INSTITUTE OF TECHNOLOGY AND MANAGEMENT
SKILLS UNIVERSITY,
KHARGHAR, NAVI MUMBAI

PYTHON PROGRAMMING LAB

Prepared by:

Name of Student: PREM ANIL THAKARE

Roll No: 02

Batch: 2023-27

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Exp. No	List of Experiment
1	1.1 Write a program to compute Simple Interest.
	1.2 Write a program to perform arithmetic, Relational operators.
	1.3 Write a program to find whether a given no is even & odd.
	1.4 Write a program to print first n natural number & their sum.
	1.5 Write a program to determine whether the character entered is a Vowel or not .
	1.6 Write a program to find whether given number is an Armstrong Number.
	1.7 Write a program using for loop to calculate factorial of a No.
	1.8 Write a program to print the following pattern
	i) * * * * * * * * * * * * * * *
	ii) 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
	iii) *
2	2.1 Write a program that define the list of defines the list of define countries that are in BRICS.

	2.2 Write a program to traverse a list in reverse order. 1.By using Reverse method. 2.By using slicing
	2.3 Write a program that scans the email address and forms a tuple of username and domain.
	2.4 Write a program to create a list of tuples from given list having number and add its cube in tuple. i/p: c= [2,3,4,5,6,7,8,9]
	2.5 Write a program to compare two dictionaries in Python? (By using == operator)
	2.6 Write a program that creates dictionary of cube of odd numbers in the range.
	2.7 Write a program for various list slicing operation. a= [10,20,30,40,50,60,70,80,90,100] i. Print Complete list ii. Print 4th element of list iii. Print list from0th to 4th index. iv. Print list -7th to 3rd element v. Appending an element to list. vi. Sorting the element of list. vii. Popping an element. viii. Removing Specified element. ix. Entering an element at specified index. x. Counting the occurrence of a specified element. xi. Extending list. xii. Reversing the list.
3	3.1 Write a program to extend a list in python by using given approach. i. By using + operator. ii. By using Append () iii. By using extend ()
	3.2 Write a program to add two matrices.

	3.3 Write a Python function that takes a list and returns a new list with distinct elements from the first list.
	3.4 Write a program to Check whether a number is perfect or not.
	3.5 Write a Python function that accepts a string and counts the number of upper- and lower-case letters. string_test= 'Today is My Best Day'
4	4.1 Write a program to Create Employee Class & add methods to get employee details & print.
	4.2 Write a program to take input as name, email & age from user using combination of keywords argument and positional arguments (*args and**kwargs) using function,
	4.3 Write a program to admit the students in the different Departments(pgdm/btech)and count the students. (Class, Object and Constructor).
	4.4 Write a program that has a class store which keeps the record of code and price of product display the menu of all product and prompt to enter the quantity of each item required and finally generate the bill and display the total amount.
	4.5 Write a program to take input from user for addition of two numbers using (single inheritance).
	4.6 Write a program to create two base classes LU and ITM and one derived class. (Multiple inheritance).
	4.7 Write a program to implement Multilevel inheritance, GrandfatheràFather-àChild to show property inheritance from grandfather to child.
	4.8 Write a program Design the Library catalogue system using inheritance take base class (library item) and derived class (Book, DVD & Journal) Each derived class should have unique attribute and methods and system should support Check in and check out the system. (Using Inheritance and Method overriding)
5	5.1 Write a program to create my_module for addition of two numbers and import it in main script.

	5.2 Write a program to create the Bank Module to perform the operations such as Check the Balance, withdraw and deposit the money in bank account and import the module in main file.
	5.3 Write a program to create a package with name cars and add different modules (such as BMW, AUDI, NISSAN) having classes and functionality and import them in main file cars.
6	6.1 Write a program to implement Multithreading. Printing “Hello” with one thread & printing “Hi” with another thread.
7.	7.1 Write a program to use ‘whether API’ and print temperature of any city, also print the sunrise and sunset times for the same humidity of that area.
	7.2 Write a program to use the ‘API’ of crypto currency.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.1

Title:

Write a program to compute Simple Interest.

Theory:

The provided Python program calculates the Simple Interest and Total Amount based on the user input for Principal Amount (p), Rate of Interest (r), and Time Period in years (t). The formula used for calculating Simple Interest is:

Simple Interest (SI) = $(p \times r \times t)$ /100

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program efficiently calculates Simple Interest and Total Amount based on user input. It adheres to the formula for Simple Interest and provides clear output. It is important to note that the program assumes correct and valid input from the user. Additionally, the program does not include error handling or validation for negative values or non-numeric inputs. Implementing such validation would enhance the program's robustness. Overall, the program serves as a simple and effective tool for computing interest in financial calculations.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.2

Title:

Write a program to perform arithmetic, Relational operators.

Theory:

The provided Python program defines a function **arithmetic_relational** that takes two numeric input values (**a** and **b**) and performs both arithmetic and relational operations on them. The arithmetic operations include addition, subtraction, multiplication, division, modulo, and exponentiation. The relational operations include checking for equality, inequality, greater than, less than, greater than or equal to, and less than or equal to.

Code:

```
def arithmetic_relational(a, b):
    equal = a == b
    not_equal = a != b
    greater = a > b
    less = a < b
    greater_less = a >= b
    less_greater = a <= b

    print(f"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}")

    print("\nRelational Operations:")
    print(f"{a} == {b} : {equal}")
    print(f"{a} != {b} : {not_equal}")
    print(f"{a} > {b} : {greater}")
    print(f"{a} < {b} : {less}")
    print(f"{a} >= {b} : {greater_less}")
    print(f"{a} <= {b} : {less_greater}")

num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

arithmetic_relational(num1, num2)
```

