Ex 1.1 **Find given number is odd or even.**
Date:

Aim: To Create a C program to find a given number is odd or even.

Procedure:

Step 1: Include the necessary header file:

Add #include <stdio.h> to include the standard input/output library.

Step 2: Declare variables:
Declare an integer variable num to store the user input.

Step 3: Get user input:
Use printf to prompt the user to enter a number.

Use scanf to read the input and store it in the num variable.

Step 4: Check if the number is odd or even:

Use an if-else statement to check the condition:

If num % 2 == 0, the number is even.

If num % 2 != 0, the number is odd.

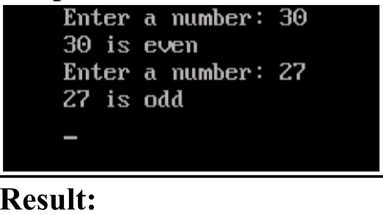
Step 5: Print the result
Use printf to display whether the number is odd or even.

Step 6: Return from the main function:
Use return 0; to indicate successful program execution.

Code:

```
#include <stdio.h>
int main()
{
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num % 2 == 0)
{
      printf("%d is even.\n", num);
   }
 Else
 {
      printf("%d is odd.\n", num);
   }
   return 0;
}
```

Output:



```
Enter a number: 30
30 is even
Enter a number: 27
27 is odd
_
```

Result:

Thus the program was executed successfully and the user-input number is odd or even is displays the result.

Ex 1.2 **FIND THE LARGEST NUMBER**
Date:

Aim:

To write a C program that finds and displays the largest of three given numbers.

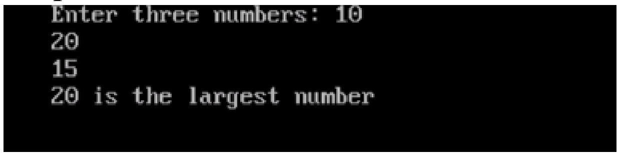
Procedure:

Step 1: Start the Program.
Step 2: Declare three integer variables a, b, and c to store the three input numbers.
Step 3: Prompt the user to enter three integers.
Step 4: Read the input values into variables a, b, and c.
Step 5: Compare the Numbers:
Step 6: Use if-else statements to compare the three numbers:
Step 7: Check if a is greater than or equal to both b and c.
Step 8: If true, a is the largest number.
Step 9: If not, check if b is greater than or equal to both a and c.
Step 10: If true, b is the largest number.
Step 11: If neither of the above conditions is true, c is the largest number.
Step 12: Display the Result:
Step 13: Print the largest number.
Step 14: End the Program.

Code:

```c
#include <stdio.h>
int main()
{
   int a, b, c;
   printf("Enter three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   if (a >= b && a >= c) {
      printf("%d is the largest number.\n", a);
   } else if (b >= a && b >= c) {
      printf("%d is the largest number.\n", b);
   } else {
      printf("%d is the largest number.\n", c);
   }
   return 0;
}
```

Output:



```
Enter three numbers: 10
20
15
20 is the largest number
```

Result:

Thus the Program was displayed the largest of given numbers Successfully.

Ex 1.3 **Factorial**

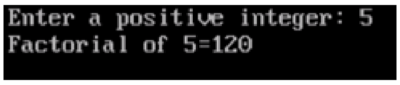
Aim: To write a C program that calculates the factorial of a given number.

Procedure:

Step 1: Start the Program
Step 2: Declare a variable factorial to hold the result of the factorial calculation, initialized to 1.
Step 3: Optionally, declare a variable i for loop iteration.
Step 4: Prompt the user to enter a non-negative integer.
Step 5: Read the input value into the variable n.
Step 6: If n is less than 0, print a message indicating that the factorial is not defined for negativ numbers.
Step 7: If n is 0, set factorial to 1 (since 0! = 1).
Step 8: Use a for loop to iterate from 1 to n:
Step 9: Multiply factorial by the loop variable i in each iteration.
Step 10: Update the value of factorial.
Step 11: Print the calculated factorial value.
Step 12: End the Program: Terminate the program execution.

Code:
```
#include <stdio.h>
int main()
{
   int n, i;
   unsigned long long factorial = 1;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   if (n < 0)
  {
      printf("Factorial is not defined for negative numbers.\n");
   }
else
{
      for (i = 1; i <= n; ++i)
{
         factorial *= i;  // factorial = factorial * i
 }
      printf("Factorial of %d = %llu\n", n, factorial);
 }
   return 0;
}
```

Output:


```
Enter a positive integer: 5
Factorial of 5=120
```

Result:
The program effectively calculates and displays the factorial of a given number, demonstrating the use of control statements and functions in C programming.

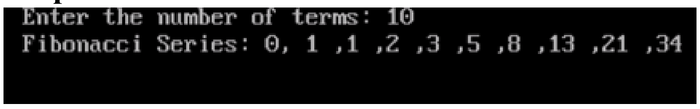
Ex 1.4 **FIBONACCI SERIES**
Date:

Aim:
To write a C program that prints the Fibonacci series up to a specified number of terms.

Procedure:

Step 1: Start the Programme
Step 2: Declare the header for input and output functions.
Step 3: Declare the main function.
Step 4: Declare Variables n: to store the number of terms in the Fibonacci series.
Step 5: t1 and t2: to store the first two terms of the series (initially set to 0 and 1).
Step 6: nextTerm: to store the next term in the series.
Step 7: Get User Input
Step 8: Use scanf to read the input.
Step 9: Print the Initial Terms
Step 10: Calculate the Fibonacci Series
Step 11: Use a loop to calculate the next terms in the series:
Step 12: Start a for loop from 3 to n (inclusive).
Step 13: Inside the loop, calculate the next term by adding t1 and t2, Print the next term.
Step 14: Update t1 and t2 for the next iteration.
Step 15: Compile and Run the Program to Run the executable to see the output.

Code:
```
#include <stdio.h>
int main() {
   int n, t1 = 0, t2 = 1, nextTerm;
   printf("Enter the number of terms: ");
   scanf("%d", &n);
   printf("Fibonacci Series: %d, %d, ", t1, t2);
   nextTerm = t1 + t2;
   for (int i = 3; i <= n; ++i) {
      printf("%d, ", nextTerm);
      t1 = t2;
      t2 = nextTerm;
      nextTerm = t1 + t2;
   }
   return 0;
}
```

Output:

```
Enter the number of terms: 10
Fibonacci Series: 0, 1 ,1 ,2 ,3 ,5 ,8 ,13 ,21 ,34
```

Result:
The program successfully prints the Fibonacci series up to the specified number of terms. The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1 in C programming.

Ex 1.5 **CALCULATION OF SIMPLE INTEREST**
Date:

Aim: To write a C program that calculates simple interest.

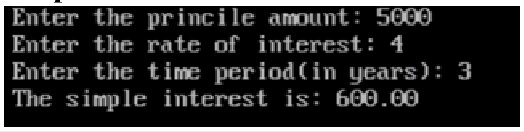
Procedure:

Step 1: Start the code.
Step 2: Define the function to create a function that takes three parameters (principal, rate, and time) and returns the calculated simple interest.
Step 3: Implement the formula inside the function, apply the simple interest formula.
Step 4: Declare variables in the main function, declare variables for principal, rate, time, and interest.
Step 5: Get the user input
Step 6: Call the function pass the user inputs to the function to calculate the simple interest.
Step 7: Display the result: Print the calculated simple interest.

Code:

```c
#include <stdio.h>
float calculateSimpleInterest(float principal, float rate, float time)
{
   return (principal * rate * time) / 100;
}
int main()
{
   float principal, rate, time, interest;
   // User input
   printf("Enter the principal amount: ");
   scanf("%f", &principal);
   printf("Enter the rate of interest: ");
   scanf("%f", &rate);
   printf("Enter the time period (in years): ");
   scanf("%f", &time);
   // Function call to calculate simple interest
   interest = calculateSimpleInterest(principal, rate, time);
   // Display the result
   printf("The simple interest is: %.2f\n", interest);
   return 0;
}
```

Output:



```
Enter the princile amount: 5000
Enter the rate of interest: 4
Enter the time period(in years): 3
The simple interest is: 600.00
```

Result:

Thus, the C program successfully calculates the simple interest based on the provided inputs for principal, rate of interest, and time period.

Ex 2.1 **CLASS WITH ATTRIBUTES**
Date:

Aim:

To write program is to demonstrate class with attributes and a method in C++.

Procedure:

Step 1: **Include Libraries**: The program begins by including the iostream and string libraries.

Step 2: **Define Class**: A class MyClass is defined with two attributes (num and str) and a method display().

