DEPARTMENT OF INFORMATICS
Faculty of Computing Science and Applications (FCSA)
Periyar Maniammai Institute of Science & Technology
Periyar Nagar, Vallam, Thanjavur - 613 403, Tamil Nadu, India
headinformatics@pmu.edu|+91-807 290 7061 |www.pmu.edu

FACULTY OF COMPUTING SCIENCES AND APPLICATIONS

DEPARTMENT OF INFORMATICS

M.Sc., (Data Science) – I YEAR

Course Code: P24DS107

Course Title: Python and R Programming

LAB MANUAL

ACADEMIC YEAR 2025-2026

DEPARTMENT OF INFORMATICS
Faculty of Computing Science and Applications (FCSA)
Periyar Maniammai Institute of Science & Technology
Periyar Nagar, Vallam, Thanjavur - 613 403, Tamil Nadu, India
headinformatics@pmu.edu|+91-807 290 7061 |www.pmu.edu

FACULTY OF COMPUTING SCIENCES AND APPLICATIONS

DEPARTMENT OF INFORMATICS

M.Sc.,(Data Science) – I YEAR

Course Code: P24DS106

Course Title: Python and R Programming

Batch: 2025-2027

Academic Year: 2025-2026(ODD)

Prepared by : Dr. M. Manikandan AP/Informatics

Signature of Course Teacher:

HOD **FACULTY DEAN/FCSA** **DEAN ACADEMIC(COURSES)**

List of Exercise:

1. Programs using input statements and operators
2. Demonstration of literals, variables, type conversion
3. Programs on arithmetic and logical operations
4. Problem-solving using decision-making and looping
5. Programs using user-defined and lambda functions
6. Recursive functions: Factorial, and Fibonacci series
7. Programs on string operations
8. Programs on list, tuple, set, and dictionary
9. File handling (text and binary files) with CSV read / write operations
10. R programs for vector, matrix, array manipulations
11. Operations using control structures and dataframes
12. Programs for missing value handling and transformations
13. Implementing R functions
14. Implement plot functions with ggplot2
15. Visualizing geometric shapes in R

Experiment 1:

Programs using Input Statements and Operators

AIM:

To write Python programs that demonstrate the usage of input statements and operators for performing computations.

PROCEDURE:

Step 1. Start the program and prompt the user to enter values using the input() function.
Step 2. Convert the input to the required data type using int() or float().
Step 3. Apply arithmetic operators (+, -, *, /, //, %, **).
Step 4. Display the computed results using the print() function.

PROGRAM:

```
# Program to demonstrate input statements and operators
# Step 1: Input values
x = float(input("Enter first number: "))
y = float(input("Enter second number: "))
# Step 2: Arithmetic operations
print("\n  Arithmetic Operations ")
print(f"{x} + {y} = {x + y}")
print(f"{x} - {y} = {x - y}")
print(f"{x} * {y} = {x * y}")
print(f"{x} / {y} = {x / y}")
print(f"{x} // {y} = {x // y}")
print(f"{x} % {y} = {x % y}")
print(f"{x} ** {y} = {x ** y}")
```

OUTPUT:

```
Enter first number: 15
Enter second number: 4
 Arithmetic Operations
15.0 + 4.0 = 19.0
15.0 - 4.0 = 11.0
15.0 * 4.0 = 60.0
15.0 / 4.0 = 3.75
15.0 // 4.0 = 3.0
15.0 % 4.0 = 3.0
15.0 ** 4.0 = 50625.0
```

Experiment 2:

Demonstration of Literals, Variables, and Type Conversion

AIM:

To demonstrate the concept of literals, variables, and type conversion in Python.

PROCEDURE:

Step 1. Define variables with literals of different data types.
Step 2. Print their values and data types.
Step 3. Convert variables from one type to another using built-in functions.
Step 4. Observe and analyze the results.