Output (screenshot):

```
Enter the first number: 23
Enter the second number: 45
Arithmetic Operations:
23.0 + 45.0 = 68.0
23.0 - 45.0 = -22.0
23.0 * 45.0 = 1035.0
23.0 / 45.0 = 0.5111111111111111
23.0 % 45.0 = 23.0
23.0 ** 45.0 = 1.8956258430116204e+61

Relational Operations:
23.0 == 45.0 : False
23.0 != 45.0 : True
23.0 > 45.0 : False
23.0 < 45.0 : True
23.0 >= 45.0 : False
23.0 <= 45.0 : True
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the first number: 3
Enter the second number: 1
Arithmetic Operations:
3.0 + 1.0 = 4.0
3.0 - 1.0 = 2.0
3.0 * 1.0 = 3.0
3.0 / 1.0 = 3.0
3.0 % 1.0 = 0.0
3.0 ** 1.0 = 3.0

Relational Operations:
3.0 == 1.0 : False
3.0 != 1.0 : True
3.0 > 1.0 : True
3.0 < 1.0 : False
3.0 >= 1.0 : True
3.0 <= 1.0 : False
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the first number: -90
Enter the second number: -12
Arithmetic Operations:
-90.0 + -12.0 = -102.0
-90.0 - -12.0 = -78.0
-90.0 * -12.0 = 1080.0
-90.0 / -12.0 = 7.5
-90.0 % -12.0 = -6.0
-90.0 ** -12.0 = 3.54070616147215e-24

Relational Operations:
-90.0 == -12.0 : False
-90.0 != -12.0 : True
-90.0 > -12.0 : False
-90.0 < -12.0 : True
-90.0 >= -12.0 : False
-90.0 <= -12.0 : True
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program serves as a versatile tool for performing basic arithmetic and relational operations on two numeric values. It provides clear and formatted output, making it user-friendly. However, it assumes that the user will enter valid numeric values, and it does not include error handling for potential input errors, such as non-numeric input. Enhancing the program with input validation would make it more robust.

Overall, the program is effective for quick calculations and comparisons between two numeric values. It showcases the use of arithmetic and relational operators in Python, demonstrating the language's simplicity and readability.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.3

Title:

Write a program to find whether a given no is even & odd.

Theory:

The provided Python program defines a function **check** that takes an integer input (**num**) and determines whether it is even or odd. It uses the modulo operator (**%**) to check if the remainder when dividing the number by 2 is zero. If the remainder is zero, the number is even; otherwise, it is odd. The program then prints the result accordingly.

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program is a straightforward and efficient way to determine whether a given number is even or odd. It uses a simple if-else statement to check the condition and provides clear output to the user. However, like many simple programs, it assumes that the user will enter a valid integer and does not include input validation.

Adding input validation, such as checking if the entered value is indeed an integer, would enhance the program's robustness. Additionally, the program could be extended to handle non-integer inputs gracefully.

In summary, the program effectively fulfills its purpose of checking whether a given number is even or odd, showcasing the simplicity and readability of Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.4

Title:

Write a program to print the first N natural number & their sum. def prime_no(n)

Theory:

The provided Python program defines a function prime_no that takes an integer input n and prints the first n natural numbers along with their sum. Inside the function, a list l is used to store the first n natural numbers, and a variable sum is used to accumulate their sum. A for loop iterates from 1 to n, appending each number to the list and adding it to the sum.

The program then prints the list of natural numbers and the sum.

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively prints the first **n** natural numbers and their sum. It demonstrates the use of a loop to generate a list of natural numbers and calculate their sum. The program assumes that the user will enter a valid positive integer and does not include input validation.

Adding input validation to ensure that the entered value is a positive integer would improve the program's robustness. Additionally, the program could be extended to handle non-integer inputs more gracefully.

In summary, the program provides a clear and simple way to generate and display the first **n** natural numbers along with their sum, showcasing the ease of working with loops and lists in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.5

Title:

Write a program to determine whether the character entered is a Vowel or not def vowel(char)

Theory:

The provided Python program defines a function **vowel** that takes a character as input and determines whether it is a vowel. The function uses a string **vowels** containing all uppercase and lowercase vowels. It checks if the input character is present in the **vowels** string and returns **True** if it is a vowel, and **False** otherwise.

The main part of the program prompts the user to enter a character. It then checks if the entered value is a single alphabetical character. If it is, the program calls the **vowel** function and prints whether the character is a vowel or not. If the entered value is not a single alphabetical character, it prompts the user to enter a valid input.

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively determines whether a given character is a vowel or not. It demonstrates the use of a function to encapsulate the logic for checking vowel status. The program assumes that the user will enter a single alphabetical character and does not include input validation for other cases.

To enhance robustness, the program could be extended to handle cases where the user enters more than one character or a non-alphabetical character. Additionally, incorporating case-insensitivity in the input check could improve the user experience.

In summary, the program provides a clear and concise way to check whether a given character is a vowel, highlighting the simplicity of working with strings and conditionals in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.6

Title:

Write a program to find whether given number is an Armstrong Number.def armstrong_no(num)

Theory:

The provided Python program defines a function **armstrong_no** that takes an integer input **num** and determines whether it is an Armstrong number. An Armstrong number (also known as a narcissistic number, pluperfect digital invariant, or pluperfect number) is a number that is the sum of its own digits each raised to the power of the number of digits.

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively determines whether a given number is an Armstrong number. It demonstrates the use of string conversion, length calculation, and a generator expression to perform the necessary calculations. The program assumes that the user will enter a positive integer and does not include input validation for other cases.

To enhance the program's robustness, input validation could be added to check if the entered value is a positive integer. Additionally, the program could provide more informative messages to the user, explaining the concept of an Armstrong number.

In summary, the program offers a clear and concise implementation of Armstrong number checking, showcasing Python's ease of handling strings and mathematical operations.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.7

Title:

Write a program using for loop to calculate factorial of a No. def factorial(n).

Theory:

The provided Python program defines a function **factorial** that calculates the factorial of a given number **n** using a for loop. The factorial of a non-negative integer *n*, denoted as *n*!, is the product of all positive integers up to *n*.

Code:

```
p=float(input("Enter the Principal Amount :"))
r=float(input("Enter the Rate of Rate :"))
t=float(input("Enter the time period (yrs):"))

si=(p*r*t)/100

print("Simple Intrest:",si)
print("Total Amount:",si+p)
```

Output (screenshot):

```
Enter the Principal Amount :25000
Enter the Rate of Rate :12
Enter the time period (yrs):5
Simple Intrest: 15000.0
Total Amount: 40000.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the Principal Amount :3500
Enter the Rate of Rate :23
Enter the time period (yrs):5
Simple Intrest: 4025.0
Total Amount: 7525.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter the Principal Amount :57000
Enter the Rate of Rate :9
Enter the time period (yrs):10
Simple Intrest: 51300.0
Total Amount: 108300.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively calculates the factorial of a given number using a for loop. It handles various cases, such as negative numbers, by providing appropriate messages. However, the program assumes that the user will enter a valid integer and does not include input validation.

To improve the program's robustness, input validation could be added to check if the entered value is an integer. Additionally, the program could provide more informative messages to guide the user.

In summary, the program demonstrates a clear and concise implementation of factorial calculation, highlighting the use of loops and conditional statements in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 1.8

Title:

Write a program to print the following pattern

i)

```
*
* *
* * *
* * * *
* * * * *
```

ii)

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

iii)

```
*
* * *
* * * * *
* * * * * * *
* * * * * * * * *
```

Theory:

The provided Python program prints three different patterns:

Pattern i) - Triangle of Asterisks:

This pattern uses nested loops to print a right-angled triangle of asterisks. The outer loop controls the number of rows, and the inner loop prints the asterisks in each row.

Pattern ii) - Number Triangle:

This pattern also uses nested loops, where the outer loop controls the number of rows, and the inner loop prints the current row number in each iteration.

Pattern iii) - Pyramid of Asterisks:

This pattern prints a pyramid-like structure of asterisks, where the number of asterisks in each row is determined by the pattern $2i-1$, where i is the current row number.

