- --- Content of the Dataset ---
- StudentID Subject Marks
- 0   101   Math   85
- 1   102 Science   90
- 2   103   Math   78
- 3   104 English   92
- 4   105 Science   88
-
-
- --- Dataset Info ---
- <class 'pandas.core.frame.DataFrame'>
- RangeIndex: 5 entries, 0 to 4
- Data columns (total 3 columns):
- #   Column   Non-Null Count Dtype
- --- ------   -------------- -----
- 0   StudentID 5 non-null   int64
- 1   Subject   5 non-null   object
- 2   Marks   5 non-null   int64
- dtypes: int64(2), object(1)
- memory usage: 248.0+ bytes

Lab 3: Install Python IDE and perform basic python programs

- **Title:** Python IDE Setup and Basic Program Execution
- **Aim:** To familiarize students with the installation and basic usage of a Python Integrated Development Environment (IDE) and to execute simple Python scripts.
- **Procedure:**
    1. **Choose IDE:** Select a Python IDE (e.g., Visual Studio Code with Python extension, PyCharm Community Edition, Anaconda Navigator with Spyder).
    2. **Installation:** Follow the official installation instructions for your chosen IDE and Python.
    3. **First Program (Hello World):**
        - Open the IDE.
        - Create a new Python file (e.g., `hello.py`).
        - Type `print("Hello, World!")`.
        - Save and run the file using the IDE's run button or command.
    4. **Arithmetic Program:**
        - Create another Python file (e.g., `arithmetic.py`).
        - Write code to perform addition, subtraction, multiplication, and division of two numbers.
        - Save and run the file.
- **Source Code:**

```
# --- hello.py ---
print("Hello, World!")

# --- arithmetic.py ---
# Define two numbers
num1 = 15
num2 = 5

# Perform arithmetic operations
sum_result = num1 + num2
difference_result = num1 - num2
product_result = num1 * num2
quotient_result = num1 / num2
floor_division_result = num1 // num2
modulo_result = num1 % num2

# Print the results
print(f"Number 1: {num1}")
print(f"Number 2: {num2}")
print(f"Sum: {sum_result}")
print(f"Difference: {difference_result}")
print(f"Product: {product_result}")
print(f"Quotient (float division): {quotient_result}")
print(f"Floor Division: {floor_division_result}")
print(f"Modulo (Remainder): {modulo_result}")
```

- **Input:**
    1. For `hello.py`: No input required.
    2. For `arithmetic.py`: The numbers `num1 = 15` and `num2 = 5` are hardcoded in the script.
- **Expected Output:**
    1. For `hello.py`:
    2. `Hello, World!`

3. For `arithmetic.py`:

```
4. Number 1: 15
5. Number 2: 5
6. Sum: 20
7. Difference: 10
8. Product: 75
9. Quotient (float division): 3.0
10.   Floor Division: 3
11.   Modulo (Remainder): 0
```

Lab 4: Apply Python built-in data types : Strings , List , Tuples , Dictionary , Set and their methods to solve any given problem

- **Title:** Exploring Python's Built-in Data Types and Their Methods
- **Aim:** To understand and apply Python's fundamental built-in data types (Strings, Lists, Tuples, Dictionaries, Sets) and their common methods to solve practical problems.
- **Procedure:**
    1. **Strings:** Declare a string, perform operations like concatenation, slicing, and use methods like `upper()`, `lower()`, `split()`, `join()`.
    2. **Lists:** Create a list, add/remove elements, access elements by index, slice, and use methods like `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`.
    3. **Tuples:** Create a tuple, understand its immutability, and perform basic indexing and slicing.
    4. **Dictionaries:** Create a dictionary, add/access/modify key-value pairs, and use methods like `keys()`, `values()`, `items()`, `get()`.
    5. **Sets:** Create a set, add/remove elements, and perform set operations like union, intersection, difference.
- **Source Code:**

```
# --- Strings ---
print("--- Strings ---")
my_string = "Hello, Data Science!"
print(f"Original String: '{my_string}'")
print(f"Length: {len(my_string)}")
print(f"First character: {my_string[0]}")
print(f"Slice (6 to 10): '{my_string[6:11]}'")
print(f"Uppercase: '{my_string.upper()}'")
print(f"Lowercase: '{my_string.lower()}'")
words = my_string.split(", ")
print(f"Split by ', ': {words}")
new_string = "-".join(words)
print(f"Joined with '-': '{new_string}'")
print("-" * 20)

# --- Lists ---
print("--- Lists ---")
my_list = [10, 20, 30, 40, 50]
print(f"Original List: {my_list}")
my_list.append(60)
print(f"After append(60): {my_list}")
my_list.insert(0, 5)
print(f"After insert(0, 5): {my_list}")
my_list.remove(30)
print(f"After remove(30): {my_list}")
popped_item = my_list.pop()
print(f"After pop() (item: {popped_item}): {my_list}")
my_list.sort(reverse=True)
print(f"After sort(reverse=True): {my_list}")
print(f"Element at index 2: {my_list[2]}")
print(f"Slice (1 to 3): {my_list[1:4]}")
print("-" * 20)

# --- Tuples ---
print("--- Tuples ---")
my_tuple = (1, 2, "apple", "banana")
print(f"Original Tuple: {my_tuple}")
```

- `print(f"Element at index 2: {my_tuple[2]}")`
- `print(f"Slice (0 to 2): {my_tuple[0:3]}")`
- `# my_tuple[0] = 99 # This would cause an error (immutability)`
- `print("-" * 20)`
-
- `# --- Dictionaries ---`
- `print("--- Dictionaries ---")`
- `my_dict = {"name": "Alice", "age": 30, "city": "New York"}`
- `print(f"Original Dictionary: {my_dict}")`
- `print(f"Name: {my_dict['name']}")`
- `print(f"Age (using get()): {my_dict.get('age')}")`
- `my_dict["age"] = 31`
- `print(f"After updating age: {my_dict}")`
- `my_dict["occupation"] = "Engineer"`
- `print(f"After adding occupation: {my_dict}")`
- `print(f"Keys: {my_dict.keys()}")`
- `print(f"Values: {my_dict.values()}")`
- `print(f"Items: {my_dict.items()}")`
- `print("-" * 20)`
-
- `# --- Sets ---`
- `print("--- Sets ---")`
- `set1 = {1, 2, 3, 4, 5}`
- `set2 = {4, 5, 6, 7, 8}`
- `print(f"Set 1: {set1}")`
- `print(f"Set 2: {set2}")`
- `print(f"Union: {set1.union(set2)}")`
- `print(f"Intersection: {set1.intersection(set2)}")`
- `print(f"Difference (set1 - set2): {set1.difference(set2)}")`
- `set1.add(9)`
- `print(f"Set 1 after add(9): {set1}")`
- `set1.remove(1)`
- `print(f"Set 1 after remove(1): {set1}")`
- `print("-" * 20)`

- **Input:** No explicit input required; values are hardcoded in the script.
- **Expected Output:**
- `--- Strings ---`
- `Original String: 'Hello, Data Science!'`
- `Length: 20`
- `First character: H`
- `Slice (6 to 10): ' Data'`
- `Uppercase: 'HELLO, DATA SCIENCE!'`
- `Lowercase: 'hello, data science!'`
- `Split by ', ': ['Hello', 'Data Science!']`
- `Joined with '-': 'Hello-Data Science!'`
- `--------------------`
- `--- Lists ---`
- `Original List: [10, 20, 30, 40, 50]`
- `After append(60): [10, 20, 30, 40, 50, 60]`
- `After insert(0, 5): [5, 10, 20, 30, 40, 50, 60]`
- `After remove(30): [5, 10, 20, 40, 50, 60]`
- `After pop() (item: 60): [5, 10, 20, 40, 50]`
- `After sort(reverse=True): [50, 40, 20, 10, 5]`
- `Element at index 2: 20`
- `Slice (1 to 3): [40, 20, 10]`

- --------------------
- --- Tuples ---
- Original Tuple: (1, 2, 'apple', 'banana')
- Element at index 2: apple
- Slice (0 to 2): (1, 2, 'apple')
- --------------------
- --- Dictionaries ---
- Original Dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}
- Name: Alice
- Age (using get()): 30
- After updating age: {'name': 'Alice', 'age': 31, 'city': 'New York'}
- After adding occupation: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
- Keys: dict_keys(['name', 'age', 'city', 'occupation'])
- Values: dict_values(['Alice', 31, 'New York', 'Engineer'])
- Items: dict_items([('name', 'Alice'), ('age', 31), ('city', 'New York'), ('occupation', 'Engineer')])
- --------------------
- --- Sets ---
- Set 1: {1, 2, 3, 4, 5}
- Set 2: {4, 5, 6, 7, 8}
- Union: {1, 2, 3, 4, 5, 6, 7, 8}
- Intersection: {4, 5}
- Difference (set1 - set2): {1, 2, 3}
- Set 1 after add(9): {1, 2, 3, 4, 5, 9}
- Set 1 after remove(1): {2, 3, 4, 5, 9}
- --------------------

