SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

COIMBATORE – 641008

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CYBER SECURITY)

22CY504– PYTHON FOR DATA ANALYSIS LABORATORY

RECORD WORK

Submitted by

Name	**:**
Register No.	**:**
Degree & Branch	**: B.E Computer Science and Engineering (Cyber Security)**
Class	**: III B.E CSE (CS)**

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

COIMBATORE – 641008

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CYBER SECURITY)

22CY504– PYTHON FOR DATA ANALYSIS LABORATORY

Continuous Assessment

Record Submitted by

Name :	**Register No. :**
Degree & Branch: B.E – CSE(CS)	**Year & Semester : III / V**

BONAFIDE CERTIFICATE

This is to certify that this record is the bonafide record of work done by Mr./Ms.__________________ during the academic year 2024 – 2025.

Staff In-charge **HOD / CSE (Cyber Security)**

Submitted for the practical examination held on ____________________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

EXP. NO	DATE	EXPERIMENT NAME	Page No.	SIGNATURE
1.		Perform Creation, indexing, slicing, concatenation and repetition operations on python built-in data types: Strings, List, Tuples, Dictionary, Set.		
2.		Apply Python built-in data types: Strings, List, Tuples, Dictionary, Set and their methods to solve any given problem.		
3.		Handle numerical operations using math and random number functions		
4.		Create user-defined functions with different types of function arguments.		
5.		Create NumPy arrays from Python Data Structures, Intrinsic NumPy objects and Random Functions.		
6.		Manipulation of NumPy arrays- Indexing, Slicing, Reshaping, Joining and Splitting.		
7.		Load an image file and do crop and flip operation using NumPy Indexing		
8.		Create Pandas Series and Data Frame from various inputs.		
9.		Import any CSV file to Pandas Data Frame and perform the given operations		
10.		Import any CSV file to Pandas Data Frame and perform the given operations		

Faculty In-Charge

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

COIMBATORE – 641008

(An AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CYBER SECURITY)

22CY504– PYTHON FOR DATA ANALYSIS LABORATORY

Reg.No: **Name:**

Components	**Experiment Nos.**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Aim & Algorithm (20)										
Coding (30)										
Compilation & Debugging (30)										
Execution & Result (20)										
TOTAL										
AVERAGE										

Signature of the Faculty

SRI KRISHNA COLLEGE OF ENGINEERING AND TECHNOLOGY

COIMBATORE - 641008

(An AUTONOMOUS INSTITUTION, AFFILIATED TO ANNA UNIVERSITY, CHENNAI)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(CYBER SECURITY)

22CY504– PYTHON FOR DATA ANALYSIS LABORATORY

ODD SEMESTER 2024-2025

Name of Faculty Members: Dr.P.D.Mahendhiran

METHOD OF CONTINUOUS EVALUATION

EVALUATION RUBRICS

Criteria	**Range of Marks**			
	Excellent	**Good**	**Average**	**Below Average**
Aim & Algorithm (20)	18-20	14-17	10-13	0-9
Coding (30)	27-30	21-26	15-20	0-14
Compilation and Debugging (30)	27-30	21-26	15-20	0-14
Execution and Result (20)	18-20	14-17	10-13	0-9

Ex.No:01

Date :

Perform Creation, indexing, slicing, concatenation and repetition operations on python built-in data types: Strings, List, Tuples, Dictionary, Set.

Aim:

To perform various operations such as creation, indexing, slicing, concatenation, and repetition on Python built-in data types: Strings, Lists, Tuples, Dictionaries, and Sets.

Algorithm:

1. **Create data types :** Initialize string, list, tuple, dictionary, and set.
2. **String Indexing:** Access an element from the string using its index.
3. **String Slicing:** Extract a substring using a range of indices.
4. **String Concatenation:** Combine two strings.
5. **String Repetition:** Repeat the string multiple times.
6. **List Indexing & Slicing:** Access and slice elements from the list.
7. **List Concatenation & Repetition:** Combine lists and repeat elements.
8. **Tuple Indexing, Slicing, Concatenation & Repetition:** Perform similar operations on tuples.
9. **Dictionary Access:** Retrieve values using keys.
10. **Set Union:** Combine sets using union (no repetition allowed)

Source Code:

```
 # 1. String
s1 = "Happy"
s2 = "Birthday"
indexed_char = s1[1]                    # Indexing
sliced_string = s1[1:4]                 # Slicing
concatenated_string = s1 + " " + s2     # Concatenation
repeated_string = s1 * 3                # Repetition

print("String Operations:")
print(f"Original Strings: '{s1}', '{s2}'")
print(f"Indexed Character: {indexed_char}")
print(f"Sliced String: {sliced_string}")
print(f"Concatenated String: {concatenated_string}")
print(f"Repeated String: {repeated_string}")

# 2. List
l1 = [32,33,34]
l2 = [53,54,55]

indexed_list_element = l1[1]
sliced_list = l1[1:]
concatenated_list = l1 + l2
repeated_list = l1 * 2

print("List Operations:")
print(f"Original Lists: {l1}, {l2}")
print(f"Indexed List Element: {indexed_list_element}")
print(f"Sliced List: {sliced_list}")
print(f"Concatenated List: {concatenated_list}")
print(f"Repeated List: {repeated_list}")

# 3. Tuple
t1 = (12,22,33)
t2 = (13,30,24)

indexed_tuple_element = t1[1]
sliced_tuple = t1[:2]
concatenated_tuple = t1 + t2
repeated_tuple = t2 * 2

print("Tuple Operations:")
print(f"Original Tuples: {t1}, {t2}")
print(f"Indexed Tuple Element: {indexed_tuple_element}")
print(f"Sliced Tuple: {sliced_tuple}")
print(f"Concatenated Tuple: {concatenated_tuple}")
print(f"Repeated Tuple: {repeated_tuple}")

# 4. Dictionary
d1 = {'x': 61, 'y': 21}
d2 = {'g': 73, 'a': 74}

indexed_dict_element = d1.get(‘x’)
concatenated_dict = {**d1, **d2}
```

```
print("Dictionary Operations:")
print(f"Original Dictionaries: {d1}, {d2}")
print(f"Indexed Dictionary Element: {indexed_dict_element}")
print(f"Concatenated Dictionary: {concatenated_dict}")
# 5. Set
s1 = {61,72,83}
s2 = {81, 91,101}

concatenated_set = s1.union(s2)

print("Set Operations:")
print(f"Original Sets: {s1}, {s2}")
print(f"Union of Sets: {concatenated_set}")
```