Step 3: **Declare Attributes**: The attributes num (integer) and str (string) are declared inside the class.

Step 4: **Define Method**: The method display() is defined to print the values of num and str.

Step 5: **Create Object**: An object obj of the class MyClass is created inside the main() function.

Step 6: **Assign Values**: The values 10 and "Hello, World!" are assigned to the attributes num and str of the object.

Step 7: **Call Method**: The display() method is called to print the assigned values to the console.

Program

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class MyClass
{
public:
   int num;
   char str[50];
   void display() {
      cout << "Number: " << num << ", String: " << str << endl;
   }
};
int main()
{
   clrscr();
   MyClass obj;
   obj.num = 10;
   strcpy(obj.str, "Hello, World!");
   obj.display();
   getch();
   return 0;
}
```

Output:

```
Number: 10, String: Hello, World!
```

Result: Thus, the C++ program for creating a class with attributes and a method in C++ was executed successfully.

Ex 2.2 **Implementing Inheritance in C++**

Date:

Aim:

To create a C++ code to implement inheritance concept.

Procedure:

Step 1:Include Libraries: The program begins by including the iostream library for input/output operations.

Step 2:Define Base Class: A class Base is defined with a method show() that prints "Base class method".

Step 3:Define Derived Class: A class Derived is defined, which publicly inherits from the Base class.

Step 4:Declare Derived Class Method: The Derived class contains its own method display(), which prints "Derived class method".

Step 5:Create Derived Class Object: In the main() function, an object obj of the Derived class is created.

Step 6:Call Base Class Method: The method show() from the Base class is called using the Derived class object.

Step 7:Call Derived Class Method: The method display() from the Derived class is called using the same object.

Code:

```
#include <iostream.h>
#include <conio.h>
class Base
{
public:
   void show()
{
      cout << "Base class method" << endl;    }};
class Derived : public Base
{
public:
   void display()
{
      cout << "Derived class method" << endl;    }};
int main()
{
   clrscr();
   Derived obj;
   obj.show();
   obj.display();
   getch();
   return 0;
}
```

Output:

```
Base class method
Derived class method
```

Result:

Thus, the C++ program successfully demonstrates the concept of inheritance by allowing the derived class to inherit properties and methods from the base class.

Ex 2.3 **VIRTUAL FUNCTIONS AND METHOD OVERRIDING IN C++**
Date:

Aim:

To create a C++ program that demonstrates virtual functions and method overriding for runtime polymorphism.

Procedure:

Step 1:Include Libraries: The program starts by including the iostream library for input/output operations.
Step 2:Define Base Class: The Base class is defined with a virtual function show() that prints "Base class show".
Step 3:Declare Virtual Function: The show() method in the Base class is marked as virtual, allowing it to be overridden by derived classes.
Step 4:Define Derived Class: The Derived class is created, inheriting from Base, and it overrides the show() method with its own implementation.
Step 5:Create Pointer: Inside the main() function, a pointer of type Base is created and initialized to point to an object of the Derived class.
Step 6:Call Overridden Method: The show() method is called using the Base class pointer, which actually invokes the Derived class's overridden version due to polymorphism.
Step 7:Clean Up Memory: The delete operator is used to free up the dynamically allocated memory.

CODE :

```
#include <iostream.h>
#include <conio.h>
class Base
{
public:
   virtual void show()
   {
      cout << "Base class show" << endl;
   }
};

class Derived : public Base
{
public:
   void show() {
      cout << "Derived class show" << endl;
   }
};
int main()
{
   clrscr();
   Base* b = new Derived();
   b->show();
   delete b;
   getch();
   return 0;
}
```

Output: Derived class show

Result: The program shows runtime polymorphism in C++ by using a base class pointer to call the derived class's overridden method with virtual functions.

Ex 2.4 **Encapsulation in C++**
Date:

Aim:
To write a C++ program to demonstrate the concept of encapsulation.

Procedure:

Step 1:Include Libraries: The program begins by including the iostream library for input/output operations.
Step 2:Define Class: A class Encapsulated is defined with a private attribute value.
Step 3:Private Attribute Declaration: The value attribute is kept private, so it cannot be directly accessed or modified from outside the class.
Step 4:Create Setter Method: A public method setValue() is defined to allow controlled modification of the value attribute.
Step 5:Create Getter Method: A public method getValue() is defined to provide controlled access to the value of the private attribute.
Step 6:Create Object and Set Value: In the main() function, an object of the class Encapsulated is created, and the setValue() method is called to set the value of the private attribute.
Step 7:Access and Display Value: The getValue() method is called to retrieve and display the value of the private attribute using std::cout.

Code

```
 #include <iostream.h>
#include <conio.h>
class Encapsulated
{
private:
   int value;
public:
   void setValue(int v) {
      value = v;    }
   int getValue()
{
      return value;  }
};
int main()
{
   clrscr();
   Encapsulated obj;
   obj.setValue(42);
   cout << "Value: " << obj.getValue() << endl;
   getch();
   return 0;
}
```

Output: Value: 42

Result: Thus the C++ program successfully execute the concept of encapsulation.

Ex 2.5 **Abstract Classes and Pure Virtual Functions in C++**
Date:

Aim:

To write a C++ program that demonstrates abstract classes and pure virtual functions, enabling derived classes to implement inherited methods.

Procedure:

Step 1:Include Libraries: The program begins by including the iostream library for input/output operations.
Step 2:Define Abstract Class: A class AbstractShape is defined with a pure virtual function draw(), making it an abstract class.
Step 3:Declare Pure Virtual Function: The draw() function is a pure virtual function, defined as virtual void draw() = 0;, meaning any derived class must implement it.
Step 4:Define Derived Class: A class Circle is defined, inheriting from AbstractShape, and it overrides the draw() method with its own implementation.
Step 5:Implement the Method:S The Circle class provides the actual implementation of the draw() method, printing "Drawing Circle".
Step 6:Create Object: In the main() function, an object circle of the Circle class is created.
Step 7:Call the Method: The draw() method of the Circle class is called using the object, which prints the message.

code

```
#include <iostream.h>
#include <conio.h>
class AbstractShape
{
public:
   virtual void draw() = 0;
};
class Circle : public AbstractShape
{
public:
   void draw()
{
      cout << "Drawing Circle" << endl;
   }
};
int main() {
   clrscr();
   Circle circle;
   circle.draw();
   getch();
   return 0;
}
```

Output:

```
Drawing Circle
```

Result:

Thus, the C++ program successfully demonstrates abstract classes and pure virtual functions with proper method implementation in derived classes.

3.1.1 **Input and Output Statements**
Date:

Aim:
To demonstrate the use of input and output statements in Python.

Procedure:
Step 1: Use `input()` to take user input.
Step 2: Use `print()` to display output.

Program
```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

Output:
```
Enter your name: BABU
Hello, BABU!
```

Result:
The program successfully takes user input and displays a greeting message.

Ex 3.1.2 **Sum and Multiply All Numbers in a List**
Date:

Aim:
To calculate the sum and product of all numbers in a list.

Procedure:
Step 1: Create a list of numbers.
Step 2: Use a loop to calculate the sum and product.

Program
```
def sum_and_multiply(numbers):
    total_sum = sum(numbers)
    total_product = 1
    for num in numbers:
        total_product *= num
    return total_sum, total_product

numbers = [1, 2, 3, 4]
sum_result, product_result = sum_and_multiply(numbers)
print(f"Sum: {sum_result}, Product: {product_result}")
```

Output:
```
Sum: 10, Product: 24
```

Result:
The program computes the sum and product of the list elements correctly.

Ex 3.1.3 **Count Upper and Lower Case Letters**
Date:

Aim:
To count the number of uppercase and lowercase letters in a string.

Procedure
Step 1: Take a string input from the user.
Step 2: Use loops to count uppercase and lowercase letters.

Program
```
def count_case(s):
    upper_count = sum(1 for c in s if c.isupper())
    lower_count = sum(1 for c in s if c.islower())
    return upper_count, lower_count

string_input = "Hello World"
upper_count, lower_count = count_case(string_input)
print(f"Uppercase letters: {upper_count}, Lowercase letters: {lower_count}")
```

Output
```
Uppercase letters: 2, Lowercase letters: 8
```

Result:
The program accurately counts the number of uppercase and lowercase letters.

Ex 3.1.4 **Display Distinct Elements from a List**
Date:

Aim:

To display distinct elements from a list that contains duplicates.

Procedure:

Step 1: Create a list with duplicate elements.
Step 2: Convert the list to a set to remove duplicates.