Code:

Pattern i) - Triangle of Asterisks:

```
a=int(input("Enter the number of rows:"))
for i in range(0,a+1):
    for j in range (0,i):
        print("*",end=" ")
    print("\n")
```

Pattern ii) - Number Triangle:

```
a = int(input("Enter the number of rows: "))
for i in range(1, a+1):
    for j in range(i):
        print(i, end=" ")
    print()
```

Pattern iii) - Pyramid of Asterisks:

```
a = int(input("Enter the number of rows: "))
for i in range(1, a+1):
    for j in range(2*i-1):
        print("*", end=" ")
    print()
```

Output (screenshot):

Pattern i) - Triangle of Asterisks:

```
Enter the number of rows:5


*

* *

* * *

* * * *

* * * * *

premthakare@2023premt_isu_ac_in Python_Lab %
```

Pattern ii) - Number Triangle:

```
Enter the number of rows: 6
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
premthakare@2023premt_isu_ac_in Python_Lab %
```

Pattern iii) - Pyramid of Asterisks:

```
Enter the number of rows: 5
*
* * *
* * * * *
* * * * * * *
* * * * * * * * *
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

The provided Python programs demonstrate the use of nested loops to generate specific patterns. Each pattern is controlled by an outer loop that determines the number of rows and an inner loop that handles the printing of characters or numbers in each row. The programs prompt the user to enter the number of rows, providing flexibility in generating patterns of different sizes.

The programs could be further enhanced by incorporating input validation to ensure the entered values are positive integers. Overall, they illustrate the simplicity and readability of Python in working with loops and patterns.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.1

Title:

Write a program that define the list of defines the list of define countries that are in BRICS.

Theory:

The provided Python program defines a list of countries that are part of the BRICS group (Brazil, Russia, India, China, South Africa). It then prompts the user to enter a country, converts the input to uppercase using **.upper()**, and checks whether the entered country is in the BRICS list. Based on the result, it prints whether the entered country is in BRICS or not.

After that, the program prints the list of BRICS countries using a for loop.

Code:

```
countries = ['BRAZIL', 'RUSSIA', 'INDIA', 'CHINA', 'SOUTH AFRICA']
ans = input("Enter the country:").upper()

if ans in countries:
    print(ans, "is in BRICS")
else:
    print(ans, "is not in BRICS")

print("\nBRICS Countries:")
for country in countries:
    print(country)
```

Output (screenshot):

```
Enter the country:India
INDIA is in BRICS

BRICS Countries:
BRAZIL
RUSSIA
INDIA
CHINA
SOUTH AFRICA
premthakare@2023premt_isu_ac_in Python_Lab %
```

Test Case Any two (screenshot):

```
Enter the country:Pakistan
PAKISTAN is not in BRICS

BRICS Countries:
BRAZIL
RUSSIA
INDIA
CHINA
SOUTH AFRICA
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program is a simple and effective way to check whether a user-entered country is part of the BRICS group. It showcases the use of a list, input validation (by converting the input to uppercase), and a conditional statement to determine membership. The program assumes that the user will enter a valid country name and does not include more advanced input validation.

To enhance the program, additional input validation could be added to handle cases where the entered value is not a country name or to handle case-insensitive input.

In summary, the program demonstrates the use of lists, input processing, and conditional statements in Python to perform a membership check and print a list of countries.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.2

Title:

Write a program to traverse a list in reverse order.

1.By using Reverse method.

2.By using slicing

Theory:

The provided Python program demonstrates two methods to traverse a list in reverse order:

- Using Reverse Method:
  - The original list is printed.
  - A new list reversed_list is created as a copy of the original list.
  - The reverse() method is applied to reversed_list to reverse its elements.
  - The reversed list is printed.
- Using Slicing:
  - The original list is printed.
  - A new list reversed_list_slice is created using list slicing ([::-1]), which effectively reverses the order of elements.
  - The reversed list is printed.

Code:

```
l = [1, 2, 3, 4, 5]
print("\nOriginal List:",l)
reversed_list = list(l)
reversed_list.reverse()

print("Reversed List using reverse method:", reversed_list)

l2 = [1, 2, 3, 4, 5]
print("\nOriginal List:",l2)
reversed_list_slice = l2[::-1]

print("Reversed List using slicing:", reversed_list_slice)
```

Output (screenshot):

```
Original List: [1, 2, 3, 4, 5]
Reversed List using reverse method: [5, 4, 3, 2, 1]

Original List: [1, 2, 3, 4, 5]
Reversed List using slicing: [5, 4, 3, 2, 1]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively demonstrates two different methods to traverse a list in reverse order. The first method uses the reverse() method, which modifies the original list in place, while the second method creates a new list using slicing, leaving the original list unchanged.

Both methods achieve the same result, showcasing the flexibility and multiple approaches available in Python for list manipulation. The program assumes a predefined list for demonstration purposes, but it could be adapted to work with user input or dynamically generated lists.

In summary, the program highlights Python's simplicity in handling list operations and provides insight into two different ways to reverse a list.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.3

Title:

Write a program that scans the email address and forms a tuple of username # and domain.

Theory:

The provided Python program defines a function **username** that takes an email address as input, extracts the username and domain by splitting the email address at the "@" symbol, and forms a tuple containing the username and domain. The program then prompts the user to enter an email address, calls the **username** function with the entered email address, and prints the extracted username and domain from the resulting tuple.

Code:

```
def username(email):

    username, domain = email.split("@")
    email = (username, domain)
    return email

email_address = input("Enter your email address: ")
result = username(email_address)
print("Username:", result[0])
print("Domain:", result[1])
```

Output (screenshot):

```
Enter your email address: premthakare@gmail.com
Username: premthakare
Domain: gmail.com
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
Enter your email address: thakareprem@yahoo.in
Username: thakareprem
Domain: yahoo.in
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively extracts the username and domain from an email address using the **split()** method and forms a tuple containing these components. The use of a function enhances modularity, allowing for easy reuse of the functionality.

The program assumes that the user will enter a valid email address with a single "@" symbol and does not include extensive validation. To improve robustness, additional validation could be added to ensure the presence of "@" and to handle potential edge cases.

In summary, the program demonstrates the simplicity of extracting information from an email address using Python's string manipulation capabilities and tuples.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.4

Title:

Write a program to create a list of tuples from given list having number and add its cube in tuple. i/p: c= [2,3,4,5,6,7,8,9]

Theory:

The provided Python program creates a list of tuples from a given list, where each tuple contains a number and its cube. This is achieved using a list comprehension. The program initializes a list **c** with some numbers, and then it creates a new list of tuples **result** using a list comprehension. Each tuple in **result** consists of a number from the original list (**num**) and its cube (**num ** 3**). The final list of tuples is printed.