Output:

```
String Operations:
Original Strings: 'Happy', 'Birthday'
Indexed Character: a
Sliced String: app
Concatenated String: HappyBirthday
Repeated String: HappyHappyHappy

List Operations:
Original Lists: [32,33,34], [53,54,55]
Indexed List Element: 33
```

```
Sliced List: [33,34]
Concatenated List: [32,33,34,53,54,55]
Repeated List: [32,33,34,32,33,34]


Tuple Operations:
Original Tuples: (12, 22, 33), (13, 30, 24)
Indexed Tuple Element: 22
Sliced Tuple: (12, 22)
Concatenated Tuple: (12, 22, 33, 13, 30, 24)
Repeated Tuple: (13, 30, 24, 13, 30, 24)

Dictionary Operations:
Original Dictionaries: {'x': 61, 'y': 21}, {'g': 73, 'a': 74}
Indexed Dictionary Element: 11
Concatenated Dictionary: {'x': 61, 'y': 21, 'g': 73, ‘a': 74}

Set Operations:
Original Sets: {61, 72, 83}, {83, 94, 105}
Union of Sets: {61, 72, 83, 81, 91, 101}
```

Result:

The operations of creation, indexing, slicing, concatenation, and repetition have been successfully performed on data types: Strings, Lists, Tuples, Dictionaries, and Sets.

Ex.No:02

Date :

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

Aim:

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

Algorithm:

1. Create a Dictionary: Initialize a dictionary where keys are student names and values are their test scores.

2. Sort Scores: Convert the dictionary values (scores) into a list and sort them.

3. Use Tuples: Group students and their scores into tuples.

4. Categorize students: Create a set to hold students who scored above 75.

5. Categorize students: Create another set to hold students who score below or equal to 75.

6. Display the sorted scores, student-score tuples, and categorized sets.

Source Code:

```
# Step 1: Initialize dictionary with student names and their scores

student_scores = {    "Alice": 85, "Bob": 75,"Charlie": 95,"David": 65,"Eva": 79}

# Step 2: Sort the scores

sorted_scores = sorted(student_scores.values())
```

```
# Step 3: Create tuples of students and their scores

student_score_tuples = [(student, score) for student, score in
student_scores.items()]

# Step 4: Categorize students based on their scores

above_75 = {student for student, score in student_scores.items() if score > 75}

below_or_equal_75 = {student for student, score in student_scores.items() if
score <= 75}

# Step 5: Print the results

print("Sorted Scores:", sorted_scores)

print("Student-Score Tuples:", student_score_tuples)

print("Students Scoring Above 75:", above_75)

print("Students Scoring Below or Equal to 75:", below_or_equal_75)
```

Output:

```
Sorted Scores: [65, 75, 79, 85, 95]

Student-Score Tuples: [('Alice', 85), ('Bob', 75), ('Charlie', 95), ('David', 65),
('Eva', 79)]

Students Scoring Above 75: {'Alice', 'Charlie', 'Eva'}

Students Scoring Below or Equal to 75: {'Bob', 'David'}
```

Result:

Thus a simple program demonstrating Python's built-in data types and their methods has been efficiently created.

Ex.No:3

Date:

Handle numerical operations using math and random number functions

Aim:

To perform numerical operations using math and random number functions in Python, demonstrating the use of mathematical functions (such as square root, power, etc.) and generating random numbers.

Algorithm:

1. Start the program.
2. Import the required libraries:
   - math for mathematical operations.
   - random for generating random numbers.
3. Define a function for each mathematical operation (e.g., square root, power, sine, etc.).
4. Generate random numbers using random functions.
5. Perform the mathematical operations using the random numbers and math functions.
6. Display the input numbers, operations performed, and the result of each operation.

Source Code:

```
import math
import random
# Math operations using math module
print("Math Operations:")
```

```
# Square root
num = 225
sqrt_num = math.sqrt(num)
print(f"Square root of {num}: {sqrt_num}")

# Power
base = 2
exp = 5
power_result = math.pow(base, exp)
print(f"{base} raised to the power {exp}: {power_result}")

# Logarithm (base 10)
log_num = 100
log_result = math.log10(log_num)
print(f"Log base 10 of {log_num}: {log_result}")

# Trigonometric functions
angle = math.radians(45)  # Convert degrees to radians
sin_value = math.sin(angle)
cos_value = math.cos(angle)
print(f"Sin(45 degrees): {sin_value}")
print(f"Cos(45 degrees): {cos_value}")



# Floor and ceil
num = 3.7
floor_value = math.floor(num)
```

```
ceil_value = math.ceil(num)
print(f"Floor value of {num}: {floor_value}")
print(f"Ceil value of {num}: {ceil_value}")

# Random operations using random module
print("\nRandom Operations:")

# Random integer between 1 and 10
rand_int = random.randint(1, 10)
print(f"Random integer between 1 and 10: {rand_int}")

# Random float between 0 and 1
rand_float = random.random()
print(f"Random float between 0 and 1: {rand_float}")

# Random choice from a list
choices = [10, 20, 30, 40, 50]
rand_choice = random.choice(choices)
print(f"Random choice from the list {choices}: {rand_choice}")

# Shuffling a list
list_to_shuffle = [1, 2, 3, 4, 5]
random.shuffle(list_to_shuffle)
print(f"Shuffled list: {list_to_shuffle}")
```

Output:

```
Math Operations:
Square root of 225: 15.0
2 raised to the power 5: 32.0
Log base 10 of 100: 2.0
Sin(45 degrees): 0.7071067811865475
Cos(45 degrees): 0.7071067811865476
Floor value of 3.7: 3
Ceil value of 3.7: 4

Random Operations:
Random integer between 1 and 10: 7
Random float between 0 and 1: 0.3620391346038981
Random choice from the list [10, 20, 30, 40, 50]: 20
Shuffled list: [5, 3, 1, 2, 4]
```

Result:

The program to demonstrate the use of math functions and random number generation has been executed successfully.

Ex.No:04

Date:

Create User-Defined Functions with Different Types of Function Arguments

Aim:

To write and implement user-defined functions in Python that demonstrate the use of different types of function arguments, including Positional Arguments, Keyword Arguments, Default Arguments, Variable-Length Arguments.

Algorithm:

1. Functions in Python are blocks of reusable code that can be executed when called.
2. **Positional Arguments**: The most common way to pass arguments, where the position in the function call matters.
3. **Keyword Arguments**: Arguments passed with the name of the parameter, which allows skipping order.
4. **Default Arguments**: Arguments that have default values. These can be omitted in the function call.
5. **Variable-Length Arguments**: Define a function that can accept a variable number of arguments using *args for non-keyworded variable-length arguments.
6. Implement and call each function to demonstrate the different behaviors.