PROGRAM:

```
# Demonstration of literals, variables and type conversion
# Step 1: Literals
integer_literal = 120
float_literal = 45.67
string_literal = "PG Python Lab"
boolean_literal = True
# Step 2: Display variables
print("Integer Literal:", integer_literal, "| Type:", type(integer_literal))
print("Float Literal:", float_literal, "| Type:", type(float_literal))
print("String Literal:", string_literal, "| Type:", type(string_literal))
print("Boolean Literal:", boolean_literal, "| Type:", type(boolean_literal))
# Step 3: Type conversion
print("\n Type Conversion ")
print("Integer to float:", float(integer_literal))
print("Float to integer:", int(float_literal))
print("Integer to string:", str(integer_literal))
print("Boolean to integer:", int(boolean_literal))
print("String to integer:", int("50"))  # Valid numeric string
```

OUTPUT:

```
Integer Literal: 120 | Type: <class 'int'>
Float Literal: 45.67 | Type: <class 'float'>
String Literal: PG Python Lab | Type: <class 'str'>
Boolean Literal: True | Type: <class 'bool'>
Type Conversion
Integer to float: 120.0
Float to integer: 45
Integer to string: 120
Boolean to integer: 1
String to integer: 50
```

Experiment 3:

Programs on Arithmetic and Logical Operations

AIM:

To implement programs that demonstrate arithmetic and logical operations in Python.

PROCEDURE:

Step 1. Read two integers from the user.
Step 2. Perform and display results of arithmetic operations.
Step 3. Perform logical operations (and, or, not) on boolean expressions.
Step 4. Interpret the results.

PROGRAM:

```
# Programs on arithmetic and logical operations
# Step 1: Input numbers
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
# Step 2: Arithmetic operations
print("\n  Arithmetic Operations")
print(f"{a} + {b} = {a + b}")
print(f"{a} - {b} = {a - b}")
print(f"{a} * {b} = {a * b}")
print(f"{a} / {b} = {a / b}")
print(f"{a} % {b} = {a % b}")
print(f"{a} ** {b} = {a ** b}")
# Step 3: Logical operations
print("\n  Logical Operations ")
print(f"({a} > 0) and ({b} > 0) = {(a > 0) and (b > 0)}")
print(f"({a} > 0) or ({b} > 0) = {(a > 0) or (b > 0)}")
print(f"not({a} > 0) = {not(a > 0)}")
```

OUTPUT:

```
Enter first number: 10
Enter second number: -5
 Arithmetic Operations
10 + -5 = 5
10 - -5 = 15
10 * -5 = -50
10 / -5 = -2.0
10 % -5 = 0
10 ** -5 = 1e-05
 Logical Operations
(10 > 0) and (-5 > 0) = False
(10 > 0) or (-5 > 0) = True
not(10 > 0) = False
```

Experiment 4:

Problem-Solving using Decision-Making and Looping

AIM:

To solve real-world problems using decision-making statements (if, elif, else) and looping constructs (for, while).

PROCEDURE:

Step 1. Accept user input for the problem.
Step 2. Use decision-making statements to select appropriate conditions.
Step 3. Apply looping constructs (for or while) to implement repetition.
Step 4. Display results.

PROGRAM (Example 1: Check Prime Number):

```
# Problem-solving using decision-making and looping
num = int(input("Enter a number: "))
if num <= 1:
    print(f"{num} is not a prime number")
else:
    is_prime = True
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(f"{num} is a prime number")
    else:
        print(f"{num} is not a prime number")
```

PROGRAM (Example 2: Factorial using loop):

```
num = int(input("Enter a number: "))
fact = 1
for i in range(1, num + 1):
    fact *= i
print(f"Factorial of {num} = {fact}")
```

SAMPLE OUTPUT:

```
Enter a number: 7
7 is a prime number
Enter a number: 5
Factorial of 5 = 120
```

Experiment 5:

Programs using User-Defined and Lambda Functions

AIM:

To write programs that use user-defined functions and anonymous (lambda) functions for computations.

PROCEDURE:

Step 1. Define user-defined functions using def keyword.
Step 2. Call functions with appropriate arguments.
Step 3. Implement lambda functions for quick computations.
Step 4. Display the results.

PROGRAM (User-Defined Functions):

```
# User-defined functions
def add(x, y):
    return x + y

def maximum(x, y):
    return x if x > y else y

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print("Addition:", add(a, b))
print("Maximum:", maximum(a, b))
```

PROGRAM (Lambda Functions):

```
# Lambda functions
square = lambda x: x * x
cube = lambda x: x * x * x
greater = lambda a, b: a if a > b else b
fact=lambda x: x*fact(x-1) if x>0 else 1
print("Square of 5:", square(5))
print("Cube of 3:", cube(3))
print("Greater of 10 and 20:", greater(10, 20))
print(“Factorial of 5 is: ”,fact(5))
```

SAMPLE OUTPUT:

```
Enter first number: 15
Enter second number: 10
Addition: 25
Maximum: 15
Square of 5: 25
Cube of 3: 27
Greater of 10 and 20: 20
Factorial of 5 is 120
```

Experiment 6:

Recursive Functions – Factorial and Fibonacci Series

AIM:

To implement recursive functions in Python to calculate the factorial of a number and generate the Fibonacci series.