Code:

```
c = [2, 3, 4, 5, 6, 7, 8, 9]

result = [(num, num ** 3) for num in c]

print("List of Tuples (Number, Cube):", result)
```

Output (screenshot):

```
List of Tuples (Number, Cube): [(2, 8), (3, 27), (4, 64), (5, 125), (6, 216), (7, 343), (8, 512), (9, 729)]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program is a concise and effective way to create a list of tuples containing numbers and their cubes. It showcases the use of list comprehensions, a powerful feature in Python for creating lists based on existing iterables.

The program assumes that the list of numbers is predefined, but it could be adapted to take user input or work with dynamically generated lists.

In summary, the program demonstrates the simplicity and readability of Python when working with list comprehensions to perform specific transformations on data.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.5

Title:

Write a program to compare two dictionaries in Python?(By using == operator)

Theory:

The provided Python program compares two dictionaries (**d1** and **d2**) using the == operator. The dictionaries are considered equal if they have the same key-value pairs. If the dictionaries have the same keys and values in the same order, the program prints "The dictionaries are Same." Otherwise, it prints "The dictionaries are not Same."

Code:

```
d1 = {'a': 2, 'b': "Prem", 'c': "Thakare"}
d2 = {'a': 2, 'b': "Prem", 'c': "Thakare"}

if d1 == d2:
    print("The dictionaries are Same.")
else:
    print("The dictionaries are not Same.")
```

```
d1 = {'a': 2, 'b': "Prem", 'c': "Thakare"}
d2 = {'a': 2, 'b': "Piyush", 'c': "Singh"}

if d1 == d2:
    print("The dictionaries are Same.")
else:
    print("The dictionaries are not Same.")
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q2.py
The dictionaries are Same.
premthakare@2023premt_isu_ac_in Python_Lab %
```

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q2.py
The dictionaries are not Same.
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively demonstrates how to use the == operator to compare two dictionaries for equality. The comparison is based on the content of the dictionaries, ensuring that both the keys and values match.

The program assumes that the dictionaries are predefined, but it could be adapted to take user input or work with dynamically generated dictionaries.

In summary, the program showcases a straightforward way to compare dictionaries in Python using the equality operator.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.6

Title:

Write a program that creates dictionary of cube of odd numbers in the range.

Theory:

The provided Python program creates a dictionary containing the cubes of odd numbers in a specified range. The program prompts the user to enter the start and end values for the range. It then uses a dictionary comprehension to generate a dictionary (**odd_cubes**). The keys are odd numbers in the specified range, and the corresponding values are the cubes of these odd numbers.

Code:

```
start = int(input("Enter the Start of range:"))
end = int(input("Enter the End of range:"))

odd_cubes= {num: num**3 for num in range(start, end + 1) if num % 2 != 0}
print("Dictionary of Cubes of Odd Numbers:", odd_cubes)
```

Output (screenshot):

```
Enter the Start of range:5
Enter the End of range:20
Dictionary of Cubes of Odd Numbers: {5: 125, 7: 343, 9: 729, 11: 1331, 13: 2197, 15: 3375, 17: 4913, 19: 6859}
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program is a concise and effective way to create a dictionary of cubes of odd numbers within a given range. It showcases the use of dictionary comprehensions in Python to build dictionaries based on specific conditions.

The program assumes that the user will enter valid integer values for the start and end of the range. Adding additional input validation could enhance the program's robustness.

In summary, the program demonstrates the simplicity and readability of Python when working with dictionary comprehensions to create dictionaries based on specific criteria.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 2.7

Title:

Write a program for various list slicing operation.

a= [10,20,30,40,50,60,70,80,90,100]

i. Print Complete list

ii. Print 4th element of list

iii. Print list from0th to 4th index.

iv. Print list -7th to 3rd element

v. Appending an element to list.

vi. Sorting the element of list.

vii. Popping an element.

viii. Removing Specified element.

ix. Entering an element at specified index.

x. Counting the occurrence of a specified element.

xi. Extending list.

xii. Reversing the list.

Theory:

The provided Python program demonstrates various list slicing and manipulation operations:

- **Print Complete List:**
  - The complete list **a** is printed.
- **Print 4th Element of List:**
  - The 4th element of the list is printed.
- **Print List from 0th to 4th Index:**
  - A sublist from the 0th to the 4th index is printed.
- **Print List from -7th to 3rd Element:**
  - A sublist from the -7th to the 3rd element is printed.
- **Appending an Element to List:**
  - The element 101 is appended to the list.
- **Sorting the Elements of List:**
  - The list is sorted in ascending order.
- **Popping an Element:**
  - The last element of the list is popped and printed.

- **Removing Specified Element:**
  - The element 60 is removed from the list.
- **Inserting an Element at a Specified Index:**
  - The element 35 is inserted at index 2.
- **Counting the Occurrence of a Specified Element:**
  - The count of the element 30 in the list is printed.
- **Extending List:**
  - The list is extended by adding elements from another list.
- **Reversing the List:**
  - The list is reversed.

Code:

1. **Print Complete List:**

```
# i. Print Complete list
a= [10,20,30,40,50,60,70,80,90,100]
print("Complete List:", a)
```

2. **Print 4th Element of List:**

```
# ii. Print 4th element of list
a= [10,20,30,40,50,60,70,80,90,100]
print("4th Element:", a[3])
```

3. **Print List from 0th to 4th Index:**

```
# iii. Print list from0th to 4th index.
a= [10,20,30,40,50,60,70,80,90,100]
print("List from 0th to 4th index:", a[0:5])
```

4. **Print List from -7th to 3rd Element:**

```
# iv. Print list -7th to 3rd element
a= [10,20,30,40,50,60,70,80,90,100]
print("List from -7th to 3rd element:", a[-7:3])
```

5. **Appending an Element to List:**

```
# v. Appending an element to list.
a= [10,20,30,40,50,60,70,80,90,100]
a.append(101)
print("List after Appending 101:", a)
```

6. **Sorting the Elements of List:**

```
# vi. Sorting the element of list.
a= [10,20,30,40,50,60,70,80,90,100]
a.sort()
print("Sorted List:", a)
```

7. **Popping an Element:**

```
#vii. Popping an element.
a= [10,20,30,40,50,60,70,80,90,100]
popped_element = a.pop()
print("Popped Element:", popped_element)
print("List after Popping:", a)
```

8. Removing Specified Element:

```
# viii. Removing Specified element.
a= [10,20,30,40,50,60,70,80,90,100]
a.remove(60)
print("List after Removing 60:", a)
```

9. Inserting an Element at a Specified Index:

```
# ix. Inserting an element at a specified index
a= [10,20,30,40,50,60,70,80,90,100]
a.insert(2, 35)
print("List after Inserting 35 at index 2:", a)
```

10. Counting the Occurrence of a Specified Element:

```
# x. Counting the occurrence of a specified element
a= [10,20,30,40,50,60,70,80,90,100]
count_30 = a.count(30)
print("Count of 30 in the List:", count_30)
```

11. Extending List:

```
# xi. Extending list
a= [10,20,30,40,50,60,70,80,90,100]
extension_list = [120, 130, 140]
a.extend(extension_list)
print("List after Extending:", a)
```