Program

```
def distinct_elements(lst):
    return list(set(lst))

duplicate_list = [1, 2, 2, 3, 4, 4]
distinct_list = distinct_elements(duplicate_list)
print(f"Distinct elements: {distinct_list}")
```

Output:

```
Distinct elements: [1, 2, 3, 4]
```

Result:

The program successfully displays distinct elements from the list.

Ex 3.1.5 **Factorial of a Given Number**
Date:

Aim:
To calculate the factorial of a given number using recursion.

Procedure:
Step 1: Define a recursive function to compute factorial.
Step 2: Take an integer input from the user.

Program
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

number = int(input("Enter a number to find its factorial: "))
fact_result = factorial(number)
print(f"Factorial of {number} is {fact_result}")
```

Output
```
Enter a number to find its factorial: 5
Factorial of 5 is 120
```

Result:
The program calculates the factorial correctly using recursion.

Ex 3.1.6 **Check if Given Number is Odd or Even**
Date:

Aim:
To check whether a given number is odd or even.

Procedure
Step 1: Take an integer input from the user.
Step 2: Use conditional statements to check if it's odd or even.

Program
```
def check_odd_even(num):
   return "Even" if num % 2 == 0 else "Odd"

number = int(input("Enter a number: "))
result = check_odd_even(number)
print(f"The number {number} is {result}.")
```

Output
```
Enter a number: 7
The number 7 is Odd.
```

Result
The program correctly identifies whether the number is odd or even.

Ex 3.1.7 **Display the Sum of Given Numbers**
Date:

Aim:
To display the sum of multiple numbers provided by the user.

Procedure
Step 1: Take multiple inputs from the user.
Step 2: Convert inputs into integers and calculate their sum.

Program

```
def sum_of_numbers(*args):
   return sum(args)

numbers_input = input("Enter numbers separated by space: ")
numbers_list = map(int, numbers_input.split())
total_sum = sum_of_numbers(*numbers_list)
print(f"The sum of given numbers is: {total_sum}")
```

Output:
```
Enter numbers separated by space: 3 5 7
The sum of given numbers is: 15
```

Result:
The program calculates the sum of multiple user-provided numbers successfully.

Ex 3.1.8 **Add Two Numbers Using Functions**
Date:

Aim:
To create a function that adds two numbers provided by the user.

Procedure

Step 1: Define an addition function.
Step 2: Take two inputs from the user and call the function with those inputs.

Program

```
def add_two_numbers(a, b):
    return a + b

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
sum_result = add_two_numbers(num1, num2)
print(f"The sum of {num1} and {num2} is {sum_result}.")
```

Output:

```
Enter first number: 10
Enter second number: 20
The sum of 10 and 20 is 30.
```

Result:
The program successfully adds two numbers using functions.

Ex 3.2 **Person Information Using Python Classes and Objects**
Date:

Aim:
To Create a Python class and objects to represent a person's information.

Procedure:

Step 1:Define the class: Create a class named Person with attributes like name, age, and city.

Step 2:Initialize the class: Implement the _init_ method to assign values to the attributes during object creation.

Step 3:Create objects: Instantiate multiple Person objects with different values.

Step 4:Access attributes: Use dot notation to access the attributes of each object.

Step 5:Modify attributes: Change the values of attributes for specific objects.

Step 6:Create methods: Define methods within the class to perform actions on the objects, such as calculating age difference or printing information.

Step 7:Call methods: Invoke the methods on the objects to execute their functionality.

code

```
class Person:
   def __init__(self, name, age, city):
      self.name = name
      self.age = age
      self.city = city
   def print_info(self):
      print(f"Name: {self.name}, Age: {self.age}, City: {self.city}")
person1 = Person("Alice", 25, "New York")
person2 = Person("Bob", 30, "Los Angeles")
print(person1.name)
person2.age = 31
person1.print_info()

person2.print_info()
```

Output:

```
Alice
Name: Alice, Age: 25, City: New York
Name: Bob, Age: 31, City: Los Angeles
```

Result:
The code will output the information for each person, including their name, age, and city. Successfully executed.

Ex 3.3 **Python Class Hierarchy and Inheritance**
Date:

Aim:
To create a Python class hierarchy with inheritance and use constructors to initialize objects.

Procedure:

Step 1:Define a base class: Create a class named Animal with common attributes and methods for all animals.
Step 2:Define derived classes: Create derived classes like Dog, Cat, and Bird that inherit from the Animal class.
Step 3:Override methods: Modify the inherited methods in the derived classes to provide specific implementations for each animal.
Step 4:Add specific attributes: Include additional attributes in the derived classes to represent unique characteristics of each animal.
Step 5:Call the base class constructor: Use super().__init__() in the derived class constructors to initialize the inherited attributes.
Step 6:Create objects: Instantiate objects of the derived classes.
Step 7:Access attributes and methods: Use dot notation to access and call the attributes and methods of the objects.

Code

```
class Animal:
   def __init__(self, name, sound):
      self.name = name
      self.sound = sound
   def make_sound(self):
      print(f"{self.name} says {self.sound}")
class Dog(Animal):
   def __init__(self, name, sound, breed):
      super().__init__(name, sound)
      self.breed = breed
class Cat(Animal):
   def __init__(self, name, sound, favorite_food):
      super().__init__(name, sound)
      self.favorite_food = favorite_food
dog = Dog("Buddy", "Woof", "Golden Retriever")
cat = Cat("Whiskers", "Meow", "Tuna")
print(dog.name, dog.breed)
dog.make_sound()
print(cat.name, cat.favorite_food)
cat.make_sound()
```

Output:

```
Buddy Golden Retriever
Buddy says Woof
Whiskers Tuna
Whiskers says Meow
```

Result:
The code will output the information for each animal, including their name, breed (for dog), favorite food (for cat), and the sound they make. **Successfully executed.**

Ex 3.4 **Understanding the Self Parameter in Python Classes**
Date:

Aim:

To understand how to use the self parameter in Python classes to access and reference object attributes and methods.

Procedure:

Step 1: **Define a class:** Create a class with attributes and methods.
Step 2: **Use self in methods:** Within methods, use self to access and modify the object's attributes.
Step 3: **Pass self to methods:** When calling methods on an object, self is automatically passed as the first argument.
Step 4: **Create objects:** Instantiate objects of the class.
Step 5: **Call methods on objects:** Invoke methods on the objects to perform actions.
Step 6: **Observe attribute changes:** Verify that the changes made within methods are reflected in the object's attributes.
Step 7: **Understand the concept of instance variables:** Recognize that attributes accessed using self are instance variables specific to each object.

Code:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age
   def greet(self):
      print(f"Hello, my name is {self.name} and I am {self.age} years old.")
person = Person("Alice", 25)
person.greet()
person.age = 26
person.greet()
```

Output:

```
Hello, my name is Alice and I am 25 years old.
Hello, my name is Alice and I am 26 years old.
```

Result: The code will output the greeting message twice, with the age updated in the second greeting.

Ex 3.5 Reading and Printing File Contents Line by Line in Python
Date:

Aim: To read the contents of a file line by line and print each line to the console.

Procedure:

Step 1:Open the file: Use the open() function to open the file in read mode ('r').
Step 2:Create a file object: The open() function returns a file object representing the opened file.
Step 3:Read lines: Use a loop to iterate over the lines of the file using the readline() method.
Step 4:Print lines: Print each line read from the file to the console.
Step 5:Close the file: Use the close() method to close the file when finished.

code

```
s="""i am PMISTIAN iam from Thanjavur this week i am going to complete the python course"""
file=open("PMIST.txt","w")
file.write(s)
print(" file is created ")
file.close()
file=open("PMIST.txt","r")
f=file.read()
print(f)
```

Output:

```
file is created
i am PMIST iam from Thanjavur this week i am going to complete the python course
```

Result:
The code will output each line of the file "my_file.txt" to the console, removing any trailing newline characters was Successfully executed.

Ex 3.6 **Creating a CSV File and Loading It into a Pandas DataFrame**
Date:

Aim: To create a CSV file and load its contents into a Pandas DataFrame for data analysis and manipulation.