Lab 5: Solve problems using decision and looping statements

- **Title:** Problem Solving with Decision and Looping Statements
- **Aim:** To practice using conditional statements (`if`, `elif`, `else`) for decision-making and looping constructs (`for`, `while`) for repetitive tasks in Python.
- **Procedure:**
    1. **Decision Making (Even/Odd Check):** Write a program that takes an integer as input and determines if it's even or odd.
    2. **Looping (Sum of N Numbers):** Write a program that calculates the sum of the first `N` natural numbers using a `for` loop.
    3. **Looping (Factorial Calculation):** Write a program to calculate the factorial of a given number using a `while` loop.
    4. **Combined Logic (Grade Calculator):** Write a program that takes a score as input and assigns a grade (A, B, C, D, F) using `if-elif-else`.
- **Source Code:**

```
# --- Even/Odd Check ---
print("--- Even/Odd Check ---")
number_to_check = 7
if number_to_check % 2 == 0:
    print(f"{number_to_check} is an even number.")
else:
    print(f"{number_to_check} is an odd number.")
print("-" * 20)

# --- Sum of First N Natural Numbers (using for loop) ---
print("--- Sum of First N Natural Numbers ---")
n = 10
sum_n = 0
for i in range(1, n + 1):
    sum_n += i
print(f"The sum of the first {n} natural numbers is: {sum_n}")
print("-" * 20)

# --- Factorial Calculation (using while loop) ---
print("--- Factorial Calculation ---")
num_factorial = 5
factorial_result = 1
i = 1
while i <= num_factorial:
    factorial_result *= i
    i += 1
print(f"The factorial of {num_factorial} is: {factorial_result}")
print("-" * 20)

# --- Grade Calculator ---
print("--- Grade Calculator ---")
score = 85
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
```

- `    grade = 'F'`
- `print(f"With a score of {score}, the grade is: {grade}")`
- `print("-" * 20)`

- **Input:**
  1. `number_to_check = 7`
  2. `n = 10`
  3. `num_factorial = 5`
  4. `score = 85` (All inputs are hardcoded in the script for demonstration.)
- **Expected Output:**
- `--- Even/Odd Check ---`
- `7 is an odd number.`
- `--------------------`
- `--- Sum of First N Natural Numbers ---`
- `The sum of the first 10 natural numbers is: 55`
- `--------------------`
- `--- Factorial Calculation ---`
- `The factorial of 5 is: 120`
- `--------------------`
- `--- Grade Calculator ---`
- `With a score of 85, the grade is: B`
- `--------------------`

Lab 6: Apply all basic python OOP Concepts

- **Title:** Applying Basic Python Object-Oriented Programming (OOP) Concepts
- **Aim:** To understand and implement fundamental Object-Oriented Programming (OOP) concepts in Python, including classes, objects, attributes, methods, inheritance, and encapsulation.
- **Procedure:**
  1. **Class and Object:** Define a simple class `Car` with attributes like `make`, `model`, `year` and a method `display_info()`. Create an object of this class.
  2. **Constructor (`__init__`):** Use the `__init__` method to initialize object attributes.
  3. **Inheritance:** Create a `ElectricCar` class that inherits from `Car` and adds a new attribute `battery_size` and overrides/extends `display_info()`.
  4. **Encapsulation (Basic):** Demonstrate basic encapsulation using conventions (e.g., prefixing attributes with _ to suggest they are "protected").
- **Source Code:**

```python
# --- Class and Object, Constructor, Methods ---
class Car:
    """
    A simple class to represent a car.
    """
    def __init__(self, make, model, year):
        """
        Constructor to initialize Car object attributes.
        :param make: The brand of the car (e.g., "Toyota")
        :param model: The model of the car (e.g., "Camry")
        :param year: The manufacturing year of the car (e.g., 2020)
        """
        self.make = make
        self.model = model
        self.year = year
        self._mileage = 0 # Example of a "protected" attribute (encapsulation)

    def display_info(self):
        """
        Displays basic information about the car.
        """
        print(f"Car: {self.year} {self.make} {self.model}")
        print(f"Current Mileage: {self._mileage} miles")

    def drive(self, miles):
        """
        Simulates driving the car and updates mileage.
        :param miles: The number of miles driven.
        """
        if miles > 0:
            self._mileage += miles
            print(f"Drove {miles} miles. New mileage: {self._mileage}")
        else:
            print("Miles driven must be positive.")

# Create an object of the Car class
my_car = Car("Honda", "Civic", 2022)
print("--- Car Object ---")
my_car.display_info()
```

```
my_car.drive(150)
my_car.display_info()
print("-" * 20)

# --- Inheritance ---
class ElectricCar(Car):
    """
    A subclass representing an electric car, inheriting from Car.
    Adds battery_size and overrides display_info.
    """
    def __init__(self, make, model, year, battery_size_kwh):
        """
        Constructor for ElectricCar, calls parent constructor and adds battery_size.
        :param make: The brand of the car.
        :param model: The model of the car.
        :param year: The manufacturing year.
        :param battery_size_kwh: The battery capacity in kWh.
        """
        super().__init__(make, model, year) # Call the parent class constructor
        self.battery_size_kwh = battery_size_kwh

    def display_info(self):
        """
        Overrides the display_info method to include battery size.
        """
        super().display_info() # Call the parent's display_info
        print(f"Battery Size: {self.battery_size_kwh} kWh")
        print("This is an electric car.")

    def charge(self):
        """
        Simulates charging the electric car.
        """
        print(f"Charging the {self.make} {self.model}...")
        print("Charge complete!")

# Create an object of the ElectricCar class
my_electric_car = ElectricCar("Tesla", "Model 3", 2023, 75)
print("--- Electric Car Object (Inheritance) ---")
my_electric_car.display_info()
my_electric_car.drive(50)
my_electric_car.charge()
print("-" * 20)

# --- Basic Encapsulation Demonstration ---
# While Python doesn't have strict private members,
# prefixing with '_' is a convention for "protected" attributes.
# We can still access it, but it signals it's for internal use.
print("--- Encapsulation (Convention) ---")
print(f"Accessing protected mileage directly: {my_car._mileage}")
my_car._mileage = 1000 # We can change it, but it's discouraged
print(f"Mileage after direct modification: {my_car._mileage}")
print("-" * 20)
```

- **Input:** No explicit input required; objects are created and methods are called within the script.
- **Expected Output:**
- `--- Car Object ---`
- `Car: 2022 Honda Civic`
- `Current Mileage: 0 miles`
- `Drove 150 miles. New mileage: 150`
- `Car: 2022 Honda Civic`
- `Current Mileage: 150 miles`
- `--------------------`
- `--- Electric Car Object (Inheritance) ---`
- `Car: 2023 Tesla Model 3`
- `Current Mileage: 0 miles`
- `Battery Size: 75 kWh`
- `This is an electric car.`
- `Drove 50 miles. New mileage: 50`
- `Charging the Tesla Model 3...`
- `Charge complete!`
- `--------------------`
- `--- Encapsulation (Convention) ---`
- `Accessing protected mileage directly: 150`
- `Mileage after direct modification: 1000`
- `--------------------`

Lab 7: Manipulation of NumPy arrays- Indexing , Slicing , Reshaping , Joining and Splitting

- **Title:** NumPy Array Manipulation: Indexing, Slicing, Reshaping, Joining, and Splitting
- **Aim:** To master fundamental NumPy array manipulation techniques, including accessing elements (indexing), extracting sub-arrays (slicing), changing array dimensions (reshaping), combining arrays (joining), and dividing arrays (splitting).
- **Procedure:**
    1. **Installation:** Ensure `numpy` is installed (`pip install numpy`).
    2. **Array Creation:** Create 1D, 2D, and 3D NumPy arrays.
    3. **Indexing:** Access individual elements and rows/columns using integer and boolean indexing.
    4. **Slicing:** Extract sub-arrays using various slicing techniques.
    5. **Reshaping:** Change the dimensions of an array (e.g., from 1D to 2D, or 2D to 1D).
    6. **Joining:** Concatenate arrays along different axes (`np.concatenate`, `np.vstack`, `np.hstack`).
    7. **Splitting:** Divide an array into multiple smaller arrays (`np.split`, `np.vsplit`, `np.hsplit`).
- **Source Code:**

```
import numpy as np

# --- Array Creation ---
print("--- Array Creation ---")
arr_1d = np.array([1, 2, 3, 4, 5, 6])
print(f"1D Array: {arr_1d}")

arr_2d = np.array([[10, 11, 12],
                   [13, 14, 15],
                   [16, 17, 18]])
print(f"2D Array:\n{arr_2d}")
print("-" * 20)

# --- Indexing ---
print("--- Indexing ---")
print(f"Element at index 2 (1D): {arr_1d[2]}") # Output: 3
print(f"Element at [1, 2] (2D): {arr_2d[1, 2]}") # Output: 15
print(f"First row (2D): {arr_2d[0, :]}") # Output: [10 11 12]
print(f"Last column (2D): {arr_2d[:, -1]}") # Output: [12 15 18]
# Boolean indexing
print(f"Elements > 3 (1D): {arr_1d[arr_1d > 3]}") # Output: [4 5 6]
print("-" * 20)

# --- Slicing ---
print("--- Slicing ---")
print(f"Slice (index 1 to 4, 1D): {arr_1d[1:5]}") # Output: [2 3 4 5]
print(f"Slice (first two rows, all columns, 2D):\n{arr_2d[0:2, :]}")
print(f"Slice (all rows, last two columns, 2D):\n{arr_2d[:, 1:3]}")
print("-" * 20)

# --- Reshaping ---
print("--- Reshaping ---")
arr_reshaped = arr_1d.reshape(2, 3)
print(f"1D array reshaped to (2, 3):\n{arr_reshaped}")
arr_flattened = arr_reshaped.flatten()
```