12. Reversing the List:

```
# xii. Reversing the list
a= [10,20,30,40,50,60,70,80,90,100]
a.reverse()
print("Reversed List:", a)
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q2.py
Complete List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
4th Element: 40
List from 0th to 4th index: [10, 20, 30, 40, 50]
List from -7th to 3rd element: []
List after Appending 101: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 101]
Sorted List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Popped Element: 100
List after Popping: [10, 20, 30, 40, 50, 60, 70, 80, 90]
List after Removing 60: [10, 20, 30, 40, 50, 70, 80, 90, 100]
List after Inserting 35 at index 2: [10, 20, 35, 30, 40, 50, 60, 70, 80, 90, 100]
Count of 30 in the List: 1
List after Extending: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 130, 140]
Reversed List: [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program demonstrates a variety of list operations, including slicing, appending, sorting, popping, removing, inserting, counting, extending, and reversing. Each operation showcases the versatility of Python lists and their manipulation capabilities.

The program assumes that the list **a** is predefined for demonstration purposes. It could be adapted to work with user input or dynamically generated lists. Additionally, some operations may modify the original list, so caution should be exercised when using them.

In summary, the program provides hands-on experience with common list operations in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 3.1

Title:

Write a program to extend a list in python by using given approach.

i. By using + operator.

ii. By using Append ()

iii. By using extend ()

Theory:

The provided Python program extends a list using three different approaches:

i. By using + Operator:

This approach uses the + operator to concatenate two lists, resulting in an extended list.

ii. By using Append():

In this approach, a loop iterates over each element in **list2**, and the **append()** method is used to add each element to **list1**, extending it.

iii. By using extend():

The **extend()** method is used to add all elements from **list2** to the end of **list1**, effectively extending the list.

Code:

i. By using + Operator:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2
print("Extended List using + operator:", list3)
```

ii. By using Append():

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
for item in list2:
    list1.append(item)
print("Extended List using append():", list1)
```

iii. By using extend():

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

list1.extend(list2)
print("Extended List using extend():", list1)
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Extended List using + operator: [1, 2, 3, 4, 5, 6]
Extended List using append(): [1, 2, 3, 4, 5, 6]
Extended List using extend(): [1, 2, 3, 4, 5, 6]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program showcases three different approaches to extend a list in Python. Each approach has its advantages and use cases:

- The + operator creates a new list by concatenating two existing lists.
- The **append()** method adds individual elements to the end of the list.
- The **extend()** method adds all elements from another iterable to the end of the list.

The choice of approach depends on the specific requirements of the task at hand. The program assumes predefined lists for demonstration purposes but could be adapted to work with user input or dynamically generated lists.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 3.2

Title:

Write a program to add two matrices.

Theory:

The provided Python program adds two matrices (**matrix1** and **matrix2**). The addition is performed element-wise, and the result is stored in a new matrix **result_matrix**. The program uses nested list comprehensions to iterate through the rows and columns of the input matrices and calculates the sum for each corresponding element.

Code:

```
matrix1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
matrix2 = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]


result_matrix = [[matrix1[i][j] + matrix2[i][j] for j in range(len(matrix1[0]))] for i in range(len(matrix1))]

print("Matrix after Addition:")
for row in result_matrix:
    print(row)
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Matrix after Addition:
[10, 10, 10]
[10, 10, 10]
[10, 10, 10]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively demonstrates how to add two matrices in Python using nested list comprehensions. The resulting matrix is then printed to display the sum of the input matrices.

The program assumes that the input matrices are predefined for demonstration purposes. However, it could be adapted to take user input or work with dynamically generated matrices.

In summary, the program showcases the simplicity and conciseness of Python when performing matrix addition using list comprehensions.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 3.3

Title:

Write a Python function that takes a list and returns a new list with distinct elements from the first list.

Theory:

The provided Python program defines a function **display_distinct_elements** that takes a list as input and returns a new list with distinct elements. The function uses the **set()** constructor to eliminate duplicate elements and then converts the set back to a list. An example is provided where the function is called with an original list containing duplicate elements, and the resulting list with distinct elements is printed.

Code:

```
def display_distinct_elements(input_list):
    return list(set(input_list))

# Example
original_list = [1, 2, 2, 3, 4, 4, 5, 6, 6]
distinct_list = display_distinct_elements(original_list)
print("Original List:", original_list)
print("List with Distinct Elements:", distinct_list)
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Original List: [1, 2, 2, 3, 4, 4, 5, 6, 6]
List with Distinct Elements: [1, 2, 3, 4, 5, 6]
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program demonstrates a simple and effective approach to remove duplicate elements from a list in Python. The use of a set ensures that only unique elements are retained, and converting it back to a list maintains the order of the elements.

The program assumes that the user will input a list, and it could be easily adapted to handle user input or work with dynamically generated lists.

In summary, the program showcases a practical way to obtain a list with distinct elements using Python's set functionality.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 3.4

Title:

Write a program to Check whether a number is perfect or not.

Theory:

The provided Python program checks whether a given number is a perfect number or not. A perfect number is a positive integer that is equal to the sum of its proper divisors, excluding itself.The program defines a function **perfect_number** that takes a number as input and calculates its divisors using a list comprehension. The sum of the divisors is then compared to the original number, and if they are equal, the number is considered perfect.An example is provided where the user is prompted to enter a number, and the program determines whether it is a perfect number or not.

Code:

```
def perfect_number(number):
    divisors = [i for i in range(1, number) if number % i == 0]
    return sum(divisors) == number

num=int(input("Enter a number:"))
if perfect_number(num):
    print(f"{num} is a Perfect Number.")
else:
    print(f"{num} is not a Perfect Number.")
```

Output (screenshot):

```
Enter a number:5
5 is not a Perfect Number.
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Enter a number:6
6 is a Perfect Number.
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Enter a number:48
48 is not a Perfect Number.
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Enter a number:24
24 is not a Perfect Number.
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively checks whether a given number is a perfect number using Python. The use of list comprehension simplifies the generation of divisors, and the sum of divisors is checked to identify perfect numbers.

The program assumes that the user will enter a positive integer, and additional input validation could be added for robustness.

In summary, the program demonstrates a straightforward approach to determine whether a number is perfect or not, utilizing list comprehension and conditional statements in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 3.5

Title:

Write a Python function that accepts a string and counts the number of upper and lower case letters. string_test= “Today is My Best Day”

Theory:

The provided Python program defines a function **count** that takes a string as input and counts the number of upper-case and lower-case letters. The function uses list comprehensions to count the occurrences of upper-case and lower-case letters separately.

An example is provided where the function is called with the string "Today is My Best Day", and the counts of upper-case and lower-case letters are printed.