PROCEDURE:

Step 1. Define recursive functions using def.
Step 2. For factorial, base case: factorial(0) = 1.
Step 3. For Fibonacci, base cases: fibonacci(0) = 0, fibonacci(1) = 1.
Step 4. Call recursive functions and display results.

PROGRAM (Factorial using Recursion):

```
# Recursive function for factorial
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
num = int(input("Enter a number: "))
print(f"Factorial of {num} = {factorial(num)}")
```

PROGRAM (Fibonacci using Recursion):

```
# Recursive function for Fibonacci series
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

terms = int(input("Enter number of terms: "))
print(f"Fibonacci Series up to {terms} terms:")
for i in range(terms):
    print(fibonacci(i), end="")
```

OUTPUT:

```
Enter a number: 5
Factorial of 5 = 120
Enter number of terms: 7
Fibonacci Series up to 7 terms:
0 1 1 2 3 5 8
```

Experiment 7:

Programs on String Operations

AIM:

To write Python programs that perform various string operations.

PROCEDURE:

Step 1. Accept a string input from the user.
Step 2. Perform basic operations like concatenation, slicing, and repetition.
Step 3. Apply built-in methods like upper(), lower(), replace(), find().
Step 4. Display the results.

PROGRAM:

```
# Programs on String Operations
s = input("Enter a string: ")
# Basic operations
print("\n Basic Operations ")
print("Original string:", s)
print("First 5 characters:", s[:5])
print("Last 3 characters:", s[-3:])
print("Repetition:", s * 2)
print("Concatenation:", s + " Python")
# Built-in methods
print("\n String Methods ")
print("Uppercase:", s.upper())
print("Lowercase:", s.lower())
print("Title Case:", s.title())
print("Replace 'a' with '@':", s.replace('a', '@'))
print("Find 'the':", s.find("the"))
print("Is Alphanumeric:", s.isalnum())
```

OUTPUT:

```
Enter a string: postgraduate python lab
 Basic Operations
Original string: postgraduate python lab
First 5 characters: postg
Last 3 characters: lab
Repetition: postgraduate python labpostgraduate python lab
Concatenation: postgraduate python lab Python
String Methods
Uppercase: POSTGRADUATE PYTHON LAB
Lowercase: postgraduate python lab
Title Case: Postgraduate Python Lab
Replace 'a' with '@': postgr@du@te python l@b
Find 'the': -1
Is Alphanumeric: False
```

Experiment 8:

Programs on List, Tuple, Set, and Dictionary

AIM:

To implement Python programs that demonstrate operations on list, tuple, set, and dictionary.

PROCEDURE:

Step 1. Define examples of list, tuple, set, and dictionary.
Step 2. Perform insertion, deletion, and access operations.
Step 3. Apply built-in methods specific to each type.
Step 4. Display results.

PROGRAM:

```
# Programs on List, Tuple, Set, and Dictionary
# List
print("\n List Operations ")
lst = [10, 20, 30]
lst.append(40)
lst.remove(20)
print("List:", lst)
print("Slicing:", lst[1:3])

# Tuple
print("\n Tuple Operations ")
tup = (1, 2, 3, 4)
print("Tuple:", tup)
print("Accessing element tup[2]:", tup[2])

# Set
print("\n  Set Operations ")
st = {10, 20, 30, 40}
st1 = {30, 40,60,80,100}
st.add(50)
st.discard(20)
print("Set:", st)
print("Set 1: ",st1)
print("Union:", st.union(st1))
print("Union(|):", st |st1)
print("Intersection:", st.intersection(st1))
print("Intersection(&):", st & st1)
print("Difference:", st.difference(st1))
print("Difference(-):", st-st1)
print("Symmetric_difference:", st.symmetric_difference(st1))
print("Symmetric_difference(^):", st^st1)
```

 # Dictionary

```
print("\n--- Dictionary Operations ---")
dct = {"name": "Alice", "age": 24}
dct["course"] = "M.Sc Data Science"
print("Dictionary:", dct)
print("Keys:", dct.keys())
print("Values:", dct.values())
print("Access name:", dct["name"])
```