Source Code:

```
# Function with Positional Arguments
def subtract(a, b):
#Subtracts second argument from the first
  return a – b

# Function with Keyword Arguments
def display_user_info(name, age, city):
#Displays user information with parameters as keyword arguments
      return f"User: {name}, Age: {age}, City: {city}"

# Function with Default Arguments
def create_account(username, account_type="Savings"):
 #Creates an account, with a default account type of 'Savings
    return f"Username: {username}, Account Type: {account_type}"

# Function with Variable-Length Arguments (*args)
def find_max(*numbers):
#Finds the maximum number from a list of numbers
    if numbers:
        return max(numbers)
    else:
        return None

# Function with Variable-Length Keyword Arguments (**kwargs)
def describe_person(**details):
```

```
#Describes a person based on provided details (any number of keyword arguments)
    description = ""
    for key, value in details.items():
        description += f"{key.capitalize()}: {value}\n"
    return description

# Positional Arguments example
result1 = subtract(25, 5)
print(f"Subtraction (Positional Arguments): {result1}")  # Output: 20

# Keyword Arguments example
result2 = display_user_info(name="Bob", age=35, city="New York")
print(f"User Info (Keyword Arguments): {result2}")

# Default Arguments example
result3 = create_account("doe")  # Uses default 'Savings'
result4 = create_account("jane", "Checking")  # Overrides default with 'Checking'
print(f"Account (Default Argument): {result3}")
print(f"Account (Overridden Default Argument): {result4}")

# Variable-Length Arguments (*args) example
result5 = find_max(10, 25, 30, 78, 50)
print(f"Maximum value (*args): {result5}")
```

```
# Variable-Length Keyword Arguments (**kwargs) example
result6 = describe_person(name="Alice", age=22, occupation="Engineer", city="Seattle")
print(f"Person Description (**kwargs):\n{result6}")
```

Output:

```
Subtraction (Positional Arguments): 20
User Info (Keyword Arguments): User: Bob, Age: 35, City: New York
Account (Default Argument): Username: doe, Account Type: Savings
Account (Overridden Default Argument): Username: jane, Account Type: Checking
Maximum value (*args): 78
Person Description (**kwargs):
Name: Alice
Age: 22
Occupation: Engineer
City: Seattle
```

`Result:

Thus user defined functions with different types of functional arguments is created successfully.

Ex.No:05

Date:

Create NumPy arrays from Python Data Structures, Intrinsic NumPy objects and Random Functions

Aim:

To understand how to create NumPy arrays from various Python data structures, intrinsic NumPy objects, and random functions, and explore basic array manipulations.

Algorithm:

1. Import NumPy Library:
   - Begin by importing the numpy module.
2. Creating NumPy Arrays from Python Data Structures:
   - Use Python lists or tuples to create NumPy arrays using np.array().
3. Creating Arrays Using Intrinsic NumPy Functions:
   - Use intrinsic NumPy functions such as np.zeros(), np.ones(), np.arange(), and np.linspace() to generate arrays.
4. Generating Arrays Using Random Functions:
   - Use np.random.rand(), np.random.randint(), and np.random.normal() to create arrays with random numbers.
5. Display Results:
   - Print the created arrays for verification and analysis.

Source Code:

```
import numpy as np
list_data = [1, 2, 3, 4, 5]
array_from_list = np.array(list_data)
print("Array from list:", array_from_list)

tuple_data = (6, 7, 8, 9)
array_from_tuple = np.array(tuple_data)
print("Array from tuple:", array_from_tuple)

array_zeros = np.zeros((3, 4))
print("\nArray of zeros:\n", array_zeros)

array_ones = np.ones((2, 3))
print("\nArray of ones:\n", array_ones)

array_range = np.arange(0, 10, 2)  # start=0, stop=10, step=2
print("\nArray with arange:\n", array_range)

# Create an array with equally spaced numbers (linspace)
array_linspace = np.linspace(0, 1, 5)  # 5 values between 0 and 1
print("\nArray with linspace:\n", array_linspace)

# Step 4: Creating arrays using random functions

# Random array with values between 0 and 1
```

```
array_random = np.random.rand(3, 3)
print("\nRandom array with values between 0 and 1:\n", array_random)

# Random integers between 10 and 20
array_randint = np.random.randint(10, 20, size=(2, 4))
print("\nRandom integers between 10 and 20:\n", array_randint)

# Random values from normal distribution
array_normal = np.random.normal(0, 1, size=(2, 2))  # mean=0, std=1
print("\nArray from normal distribution:\n", array_normal)
```

Output:

```
Array from list: [1 2 3 4 5]
Array from tuple: [6 7 8 9]

Array of zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Array of ones:
 [[1. 1. 1.]
 [1. 1. 1.]]

Array with arange:
 [0 2 4 6 8]
```

```
Array with linspace:
 [0.   0.25 0.5  0.75 1.  ]

Random array with values between 0 and 1:
 [[0.51203515 0.31620874 0.80739132]
 [0.58591466 0.70102781 0.45036532]
 [0.17841973 0.36650291 0.32934869]]

Random integers between 10 and 20:
 [[17 11 12 16]
 [13 17 15 11]]

Array from normal distribution:
 [[-1.12352855  0.35737266]
 [-0.48275892 -0.23457924]]
```

Result:

Thus NumPy arrays from Python Data Structures, Intrinsic NumPy objects and Random Functions is successfully created.

Ex.No:06

Date:

Manipulation of NumPy arrays- Indexing, Slicing, Reshaping, Joining and Splitting.

Aim:

To understand the manipulation of NumPy arrays through various operations, including indexing, slicing, reshaping, joining, and splitting.

Algorithm:

1. Import NumPy Library:
   Start by importing the numpy library.
2. Create NumPy Arrays:
   Initialize arrays for demonstrating indexing, slicing, reshaping, joining, and splitting.
3. Indexing:
   Access specific elements using their indexes.
4. Slicing:
   Extract portions of an array using slicing.
5. Reshaping:
   Change the shape of an array using reshape().
6. Joining:
   Combine multiple arrays using concatenate() and stack().
7. Splitting:
   Split an array into smaller sub-arrays using split() or array_split().
8. Display Output:
   Print the outputs of the various operations.
9. Result:
   Summarize the successful execution of all manipulations.