Code:

```
def count(string):
    upper_count = sum(1 for char in string if char.isupper())
    lower_count = sum(1 for char in string if char.islower())
    return upper_count, lower_count

string_test = 'Today is My Best Day'
print(string_test)
upper, lower = count(string_test)
print(f"Number of Upper-case letters: {upper}")
print(f"Number of Lower-case letters: {lower}")
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q3.py
Today is My Best Day
Number of Upper-case letters: 4
Number of Lower-case letters: 12
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program effectively counts the number of upper-case and lower-case letters in a given string using Python. The use of list comprehensions simplifies the counting process, and the counts are returned by the function.

The program assumes that the input string contains alphabetic characters, and additional input validation could be added if needed.

In summary, the program demonstrates a practical approach to count upper-case and lower-case letters in a string, leveraging Python's string methods and list comprehensions.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.1

Title:

Write a program to Create Employee Class & add methods to get employee details & print.

Theory:

The provided Python program defines an **Employee** class with an **__init__** method for initializing the attributes (**name**, **employee_id**, **department**). Additionally, the class has a **details** method that returns a formatted string containing the employee details.

An instance of the **Employee** class is created (in this case, **employee1**), and its details are printed using the **details** method.

Code:

```
class Employee:
    def __init__(self, name, employee_id, department):
        self.name = name
        self.employee_id = employee_id
        self.department = department

    def details(self):
        return f"Name: {self.name}\nEmployee ID: {self.employee_id}\nDepartment: {self.department}"

employee1 = Employee("Prem Thakare", "02", "Btech")
print(employee1.details())
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
Name: Prem Thakare
Employee ID: 02
Department: Btech
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program demonstrates the creation of a simple **Employee** class in Python with attributes and methods. The **__init__** method initializes the attributes during object creation, and the **details** method returns a formatted string containing employee information.

The program assumes predefined values for the employee instance, but it could be adapted to take user input or work with dynamically generated employee details.

In summary, the program illustrates the basics of creating a class, initializing attributes, and defining methods in Python. The class provides a blueprint for creating and interacting with employee objects.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.2

Title:

Write a program to take input as name, email & age from user using combination of keywords argument and positional arguments (*args and **kwargs) using function,

Theory:

The provided Python program defines a function **details** that takes a combination of positional arguments (***args**) and keyword arguments (****kwargs**). The function prompts the user to input their name, email, and age using the **input** function and then prints additional details provided through positional and keyword arguments.

An example function call is made with additional arguments and keyword arguments.

Code:

```python
def details(*args, **kwargs):
    name = input("Enter Name: ")
    email = input("Enter Email: ")
    age = int(input("Enter Age: "))

    print("Additional Details:")
    for arg in args:
        print(arg)

    for key, value in kwargs.items():
        print(f"{key}: {value}")
details("Additional Arg1", "Additional Arg2", additional_kwarg="Additional KWarg")
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
Enter Name: Prem Thakare
Enter Email: premthakare@gmail.com
Enter Age: 18
Additional Details:
Additional Arg1
Additional Arg2
additional_kwarg: Additional KWarg
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program showcases the use of a function with a combination of positional (***args**) and keyword (****kwargs**) arguments. The function collects basic information (name, email, age) from the user using the **input** function and then prints additional details provided through the function call.

The program assumes that the user will input a valid integer for age, and additional input validation could be added for robustness.

In summary, the program demonstrates a flexible approach to gathering and processing user information through a combination of different types of function arguments in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.3

Title:

Write a program to admit the students in the different Departments(pgdm/btech) and count the students. (Class, Object and Constructor).

Theory:

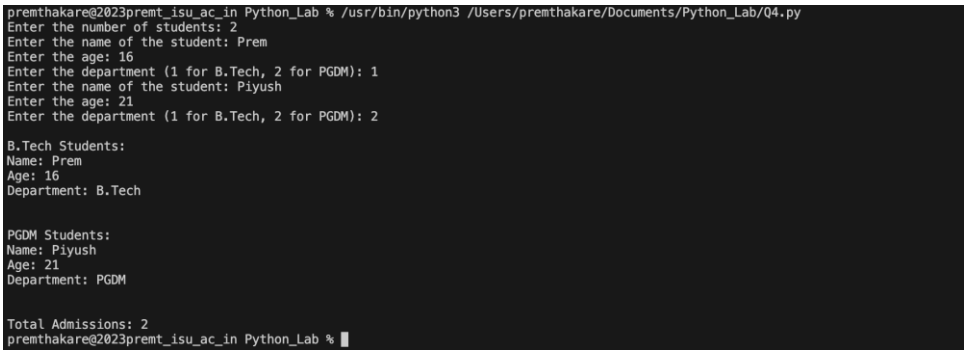
The provided Python program uses classes and objects to simulate the admission of students into different departments (B.Tech and PGDM). The program defines a class **ITM** with a class variable **count** to keep track of the total number of students admitted. The class has methods **get** to get details from the user and **display** to display the student's information.

The program then prompts the user to enter the number of students and iteratively creates instances of the **ITM** class to get details and classify students into B.Tech or PGDM based on the department code. Finally, it displays the details of the admitted students and the total number of admissions.

Code:

```python
class ITM:
    count = 0
    def get(self):
        self.name = input("Enter the name of the student: ")
        self.age = int(input("Enter the age: "))
        self.dep = int(input("Enter the department (1 for B.Tech, 2 for PGDM): "))
        ITM.count += 1

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
        if self.dep == 1:
            print("Department: B.Tech")
        elif self.dep == 2:
            print("Department: PGDM")
        print()


students = int(input("Enter the number of students: "))

btech_students = []
pgdm_students = []

for i in range(students):
    student = ITM()
    student.get()

    if student.dep == 1:
        btech_students.append(student)
    elif student.dep == 2:
        pgdm_students.append(student)

print("\nB.Tech Students:")
for student in btech_students:
    student.display()

print("\nPGDM Students:")
for student in pgdm_students:
    student.display()

print("\nTotal Admissions:", ITM.count)
```

Output (screenshot):



Conclusion:

This program demonstrates the use of classes, objects, and constructors in Python to manage student admissions in different departments. It uses class methods to get and display information, and class variables to keep track of the total number of admissions.

The program assumes that the user will enter valid inputs for name, age, and department code. Additional input validation could be added for robustness.

In summary, the program provides a basic example of using object-oriented concepts in Python to model a scenario involving student admissions and department classification.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.4

Title:

Write a program that has a class store which keeps the record of code and price of product display the menu of all product and prompt to enter the quantity of each item required and finally generate the bill and display the total amount.

Theory:

The provided Python program defines a **Store** class that keeps a record of product codes and prices. It includes methods to display the product menu, take orders, generate bills, and display the total amount. The program creates an instance of the **Store** class, displays the menu, takes orders for specific products and quantities, and then generates and displays the bill with the total amount.

Code:

```python
class Store:
    def __init__(self):
        self.products = {"Product1": 10, "Product2": 20, "Product3": 30}
        self.bill = {}

    def display_menu(self):
        print("Product Menu:")
        for product, price in self.products.items():
            print(f"{product}: Rs.{price}")

    def generate_bill(self):
        print("Your Bill:")
        total_amount = 0
        for product, quantity in self.bill.items():
            price = self.products[product]
            total_price = price * quantity
            print(f"{product} x {quantity}: Rs.{total_price}")
            total_amount += total_price
        print("Total Amount: Rs.{}".format(total_amount))

    def take_order(self, product, quantity):
        if product in self.products and quantity > 0:
            self.bill[product] = quantity
            print(f"{quantity} {product}(s) added to your bill.")
        else:
            print("Invalid product or quantity.")

store = Store()
store.display_menu()
store.take_order("Product1", 2)
store.take_order("Product3", 1)
store.generate_bill()
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
Product Menu:
Product1: Rs.10
Product2: Rs.20
Product3: Rs.30
2 Product1(s) added to your bill.
1 Product3(s) added to your bill.
Your Bill:
Product1 x 2: Rs.20
Product3 x 1: Rs.30
Total Amount: Rs.50
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program demonstrates the use of a class to model a store with products, prices, and an ordering system. The class includes methods for displaying the menu, taking orders, and generating bills. It uses a dictionary to store products and their prices, and another dictionary (**bill**) to keep track of the customer's order.