- print(f"Reshaped array flattened back to 1D: {arr_flattened}")
- print("-" * 20)
-
- # --- Joining Arrays ---
- print("--- Joining Arrays ---")
- arr_a = np.array([[1, 2], [3, 4]])
- arr_b = np.array([[5, 6], [7, 8]])
- print(f"Array A:\n{arr_a}")
- print(f"Array B:\n{arr_b}")
-
- # Concatenate along axis 0 (rows)
- arr_concat_axis0 = np.concatenate((arr_a, arr_b), axis=0)
- print(f"Concatenated along axis 0:\n{arr_concat_axis0}")
-
- # Concatenate along axis 1 (columns)
- arr_concat_axis1 = np.concatenate((arr_a, arr_b), axis=1)
- print(f"Concatenated along axis 1:\n{arr_concat_axis1}")
-
- # Vertical stack
- arr_vstack = np.vstack((arr_a, arr_b))
- print(f"Vertically stacked:\n{arr_vstack}")
-
- # Horizontal stack
- arr_hstack = np.hstack((arr_a, arr_b))
- print(f"Horizontally stacked:\n{arr_hstack}")
- print("-" * 20)
-
- # --- Splitting Arrays ---
- print("--- Splitting Arrays ---")
- arr_to_split = np.array([10, 20, 30, 40, 50, 60, 70, 80])
- print(f"Array to split: {arr_to_split}")
- split_arrays_1d = np.split(arr_to_split, 4) # Split into 4 equal arrays
- print(f"Split into 4 equal parts (1D): {split_arrays_1d}")
-
- arr_2d_to_split = np.array([[1, 2, 3, 4],
-                             [5, 6, 7, 8],
-                             [9, 10, 11, 12],
-                             [13, 14, 15, 16]])
- print(f"2D Array to split:\n{arr_2d_to_split}")
-
- # Split horizontally (columns)
- hsplit_arrays = np.hsplit(arr_2d_to_split, 2) # Split into 2 equal parts horizontally
- print(f"Horizontally split into 2 parts:\n{hsplit_arrays[0]}\n{hsplit_arrays[1]}")
-
- # Split vertically (rows)
- vsplit_arrays = np.vsplit(arr_2d_to_split, 2) # Split into 2 equal parts vertically
- print(f"Vertically split into 2 parts:\n{vsplit_arrays[0]}\n{vsplit_arrays[1]}")
- print("-" * 20)

- **Input:** No explicit input required; arrays are created and manipulated within the script.
- **Expected Output:**
- --- Array Creation ---
- 1D Array: [1 2 3 4 5 6]

- 2D Array:
- [[10 11 12]
- [13 14 15]
- [16 17 18]]
- --------------------
- --- Indexing ---
- Element at index 2 (1D): 3
- Element at [1, 2] (2D): 15
- First row (2D): [10 11 12]
- Last column (2D): [12 15 18]
- Elements > 3 (1D): [4 5 6]
- --------------------
- --- Slicing ---
- Slice (index 1 to 4, 1D): [2 3 4 5]
- Slice (first two rows, all columns, 2D):
- [[10 11 12]
- [13 14 15]]
- Slice (all rows, last two columns, 2D):
- [[11 12]
- [14 15]
- [17 18]]
- --------------------
- --- Reshaping ---
- 1D array reshaped to (2, 3):
- [[1 2 3]
- [4 5 6]]
- Reshaped array flattened back to 1D: [1 2 3 4 5 6]
- --------------------
- --- Joining Arrays ---
- Array A:
- [[1 2]
- [3 4]]
- Array B:
- [[5 6]
- [7 8]]
- Concatenated along axis 0:
- [[1 2]
- [3 4]
- [5 6]
- [7 8]]
- Concatenated along axis 1:
- [[1 2 5 6]
- [3 4 7 8]]
- Vertically stacked:
- [[1 2]
- [3 4]
- [5 6]
- [7 8]]
- Horizontally stacked:
- [[1 2 5 6]
- [3 4 7 8]]
- --------------------
- --- Splitting Arrays ---
- Array to split: [10 20 30 40 50 60 70 80]
- Split into 4 equal parts (1D): [array([10, 20]), array([30, 40]), array([50, 60]), array([70, 80])]
- 2D Array to split:
- [[1 2 3 4]

- [5 6 7 8]
- [9 10 11 12]
- [13 14 15 16]]
- Horizontally split into 2 parts:
- [[1 2]
- [5 6]
- [9 10]
- [13 14]]
- [[3 4]
- [7 8]
- [11 12]
- [15 16]]
- Vertically split into 2 parts:
- [[1 2 3 4]
- [5 6 7 8]]
- [[9 10 11 12]
- [13 14 15 16]]
- --------------------

Lab 8: Perform array operations

- **Title:** Performing Basic Array Operations with NumPy
- **Aim:** To execute common mathematical and logical operations on NumPy arrays, including element-wise operations, aggregation functions, and broadcasting.
- **Procedure:**
    1. **Installation:** Ensure `numpy` is installed.
    2. **Element-wise Operations:** Perform addition, subtraction, multiplication, and division between arrays and between an array and a scalar.
    3. **Aggregation Functions:** Use `sum()`, `mean()`, `max()`, `min()`, `std()` on arrays, both for the entire array and along specific axes.
    4. **Broadcasting:** Demonstrate how NumPy automatically handles operations on arrays of different shapes under certain conditions.
    5. **Linear Algebra (Optional but good to include):** Perform dot product of two arrays.
- **Source Code:**

```
import numpy as np

# --- Element-wise Operations ---
print("--- Element-wise Operations ---")
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])
scalar = 10

print(f"Array 1: {arr1}")
print(f"Array 2: {arr2}")
print(f"Scalar: {scalar}")

print(f"Addition (arr1 + arr2): {arr1 + arr2}")
print(f"Subtraction (arr2 - arr1): {arr2 - arr1}")
print(f"Multiplication (arr1 * arr2): {arr1 * arr2}")
print(f"Division (arr2 / arr1): {arr2 / arr1}")
print(f"Array 1 + Scalar: {arr1 + scalar}")
print(f"Array 1 * Scalar: {arr1 * scalar}")
print("-" * 20)

# --- Aggregation Functions ---
print("--- Aggregation Functions ---")
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
print(f"Matrix:\n{matrix}")

print(f"Sum of all elements: {matrix.sum()}")
print(f"Mean of all elements: {matrix.mean()}")
print(f"Maximum element: {matrix.max()}")
print(f"Minimum element: {matrix.min()}")
print(f"Standard deviation: {matrix.std()}")

print(f"Sum along axis 0 (columns): {matrix.sum(axis=0)}") # Sum of each column
print(f"Sum along axis 1 (rows): {matrix.sum(axis=1)}")    # Sum of each row
print("-" * 20)

# --- Broadcasting ---
```

- `print("--- Broadcasting ---")`
- `matrix_b = np.array([[10, 20, 30],`
- `                     [40, 50, 60]])`
- `row_vector = np.array([1, 2, 3])`
-
- `print(f"Matrix B:\n{matrix_b}")`
- `print(f"Row Vector: {row_vector}")`
- `print(f"Matrix B + Row Vector (Broadcasting):\n{matrix_b + row_vector}")`
-
- `col_vector = np.array([[100], [200]])`
- `print(f"Column Vector:\n{col_vector}")`
- `print(f"Matrix B + Column Vector (Broadcasting):\n{matrix_b + col_vector}")`
- `print("-" * 20)`
-
- `# --- Linear Algebra: Dot Product ---`
- `print("--- Linear Algebra: Dot Product ---")`
- `matrix_c = np.array([[1, 2], [3, 4]])`
- `matrix_d = np.array([[5, 6], [7, 8]])`
- `print(f"Matrix C:\n{matrix_c}")`
- `print(f"Matrix D:\n{matrix_d}")`
- `print(f"Dot product (C @ D):\n{matrix_c @ matrix_d}")`
- `print(f"Dot product (np.dot(C, D)):\n{np.dot(matrix_c, matrix_d)}")`
- `print("-" * 20)`

- **Input:** No explicit input required; arrays are created and operations are performed within the script.
- **Expected Output:**
- `--- Element-wise Operations ---`
- `Array 1: [1 2 3 4]`
- `Array 2: [5 6 7 8]`
- `Scalar: 10`
- `Addition (arr1 + arr2): [ 6  8 10 12]`
- `Subtraction (arr2 - arr1): [4 4 4 4]`
- `Multiplication (arr1 * arr2): [ 5 12 21 32]`
- `Division (arr2 / arr1): [5.         3.         2.33333333 2.        ]`
- `Array 1 + Scalar: [11 12 13 14]`
- `Array 1 * Scalar: [10 20 30 40]`
- `--------------------`
- `--- Aggregation Functions ---`
- `Matrix:`
- `[[1 2 3]`
- ` [4 5 6]`
- ` [7 8 9]]`
- `Sum of all elements: 45`
- `Mean of all elements: 5.0`
- `Maximum element: 9`
- `Minimum element: 1`
- `Standard deviation: 2.581988897471611`
- `Sum along axis 0 (columns): [12 15 18]`
- `Sum along axis 1 (rows): [ 6 15 24]`
- `--------------------`
- `--- Broadcasting ---`
- `Matrix B:`
- `[[10 20 30]`
- ` [40 50 60]]`