Procedure:

- **Step 1:Create a list of dictionaries:** Prepare a list of dictionaries where each dictionary represents a row of data.
- **Step 2:Import Pandas:** Import the Pandas library for data manipulation.
- **Step 3:Create a DataFrame:** Use the pandas.DataFrame() function to create a DataFrame from the list of dictionaries.
- **Step 4:Save as CSV:** Use the DataFrame.to_csv() method to save the DataFrame as a CSV file.
- **Step 5:Read the CSV file:** Use the pandas.read_csv() function to read the CSV file back into a DataFrame.
- **Step 6:Inspect the DataFrame:** Use methods like head(), tail(), info(), and describe() to examine the structure and contents of the DataFrame.
- **Step 7:Perform analysis:** Utilize Pandas' data analysis capabilities to manipulate, filter, and analyze the data within the DataFrame.

code

```
import pandas as pd
data = [
   {'Name': 'Alice', 'Age': 25, 'City': 'New York'},
   {'Name': 'Bob', 'Age': 30, 'City': 'Los Angeles'},
   {'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
]
df = pd.DataFrame(data)
df.to_csv('people.csv', index=False)
df_from_csv = pd.read_csv('people.csv')
print(df_from_csv.head())
print(df_from_csv.info())
```

Output:

```
Name  Age        City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     3 non-null      int64
 2   City    3 non-null      object
dtypes: int64(1), object(2)
memory usage: 200.0+ bytes
None
```

Result:
The output will display the first few rows of the DataFrame and its information Sucessfully.

Ex 3.7 Inserting a Record in a File Using Sequential and Random Access

Date:

Aim: To insert a record at a specified position in a file using both sequential and random access modes.

Procedure:

Sequential Access:

- **Step 1:Open the file in append mode:** Use the 'a' mode to append data to the end of the file.
- **Step 2:Read the file contents:** Read the entire file into a list.
- **Step 3:Insert the new record:** Insert the new record at the desired position in the list.
- **Step 4:Write the updated list:** Write the updated list back to the file, overwriting the original contents.
- **Step 5:Close the file:** Close the file using the close() method.

Random Access:

- **Step 1:Open the file in read-write mode:** Use the 'r+' mode to open the file for both reading and writing.
- **Step 2:Calculate the byte offset:** Determine the byte offset where the new record should be inserted based on the record size and the desired position.
- **Step 3:Seek to the offset:** Use the seek() method to move the file pointer to the calculated offset.
- **Step 4:Write the new record:** Write the new record to the file using the write() method.
- **Step 5:Shift subsequent records:** If necessary, shift subsequent records to accommodate the new record.
- **Step 6:Truncate the file:** If the file size has increased, truncate it to remove any unused space.
- **Step 7:Close the file:** Close the file using the close() method.

Code:

```
def insert_record_sequentially(filename, new_record, position):
    with open(filename, 'a') as file:
        lines = file.readlines()
        lines.insert(position, new_record + '\n')
        file.seek(0)
        file.writelines(lines)
def insert_record_randomly(filename, new_record, position, record_size):
    with open(filename, 'r+') as file:
        offset = position * record_size
        file.seek(offset)
        file.write(new_record + '\n')
        file.seek(0, 2)
        file.truncate()
filename = "my_file.txt"
new_record = "New record"
position = 2
record_size = 20
insert_record_sequentially(filename, new_record, position)
insert_record_randomly(filename, new_record, position, record_size)
```

Result:

The code inserts a record at the specified position in the file using sequential and random access methods. Output varies based on the file and position.

Ex 4.1 Setting up a Flask development environment

Date:

Aim:

To set up a Flask development environment for building and running web applications in Python.

Procedure:

1. **Install Python**

Ensure that Python is installed on your machine. You can check this by running the following command in your terminal or command prompt:

2. Set up a Virtual Environment

Creating a virtual environment ensures that your Flask project uses specific versions of libraries, isolated from other projects.

2. **Install virtualenv**

If you don’t have virtualenv installed, you can install it via pip:

```
pip install virtualenv
```

b) Create a virtual environment

Navigate to your project directory and create a virtual environment:

```
python –m venv venv
```

c) Activate the virtual environment

Activate the virtual environment using the following commands:

```
venv\Scripts\activate
```

You should see the virtual environment’s name (e.g., (venv)) appear in your terminal prompt.

3. Install Flask

With the virtual environment activated, install Flask using pip:

```
pip install Flask
```

You can verify the installation by checking the Flask version:

```
flask –version
```

3. **Create a Flask Application**

Next, create a basic Flask application to ensure everything is set up correctly.

4. **Create a Python file**

In your project directory, create a file called app.py with the following content:

```
from flask import Flask
app = Flask(__name__)
@app.route(‘/’)
def home():
    return “Hello, Flask!”

if __name__ == ‘__main__’:
    app.run(debug=True)
```

b) Set up the FLASK_APP environment variable

Before running your app, set the FLASK_APP environment variable to the name of your Python file:

```
# On Windows (CMD)
set FLASK_APP=app.py

# On Windows (Powershell)
$env:FLASK_APP = “app.py”
```

5. **Run the Flask Application**

You can now run your Flask app by using the following command:

```
flask run
```

By default, the app will be available at http://127.0.0.1:5000/. Open that link in a web browser, and you should see the message **“Hello, Flask!”**.

6. **Configure Auto-Reloading (Optional)**

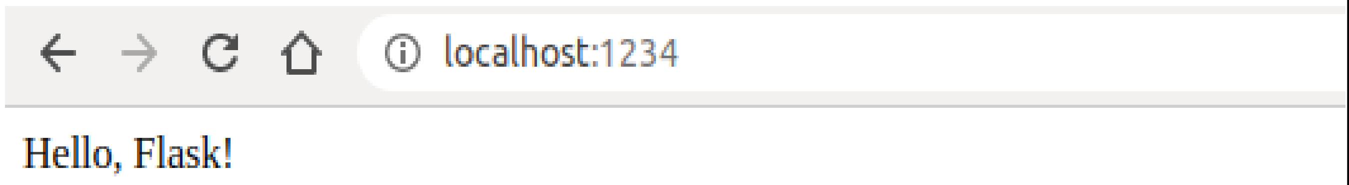
For easier development, you can enable debug mode, which allows Flask to automatically reload the app when you make changes:

```
export FLASK_ENV=development
```

Alternatively, set debug=True in the app.run() function as shown earlier.

Code:

```
pip install virtualenv

python –m venv venv

pip install Flask

flask –version

from flask import Flask
app = Flask(__name__)
@app.route('/')
def home():
    return "Hello, Flask!"

if __name__ == '__main__':
    app.run(debug=True)

# On Windows (CMD)
set FLASK_APP=app.py

# On Windows (Powershell)
$env:FLASK_APP = "app.py"


flask run

export FLASK_ENV=development
```



```
localhost:1234

Hello, Flask!
```

Result:

Thus, the Flask development environment is successfully set up, allowing for the creation and execution of web applications.

Ex 4.2 **Install MySQL**
Date:

Aim:

To install MySQL on a system for managing and interacting with databases.

Procedure:

Step 1: Install MySQL

Windows/macOS: Download the MySQL Community Server from the official MySQL website.

Step 2: Install the MySQL Connector for Python

To connect MySQL with Python, you need the mysql-connector-python package. You can install it via pip:

```
pip install mysql-connector-python
```

Step 3: Create a Database in MySQL

Once logged into MySQL via the terminal, you can create a database and a table to experiment with:

```
CREATE DATABASE test_db;
USE test_db;

CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100),
   age INT,
   email VARCHAR(100)
);
```

You now have a database test_db with a users table.

Step 4: Connect Python to MySQL and Experiment with SQL Commands

Let's create a Python script that connects to the MySQL database and runs some basic SQL commands.

a) Basic Connection and Insertion

Create a file mysql_experiment.py with the following content:

```
import mysql.connector

# Establish a database connection
connection = mysql.connector.connect(
   host="localhost",
   user="root",      # Change this if you use a different MySQL user
   password="your_password",  # Update with your MySQL root password
   database="test_db"  # Database you want to connect to
)

cursor = connection.cursor()

# Create a new record
insert_query = "INSERT INTO users (name, age, email) VALUES (%s, %s, %s)"
values = ("John Doe", 25, "john.doe@example.com")

cursor.execute(insert_query, values)
connection.commit()

print(f"Inserted {cursor.rowcount} record into users table.")

# Close the connection
cursor.close()
connection.close()
```

Run this script:

```
python mysql_experiment.py
```

This script connects to the MySQL database, inserts a new record into the users table, and then closes the connection.

b) Query Data from MySQL

Let’s extend the script to retrieve and print the data we just inserted:

cc

Step 5: Experiment with SQL Commands

You can now experiment with various SQL commands such as:

- **SELECT**: To fetch data from tables.
- **INSERT**: To insert new records into a table.
- **UPDATE**: To modify existing records.
- **DELETE**: To remove records from a table.
- **JOIN**: To fetch data from multiple tables.