Source Code:

```
# Step 1: Import numpy library
import numpy as np

# Step 2: Create NumPy arrays for manipulation
array1 = np.array([1, 2, 3, 4, 5, 6])
array2 = np.array([7, 8, 9, 10, 11, 12])

# Step 3: Indexing - Access elements by their index
print("Element at index 2 of array1:", array1[2])

# Step 4: Slicing - Extract a sub-array
print("Slice of array1 (index 1 to 4):", array1[1:5])

# Step 5: Reshaping - Change the shape of an array
reshaped_array = array1.reshape(2, 3)  # Reshape to 2 rows, 3 columns
print("\nReshaped Array (2x3):\n", reshaped_array)

# Step 6: Joining - Concatenate two arrays
joined_array = np.concatenate((array1, array2))
print("\nJoined Array:\n", joined_array)

# Vertical stacking
vstacked_array = np.vstack((array1.reshape(2, 3), array2.reshape(2, 3)))
print("\nVertically Stacked Array:\n", vstacked_array)
```

```
# Step 7: Splitting - Split an array into smaller arrays
split_array = np.split(array1, 2)  # Split into 2 equal parts
print("\nSplit Array (2 equal parts):", split_array)

# Unequal splitting using array_split
unequal_split = np.array_split(array1, 4)  # Split into 4 parts (not equal)
print("\nUnequal Split Array:", unequal_split)
```

Output:

```
Element at index 2 of array1: 3
Slice of array1 (index 1 to 4): [2 3 4 5]
Reshaped Array (2x3):
 [[1 2 3]
  [4 5 6]]
Joined Array:
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
Vertically Stacked Array:
 [[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]
  [10 11 12]]
Split Array (2 equal parts): [array([1, 2, 3]), array([4, 5, 6])]
Unequal Split Array: [array([1, 2]), array([3, 4]), array([5]), array([6])]
```

Result:

Thus the Manipulation of Numpy Arrays is demonstrated successfully.

Ex.No:07

Date:

Load an image file and do crop and flip Operation using NumPy Indexing

Aim:

To load an image file and perform cropping and flipping operations using NumPy indexing.

Algorithm:

1. Import Required Libraries:
Import numpy for manipulation and PIL (Python Imaging Library) for loading and saving the image.

2. Load the Image:
Use the Image module from PIL to load the image as an array.

3. Convert Image to NumPy Array:
Convert the loaded image into a NumPy array to allow array manipulations.

4. Crop the Image:
Use NumPy indexing to extract a region of interest (crop).

5. Flip the Image:
Use NumPy operations to flip the image horizontally and vertically.

6. Convert Arrays Back to Image:
Convert the manipulated arrays back to images using PIL.

7. Display the Results:
Print or display the cropped and flipped images.

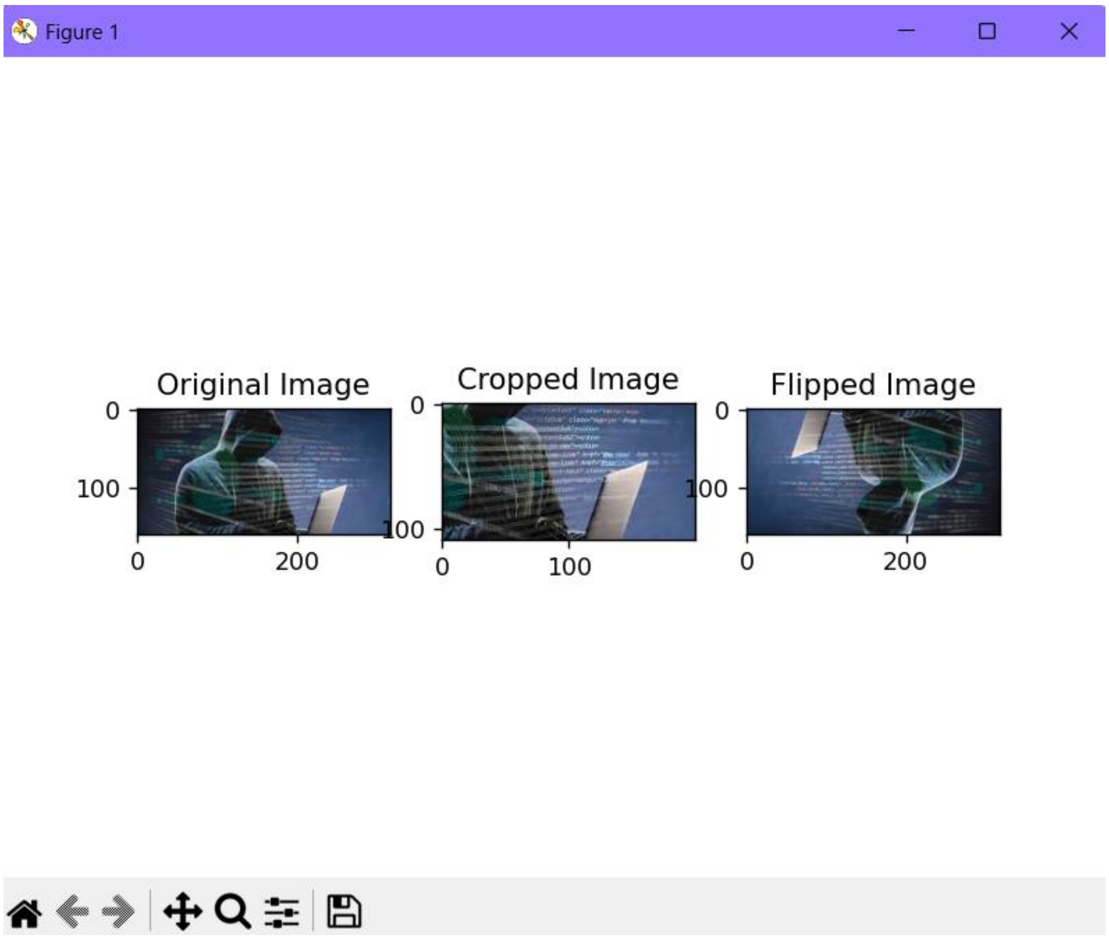
8. Save the Results (Optional):
Save the output images to verify the operations.

Source Code:

```
# Step 1: Import necessary libraries
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Step 2: Load the image and convert it to a NumPy array
image = Image.open(r"sample_image.jpg")  # Replace with your image path
image_array = np.array(image)

# Display the original image
plt.subplot(1, 3, 1)
plt.imshow(image_array)
plt.title("Original Image")

# Step 3: Crop the image using NumPy indexing
# Crop the central region (e.g., height: 50 to 200, width: 100 to 300)
cropped_image_array = image_array[50:200, 100:300]

# Display the cropped image
plt.subplot(1, 3, 2)
plt.imshow(cropped_image_array)
plt.title("Cropped Image")

# Step 4: Flip the image horizontally and vertically using NumPy operations
```

```
flipped_image_array = np.flipud(np.fliplr(image_array))  # Flip both vertically and horizontally

# Display the flipped image

plt.subplot(1, 3, 3)

plt.imshow(flipped_image_array)

plt.title("Flipped Image")

plt.show()
```

Output:



Result:

Thus, loading an image file and performing cropping and flipping operations using NumPy indexing is done successfully.