OUTPUT:

```
--- List Operations ---
List: [10, 30, 40]
Slicing: [30, 40]
--- Tuple Operations ---
Tuple: (1, 2, 3, 4)
Accessing element tup[2]: 3
--- Set Operations ---
Set: {40, 10, 50, 30}
Set 1:  {80, 100, 40, 60, 30}
Union: {100, 40, 10, 80, 50, 60, 30}
Union(|): {100, 40, 10, 80, 50, 60, 30}
Intersection: {40, 30}
Intersection(&): {40, 30}
Difference: {10, 50}
Difference(-): {10, 50}
Symmetric_difference: {80, 50, 100, 10, 60}
Symmetric_difference(^): {80, 50, 100, 10, 60}
--- Dictionary Operations ---
Dictionary: {'name': 'Alice', 'age': 24, 'course': 'M.Sc Data Science'}
Keys: dict_keys(['name', 'age', 'course'])
Values: dict_values(['Alice', 24, 'M.Sc Data Science'])
Access name: Alice
```

Experiment 9:

File Handling (Text and Binary Files) with CSV Read/Write

AIM:

To demonstrate file handling operations in Python including text, binary, and CSV files.

PROCEDURE:

Step 1. Open a text file using open() in read/write mode and perform operations.
Step 2. Use pickle for binary file operations.
Step 3. Use the csv module to read and write CSV files.
Step 4. Close files properly after operations.

PROGRAM (Text File Handling):

```
# Text File Handling
with open("sample.txt", "w") as f:
    f.write("Hello PG Students!\nWelcome to Python Lab.")
with open("sample.txt", "r") as f:
    content = f.read()
    print("\n  Text File Content \n", content)
```

PROGRAM (Binary File Handling with Pickle):

```
import pickle
data = {"name": "Alice", "age": 24, "course": "Data Science"}
# Write to binary file
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)
# Read from binary file
with open("data.pkl", "rb") as f:
    obj = pickle.load(f)
    print("\n  Binary File Content \n", obj)
```

PROGRAM (CSV File Handling):

```
import csv
# Write to CSV file
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Age", "Course"])
    writer.writerow(["Alice", 24, "Data Science"])
    writer.writerow(["Bob", 23, "AI & ML"])

# Read from CSV file
with open("students.csv", "r") as f:
    reader = csv.reader(f)
    print("\n  CSV File Content  ")
    for row in reader:
        print(row)
```

OUTPUT:

```
 Text File Content
 Hello PG Students!
Welcome to Python Lab.

 Binary File Content
 {'name': 'Alice', 'age': 24, 'course': 'Data Science'}

 CSV File Content
['Name', 'Age', 'Course']
['Alice', '24', 'Data Science']
['Bob', '23', 'AI & ML']
```

Experiment 10:

R Programs for Vector, Matrix, and Array Manipulations

AIM:

To write R programs demonstrating operations on vectors, matrices, and arrays.

PROCEDURE:

Step 1. Define vectors and apply arithmetic and indexing operations.

Step 2. Create matrices using the matrix() function and perform addition, multiplication, transpose, and inversion.

Step 3. Create arrays using the array() function and access elements.

PROGRAM:

```
# Vector operations
v1 <- c(10, 20, 30, 40)
v2 <- c(1, 2, 3, 4)
cat("\n--- Vector Operations ---\n")
print(v1 + v2)
print(v1 * v2)
print(v1-v2)
print(v1 / v2)
print(v1[2:4])
print(v1[3:3])
# Matrix operations
m1 <- matrix(1:6, nrow=2, ncol=3)
print(m1)
m2 <- matrix(7:12, nrow=2, ncol=3)
print(m2)
cat("\n--- Matrix Operations ---\n")
print(m1 + m2)
print(t(m1))  # Transpose
# Array operations
arr <- array(1:24, dim=c(3,4,2))
cat("\n--- Array Elements ---\n")
print(arr[,,1])
print(arr[,,2])
print(arr)
```