- Row Vector: [1 2 3]
- Matrix B + Row Vector (Broadcasting):
- [[11 22 33]
- [41 52 63]]
- Column Vector:
- [[100]
- [200]]
- Matrix B + Column Vector (Broadcasting):
- [[110 120 130]
- [240 250 260]]
- --------------------
- --- Linear Algebra: Dot Product ---
- Matrix C:
- [[1 2]
- [3 4]]
- Matrix D:
- [[5 6]
- [7 8]]
- Dot product (C @ D):
- [[19 22]
- [43 50]]
- Dot product (np.dot(C, D)):
- [[19 22]
- [43 50]]
- --------------------

Lab 9: Implement Random Walks

- **Title:** Implementing Random Walks
- **Aim:** To understand and implement the concept of a random walk, a mathematical process that describes a path consisting of a succession of random steps.
- **Procedure:**
    1. **Installation:** Ensure `numpy` and `matplotlib` are installed (`pip install numpy matplotlib`).
    2. **Define Parameters:** Set the number of steps and the starting position.
    3. **Generate Steps:** Use NumPy's `random.choice` or `random.randint` to generate random steps (e.g., +1 or -1).
    4. **Calculate Positions:** Accumulate the steps to get the position at each time point.
    5. **Visualize:** Plot the random walk using `matplotlib` to observe its path.
- **Source Code:**

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

# --- Parameters for the Random Walk ---
num_steps = 1000 # Number of steps in the walk
start_position = 0 # Starting position

# --- Generate Random Steps ---
# Each step can be +1 (move right) or -1 (move left)
# np.random.choice([1, -1], size=num_steps) generates an array of 1s and -1s
steps = np.random.choice([1, -1], size=num_steps)

# --- Calculate Positions ---
# The position at each step is the cumulative sum of the steps
# np.cumsum() calculates the cumulative sum
positions = np.cumsum(steps)

# Add the starting position to all calculated positions
# This shifts the entire walk if start_position is not 0
positions = np.insert(positions, 0, start_position) # Add start_position at the beginning

# Create an array for the time points (steps)
time_points = np.arange(num_steps + 1)

# --- Visualize the Random Walk ---
plt.figure(figsize=(10, 6)) # Set figure size for better readability
plt.plot(time_points, positions, linestyle='-', color='blue', alpha=0.7)
plt.title('1D Random Walk')
plt.xlabel('Number of Steps')
plt.ylabel('Position')
plt.grid(True, linestyle='--', alpha=0.6)
plt.axhline(0, color='red', linestyle=':', linewidth=0.8, label='Starting Line') # Mark the starting line
plt.legend()
plt.show()

# --- Basic Analysis (Optional) ---
print(f"--- Random Walk Analysis ---")
print(f"Total steps: {num_steps}")
```

- `print(f"Final position: {positions[-1]}")`
- `print(f"Maximum position reached: {np.max(positions)}")`
- `print(f"Minimum position reached: {np.min(positions)}")`
- `print("-" * 20)`

- **Input:** No explicit input required; parameters are set within the script.
- **Expected Output:**
  1. A plot showing the 1D random walk over 1000 steps, starting at 0. The path will be random each time the script is run.
  2. Console output similar to:
  3. `--- Random Walk Analysis ---`
  4. `Total steps: 1000`
  5. `Final position: -24`
  6. `Maximum position reached: 32`
  7. `Minimum position reached: -32`
  8. `--------------------`

  (Note: The final position, max, and min will vary due to randomness.)

Lab 10: Perform operations on Data Frames using Python

- **Title:** Performing Operations on DataFrames using Python (Part 1)
- **Aim:** To gain proficiency in manipulating and analyzing data stored in Pandas DataFrames, focusing on basic operations like selection, filtering, and adding/modifying columns.
- **Procedure:**
  1. **Installation:** Ensure `pandas` is installed.
  2. **DataFrame Creation:** Create a DataFrame from a dictionary or list of lists.
  3. **Selection:** Select specific columns and rows using various methods (e.g., `df['column']`, `df[['col1', 'col2']]`, `df.loc[]`, `df.iloc[]`).
  4. **Filtering:** Filter rows based on conditions (e.g., `df[df['column'] > value]`).
  5. **Adding/Modifying Columns:** Create new columns or update existing ones.
  6. **Dropping Columns/Rows:** Remove unwanted columns or rows.
- **Source Code:**

```
import pandas as pd
import numpy as np

# --- DataFrame Creation ---
print("--- DataFrame Creation ---")
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Age': [24, 27, 22, 29, 25, 30],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',
'Phoenix', 'New York'],
    'Score': [88, 92, 78, 95, 85, 70]
}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)
print("-" * 20)

# --- Selection ---
print("--- Selection ---")
# Select a single column
print("\n'Name' column:\n", df['Name'])

# Select multiple columns
print("\n'Name' and 'Score' columns:\n", df[['Name', 'Score']])

# Select rows by label (using .loc)
print("\nRow with index 1 (using .loc):\n", df.loc[1])

# Select rows by integer position (using .iloc)
print("\nRow with index 0 (using .iloc):\n", df.iloc[0])

# Select specific cell
print(f"\nScore of Alice (iloc[0, 3]): {df.iloc[0, 3]}")
print("-" * 20)

# --- Filtering ---
print("--- Filtering ---")
# Filter rows where Age > 25
df_filtered_age = df[df['Age'] > 25]
print("\nDataFrame where Age > 25:\n", df_filtered_age)

# Filter rows where City is 'New York'
```

- df_filtered_city = df[df['City'] == 'New York']
- print("\nDataFrame where City is 'New York':\n", df_filtered_city)
-
- # Filter with multiple conditions (Age > 25 AND Score > 90)
- df_filtered_complex = df[(df['Age'] > 25) & (df['Score'] > 90)]
- print("\nDataFrame where Age > 25 AND Score > 90:\n", df_filtered_complex)
- print("-" * 20)
-
- # --- Adding/Modifying Columns ---
- print("--- Adding/Modifying Columns ---")
- # Add a new column 'Passed' based on 'Score'
- df['Passed'] = np.where(df['Score'] >= 75, 'Yes', 'No')
- print("\nDataFrame after adding 'Passed' column:\n", df)
-
- # Add a new column 'Age_in_Months'
- df['Age_in_Months'] = df['Age'] * 12
- print("\nDataFrame after adding 'Age_in_Months' column:\n", df)
-
- # Modify an existing column (e.g., increase all scores by 2)
- df['Score'] = df['Score'] + 2
- print("\nDataFrame after increasing 'Score' by 2:\n", df)
- print("-" * 20)
-
- # --- Dropping Columns/Rows ---
- print("--- Dropping Columns/Rows ---")
- # Drop a column
- df_no_city = df.drop(columns=['City'])
- print("\nDataFrame after dropping 'City' column:\n", df_no_city)
-
- # Drop multiple columns
- df_less_cols = df.drop(columns=['Passed', 'Age_in_Months'])
- print("\nDataFrame after dropping 'Passed' and 'Age_in_Months' columns:\n", df_less_cols)
-
- # Drop a row by index
- df_no_row_0 = df.drop(index=0)
- print("\nDataFrame after dropping row with index 0:\n", df_no_row_0)
- print("-" * 20)

- **Input:** No explicit input required; DataFrame is created and manipulated within the script.
- **Expected Output:**
- --- DataFrame Creation ---
- Original DataFrame:
-     Name Age   City Score
- 0   Alice   24   New York   88
- 1   Bob   27 Los Angeles   92
- 2 Charlie   22   Chicago   78
- 3   David   29   Houston   95
- 4   Eve   25   Phoenix   85
- 5   Frank   30   New York   70
- --------------------
- --- Selection ---
-
- 'Name' column:
- 0   Alice

- 1   Bob
- 2   Charlie
- 3   David
- 4   Eve
- 5   Frank
- Name: Name, dtype: object
-
- 'Name' and 'Score' columns:
-     Name Score
- 0   Alice   88
- 1   Bob   92
- 2 Charlie   78
- 3   David   95
- 4   Eve   85
- 5   Frank   70
-
- Row with index 1 (using .loc):
- Name   Bob
- Age   27
- City   Los Angeles
- Score   92
- Name: 1, dtype: object
-
- Row with index 0 (using .iloc):
- Name   Alice
- Age   24
- City   New York
- Score   88
- Name: 0, dtype: object
-
- Score of Alice (iloc[0, 3]): 88
- --------------------
- --- Filtering ---
-
- DataFrame where Age > 25:
-     Name Age   City Score
- 1   Bob   27 Los Angeles   92
- 3 David   29   Houston   95
- 5 Frank   30   New York   70
-
- DataFrame where City is 'New York':
-     Name Age   City Score
- 0   Alice   24 New York   88
- 5   Frank   30 New York   70
-
- DataFrame where Age > 25 AND Score > 90:
-     Name Age   City Score
- 1   Bob   27 Los Angeles   92
- 3 David   29   Houston   95
- --------------------
- --- Adding/Modifying Columns ---
-
- DataFrame after adding 'Passed' column:
-     Name Age   City Score Passed
- 0   Alice   24   New York   88   Yes
- 1   Bob   27 Los Angeles   92   Yes
- 2 Charlie   22   Chicago   78   Yes

- 3   David   29   Houston   95   Yes
- 4   Eve   25   Phoenix   85   Yes
- 5   Frank   30   New York   70   No
-
- DataFrame after adding 'Age_in_Months' column:
-     Name Age   City Score Passed Age_in_Months
- 0   Alice   24   New York   88   Yes   288
- 1   Bob   27 Los Angeles   92   Yes   324
- 2 Charlie   22   Chicago   78   Yes   264
- 3   David   29   Houston   95   Yes   348
- 4   Eve   25   Phoenix   85   Yes   300
- 5   Frank   30   New York   70   No   360
-
- DataFrame after increasing 'Score' by 2:
-     Name Age   City Score Passed Age_in_Months
- 0   Alice   24   New York   90   Yes   288
- 1   Bob   27 Los Angeles   94   Yes   324
- 2 Charlie   22   Chicago   80   Yes   264
- 3   David   29   Houston   97   Yes   348
- 4   Eve   25   Phoenix   87   Yes   300
- 5   Frank   30   New York   72   No   360
- --------------------
- --- Dropping Columns/Rows ---
-
- DataFrame after dropping 'City' column:
-     Name Age Score Passed Age_in_Months
- 0   Alice   24   90   Yes   288
- 1   Bob   27   94   Yes   324
- 2 Charlie   22   80   Yes   264
- 3   David   29   97   Yes   348
- 4   Eve   25   87   Yes   300
- 5   Frank   30   72   No   360
-
- DataFrame after dropping 'Passed' and 'Age_in_Months' columns:
-     Name Age   City Score
- 0   Alice   24   New York   90
- 1   Bob   27 Los Angeles   94
- 2 Charlie   22   Chicago   80
- 3   David   29   Houston   97
- 4   Eve   25   Phoenix   87
- 5   Frank   30   New York   72
-
- DataFrame after dropping row with index 0:
-     Name Age   City Score Passed Age_in_Months
- 1   Bob   27 Los Angeles   94   Yes   324
- 2 Charlie   22   Chicago   80   Yes   264
- 3   David   29   Houston   97   Yes   348
- 4   Eve   25   Phoenix   87   Yes   300
- 5   Frank   30   New York   72   No   360
- --------------------

Lab 11: Perform operations on Data Frames using Python

- **Title:** Performing Operations on DataFrames using Python (Part 2)
- **Aim:** To further enhance skills in DataFrame manipulation, including grouping data, aggregation, merging/joining DataFrames, and handling missing values.
- **Procedure:**
    1. **Installation:** Ensure `pandas` is installed.
    2. **Grouping and Aggregation:** Use `groupby()` to group data by one or more columns and apply aggregation functions (`sum`, `mean`, `count`, `max`, `min`).
    3. **Merging/Joining:** Combine two DataFrames based on a common column using `pd.merge()`.
    4. **Handling Missing Values:** Identify, count, and handle missing values (e.g., `dropna()`, `fillna()`).
    5. **Apply Function:** Apply a custom function to a column or row.
- **Source Code:**