For example, a JOIN query:

```
join_query = """
SELECT u.name, o.order_date
FROM users u
JOIN orders o ON u.id = o.user_id
"""
cursor.execute(join_query)
results = cursor.fetchall()
for row in results:
    print(row)
```

Query:

```
pip install mysql-connector-python

CREATE DATABASE test_db;
USE test_db;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    email VARCHAR(100)
);



import mysql.connector

# Establish a database connection
connection = mysql.connector.connect(
    host="localhost",
    user="root",      # Change this if you use a different MySQL user
    password="your_password",  # Update with your MySQL root password
    database="test_db"  # Database you want to connect to
)

cursor = connection.cursor()

# Create a new record
insert_query = "INSERT INTO users (name, age, email) VALUES (%s, %s, %s)"
values = ("John Doe", 25, "john.doe@example.com")

cursor.execute(insert_query, values)
connection.commit()

print(f"Inserted {cursor.rowcount} record into users table.")

# Close the connection
cursor.close()
connection.close()
```

Run this script:

```
python mysql_experiment.py

join_query = """
SELECT u.name, o.order_date
FROM users u
JOIN orders o ON u.id = o.user_id
"""
cursor.execute(join_query)
results = cursor.fetchall()
for row in results:
    print(row)
```

Result:

connect MySQL with Python and interact with databases via SQL commands in a Python environment

Ex 4.3 **Sample MySQL Table**
Date:

Aim:
To create a sample MySQL table to demonstrate database structure and data manipulation.

Procedure:
Step 1: Create a Sample MySQL Table
First, make sure you have a table to work with. You can run this SQL in your MySQL database:

```
CREATE DATABASE order_by_test;
USE order_by_test;

CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(100),
    salary INT
);

INSERT INTO employees (name, position, salary)
VALUES
    ('Alice', 'Developer', 60000),
    ('Bob', 'Designer', 55000),
    ('Charlie', 'Manager', 70000),
    ('David', 'HR', 45000),
    ('Eve', 'Developer', 62000);
```

Step 2: Python Script to Select Records in Ascending/Descending Order
Create a Python file named order_by_example.py with the following content:

```
import mysql.connector

def get_sorted_records(order_by_column, sort_order):
    # Establish a database connection
    connection = mysql.connector.connect(
        host="localhost",
        user="root",  # Use your MySQL user
        password="your_password",  # Use your MySQL password
        database="order_by_test"  # Database to connect to
    )
    cursor = connection.cursor()

    # Validate input for sort order (ASC or DESC)
    if sort_order.lower() not in ['asc', 'desc']:
        print("Invalid sort order! Use 'asc' for ascending or 'desc' for descending.")
        return

    # Construct the query dynamically
    query = f"SELECT * FROM employees ORDER BY {order_by_column} {sort_order.upper()}"

    try:
        # Execute the query
        cursor.execute(query)
        # Fetch all results
        results = cursor.fetchall()
        # Print the results
        print(f"Employees sorted by {order_by_column} in {sort_order.upper()} order:")
        print(f"{'ID':<5}{'Name':<15}{'Position':<15}{'Salary':<10}")
        print("-" * 45)
        for row in results:
            print(f"{row[0]:<5}{row[1]:<15}{row[2]:<15}{row[3]:<10}")
    except mysql.connector.Error as err:
        print(f"Error: {err}")
    finally:
        # Close the cursor and connection
        cursor.close()
        connection.close()


# Main block
if __name__ == '__main__':
    # Get the sorting preference from the user
    column = input("Enter the column to sort by (id, name, position, salary): ").strip()
    order = input("Enter sort order (asc for ascending, desc for descending): ").strip()
    # Call the function with user inputs
    get_sorted_records(column, order)
```

To Execute:

```
python order_by_example.py
```

Query:

```
CREATE DATABASE order_by_test;
USE order_by_test;

CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(100),
    salary INT
);

INSERT INTO employees (name, position, salary)
VALUES
    ('Alice', 'Developer', 60000),
    ('Bob', 'Designer', 55000),
    ('Charlie', 'Manager', 70000),
    ('David', 'HR', 45000),
    ('Eve', 'Developer', 62000);
```

```
import mysql.connector
def get_sorted_records(order_by_column, sort_order):
    # Establish a database connection
    connection = mysql.connector.connect(
        host="localhost",
        user="root",  # Use your MySQL user
        password="your_password",  # Use your MySQL password
        database="order_by_test"  # Database to connect to
    )
    cursor = connection.cursor()

    # Validate input for sort order (ASC or DESC)
    if sort_order.lower() not in ['asc', 'desc']:
        print("Invalid sort order! Use 'asc' for ascending or 'desc' for descending.")
        return

    # Construct the query dynamically
    query = f"SELECT * FROM employees ORDER BY {order_by_column} {sort_order.upper()}"

    try:
        # Execute the query
        cursor.execute(query)

        # Fetch all results
        results = cursor.fetchall()

        # Print the results
        print(f"Employees sorted by {order_by_column} in {sort_order.upper()} order:")
        print(f"{'ID':<5}{'Name':<15}{'Position':<15}{'Salary':<10}")
        print("-" * 45)
        for row in results:
            print(f"{row[0]:<5}{row[1]:<15}{row[2]:<15}{row[3]:<10}")
    except mysql.connector.Error as err:
        print(f"Error: {err}")
    finally:
        # Close the cursor and connection
        cursor.close()
        connection.close()

# Main block
if __name__ == '__main__':
    # Get the sorting preference from the user
    column = input("Enter the column to sort by (id, name, position, salary): ").strip()
    order = input("Enter sort order (asc for ascending, desc for descending): ").strip()

    # Call the function with user inputs
    get_sorted_records(column, order)
```

To Execute:

```
python order_by_example.py
```

Output:

The terminal interaction will look like this:

```
Enter the column to sort by (id, name, position, salary): salary
Enter sort order (asc for ascending, desc for descending): asc
Employees sorted by salary in ASC order:
ID   Name           Position       Salary
---------------------------------------------
4    David          HR             45000
2    Bob            Designer       55000
1    Alice          Developer      60000
5    Eve            Developer      62000
3    Charlie        Manager        70000

Enter the column to sort by (id, name, position, salary): name
Enter sort order (asc for ascending, desc for descending): desc
Employees sorted by name in DESC order:
ID   Name           Position       Salary
---------------------------------------------
5    Eve            Developer      62000
4    David          HR             45000
3    Charlie        Manager        70000
2    Bob            Designer       55000
1    Alice          Developer      60000
```

Result:

This script provides a flexible and user-driven way to experiment with sorting MySQL records directly from Python.

Ex 4.4 **Flask Web Application**
Date:

Aim:
To create a Flask application that demonstrates routing, including static and dynamic routes, and handling different HTTP methods.

Procedure: Step 1: Set Up Flask

1. **Install Flask** if you haven't already:

```
pip install Flask
```

2. **Create a new directory** for your Flask project and navigate into it:

```
mkdir flask_routing_demo
cd flask_routing_demo
```

3. **Create a new file** named app.py and add the following code:

```
from flask import Flask, jsonify

app = Flask(__name__)

# Home route
@app.route('/')
def home():
    return "Welcome to the Flask Routing Demo!"

# About route
@app.route('/about')
def about():
    return "This is the About page."

# User route with variable
@app.route('/user/<username>')
def show_user_profile(username):
    return f"User: {username}"

# Route for different HTTP methods
@app.route('/api/data', methods=['GET', 'POST'])
def api_data():
    if request.method == 'GET':
        return jsonify({"message": "GET request received!"})
    elif request.method == 'POST':
        return jsonify({"message": "POST request received!"})

if __name__ == '__main__':
    app.run(debug=True)
```

Step 2: Run the Flask Application
Run the Flask app:
python app.py
Step 3: Test the Routes
You can test the routes by visiting the following URLs in your browser:

- http://127.0.0.1:5000/ - Home Page
- http://127.0.0.1:5000/about - About Page
- http://127.0.0.1:5000/user/<username> - Replace <username> with any username (e.g., http://127.0.0.1:5000/user/johndoe)

Code:

```
pip install Flask
```

```
mkdir flask_routing_demo
cd flask_routing_demo

    from flask import Flask, jsonify

    app = Flask(__name__)

    # Home route
    @app.route('/')
    def home():
        return "Welcome to the Flask Routing Demo!"

    # About route
    @app.route('/about')
    def about():
        return "This is the About page."

    # User route with variable
    @app.route('/user/<username>')
    def show_user_profile(username):
        return f"User: {username}"

    # Route for different HTTP methods
    @app.route('/api/data', methods=['GET', 'POST'])
    def api_data():
        if request.method == 'GET':
            return jsonify({"message": "GET request received!"})
        elif request.method == 'POST':
            return jsonify({"message": "POST request received!"})

    if __name__ == '__main__':
        app.run(debug=True)
```

python app.py

You can test the routes by visiting the following URLs in your browser:

- http://127.0.0.1:5000/ - Home Page
- http://127.0.0.1:5000/about - About Page
- http://127.0.0.1:5000/user/<username> - Replace <username> with any username (e.g., http://127.0.0.1:5000/user/johndoe)

Result:

Thus, the Flask application successfully sets up routes for the home page, about page, dynamic user profiles, and API requests for both GET and POST methods, with responses displayed as expected in the browser.