Ex.No:08

Date:

Creating Pandas Series and DataFrame from Various Inputs

Aim:

To create and manipulate Pandas Series and DataFrames from different types of inputs, such as lists, dictionaries, and NumPy arrays.

Algorithm:

1. **Import the Pandas and NumPy libraries:**
   Use import pandas as pd and import numpy as np.
2. **Create a Pandas Series:**
   Create Series objects from:
   - Python lists
   - NumPy arrays
   - Python dictionaries
3. **Create a DataFrame:**
   Create DataFrames from:
   - Dictionaries of lists or arrays
   - NumPy arrays
   - List of dictionaries
4. **Manipulate and Display Data:**
   Print the created Series and DataFrames for verification.
5. **Result:**
   Validate the outputs to ensure correct creation of Series and DataFrames.

Source Code:

```
# Step 1: Import necessary libraries
import pandas as pd
import numpy as np

# Step 2: Create Pandas Series from various inputs

# Series from a list
series_from_list = pd.Series([10, 20, 30, 40, 50])
print("Series from List:\n", series_from_list)

# Series from a NumPy array
array = np.array([1.1, 2.2, 3.3, 4.4])
series_from_array = pd.Series(array)
print("\nSeries from NumPy Array:\n", series_from_array)

# Series from a dictionary
dict_data = {'a': 100, 'b': 200, 'c': 300}
series_from_dict = pd.Series(dict_data)
print("\nSeries from Dictionary:\n", series_from_dict)

# Step 3: Create DataFrames from various inputs

# DataFrame from a dictionary of lists
data_dict = {
   'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'San Francisco', 'Los Angeles']
}
df_from_dict = pd.DataFrame(data_dict)
print("\nDataFrame from Dictionary of Lists:\n", df_from_dict)

# DataFrame from a NumPy array
array_data = np.array([[1, 2], [3, 4], [5, 6]])
df_from_array = pd.DataFrame(array_data, columns=['Column1', 'Column2'])
print("\nDataFrame from NumPy Array:\n", df_from_array)

# DataFrame from a list of dictionaries
list_of_dicts = [
    {'Name': 'David', 'Age': 28},
    {'Name': 'Eva', 'Age': 24},
    {'Name': 'Frank', 'Age': 32}]
df_from_list_of_dicts = pd.DataFrame(list_of_dicts)
print("\nDataFrame from List of Dictionaries:\n", df_from_list_of_dicts)
```

Output:

```
Series from List:
0    10
1    20
2    30
3    40
4    50
```

```
Series from NumPy Array:
0    1.1
1    2.2
2    3.3
3    4.4

Series from Dictionary:
a    100
b    200
c    300

DataFrame from Dictionary of Lists:
      Name     Age      City
0    Alice      25      New York
1    Bob        30    San Francisco
2    Charlie    35     Los Angeles

DataFrame from NumPy Array:
       Column1   Column2
0          1             2
1          3             4
2          5             6
```

```
DataFrame from List of Dictionaries:
       Name     Age
0      David     28
1      Eva       24
2      Frank     32
```

Result:

Thus the creation of Pandas Series and Dataframes is done successfully.

Ex.No:09

Date:

Import any CSV file to Pandas Data Frame and perform the given operations

Aim:

To import a CSV file into a Pandas DataFrame and perform the given operations:

(a) Visualize the first and last 10 records

(b) Get the shape, index and column details

(c) Select/Delete the records (rows)/columns based on conditions.

(d) Perform ranking and sorting operations.

(e) Do required statistical operations on the given columns.

(f) Find the count and uniqueness of the given categorical values.

(g) Rename single/multiple columns.

Algorithm:

1. Import the necessary libraries.
2. Load the CSV file into a Pandas DataFrame.
3. Visualize the first and last 10 records.
4. Retrieve the shape, index, and column details of the DataFrame.
5. Select or delete records or columns based on specified conditions.
6. Perform ranking and sorting operations on the DataFrame.
7. Conduct statistical operations on specified columns.
8. Find the count and uniqueness of categorical values.
9. Rename Single/Multiple columns.

Source Code:

```
# Importing necessary libraries
import pandas as pd
# Load the CSV file into a DataFrame (replace 'data.csv' with your file path)
df = pd.read_csv('data.csv')

# (a) Visualize the first and last 10 records
print("First 10 records:")
print(df.head(10))
print("\nLast 10 records:")
print(df.tail(10))

# (b) Get the shape, index, and column details
print("\nShape of the DataFrame:", df.shape)
print("Index of the DataFrame:", df.index)
print("Columns in the DataFrame:", df.columns)

# (c) Select/Delete rows or columns based on conditions
# Example: Select rows where column 'A' > 10
selected_rows = df[df['A'] > 10]
print("\nSelected rows where column 'A' > 10:")
print(selected_rows)
# Example: Delete rows where column 'B' has null values
df_dropped = df.dropna(subset=['B'])
print("\nDataFrame after deleting rows where column 'B' has null values:")
print(df_dropped)

# (d) Perform ranking and sorting operations
# Example: Rank the rows based on values in column 'D'
df['Rank_D'] = df['D'].rank()
print("\nDataFrame with ranking based on column 'D':")
print(df[['D', 'Rank_D']])
# Example: Sort the DataFrame by column 'D' in descending order
df_sorted = df.sort_values(by='D', ascending=False)
print("\nDataFrame sorted by column 'D' in descending order:")
print(df_sorted)

# (e) Perform statistical operations on given columns
# Example: Compute mean and standard deviation for column 'A'
mean_A = df['A'].mean()
std_dev_A = df['A'].std()
print(f"\nMean of column 'A': {mean_A}")
print(f"Standard deviation of column 'A': {std_dev_A}")

# (f) Find count and uniqueness of categorical values
# Example: Count and unique values in column 'C'
count_C = df['C'].value_counts()
unique_C = df['C'].nunique()
print("\nCount of each unique value in column 'C':")
print(count_C)
print(f"Number of unique values in column 'C': {unique_C}")
```

Output:

(a) Visualize the First and Last 10 Records

```
First 10 records:
    A    B  C    D
0   5   1.0  X  100
1  15   NaN  Y  200
2  10   3.0  X  150
3  20   NaN  Y  250
4  25   5.0  Z  300
5  30   6.0  X  350
6   5   7.0  Y  100
7  10   NaN  X  150
8  15   9.0  Y  200
9  20  10.0  Z  250
Last 10 records:
    A    B  C    D
0   5   1.0  X  100
1  15   NaN  Y  200
2  10   3.0  X  150
3  20   NaN  Y  250
4  25   5.0  Z  300
5  30   6.0  X  350
6   5   7.0  Y  100
7  10   NaN  X  150
8  15   9.0  Y  200
```

```
9   20  10.0  Z  250
```