```
import pandas as pd
import numpy as np

# --- Initial DataFrame for demonstration ---
data = {
    'Department': ['HR', 'IT', 'HR', 'IT', 'Finance', 'HR', 'IT'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank', 'Grace'],
    'Salary': [60000, 75000, 62000, 80000, 70000, 58000, 85000],
    'Experience': [5, 8, 4, 10, 7, 3, 9],
    'Bonus': [5000, np.nan, 4500, 7000, np.nan, 4000, 8000] # Introduce some NaN values
}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)
print("-" * 20)

# --- Grouping and Aggregation ---
print("--- Grouping and Aggregation ---")
# Group by 'Department' and calculate mean salary
avg_salary_by_dept = df.groupby('Department')['Salary'].mean()
print("\nAverage Salary by Department:\n", avg_salary_by_dept)

# Group by 'Department' and get multiple aggregations
dept_stats = df.groupby('Department').agg(
    Total_Employees=('Employee', 'count'),
    Average_Salary=('Salary', 'mean'),
    Max_Experience=('Experience', 'max')
)
print("\nDepartment Statistics:\n", dept_stats)
print("-" * 20)

# --- Merging/Joining DataFrames ---
print("--- Merging/Joining DataFrames ---")
# Create another DataFrame for department details
dept_info_data = {
    'Department': ['HR', 'IT', 'Finance', 'Marketing'],
    'Head': ['John Doe', 'Jane Smith', 'Peter Jones', 'Sarah Lee'],
    'Location': ['Building A', 'Building B', 'Building A', 'Building C']
}
```

```python
df_dept_info = pd.DataFrame(dept_info_data)
print("\nDepartment Info DataFrame:\n", df_dept_info)

# Merge df with df_dept_info on 'Department'
merged_df = pd.merge(df, df_dept_info, on='Department', how='left')
print("\nMerged DataFrame (left join):\n", merged_df)
print("-" * 20)

# --- Handling Missing Values ---
print("--- Handling Missing Values ---")
print("\nMissing values before handling:\n", df.isnull().sum())

# Drop rows with any missing values
df_dropped_na = df.dropna()
print("\nDataFrame after dropping rows with NaN:\n", df_dropped_na)

# Fill missing 'Bonus' values with the mean of the 'Bonus' column
df_filled_bonus = df.copy() # Make a copy to avoid modifying original df
mean_bonus = df_filled_bonus['Bonus'].mean()
df_filled_bonus['Bonus'].fillna(mean_bonus, inplace=True)
print(f"\nDataFrame after filling 'Bonus' NaN with mean ({mean_bonus:.2f}):\n", df_filled_bonus)
print("-" * 20)

# --- Apply Function ---
print("--- Apply Function ---")
# Define a function to categorize salary
def categorize_salary(salary):
    if salary >= 75000:
        return 'High'
    elif salary >= 60000:
        return 'Medium'
    else:
        return 'Low'

# Apply the function to the 'Salary' column to create a new 'Salary_Category' column
df['Salary_Category'] = df['Salary'].apply(categorize_salary)
print("\nDataFrame after adding 'Salary_Category' column:\n", df)
print("-" * 20)
```

- **Input:** No explicit input required; DataFrames are created and manipulated within the script.
- **Expected Output:**

```
Original DataFrame:
  Department Employee  Salary  Experience    Bonus
0         HR    Alice   60000           5   5000.0
1         IT      Bob   75000           8      NaN
2         HR  Charlie   62000           4   4500.0
3         IT    David   80000          10   7000.0
4    Finance      Eve   70000           7      NaN
5         HR    Frank   58000           3   4000.0
6         IT    Grace   85000           9   8000.0
--------------------
--- Grouping and Aggregation ---

```

- Average Salary by Department:
- Department
- Finance   70000.0
- HR   60000.0
- IT   80000.0
- Name: Salary, dtype: float64
-
- Department Statistics:
-             Total_Employees Average_Salary Max_Experience
- Department
- Finance   1   70000.0   7
- HR   3   60000.0   5
- IT   3   80000.0   10
- --------------------
- --- Merging/Joining DataFrames ---
-
- Department Info DataFrame:
-   Department   Head   Location
- 0   HR   John Doe Building A
- 1   IT   Jane Smith Building B
- 2   Finance Peter Jones Building A
- 3 Marketing   Sarah Lee Building C
-
- Merged DataFrame (left join):
-   Department Employee Salary Experience   Bonus   Head Location
- 0   HR   Alice   60000   5   5000.0   John Doe Building A
- 1   IT   Bob   75000   8   NaN   Jane Smith Building B
- 2   HR Charlie   62000   4   4500.0   John Doe Building A
- 3   IT   David   80000   10   7000.0   Jane Smith Building B
- 4   Finance   Eve   70000   7   NaN Peter Jones Building A
- 5   HR   Frank   58000   3   4000.0   John Doe Building A
- 6   IT   Grace   85000   9   8000.0   Jane Smith Building B
- --------------------
- --- Handling Missing Values ---
-
- Missing values before handling:
- Department   0
- Employee   0
- Salary   0
- Experience   0
- Bonus   2
- dtype: int64
-
- DataFrame after dropping rows with NaN:
-   Department Employee Salary Experience   Bonus
- 0   HR   Alice   60000   5 5000.0
- 2   HR Charlie   62000   4 4500.0
- 3   IT   David   80000   10 7000.0
- 5   HR   Frank   58000   3 4000.0
- 6   IT   Grace   85000   9 8000.0
-

- DataFrame after filling 'Bonus' NaN with mean (5900.00):
-   Department Employee Salary Experience   Bonus
- 0   HR   Alice   60000   5   5000.0
- 1   IT   Bob   75000   8   5900.0
- 2   HR Charlie   62000   4   4500.0
- 3   IT   David   80000   10   7000.0
- 4   Finance   Eve   70000   7   5900.0
- 5   HR   Frank   58000   3   4000.0
- 6   IT   Grace   85000   9   8000.0
- --------------------
- --- Apply Function ---
-
- DataFrame after adding 'Salary_Category' column:
-   Department Employee Salary Experience   Bonus Salary_Category
- 0   HR   Alice   60000   5   5000.0   Medium
- 1   IT   Bob   75000   8   NaN   High
- 2   HR Charlie   62000   4   4500.0   Medium
- 3   IT   David   80000   10   7000.0   High
- 4   Finance   Eve   70000   7   NaN   Medium
- 5   HR   Frank   58000   3   4000.0   Low
- 6   IT   Grace   85000   9   8000.0   High
- --------------------

Lab 12: Install , Import Pandas Learn and Explore a Sample Dataset with it

- **Title:** Installing, Importing, and Exploring a Sample Dataset with Pandas
- **Aim:** To guide students through the process of setting up the Pandas library, importing it into a Python environment, and performing initial exploratory data analysis on a sample dataset.
- **Procedure:**
  1. **Install Pandas:** Use `pip install pandas` in your terminal or command prompt.
  2. **Import Pandas:** Start your Python script by importing the pandas library, typically as `pd`.
  3. **Obtain Sample Dataset:** For this lab, we will use a small, built-in dataset or create a simple one to demonstrate. Alternatively, you can download a small CSV file (e.g., from Kaggle or UCI Machine Learning Repository).
  4. **Load Dataset:** Load the sample dataset into a Pandas DataFrame.
  5. **Initial Exploration:**
     - Display the first few rows (`.head()`).
     - Get a summary of the DataFrame (`.info()`).
     - View descriptive statistics (`.describe()`).
     - Check for unique values in categorical columns (`.unique()`, `.value_counts()`).
     - Check for missing values (`.isnull().sum()`).
- **Source Code:**