OUTPUT:

```
--- Vector Operations ---
[1] 11 22 33 44
[1]  10  40  90 160
[1]  9 18 27 36
[1] 10 10 10 10
[1] 20 30 40
[1] 30
      [,1] [,2] [,3]
[1,]    1    3    5
```

```
[2,]    2    4    6
      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

--- Matrix Operations ---
       [,1] [,2] [,3]
[1,]    8   12   16
[2,]   10   14   18
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6

--- Array Elements ---
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
       [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
, , 1

      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

       [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Experiment 11:

Operations using Control Structures and DataFrames

AIM:

To implement R programs using control structures (if, for, while) and perform operations on DataFrames.

PROCEDURE:

Step 1. Use if-else, for, and while for decision-making and iteration.
Step 2. Create a DataFrame using data.frame().
Step 3. Access rows, columns, and apply operations like filtering.

PROGRAM:

```
# Control structures
num <- 7
cat("\n--- If-Else Example ---\n")
if (num %% 2 == 0)
{
  print("Even number")
}
else
{
  print("Odd number")
}
cat("\n--- For Loop Example ---\n")
for (i in 1:5)
{
  print(i^2)
}
cat("\n--- While Loop Example ---\n")
x <- 1
while (x <= 5)
{
  print(x)
  x <- x + 1
}
# DataFrame operations
cat("\n--- DataFrame Example ---\n")
students <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(24, 23, 22),
  Marks = c(85, 90, 78)
)
print(students)
```

```
# Select column
print(students$Name)
# Filter rows
print(subset(students, Marks > 80))
```

OUTPUT:

```
--- If-Else Example ---
[1] "Odd number"
--- For Loop Example ---
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25

--- While Loop Example ---
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5

--- DataFrame Example ---

     Name Age Marks
1   Alice  24    85
2     Bob  23    90
3 Charlie  22    78

[1] "Alice""Bob""Charlie"

     Name Age Marks
1   Alice  24    85
2     Bob  23    90
```

Experiment 12:

Programs for Missing Value Handling and Transformations

AIM:

To demonstrate R programs for handling missing values (NA) and performing data transformations.

PROCEDURE:

Step 1. Create a vector or DataFrame with missing values (NA).
Step 2. Use functions like is.na(), na.omit(), mean() with na.rm=TRUE.
Step 3. Apply data transformations such as scaling and normalization.

PROGRAM:

```
# Missing values
vec <- c(10, 20, NA, 40, NA, 60)
cat("\n--- Missing Values ---\n")
print(vec)
# Identify missing values
print(is.na(vec))
# Remove missing values
print(na.omit(vec))
# Replace NA with mean
vec[is.na(vec)] <- mean(vec, na.rm=TRUE)
print(vec)
# Data transformations
data <- c(5, 10, 15, 20, 25)
cat("\n--- Transformations ---\n")
# Normalization (0-1 scaling)
norm <- (data - min(data)) / (max(data) - min(data))
print(norm)
# Log transformation
print(log(data))
# Scaling (mean 0, sd 1)
scaled <- scale(data)
print(scaled)
```

OUTPUT:

```
--- Missing Values ---
[1] 10 20 NA 40 NA 60
[1] FALSE FALSE TRUE FALSE TRUE FALSE
[1] 10 20 40 60
[1] 10 20 32.5 40 32.5 60
```

--- Transformations ---
[1] 0.00 0.25 0.50 0.75 1.00
[1] 1.609 2.303 2.708 2.996 3.219

```
           [,1]
[1,] -1.2649111
[2,] -0.6324555
[3,]  0.0000000
[4,]  0.6324555
[5,]  1.2649111
```

Experiment 13:

Implementing R Functions

AIM:

To write and execute user-defined functions in R for modular programming.

PROCEDURE:

Step 1. Define a function using the syntax fun_name <- function(arguments){...}.
Step 2. Call the function with appropriate inputs.
Step 3. Return results using the return() statement.
Step 4. Test functions with different inputs.

PROGRAM:

```
# Function to calculate factorial
factorial_fun <- function(n) {
  if (n == 0) {
    return(1)
  } else {
    return(n * factorial_fun(n - 1))
  }
}
# Function to calculate mean
mean_fun <- function(vec) {
  return(sum(vec) / length(vec))
}
# Function call
cat("\nFactorial of 5:", factorial_fun(5), "\n")
cat("Mean of c(10,20,30,40):", mean_fun(c(10,20,30,40)), "\n")
```

OUTPUT:

```
Factorial of 5: 120
Mean of c(10,20,30,40): 25
```

Experiment 14:

Implement Plot Functions with ggplot2

AIM:

To create various plots in R using the ggplot2 package.

PROCEDURE:

Step 1. Install and load the ggplot2 library.
Step 2. Create a DataFrame with sample data.
Step 3. Use ggplot() with geom_point(), geom_bar(), geom_line() for plots.
Step 4. Add titles, axis labels, and color mapping.

PROGRAM:

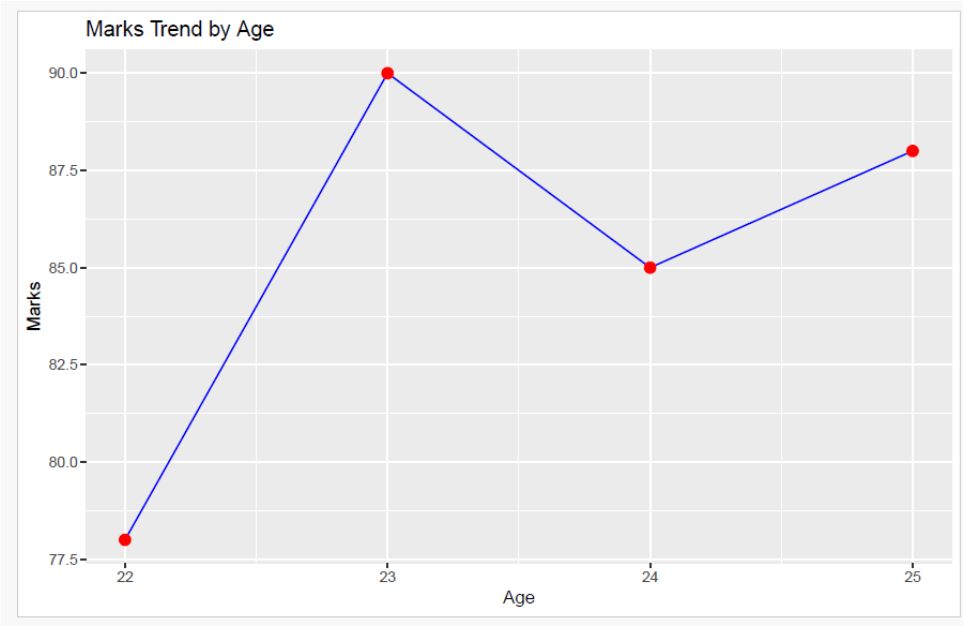
```
# Install package if not installed
# install.packages("ggplot2")
library(ggplot2)
# Sample data
data <- data.frame(
  Name = c("Alice", "Bob", "Charlie", "David"),
  Marks = c(85, 90, 78, 88),
  Age = c(24, 23, 22, 25)
)
# Bar plot
ggplot(data, aes(x=Name, y=Marks, fill=Name)) +
  geom_bar(stat="identity") +
  ggtitle("Marks of Students")
# Scatter plot
ggplot(data, aes(x=Age, y=Marks, color=Name)) +
  geom_point(size=3) +
  ggtitle("Marks vs Age")
# Line plot
ggplot(data, aes(x=Age, y=Marks)) +
  geom_line(color="blue") +
  geom_point(color="red", size=3) +
  ggtitle("Marks Trend by Age")
```

OUTPUT (Visuals):

A bar chart showing student names vs marks.

A scatter plot showing relationship between Age and Marks.

A line plot showing trend of Marks with Age.



Experiment 15:

Visualizing Geometric Shapes in R

AIM:

To visualize geometric shapes such as circles, rectangles, and polygons in R.

PROCEDURE:

Step 1. Use plot() function to create blank coordinate space.
Step 2. Draw rectangles using rect().
Step 3. Draw circles using parametric equations with lines().
Step 4. Draw polygons using polygon().

PROGRAM:

```
# Blank plot
plot(0, 0, xlim=c(-10, 10), ylim=c(-10, 10), type="n", main="Geometric Shapes")
# Rectangle
rect(-5, -2, 5, 2, border="blue", col="lightblue")
# Circle
theta <- seq(0, 2*pi, length=200)
x <- 3 * cos(theta)
y <- 3 * sin(theta)
lines(x, y, col="red", lwd=2)
# Triangle (Polygon)
polygon(c(-4, 0, 4), c(-5, -9, -5), col="green", border="darkgreen")
```

OUTPUT (Visuals):

A rectangle centered on the X-axis.
A circle drawn using sine and cosine.
A triangle (polygon) below the rectangle.