(b) Get the Shape, Index, and Column Details:

```
Shape of the DataFrame: (10, 4)

Index of the DataFrame: RangeIndex(start=0, stop=10, step=1)

Columns in the DataFrame: Index(['A', 'B', 'C', 'D'], dtype='object')
```

(c) Select/Delete Rows or Columns Based on Conditions:

- Select rows where A > 10:

```
Selected rows where column 'A' > 10:
    A    B  C    D
1  15   NaN  Y  200
3  20   NaN  Y  250
4  25   5.0  Z  300
5  30   6.0  X  350
8  15   9.0  Y  200
9  20  10.0  Z  250
```

- Delete rows where B has null values:

```
DataFrame after deleting rows where column 'B' has null values:
    A    B  C    D
0   5   1.0  X  100
2  10   3.0  X  150
4  25   5.0  Z  300
5  30   6.0  X  350
6   5   7.0  Y  100
8  15   9.0  Y  200
9  20  10.0  Z  250
```

(d) Perform Ranking and Sorting Operations:

- **Rank rows based on values in column D:**

```
DataFrame with ranking based on column 'D':

    D  Rank_D

0  100     1.0

1  200     4.5
```

```
2  150      3.0
3  250      6.5
4  300      8.0
5  350      9.0
6  100      1.0
7  150      3.0
8  200      4.5
9  250      6.5
```

- Sort DataFrame by column D in descending order:

```
DataFrame sorted by column 'D' in descending order:
    A    B  C    D
5   30   6.0  X  350
4   25   5.0  Z  300
3   20   NaN  Y  250
9   20  10.0  Z  250
1   15   NaN  Y  200
8   15   9.0  Y  200
2   10   3.0  X  150
7   10   NaN  X  150
0    5   1.0  X  100
6    5   7.0  Y  100
```

(e) Perform Statistical Operations on Column A

```
Mean of column 'A': 17.5
Standard deviation of column 'A': 8.539125638299666
```

(f) Find Count and Uniqueness of Categorical Values

- **Count and unique values in column C:**

```
Count of each unique value in column 'C':
X    4
Y    4
Z    2
Name: C, dtype: int64
Number of unique values in column 'C': 3
```

(g)Rename Single/Multiple Columns:

- **DataFrame after Renaming a Single Column (A to Alpha)**:

```
DataFrame after renaming a single column:
   Alpha    B  C  D
0     10  100  X  1
1     20  200  Y  2
2     30  300  Z  3
3     40  400  X  4
4     50  500  Y  5
```

- **DataFrame after Renaming Multiple Columns (B to Beta and C to Gamma)**:

```
DataFrame after renaming multiple columns:
   A  Beta Gamma  D
0  10   100     X  1
1  20   200     Y  2
2  30   300     Z  3
3  40   400     X  4
4  50   500     Y  5
```

Result:

Hence a CSV file is converted into pyhton data frame and the given operations are performed successfully.

Ex.No:10

Date:

Import any CSV file to Pandas Data Frame and perform the given operations

Aim:

To import a CSV file into a Pandas DataFrame and perform data cleaning, transformation, outlier detection, string operations, and data visualization using various Pandas methods and Matplotlib.

Algorithm:

1. Import Libraries: Import necessary libraries (pandas, numpy, matplotlib, and seaborn).
2. Read CSV File: Load the CSV file into a Pandas DataFrame.
3. Handle Missing Data:
    - Detect missing values.
    - Drop or fill missing values.
4. Transform Data:
    - Use apply() and map() to perform transformations.
5. Detect and Filter Outliers:
    - Use IQR (Interquartile Range) to detect and remove outliers.
6. Perform Vectorized String Operations:
    - Apply string operations on a Pandas Series (e.g., converting to uppercase).
7. Visualize Data:
    - Create line plots, bar plots, histograms, and density plots using Matplotlib and Seaborn.

Source Code:

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

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

# Display the first few rows of the DataFrame
print("Initial DataFrame:")
print(df.head())

# (a) Handle missing data
print("\nMissing Data Information:")
print(df.isnull().sum())  # Detect missing values

# Option 1: Drop rows with any missing values
df_dropped = df.dropna()
print("\nDataFrame after dropping missing values:")
print(df_dropped.head())

# Option 2: Fill missing values with the column mean
# It's safer to use numeric_only=True to avoid errors with non-numeric
columns
df_filled = df.fillna(df.mean(numeric_only=True))
print("\nDataFrame after filling missing values with column mean:")
print(df_filled.head())

# (b) Transform data using apply() and map()
# Example: Apply a transformation to add 10 to each value in a numeric
column (e.g., 'A')
if 'A' in df.columns:
   df['A_transformed'] = df['A'].apply(lambda x: x + 10)
   print("\nDataFrame after applying transformation to column 'A':")
   print(df[['A', 'A_transformed']].head())
else:
   print("\nColumn 'A' not found in DataFrame.")

# Example: Map categorical values to numeric values in column 'C'
if 'C' in df.columns:
   df['C_mapped'] = df['C'].map({'X': 1, 'Y': 2, 'Z': 3})
   print("\nDataFrame after mapping values in column 'C':")
   print(df[['C', 'C_mapped']].head())
else:
   print("\nColumn 'C' not found in DataFrame.")

# (c) Detect and filter outliers
# Using IQR to filter out outliers in column 'A'
if 'A' in df.columns:
   Q1 = df['A'].quantile(0.25)
   Q3 = df['A'].quantile(0.75)
   IQR = Q3 - Q1
   df_outliers_removed = df[~((df['A'] < (Q1 - 1.5 * IQR)) | (df['A'] > (Q3 +
1.5 * IQR)))]
   print("\nDataFrame after removing outliers in column 'A':")
   print(df_outliers_removed[['A']].head())
else:
   print("\nColumn 'A' not found in DataFrame.")

# (d) Perform Vectorized String operations on a Pandas Series
# Example: Convert values in a string column (e.g., 'D') to uppercase
if 'D' in df.columns:
   df['D_upper'] = df['D'].str.upper()
   print("\nDataFrame after converting values in column 'D' to
uppercase:")
   print(df[['D', 'D_upper']].head())
else:
   print("\nColumn 'D' not found in DataFrame.")

# (e) Visualize data using Line Plots, Bar Plots, Histograms, and Density
Plots
# Line Plot for column 'A'
if 'A' in df.columns:
   plt.figure(figsize=(10, 6))
   plt.plot(df.index, df['A'], label='A', marker='o', color='blue')
   plt.title("Line Plot for Column 'A'")
   plt.xlabel("Index")
   plt.ylabel("Values of A")
   plt.legend()
   plt.grid()
   plt.show()
else:
   print("\nColumn 'A' not found for plotting.")