```
import pandas as pd
import numpy as np # Used for creating NaN values in dummy data

# --- Step 1 & 2: Installation (done via pip) and Importing Pandas ---
# Installation: Open your terminal/command prompt and run: pip install pandas
# Importing: Already done at the top of this script.

print("Pandas library imported successfully as 'pd'.\n")

# --- Step 3: Obtain Sample Dataset (Creating a dummy one for demonstration) ---
# In a real scenario, you might download a CSV like 'titanic.csv' or 'iris.csv'
# For this lab, we'll create a simple dummy dataset.
data = {
    'ProductID': [1, 2, 3, 4, 5, 6, 7, 8],
    'Category': ['Electronics', 'Books', 'Electronics', 'Home', 'Books', 'Electronics', 'Books', 'Home'],
    'Price': [1200.00, 25.50, 800.00, 150.75, 30.00, 1500.00, 18.25, 90.00],
    'Stock': [50, 120, 30, 80, 200, 40, 150, 60],
    'Rating': [4.5, 3.8, 4.2, np.nan, 4.0, 4.7, 3.5, 4.1] # Introducing a missing value
}
df_sample = pd.DataFrame(data)
print("Dummy sample dataset created successfully.\n")

# --- Step 4: Load Dataset (Already loaded as df_sample) ---
print("--- Initial Exploration of the Sample Dataset ---")

# --- Step 5: Initial Exploration ---
print("\n1. Display the first 5 rows (df_sample.head()):")
```

- print(df_sample.head())
-
- print("\n2. Get a concise summary of the DataFrame (df_sample.info()):")
- df_sample.info()
-
- print("\n3. View descriptive statistics (df_sample.describe()):")
- print(df_sample.describe())
-
- print("\n4. Check unique values and their counts in 'Category' column:")
- print("Unique Categories:", df_sample['Category'].unique())
- print("Value Counts for Categories:\n", df_sample['Category'].value_counts())
-
- print("\n5. Check for missing values (df_sample.isnull().sum()):")
- print(df_sample.isnull().sum())
-
- print("\n6. Check data types of columns (df_sample.dtypes):")
- print(df_sample.dtypes)

- **Input:** No explicit input required; a dummy DataFrame is created within the script.
- **Expected Output:**
- Pandas library imported successfully as 'pd'.
-
- Dummy sample dataset created successfully.
-
- --- Initial Exploration of the Sample Dataset ---
-
- 1. Display the first 5 rows (df_sample.head()):
-    ProductID   Category   Price Stock Rating
- 0   1 Electronics 1200.00   50   4.5
- 1   2   Books   25.50   120   3.8
- 2   3 Electronics   800.00   30   4.2
- 3   4   Home   150.75   80   NaN
- 4   5   Books   30.00   200   4.0
-
- 2. Get a concise summary of the DataFrame (df_sample.info()):
- <class 'pandas.core.frame.DataFrame'>
- RangeIndex: 8 entries, 0 to 7
- Data columns (total 5 columns):
- #   Column   Non-Null Count Dtype
- --- ------   -------------- -----
- 0   ProductID 8 non-null   int64
- 1   Category   8 non-null   object
- 2   Price   8 non-null   float64
- 3   Stock   8 non-null   int64
- 4   Rating   7 non-null   float64
- dtypes: float64(2), int64(2), object(1)
- memory usage: 448.0+ bytes
-
- 3. View descriptive statistics (df_sample.describe()):
-    ProductID   Price   Stock   Rating
- count   8.000000   8.000000   8.000000 7.000000
- mean   4.500000   479.312500   91.250000 4.114286
- std   2.449490   580.491295   59.816654 0.403565
- min   1.000000   18.250000   30.000000 3.500000

- 25%   2.750000   28.875000   47.500000 3.900000
- 50%   4.500000   120.375000   70.000000 4.100000
- 75%   6.250000   900.000000 132.500000 4.350000
- max   8.000000 1500.000000 200.000000 4.700000
-
- 4. Check unique values and their counts in 'Category' column:
- Unique Categories: ['Electronics' 'Books' 'Home']
- Value Counts for Categories:
- Category
- Electronics   3
- Books   3
- Home   2
- Name: count, dtype: int64
-
- 5. Check for missing values (df_sample.isnull().sum()):
- ProductID   0
- Category   0
- Price   0
- Stock   0
- Rating   1
- dtype: int64
-
- 6. Check data types of columns (df_sample.dtypes):
- ProductID   int64
- Category   object
- Price   float64
- Stock   int64
- Rating   float64
- dtype: object

Lab 13: Perform data transformations using python

- **Title:** Performing Data Transformations using Python
- **Aim:** To learn and apply various data transformation techniques in Python using Pandas, including data type conversion, feature scaling, encoding categorical variables, and creating new features.
- **Procedure:**
    1. **Installation:** Ensure `pandas` and `scikit-learn` are installed (`pip install pandas scikit-learn`).
    2. **Data Type Conversion:** Convert columns to appropriate data types (e.g., `object` to `category`, `float` to `int`).
    3. **Categorical Encoding:** Convert categorical (text) data into numerical format using techniques like One-Hot Encoding or Label Encoding.
    4. **Feature Scaling:** Apply scaling techniques (e.g., Min-Max Scaling, Standardization) to numerical features.
    5. **Creating New Features:** Derive new features from existing ones (e.g., `Total_Sales` from `Price` and `Quantity`).
    6. **Discretization/Binning:** Convert continuous numerical data into discrete bins.
- **Source Code:**

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler, LabelEncoder, OneHotEncoder

# --- Create a sample DataFrame for transformations ---
data = {
    'CustomerID': [1, 2, 3, 4, 5, 6, 7],
    'Age': [25, 30, 22, 35, 28, 40, 32],
    'Gender': ['Male', 'Female', 'Male', 'Female', 'Male', 'Male', 'Female'],
    'Income': [50000, 75000, 45000, 90000, 60000, 120000, 80000],
    'Product_Category': ['Electronics', 'Books', 'Electronics', 'Home', 'Books', 'Electronics', 'Home'],
    'Quantity': [2, 1, 3, 1, 2, 4, 1],
    'Price_Per_Unit': [250, 30, 200, 100, 25, 300, 90]
}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)
print("-" * 20)

# --- 1. Data Type Conversion ---
print("--- Data Type Conversion ---")
# Convert 'Gender' and 'Product_Category' to 'category' dtype for memory efficiency
df['Gender'] = df['Gender'].astype('category')
df['Product_Category'] = df['Product_Category'].astype('category')
print("\nDataFrame dtypes after category conversion:\n", df.dtypes)
print("-" * 20)

# --- 2. Categorical Encoding ---
print("--- Categorical Encoding ---")
# Label Encoding for 'Gender'
le = LabelEncoder()
df['Gender_Encoded'] = le.fit_transform(df['Gender'])
print("\nDataFrame after Label Encoding 'Gender':\n", df[['Gender', 'Gender_Encoded']])
```

-
- # One-Hot Encoding for 'Product_Category'
- # Create an OneHotEncoder object
- ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
- # Fit and transform the 'Product_Category' column
- ohe_features = ohe.fit_transform(df[['Product_Category']])
- # Create a DataFrame from the one-hot encoded features
- ohe_df = pd.DataFrame(ohe_features, columns=ohe.get_feature_names_out(['Product_Category']), index=df.index)
- # Concatenate the new one-hot encoded columns with the original DataFrame
- df = pd.concat([df, ohe_df], axis=1)
- print("\nDataFrame after One-Hot Encoding 'Product_Category':\n", df[['Product_Category', 'Product_Category_Books', 'Product_Category_Electronics', 'Product_Category_Home']].head())
- print("-" * 20)
-
- # --- 3. Feature Scaling ---
- print("--- Feature Scaling ---")
- # Min-Max Scaling for 'Income'
- min_max_scaler = MinMaxScaler()
- df['Income_MinMaxScaled'] = min_max_scaler.fit_transform(df[['Income']])
- print("\nDataFrame after Min-Max Scaling 'Income':\n", df[['Income', 'Income_MinMaxScaled']])
-
- # Standardization (Z-score scaling) for 'Age'
- standard_scaler = StandardScaler()
- df['Age_StandardScaled'] = standard_scaler.fit_transform(df[['Age']])
- print("\nDataFrame after Standardization 'Age':\n", df[['Age', 'Age_StandardScaled']])
- print("-" * 20)
-
- # --- 4. Creating New Features ---
- print("--- Creating New Features ---")
- # Create 'Total_Sales' from 'Quantity' and 'Price_Per_Unit'
- df['Total_Sales'] = df['Quantity'] * df['Price_Per_Unit']
- print("\nDataFrame after creating 'Total_Sales' column:\n", df[['Quantity', 'Price_Per_Unit', 'Total_Sales']])
- print("-" * 20)
-
- # --- 5. Discretization/Binning ---
- print("--- Discretization/Binning ---")
- # Bin 'Age' into categories
- df['Age_Group'] = pd.cut(df['Age'], bins=[20, 30, 40, 50], labels=['20-30', '30-40', '40-50'], right=False)
- print("\nDataFrame after binning 'Age' into 'Age_Group':\n", df[['Age', 'Age_Group']])
- print("-" * 20)
-
- print("\nFinal DataFrame after various transformations (showing relevant columns):\n", df[['CustomerID', 'Age', 'Age_Group', 'Gender', 'Gender_Encoded', 'Income', 'Income_MinMaxScaled', 'Product_Category', 'Product_Category_Books', 'Product_Category_Electronics', 'Product_Category_Home', 'Total_Sales']].head(7))