The program assumes that the user will enter valid product names and quantities. Additional input validation could be added for robustness.

In summary, the program provides a basic example of how a store might use a class to manage products, orders, and generate bills in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.5

Title:

Write a program to take input from user for addition of two numbers using (single inheritance).

Theory:

The provided Python program demonstrates single inheritance, where the **UserInput** class inherits from the **Addition** class. The **Addition** class has a method **add_numbers** that adds two numbers, and the **UserInput** class has a method **get_user_input** that takes user input for two numbers and calls the **add_numbers** method to display the sum.

Code:

```
class Addition:
    def add_numbers(self, num1, num2):
        return num1 + num2

class UserInput(Addition):
    def get_user_input(self):
        num1 = int(input("Enter the first number: "))
        num2 = int(input("Enter the second number: "))
        result = self.add_numbers(num1, num2)
        print(f"The sum of {num1} and {num2} is: {result}")


user_input = UserInput()
user_input.get_user_input()
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
Enter the first number: 23
Enter the second number: 12
The sum of 23 and 12 is: 35
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program illustrates the concept of single inheritance in Python, where the **UserInput** class inherits the functionality of the **Addition** class. The **UserInput** class extends the functionality by providing a method to get user input and use the inherited method to perform addition.

The program assumes that the user will input valid integers for the addition. Additional input validation could be added for robustness.

In summary, the program showcases the basic structure of a class hierarchy using single inheritance in Python to perform addition based on user input.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.6

Title:

Write a program to create two base classes LU and ITM and one derived class.(Multiple inheritance).

Theory:

The provided Python program demonstrates multiple inheritance, where there are two base classes (**LU** and **ITM**) and one derived class (**Derived**) that inherits from both base classes. The base classes have methods (**display_LU** and **display_ITM**), and the derived class inherits these methods.

Code:

```
class LU:
    def display_LU(self):
        print("LU Class")

class ITM:
    def display_ITM(self):
        print("ITM Class")

class Derived(LU, ITM):
    pass


derived_object = Derived()
derived_object.display_LU()
derived_object.display_ITM()
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
LU Class
ITM Class
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program showcases the concept of multiple inheritance in Python. The **Derived** class inherits from both the **LU** and **ITM** classes, and an instance of the **Derived** class is created to demonstrate the use of methods from both base classes.

The program doesn't add any new methods or attributes to the **Derived** class, but it inherits the behavior from both base classes.

In summary, the program illustrates the use of multiple inheritance in Python, where a derived class inherits from more than one base class.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.7

Title:

Write a program to implement Multilevel inheritance, Grandfather Father Child to show property inheritance from grandfather to child.

Theory:

The provided Python program demonstrates multilevel inheritance with three classes: **Grandfather**, **Father**, and **Child**. Each class inherits from the previous one, forming a chain of inheritance. The program collects information about the grandfather's inheritance, father's name and purchased property, and finally, the child's name and purchased property. The total assets inherited by the child are then displayed.

Code:

```python
class Grandfather:
    def __init__(self):
        self.gfname = " Thakare"
        self.inheritance = 70000000

class Father(Grandfather):
    def __init__(self):
        super(Father, self).__init__()
        self.fname = input("Enter the Father Name: ") + self.gfname
        self.finheritance = float(input("Enter the Property purchased by Father: ")) + self.inheritance

class Child(Father):
    def __init__(self):
        super(Child, self).__init__()
        self.child = input("Enter the Child Name: ") + " " + self.fname
        self.cinheritance = float(input("Enter the Property purchased by " + self.child + " is: ")) + self.finheritance

c = Child()
print("\nHi,", c.child)
print("\nYour Total Assets is:\nInherited:", c.inheritance, "\nPurchased:", c.finheritance, "\nYour Property:", c.cinheritance)
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py
Enter the Father Name: Anil
Enter the Property purchased by Father: 10000000
Enter the Child Name: Prem
Enter the Property purchased by Prem Anil Thakare is: 509128421

Hi, Prem Anil Thakare

Your Total Assets is:
Inherited: 70000000
Purchased: 80000000.0
Your Property: 589128421.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

This program illustrates the concept of multilevel inheritance in Python. Each class in the hierarchy (**Grandfather**, **Father**, **Child**) extends the properties of the previous one. The **Child** class inherits from both the **Father** and **Grandfather** classes.

The program collects input for the names and property purchases of each family member and calculates the total assets inherited by the child.

The user is prompted to enter information in a structured manner, and the program assumes that the user will input valid values. Additional input validation could be added for robustness.

In summary, the program showcases multilevel inheritance to represent a family tree and inheritance of properties from the grandfather to the child in Python.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 4.8

Title:

Write a program Design the Library catalogue system using inheritance take base class (library item) and derived class (Book, DVD & Journal) Each derived class should have unique

attribute and methods and system should support Check in and check out the system. (Using Inheritance and Method overriding)

Theory:

The provided Python program demonstrates the design of a Library Catalog System using inheritance. It defines a base class **LibraryItem** and three derived classes **Book**, **DVD**, and **Journal**. Each derived class has unique attributes and methods, and the system supports the check-in and check-out functionality using method overriding.

Code:

```
class LibraryItem:
    def __init__(self, title, item_id, no_copies):
        self.title = title
        self.item_id = item_id
        self.no_copies = no_copies
    def display(self):
        print("\nTitle: ",self.title)
        print("Item ID: ",self.item_id)
        print("Num Copies: ",self.no_copies)
    def checkout(self):
        cho=input("Enter the item ID:")
        if (cho=="B12"):
            ch=int(input("Enter the No of Copies: "))
            if (ch<=self.no_copies):
                self.no_copies-=1
            else:
                print("Insufficient Copies")

class Book(LibraryItem):
    def __init__(self, title, item_id, no_copies):
        super().__init__(title, item_id, no_copies)
        self.title = title
    def display(self):
        super().display()

class DVD(LibraryItem):
    def __init__(self, title, item_id, no_copies):
        super().__init__(title, item_id, no_copies)
        self.title = title
    def display(self):
        super().display()


class Journal(LibraryItem):
    def __init__(self, title, item_id, no_copies):
        super().__init__(title, item_id, no_copies)
        self.titler = title
    def display(self):
        super().display()
```

```
book = Book("C++ Obj.", "B12", 10)
dvd = DVD("Animal", "D13", 5)
journal = Journal("Indian Journal", "J14",15)

book.display()
dvd.display()
journal.display()
```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q4.py

Title:  C++ Obj.
Item ID:  B12
Num Copies:  10

Title:  Animal
Item ID:  D13
Num Copies:  5

Title:  Indian Journal
Item ID:  J14
Num Copies:  15
```

Conclusion:

- **Base Class (LibraryItem):** The base class includes common attributes like title, item ID, and the number of copies. It also includes a method for displaying the details and a method for checking out items.
- **Derived Classes (Book, DVD, Journal):** Each derived class inherits from the base class and adds its unique attributes. The **display** method is overridden to display details specific to each item type.
- **Check-Out Functionality:** The **checkout** method is implemented in the base class. The overridden method in each derived class allows checking out items based on the item ID and the number of copies.
- **Instances and Display:** Instances of each class (**Book**, **DVD**, **Journal**) are created, and the **display** method is called to showcase the details of each item.