Ex 4.5. **Django web Application**
Date:

Aim:
To create a Django project that demonstrates the handling of HTTP methods, specifically using a POST request to submit form data and display it on the same page.

Procedure:
Set Up Django

Step 1: **Install Django**

Step 1: Set Up Django

```
pip install Django
```

Step 2: **Create a new Django project**:

```
django-admin startproject django_http_methods_demo
cd django_http_methods_demo
```

Step 3: **Create a new Django app**:

```
python manage.py startapp demo
```

Step 4: **Add the app to your project**: Open django_http_methods_demo/settings.py and add 'demo', to the INSTALLED_APPS list.

Step 5: **Create a file named urls.py in the demo app directory** (demo/urls.py) and add the following code:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
]
```

Step 6: **Update the project's urls.py** (django_http_methods_demo/urls.py) to include the app's URLs:

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('demo.urls')),  # Include the demo app's URLs
]
```

7. **Define the view in demo/views.py**:

```
from django.http import HttpResponse
from django.shortcuts import render

def home(request):
    message = None
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        message = f"Received Name: {name} and Email: {email}"

    return render(request, 'demo/home.html', {'message': message})
```

8. **Create a template directory** in your app (demo/templates/demo/) and create a file named home.html inside it with the following content:

```
<!doctype html>
```

```
<html>
   <head>
      <title>Django HTTP Methods Demo</title>
   </head>
   <body>
      <h1>Submit Your Information</h1>
      <form method="POST">
         {% csrf_token %}
         <label for="name">Name:</label>
         <input type="text" id="name" name="name" required>
         <br><br>
         <label for="email">Email:</label>
         <input type="email" id="email" name="email" required>
         <br><br>
         <input type="submit" value="Submit">
      </form>
      {% if message %}
      <h2>{{ message }}</h2>
      {% endif %}
   </body>
</html>
```

Step 2: Run the Django Application

Run the Django development server:

```
python manage.py runserver
```

Step 3: Test the Application

Open your browser and go to http://127.0.0.1:8000/.

You will see a form to enter your name and email.

After submitting the form, the application will display the submitted data on the same page.

Output:

Result:

Thus, the Django project successfully demonstrates handling HTTP methods by allowing users to submit their name and email via a form, which is then displayed on the same page after submission.

Ex 5.1 **R Programming Data Frames.**
Date:

Aim:
To perform the following operations using R programming:
1. Create a DataFrame.
2. Combine two DataFrames.
3. Convert a list into a DataFrame.
4. Extract, drop, and reorder columns in a DataFrame.
5. Split the DataFrame into subsets.

Procedure:

Step 1: Create a DataFrame
Use the data.frame() function to create a DataFrame.

Step 2: Combine Two DataFrames
Use the rbind() function for combining DataFrames row-wise or cbind() for column-wise.

Step 3: Convert a List into a DataFrame
Use the data.frame() function to convert a list into a DataFrame.

Step 4: Extract, Drop, and Reorder Columns in a DataFrame
1. **Extract columns** by using column indexing or names.
2. **Drop columns** using negative indexing.
3. **Reorder columns** by specifying the desired order.

Step 5: Split the DataFrame into Subsets

Use the split() function to split a DataFrame based on a column.

Code:
```
# Creating a DataFrame
df1 <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Gender = c("F", "M", "M")
)

print(df1)

# Creating another DataFrame
df2 <- data.frame(
  Name = c("David", "Eve"),
  Age = c(40, 28),
  Gender = c("M", "F")
)

# Combining two DataFrames row-wise
combined_df <- rbind(df1, df2)

print(combined_df)

# Extracting 'Name' and 'Age' columns
extracted_columns <- combined_df[, c("Name", "Age")]

print(extracted_columns)

# Dropping 'Gender' column
```

```
dropped_column_df <- combined_df[, -which(names(combined_df) == "Gender")]

print(dropped_column_df)

# Reordering columns: Age, Name, Gender
reordered_df <- combined_df[, c("Age", "Name", "Gender")]

print(reordered_df)

# Splitting the DataFrame based on Gender
split_df <- split(combined_df, combined_df$Gender)

print(split_df)
```

Output:

```
Create a DataFrame
  Name   Age Gender
1 Alice   25     F
2  Bob   30     M
3 Charlie 35     M

Combine Two DataFrames
    Name   Age Gender
1   Alice   25     F
2     Bob   30     M
3 Charlie   35     M
4   David   40     M
5     Eve   28     F

Convert a List into a DataFrame
  Name   Age Gender
1 Frank   32     M
2 Grace   26     F

Extract, Drop, and Reorder Columns in a DataFrame

Extract 'Name' and 'Age' columns
    Name Age
1   Alice  25
2     Bob  30
3 Charlie  35
4   David  40
5     Eve  28

Drop the 'Gender' column
    Name   Age
1   Alice   25
2     Bob   30
3 Charlie   35
4   David   40
5     Eve   28

Reorder columns: Age, Name, Gender
    Age     Name   Gender
1    25    Alice      F
```

```
2   30      Bob      M
3   35   Charlie     M
4   40    David      M
5   28      Eve      F
```

Split the DataFrame into Subsets

For Gender = F

```
$F
  Name   Age Gender
1 Alice   25     F
5   Eve   28     F
```

For Gender = M

```
$M
     Name   Age Gender
2     Bob   30     M
3 Charlie   35     M
4   David   40     M
```

Result:

1. A DataFrame was created successfully.
2. Two DataFrames were combined row-wise.
3. A list was converted into a DataFrame.
4. Columns were extracted, dropped, and reordered.
5. The DataFrame was split into subsets based on a column value.

This process allows you to manipulate and manage DataFrames effectively in R programming.

Ex 5.2 **Reading the CSV file into DataFrames**

Date:

Aim:

To read a CSV file into a DataFrame using R programming and display the content.

Procedure:

Step 1:Install R and Set Up Environment: Ensure R is installed and running on your system.

Step 2:Place the CSV File: Ensure the CSV file is saved on your local system and note its file path.

Step 3:Read the CSV File: Use the read.csv() function to load the CSV file into a DataFrame.

Step 4:Inspect the DataFrame: Use the head() function to display the first few rows of the DataFrame.

Step 5:Handle Missing Headers (if applicable): If the CSV file lacks headers, add header=FALSE in read.csv() to read the data without column names.

Code:

```
data <- read.csv("path_to_your_file.csv")

print(head(data))
```

Result:

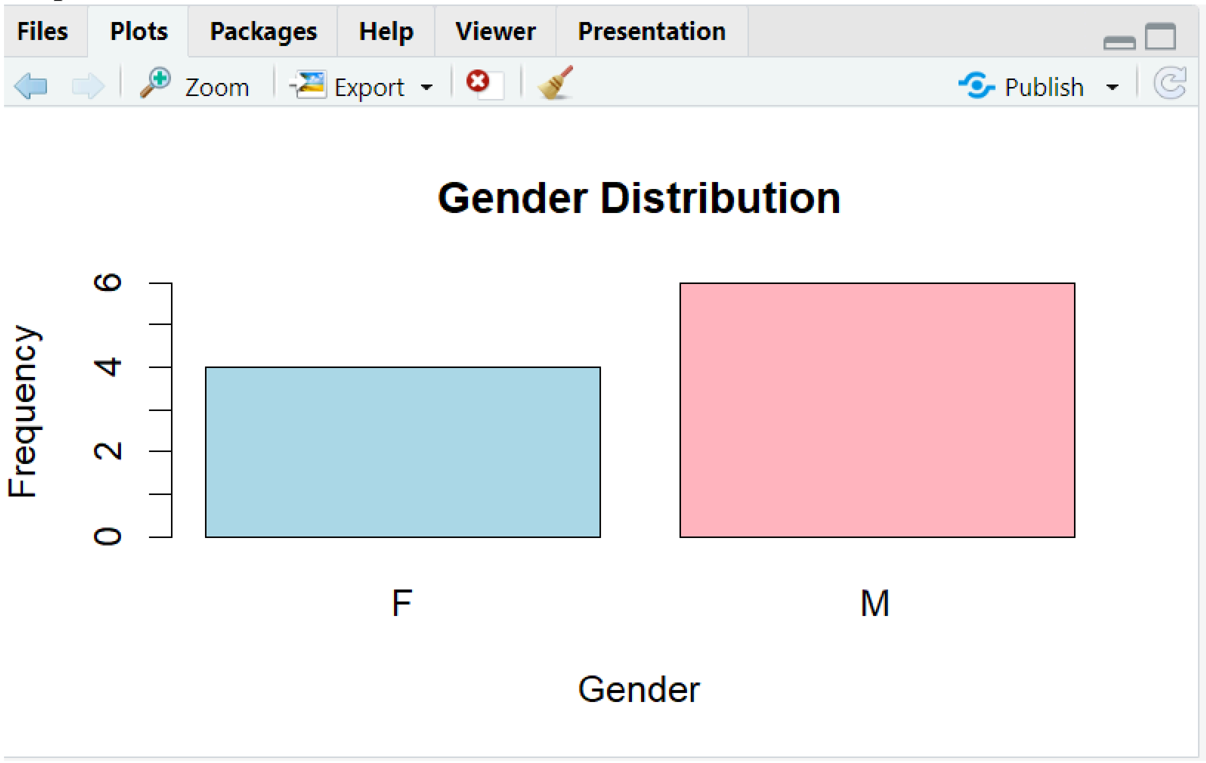
The CSV file was successfully loaded into a DataFrame, and the first few rows of the data were displayed, showing the columns and their respective values using R Programming..