- **Input:** No explicit input required; a dummy DataFrame is created and transformed within the script.
- **Expected Output:**
- Original DataFrame:
-    CustomerID Age Gender Income Product_Category Quantity Price_Per_Unit
- 0   1   25   Male   50000   Electronics   2 250
- 1   2   30 Female   75000   Books   1 30
- 2   3   22   Male   45000   Electronics   3 200
- 3   4   35 Female   90000   Home   1 100
- 4   5   28   Male   60000   Books   2 25
- 5   6   40   Male 120000   Electronics   4 300
- 6   7   32 Female   80000   Home   1 90
- --------------------
- --- Data Type Conversion ---
-
- DataFrame dtypes after category conversion:
- CustomerID   int64
- Age   int64
- Gender   category
- Income   int64
- Product_Category   category
- Quantity   int64
- Price_Per_Unit   int64
- dtype: object
- --------------------
- --- Categorical Encoding ---
-
- DataFrame after Label Encoding 'Gender':
-    Gender Gender_Encoded
- 0   Male   1
- 1 Female   0
- 2   Male   1
- 3 Female   0
- 4   Male   1
- 5   Male   1
- 6 Female   0
-
- DataFrame after One-Hot Encoding 'Product_Category':
-   Product_Category Product_Category_Books Product_Category_Electronics Product_Category_Home
- 0   Electronics   0.0   1.0 0.0
- 1   Books   1.0   0.0 0.0
- 2   Electronics   0.0   1.0 0.0
- 3   Home   0.0   0.0 1.0
- 4   Books   1.0   0.0 0.0
- --------------------

- --- Feature Scaling ---
-
- DataFrame after Min-Max Scaling 'Income':
-    Income Income_MinMaxScaled
- 0   50000   0.058824
- 1   75000   0.470588
- 2   45000   0.000000
- 3   90000   0.764706
- 4   60000   0.235294
- 5 120000   1.000000
- 6   80000   0.617647
-
- DataFrame after Standardization 'Age':
-    Age Age_StandardScaled
- 0   25   -0.801784
- 1   30   0.133631
- 2   22   -1.369527
- 3   35   1.069050
- 4   28   -0.234112
- 5   40   2.004469
- 6   32   0.417573
- --------------------
- --- Creating New Features ---
-
- DataFrame after creating 'Total_Sales' column:
-    Quantity Price_Per_Unit Total_Sales
- 0   2   250   500
- 1   1   30   30
- 2   3   200   600
- 3   1   100   100
- 4   2   25   50
- 5   4   300   1200
- 6   1   90   90
- --------------------
- --- Discretization/Binning ---
-
- DataFrame after binning 'Age' into 'Age_Group':
-    Age Age_Group
- 0   25   20-30
- 1   30   30-40
- 2   22   20-30
- 3   35   30-40
- 4   28   20-30
- 5   40   40-50
- 6   32   30-40
- --------------------
-
- Final DataFrame after various transformations (showing relevant columns):
-    CustomerID Age Age_Group Gender Gender_Encoded Income Income_MinMaxScaled Product_Category Product_Category_Books Product_Category_Electronics Product_Category_Home Total_Sales
- 0   1   25   20-30   Male   1   50000 0.058824   Electronics   0.0 1.0   0.0   500
- 1   2   30   30-40 Female   0   75000 0.470588   Books   1.0 0.0   0.0   30

- 2 3 22 20-30 Male 1 45000
  0.000000 Electronics 0.0
  1.0 0.0 600
- 3 4 35 30-40 Female 0 90000
  0.764706 Home 0.0
  0.0 1.0 100
- 4 5 28 20-30 Male 1 60000
  0.235294 Books 1.0
  0.0 0.0 50
- 5 6 40 40-50 Male 1 120000
  1.000000 Electronics 0.0
  1.0 0.0 1200
- 6 7 32 30-40 Female 0 80000
  0.617647 Home 0.0
  0.0 1.0 90

Lab 14: Install , Import Matplotlib . Explore all the Data Visualization Graphs

- **Title:** Installing, Importing Matplotlib, and Exploring Data Visualization Graphs
- **Aim:** To introduce students to the Matplotlib library for data visualization in Python and to demonstrate how to create various types of plots commonly used in data science.
- **Procedure:**
    1. **Installation:** Ensure `matplotlib` and `pandas` (for data handling) are installed (`pip install matplotlib pandas`).
    2. **Import Matplotlib:** Import the `matplotlib.pyplot` module, typically as `plt`.
    3. **Prepare Data:** Create or load sample data suitable for different plot types.
    4. **Create Plots:**
        - Line Plot
        - Scatter Plot
        - Bar Chart
        - Histogram
        - Pie Chart
        - Box Plot
        - Subplots (combining multiple plots)
    5. **Customize Plots:** Add titles, labels, legends, and adjust colors/styles.
    6. **Display Plot:** Use `plt.show()` to display the generated plots.
- **Source Code:**

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

print("Matplotlib and NumPy/Pandas imported successfully.\n")

# --- Sample Data for Visualization ---
# Line Plot Data
x_line = np.linspace(0, 10, 100)
y_line = np.sin(x_line)

# Scatter Plot Data
np.random.seed(42) # for reproducibility
x_scatter = np.random.rand(50) * 10
y_scatter = np.random.rand(50) * 10
colors = np.random.rand(50)
sizes = np.random.rand(50) * 100 + 50 # Random sizes for points

# Bar Chart Data
categories = ['A', 'B', 'C', 'D', 'E']
values = [23, 45, 56, 12, 39]

# Histogram Data (random normal distribution)
data_hist = np.random.randn(1000) * 10 + 50 # Mean 50, Std Dev 10

# Pie Chart Data
labels_pie = ['Apples', 'Bananas', 'Cherries', 'Dates']
sizes_pie = [15, 30, 45, 10]
explode_pie = (0, 0.1, 0, 0) # Explode the 2nd slice (Bananas)

# Box Plot Data
data_box = [np.random.normal(0, std, 100) for std in range(1, 4)]
# This creates 3 datasets with different standard deviations
```

```
# --- 1. Line Plot ---
plt.figure(figsize=(8, 5)) # Set figure size
plt.plot(x_line, y_line, color='blue', linestyle='-', linewidth=2, label='sin(x)')
plt.title('Line Plot of Sine Wave')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.show()

# --- 2. Scatter Plot ---
plt.figure(figsize=(8, 5))
plt.scatter(x_scatter, y_scatter, c=colors, s=sizes, alpha=0.7, cmap='viridis')
plt.title('Scatter Plot with Color and Size Variation')
plt.xlabel('Feature X')
plt.ylabel('Feature Y')
plt.colorbar(label='Color Intensity')
plt.grid(True, linestyle=':', alpha=0.5)
plt.show()

# --- 3. Bar Chart ---
plt.figure(figsize=(8, 5))
plt.bar(categories, values, color=['skyblue', 'lightcoral', 'lightgreen', 'gold', 'plum'])
plt.title('Bar Chart of Category Values')
plt.xlabel('Category')
plt.ylabel('Value')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# --- 4. Histogram ---
plt.figure(figsize=(8, 5))
plt.hist(data_hist, bins=30, color='teal', edgecolor='black', alpha=0.7)
plt.title('Histogram of Sample Data')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# --- 5. Pie Chart ---
plt.figure(figsize=(7, 7))
plt.pie(sizes_pie, explode=explode_pie, labels=labels_pie, autopct='%1.1f%%',
        shadow=True, startangle=90, colors=['#ff9999','#66b3ff','#99ff99','#ffcc99'])
plt.title('Distribution of Fruits')
plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()

# --- 6. Box Plot ---
plt.figure(figsize=(8, 5))
plt.boxplot(data_box, patch_artist=True,
            boxprops=dict(facecolor='lightblue', color='blue'),
```

```
            medianprops=dict(color='red'))
plt.title('Box Plot of Multiple Distributions')
plt.xlabel('Distribution')
plt.ylabel('Value')
plt.xticks([1, 2, 3], ['Dataset 1', 'Dataset 2', 'Dataset 3'])
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

# --- 7. Subplots (Example: Line and Scatter in one figure) ---
fig, axes = plt.subplots(1, 2, figsize=(12, 5)) # 1 row, 2 columns

# Plot 1: Line Plot
axes[0].plot(x_line, y_line, color='purple', linestyle='-',
label='sin(x)')
axes[0].set_title('Line Plot')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Y')
axes[0].grid(True, linestyle='--', alpha=0.7)
axes[0].legend()

# Plot 2: Scatter Plot
axes[1].scatter(x_scatter, y_scatter, color='orange', alpha=0.8)
axes[1].set_title('Scatter Plot')
axes[1].set_xlabel('Feature X')
axes[1].set_ylabel('Feature Y')
axes[1].grid(True, linestyle=':', alpha=0.5)

plt.tight_layout() # Adjust layout to prevent overlapping titles/labels
plt.suptitle('Combined Plots: Line and Scatter', y=1.02, fontsize=16) #
Super title for the figure
plt.show()

print("\nAll requested plots have been generated and displayed.")
```

- **Input:** No explicit input required; data for plots is generated within the script.
- **Expected Output:** The execution will generate and display a series of separate plots (Line, Scatter, Bar, Histogram, Pie, Box, and a combined subplot figure), each appearing in its own window or inline if using an environment like Jupyter Notebook. The console will show:

```
Matplotlib and NumPy/Pandas imported successfully.