# Bar Plot for average values of each category in column 'C'
if 'C' in df.columns and 'A' in df.columns:
   plt.figure(figsize=(10, 6))
   df.groupby('C')['A'].mean().plot(kind='bar', color='skyblue')
   plt.title("Bar Plot of Average 'A' by Category 'C'")
   plt.xlabel("Category C")
   plt.ylabel("Average of A")
   plt.grid()
   plt.show()
else:
   print("\nColumns 'C' or 'A' not found for plotting.")

# Histogram for column 'A'
if 'A' in df.columns:
   plt.figure(figsize=(10, 6))
   plt.hist(df['A'], bins=10, color='green', edgecolor='black')
   plt.title("Histogram of Column 'A'")
   plt.xlabel("Values of A")
   plt.ylabel("Frequency")
   plt.grid()
   plt.show()
else:
   print("\nColumn 'A' not found for plotting.")

# Density Plot for column 'A'
if 'A' in df.columns:
   plt.figure(figsize=(10, 6))
   sns.kdeplot(df['A'], shade=True, color="purple")
   plt.title("Density Plot of Column 'A'")
   plt.xlabel("Values of A")
   plt.grid()
   plt.show()
else:
   print("\nColumn 'A' not found for plotting.")
```

Output:

(a) Handling Missing Data:

- Missing Data Information:

```
Missing Data Information:
A    0
B    3
C    0
D    0
dtype: int64
```

- **DataFrame after Dropping Rows with Missing Values**:

```
    A      B  C       D
0  10  100.0  X  Apple
2  30  300.0  Z  Cherry
4  50  500.0  Y   Fig
```

- **DataFrame after Filling Missing Values with Column Mean**:

```
    A      B  C       D
0  10  100.0  X  Apple
1  20  300.0  Y  Banana
2  30  300.0  Z  Cherry
3  40  300.0  X    Date
4  50  500.0  Y     Fig
5 100  300.0  Z   Grape
```

(b) **Transform Data using apply() and map():**

- **DataFrame after Applying Transformation to Column A (adding 10 to each value)**:

```
    A  A_transformed
0  10             20
1  20             30
2  30             40
3  40             50
4  50             60
5 100            110
```

- **DataFrame after Mapping Values in Column C (X=1, Y=2, Z=3)**:

```
   C  C_mapped
0  X         1
1  Y         2
2  Z         3
3  X         1
4  Y         2
5  Z         3
```

(c) **Detect and Filter Outliers:**

DataFrame after Removing Outliers in Column A (assuming values above 75 are outliers):

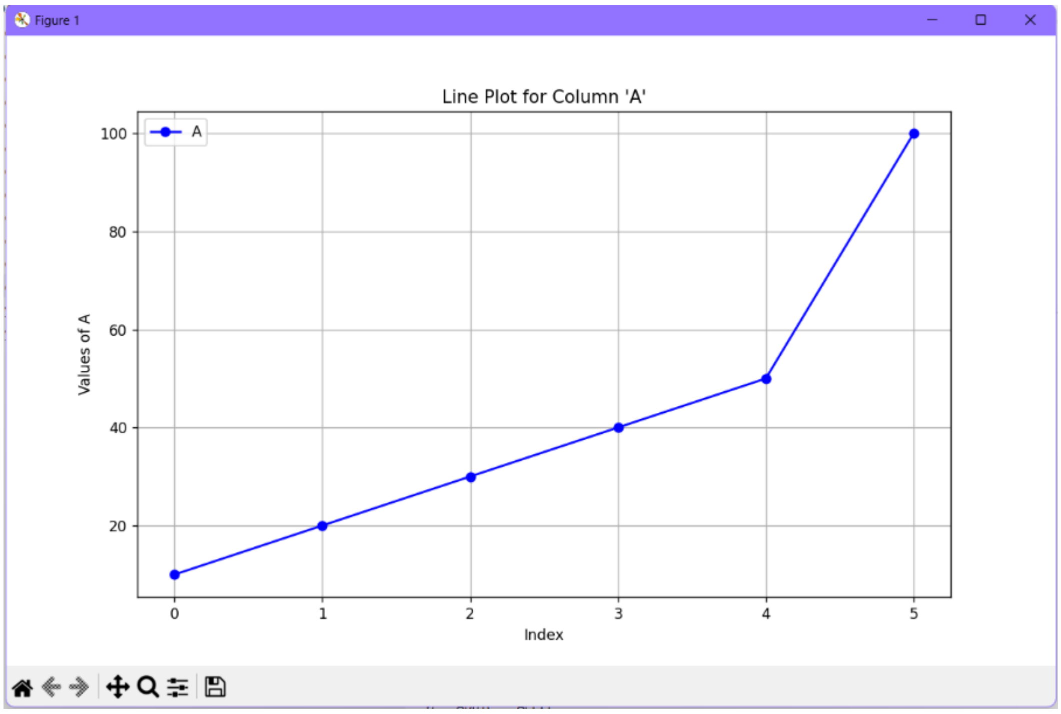
```
    A
0 10
1 20
2 30
3 40
4 50
```

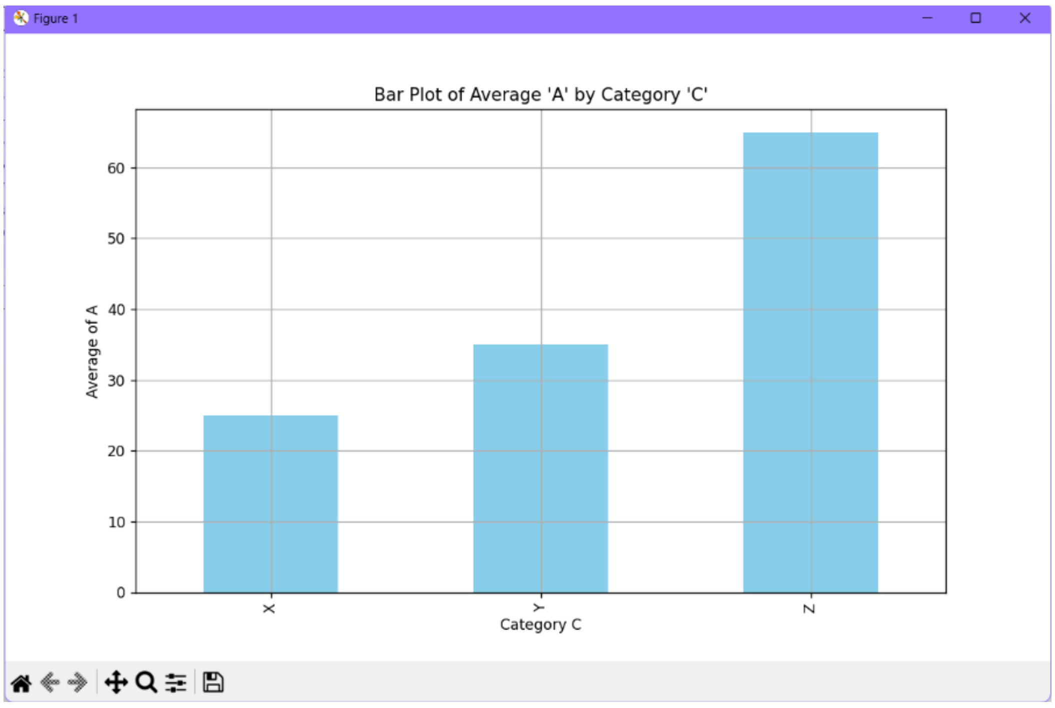
(d) **Perform Vectorized String Operations on a Pandas Series:**

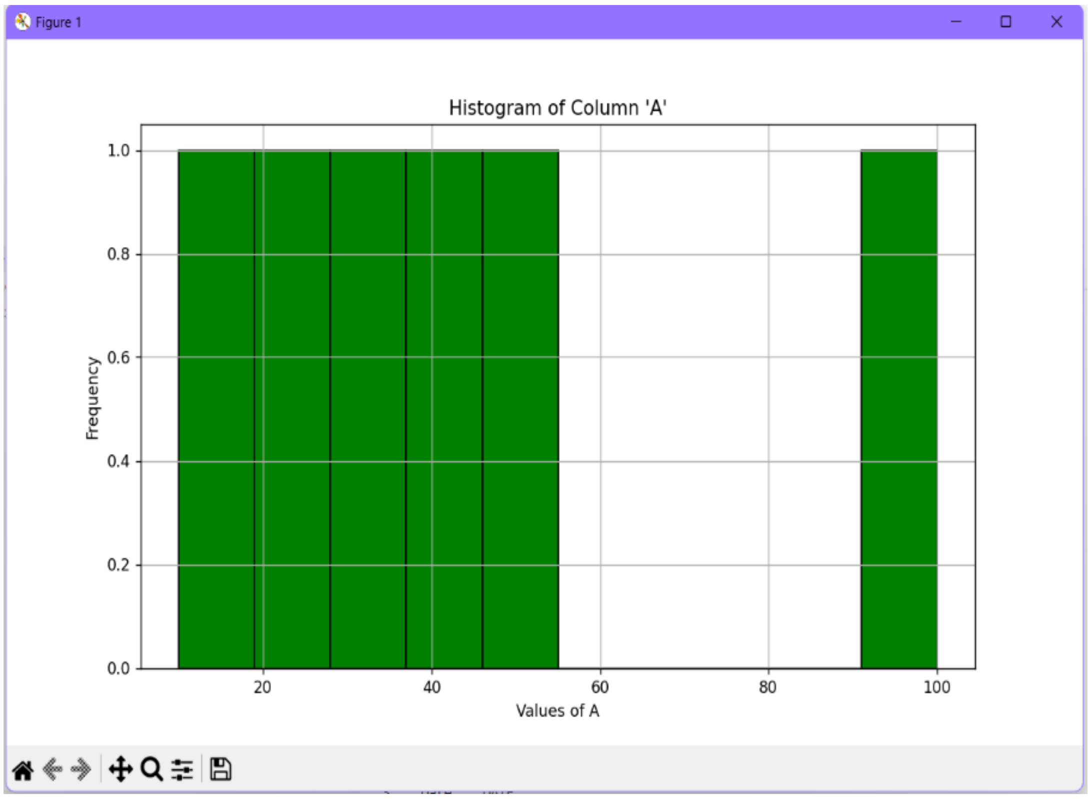
DataFrame after Converting Values in Column D to Uppercase:

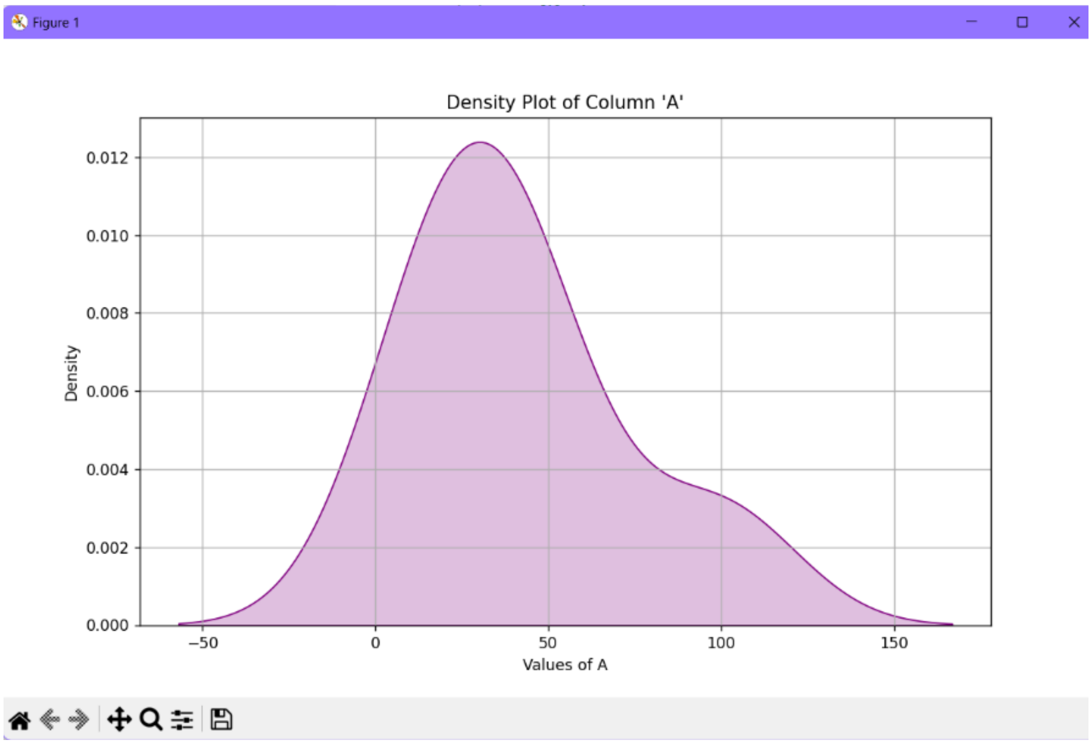
```
      D  D_upper
0   Apple   APPLE
1  Banana  BANANA
2  Cherry  CHERRY
3    Date    DATE
4     Fig     FIG
5   Grape   GRAPE
```

(e) **Visualize Data using Line Plots, Bar Plots, Histograms, Density Plots:**









RESULT:

Hence a CSV is converted to python data frame and the given operations are performed successfully.