Ex 5.2.1 **Bar Plot Visualization**
Date:

Aim:
To create a bar plot to visualize the distribution of gender in the dataset using R Programming.

Procedure:
Step 1: Create a sample DataFrame containing the data.
Step 2: Use the `barplot()` function to visualize the distribution of the categorical variable "Gender".
Step 3: Add labels for the x-axis, y-axis, and title for better readability.

Code:
```
data <- read.csv("heightweight.csv")
View(data)
barplot(table(data$Gender),
        main = "Gender Distribution",
        xlab = "Gender",
        ylab = "Frequency",
        col = c("lightblue", "lightpink"))
```

Output:


The bar plot shows the count of males and females in the dataset, with different colors representing gender.

Result:
A bar plot was successfully generated, illustrating the distribution of gender in the dataset using R Programming.

Ex 5.2.2 **Histogram Visualization**
Date:

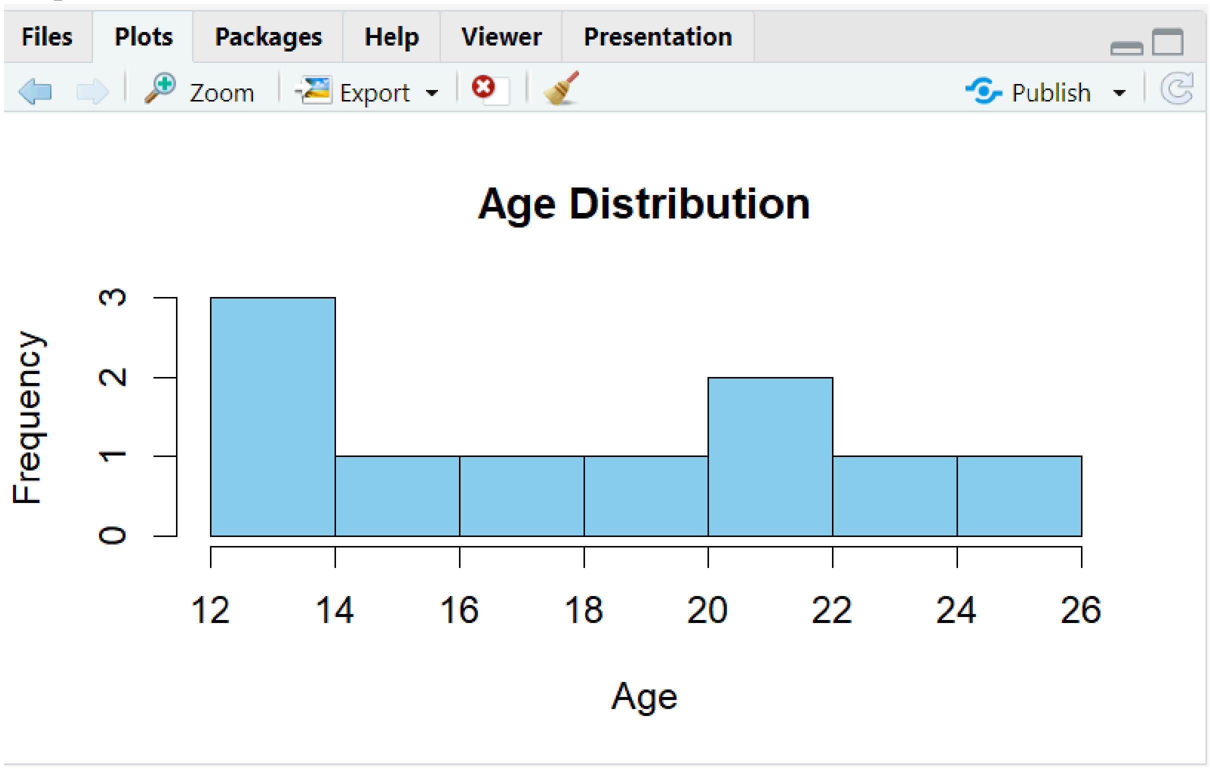
Aim:
To create a histogram to visualize the distribution of ages in the dataset using R Programming.

Procedure:

Step 1: Create a sample DataFrame.
Step 2: Use the `hist()` function to plot the frequency distribution of ages.
Step 3: Customize the bin width and colors.

Code:
```
hist(data$Age,
    main = "Age Distribution",
    xlab = "Age",
    col = "skyblue",
    border = "black",
    breaks = 5)
```

Output:



A histogram displaying the frequency of different age groups in the dataset.

Result:
A histogram was successfully generated, showing the distribution of ages within the dataset using R Programming.

Ex 5.2.3 **Scatter Plot Visualization**
Date:

Aim:
To visualize the relationship between height and weight using a scatter plot using R Programming.

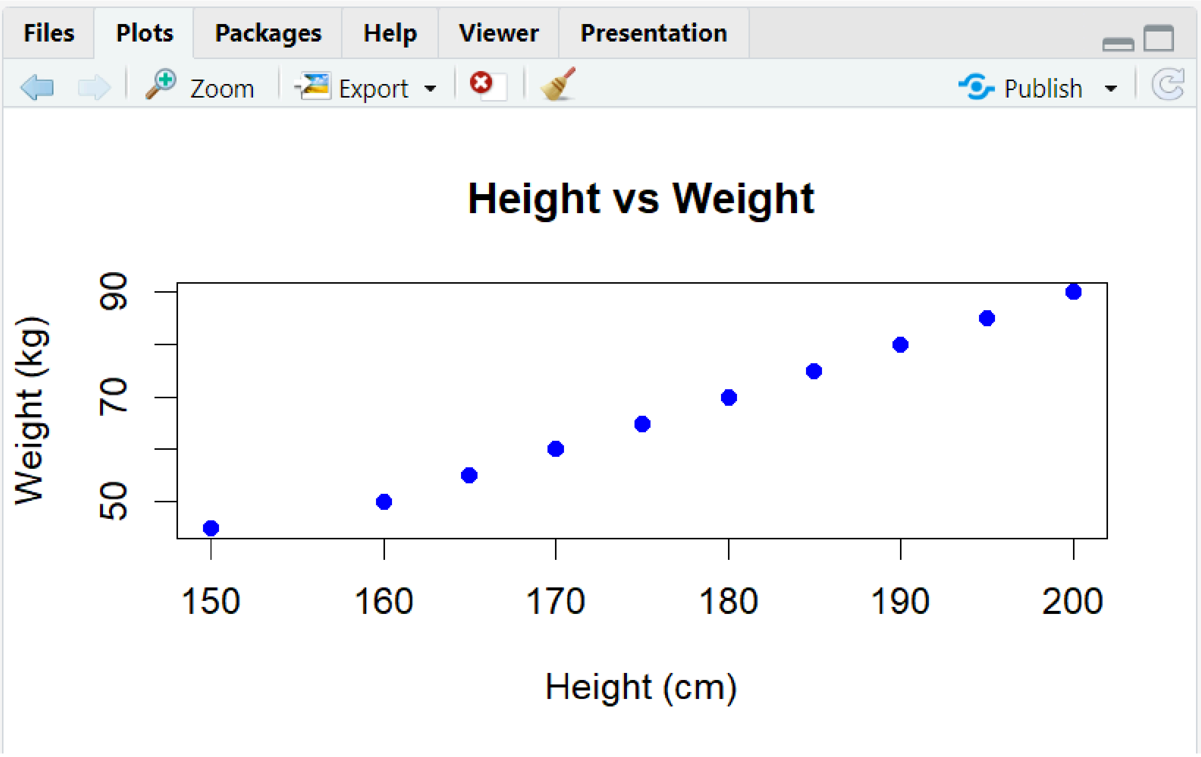
Procedure:

Step 1: Create a sample DataFrame.
Step 2: Use the `plot()` function to generate a scatter plot with height on the x-axis and weight on the y-axis.
Step 3: Customize point colors and labels.