All requested plots have been generated and displayed.
```

(The actual plots are visual and cannot be represented in text, but the code will produce them.)

Lab 15: Install , Import Scikit Learn and Explore Iris Dataset with Pandas for ML Modelling

- **Title:** Installing, Importing Scikit-learn, and Exploring Iris Dataset for ML Modeling
- **Aim:** To introduce students to the Scikit-learn library, a powerful tool for machine learning in Python, and to perform initial data exploration and preparation on the famous Iris dataset for potential machine learning modeling.
- **Procedure:**
  1. **Installation:** Ensure `scikit-learn`, `pandas`, and `matplotlib` are installed (`pip install scikit-learn pandas matplotlib`).
  2. **Import Libraries:** Import necessary modules from `sklearn.datasets`, `pandas`, and `matplotlib.pyplot`.
  3. **Load Iris Dataset:** Load the Iris dataset using `load_iris()` from `sklearn.datasets`.
  4. **Convert to DataFrame:** Convert the Iris dataset (which is initially a Bunch object) into a Pandas DataFrame for easier manipulation.
  5. **Initial Data Exploration:**
     - Display head, info, describe.
     - Check class distribution.
     - Visualize relationships between features (e.g., using scatter plots).
  6. **Data Preparation (Basic):**
     - Separate features (X) and target (y).
     - (Optional) Check for missing values (Iris is clean, but good practice).
- **Source Code:**

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import seaborn as sns # Often used with matplotlib for nicer plots

print("Scikit-learn, Pandas, Matplotlib, and Seaborn imported successfully.\n")

# --- 1. Load Iris Dataset ---
# The Iris dataset is a classic and is included in scikit-learn
iris = load_iris()
print("Iris dataset loaded successfully.")
print(f"Keys in Iris dataset: {iris.keys()}\n")
print(f"Description of Iris dataset:\n{iris.DESCR[:500]}...\n") # Print first 500 chars

# --- 2. Convert to DataFrame ---
# Create a DataFrame from the data and feature names
df_iris = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Add the target variable (species) to the DataFrame
# The target is numerical (0, 1, 2), so we map it to actual species names
df_iris['species'] = iris.target
df_iris['species_name'] = df_iris['species'].map({0: 'setosa', 1: 'versicolor', 2: 'virginica'})

print("Iris dataset converted to Pandas DataFrame.\n")

# --- 3. Initial Data Exploration ---
print("--- Initial Data Exploration ---")
```

-
- `print("\n1. Display the first 5 rows (df_iris.head()):")`
- `print(df_iris.head())`
-
- `print("\n2. Get a concise summary of the DataFrame (df_iris.info()):")`
- `df_iris.info()`
-
- `print("\n3. View descriptive statistics (df_iris.describe()):")`
- `print(df_iris.describe())`
-
- `print("\n4. Check class distribution of 'species_name':")`
- `print(df_iris['species_name'].value_counts())`
-
- `print("\n5. Check for missing values (df_iris.isnull().sum()):")`
- `print(df_iris.isnull().sum()) # Iris dataset is clean, so all should be 0`
-
- `# --- 6. Visualize Relationships (Pair Plot) ---`
- `print("\n--- Visualizing Relationships (Pair Plot) ---")`
- `# Using Seaborn's pairplot to visualize relationships between features`
- `# and distributions, colored by species.`
- `sns.pairplot(df_iris, hue='species_name', palette='viridis')`
- `plt.suptitle('Pair Plot of Iris Dataset Features by Species', y=1.02) # Add a main title`
- `plt.show()`
-
- `# --- 7. Data Preparation (Basic) ---`
- `print("\n--- Data Preparation (Basic) ---")`
- `# Separate features (X) and target (y)`
- `X = df_iris[iris.feature_names] # Features are the original column names`
- `y = df_iris['species'] # Target is the numerical species column`
-
- `print(f"\nFeatures (X) shape: {X.shape}")`
- `print("First 5 rows of X:\n", X.head())`
-
- `print(f"\nTarget (y) shape: {y.shape}")`
- `print("First 5 values of y:\n", y.head())`
-
- `print("\nIris dataset exploration and basic preparation complete. Ready for ML modeling.")`

- **Input:** No explicit input required; the Iris dataset is loaded from `sklearn.datasets`.
- **Expected Output:**
    1. Console output detailing the dataset's keys, a partial description, DataFrame conversion confirmation, and various statistical summaries (head, info, describe, value counts, null sums).
    2. A pair plot visualization (generated by `seaborn.pairplot`) showing scatter plots for all pairs of features and histograms/KDE plots for individual features, with points colored by species. This plot will appear in a separate window or inline.
- `Scikit-learn, Pandas, Matplotlib, and Seaborn imported successfully.`
-
- `Iris dataset loaded successfully.`
- `Keys in Iris dataset: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename', 'data_module'])`
-
- `Description of Iris dataset:`

- .. _iris_dataset:
-
- Iris plants dataset
- --------------------
-
- **Data Set Characteristics:**
-
- :Number of Instances: 150 (50 in each of three classes)
- :Number of Attributes: 4 numeric, predictive attributes and the class
- :Attribute Information:
- - sepal length in cm
- - sepal width in cm
- - petal length in cm
- - petal width in cm
- - class:
- - Iris-Setosa
- - Iris-Versicol...
-
- Iris dataset converted to Pandas DataFrame.
-
- --- Initial Data Exploration ---
-
- 1. Display the first 5 rows (df_iris.head()):
- sepal length (cm) sepal width (cm) petal length (cm) petal width (cm) species species_name
- 0   5.1   3.5   1.4   0.2   0   setosa
- 1   4.9   3.0   1.4   0.2   0   setosa
- 2   4.7   3.2   1.3   0.2   0   setosa
- 3   4.6   3.1   1.5   0.2   0   setosa
- 4   5.0   3.6   1.4   0.2   0   setosa
-
- 2. Get a concise summary of the DataFrame (df_iris.info()):
- <class 'pandas.core.frame.DataFrame'>
- RangeIndex: 150 entries, 0 to 149
- Data columns (total 6 columns):
- #   Column   Non-Null Count Dtype
- --- ------   -------------- -----
- 0   sepal length (cm) 150 non-null   float64
- 1   sepal width (cm)   150 non-null   float64
- 2   petal length (cm) 150 non-null   float64
- 3   petal width (cm)   150 non-null   float64
- 4   species   150 non-null   int64
- 5   species_name   150 non-null   object
- dtypes: float64(4), int64(1), object(1)
- memory usage: 7.2+ KB
-
- 3. View descriptive statistics (df_iris.describe()):
- sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)   species
- count   150.000000   150.000000   150.000000   150.000000 150.000000
- mean   5.843333   3.057333   3.758000   1.199333   1.000000

- std 0.828066 0.435866 1.765298 0.762238 0.819232
- min 4.300000 2.000000 1.000000 0.100000 0.000000
- 25% 5.100000 2.800000 1.600000 0.300000 0.000000
- 50% 5.800000 3.000000 4.350000 1.300000 1.000000
- 75% 6.400000 3.300000 5.100000 1.800000 2.000000
- max 7.900000 4.400000 6.900000 2.500000 2.000000
-
- 4. Check class distribution of 'species_name':
- species_name
- setosa 50
- versicolor 50
- virginica 50
- Name: count, dtype: int64
-
- 5. Check for missing values (df_iris.isnull().sum()):
- sepal length (cm) 0
- sepal width (cm) 0
- petal length (cm) 0
- petal width (cm) 0
- species 0
- species_name 0
- dtype: int64
-
- --- Visualizing Relationships (Pair Plot) ---
- (A graphical plot will be displayed here)
-
- --- Data Preparation (Basic) ---
-
- Features (X) shape: (150, 4)
- First 5 rows of X:
- sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)
- 0 5.1 3.5 1.4 0.2
- 1 4.9 3.0 1.4 0.2
- 2 4.7 3.2 1.3 0.2
- 3 4.6 3.1 1.5 0.2
- 4 5.0 3.6 1.4 0.2
-
- Target (y) shape: (150,)
- First 5 values of y:
- 0 0
- 1 0
- 2 0
- 3 0
- 4 0
- Name: species, dtype: int64
-

- Iris dataset exploration and basic preparation complete. Ready for ML modeling.