Code:

```
plot(data$Height, data$Weight,
     main = "Height vs Weight",
     xlab = "Height (cm)",
     ylab = "Weight (kg)",
     col = "blue",
     pch = 16)
```

Output:



A scatter plot showing the relationship between height and weight, with points representing individuals.

Result:
A scatter plot was successfully created, showing a visual relationship between height and weight in the dataset using R Programming.

Ex 5.2.4 **Line Plot Visualization**

Date:

Aim:

To create a line plot to visualize the relationship between age and height using R Programming..

Procedure:

Step 1: Create a sample DataFrame.

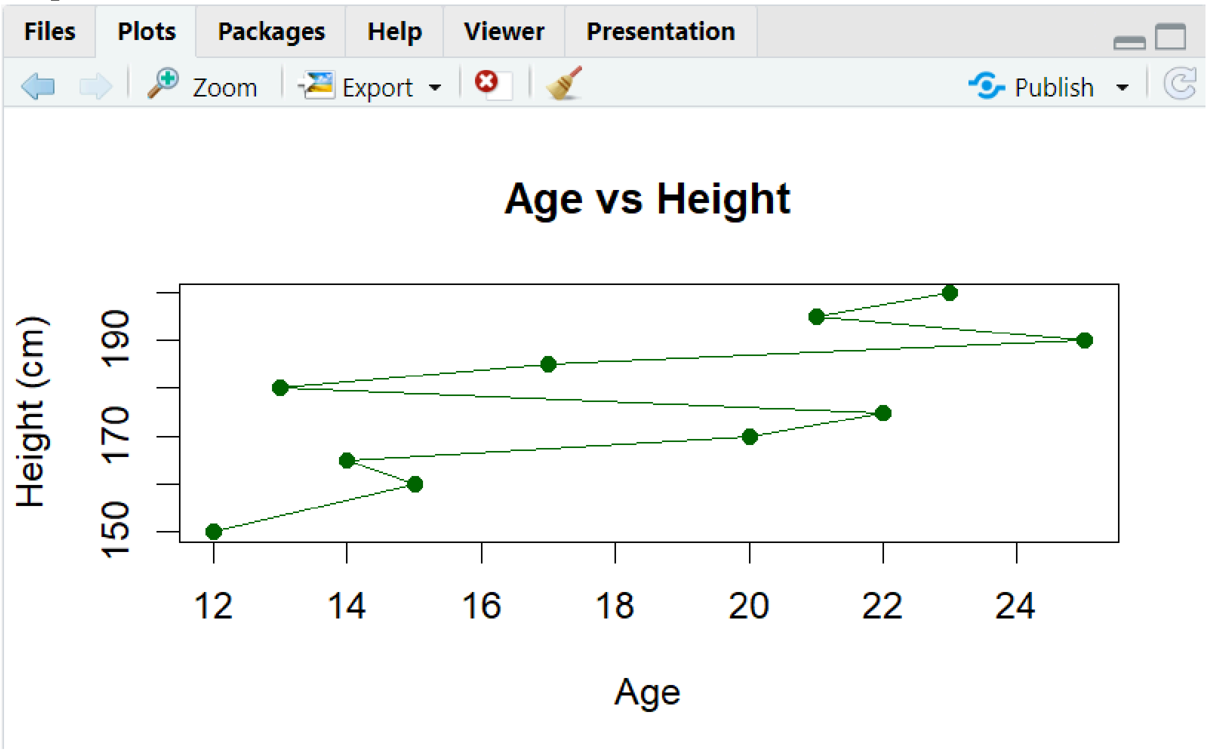
Step 2: Use the `plot()` function to plot a line graph with age on the x-axis and height on the y-axis.

Step 3: Customize the line style and point markers.

Code:

```
plot(data$Age, data$Height, type = "o",
    main = "Age vs Height",
    xlab = "Age",
    ylab = "Height (cm)",
    col = "darkgreen",
    pch = 16)
```

Output:



A line plot showing how height varies with age, with points connected by lines.

Result:

A line plot was successfully generated, displaying the trend between age and height in the dataset.

Ex 5.2.5 **Box Plot Visualization**

Date:

Aim: To create a box plot to visualize the distribution of height by gender using R Programming.

Procedure:

Step 1: Create a sample DataFrame.

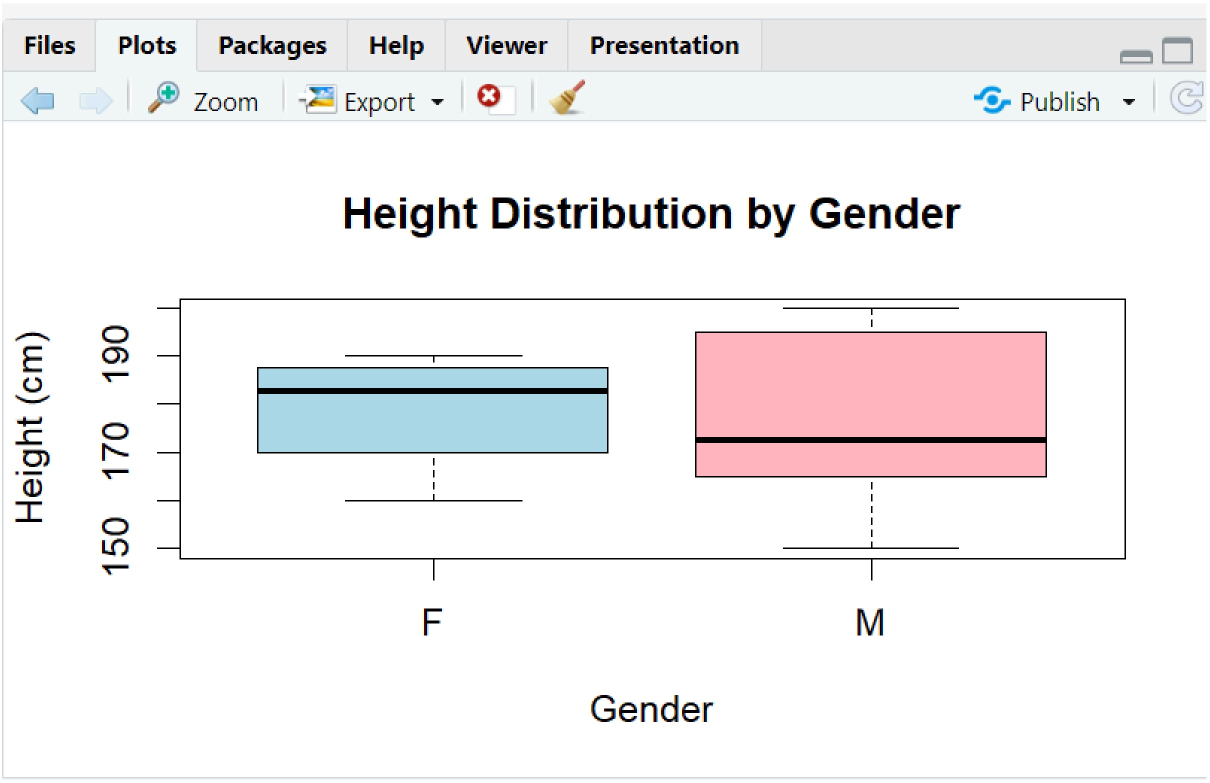
Step 2: Use the `boxplot()` function to create a box plot for the height variable, separated by gender.

Step 3: Customize the colors and labels for clarity.

Code:

```
boxplot(Height ~ Gender, data = data,
        main = "Height Distribution by Gender",
        xlab = "Gender",
        ylab = "Height (cm)",
        col = c("lightblue", "lightpink"))
```

Output:



A box plot displaying the distribution of height for males and females, showing the median, quartiles, and potential outliers.

Result:

A box plot was successfully created, showing the height distribution by gender in the dataset.