The program assumes a simple check-out mechanism where the user needs to input the item ID and the number of copies they want to check out.

In summary, the program demonstrates the use of inheritance and method overriding to design a Library Catalog System in Python. Each item type inherits common functionality from the base class while having its specific attributes and behavior.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 5.1

Title:

Write a program to create my_module for addition of two numbers and import it in main script.

Theory:

The provided Python program demonstrates the creation of a module (**my_module**) for the addition of two numbers and its import in the main script. The module contains a function **addition** that performs the addition operation.

Code:

my_module.py > ...

```python
# 5.1 Write a program to create my_module for addition of two numbers and import
# it in main script.

def addition(num1, num2):
    return num1 + num2

```

Q5A_main.py > ...

```python
# 5.1 Write a program to create my_module for addition of two numbers and import
# it in main script.

import my_module

num1=float(input("Enter the first number:"))
num2=float(input("Enter the second number:"))
result = my_module.addition(num1, num2)

print("Result of addition:", result)

```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q5A_main.py
Enter the first number:12
Enter the second number:45
Result of addition: 57.0
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

- **Module Creation (my_module):** The module is created with a single function **addition** that takes two numbers as input and returns their sum.
- **Importing the Module:** In the main script, the **my_module** is imported using the **import** statement.
- **User Input and Function Call:** The user is prompted to input two numbers. The **addition** function from the imported module is then called with these numbers, and the result is displayed.

The program showcases the concept of modular programming, where the functionality is encapsulated in a separate module, promoting code organization and reusability.

In summary, the program demonstrates the creation of a module for a specific functionality (addition of two numbers) and its import in the main script to perform the operation.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 5.2

Title:

Write a program to create the Bank Module to perform the operations such as Check the Balance, withdraw and deposit the money in bank account and import the module in main file.

Theory:

The provided Python program demonstrates the creation of a Bank Module (**bank_module**) with a class **Banking** that performs operations such as checking the balance, withdrawing, and depositing money. The module is then imported into the main script for use.

Code:

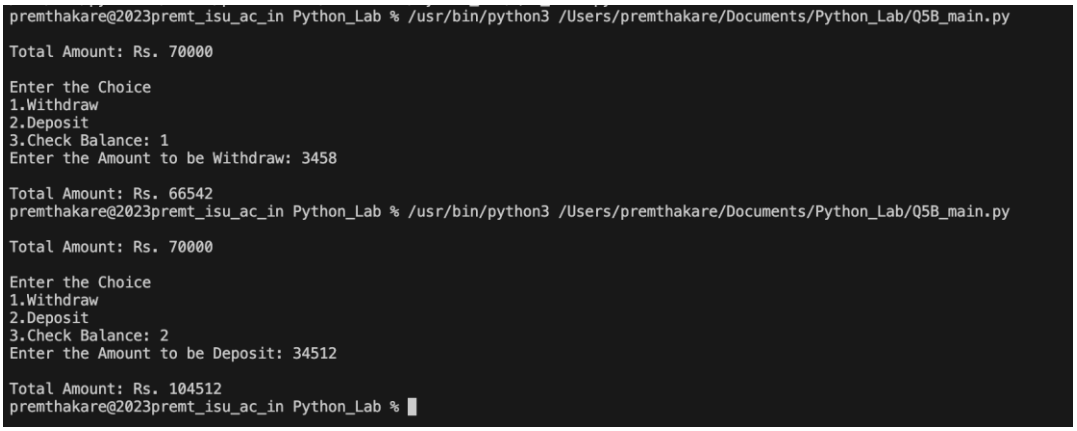
bank_module.py > ...

```python
# 5.2 Write a program to create the Bank Module to perform the operations such as
# Check the Balance, withdraw and deposit the money in bank account and import
# the module in main file.

class Banking:
    def __init__(self):
        self.bala = 70000

    def operation(self, choice, withd, depo):
        if (choice == 1):
            if withd <= self.bala:
                self.bala -= withd
                print("\nTotal Amount: Rs.", self.bala)
            else:
                print("Insufficient Balance to Withdraw")
        elif (choice == 2):
            self.bala += depo
            print("\nTotal Amount: Rs.", self.bala)
        elif (choice == 3):
            print("\nTotal Amount: Rs.", self.bala)
        else:
            print("Invalid Input")
```

Q5B_main.py > ...

```python
# 5.2 Write a program to create the Bank Module to perform the operations such as
# Check the Balance, withdraw and deposit the money in bank account and import
# the module in main file.

import bank_module as balala

def atm():
    b = balala.Banking()
    print("\nTotal Amount: Rs.", b.bala,"\n")
    choice = int(input("Enter the Choice \n1.Withdraw\n2.Deposit\n3.Check Balance: "))

    if choice == 1:
        withd = int(input("Enter the Amount to be Withdraw: "))
        b.operation(choice, withd, None)

    elif choice == 2:
        depo = int(input("Enter the Amount to be Deposit: "))
        b.operation(choice, None, depo)

    elif choice == 3:
        b.operation(choice, None, None)

    else:
        print("Invalid Input")

atm()
```

Output (screenshot):



Conclusion:

- **Bank Module (bank_module):** The module contains a class **Banking** with an **__init__** method initializing the initial balance and an **operation** method to perform operations based on user input.
- **Importing the Module:** In the main script, the **bank_module** is imported using the **import** statement with an alias (**as balala**).
- **User Input and Operation:** The **atm** function in the main script prompts the user to input the type of operation they want to perform (withdraw, deposit, or check balance). Based on the user's choice, the **operation** method from the imported module is called, and the corresponding operation is performed.
- **Executing the ATM Function:** The **atm** function is called to simulate an ATM-like interaction where the user can check the balance, withdraw, or deposit money.

The program demonstrates the use of modular programming by creating a separate module for banking operations and importing it into the main script for practical use. This modular approach enhances code organization and reusability.

In summary, the program showcases a basic ATM simulation with banking operations encapsulated in a separate module.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 5.3

Title:

Write a program to create a package with name cars and add different modules (such as BMW, AUDI, NISSAN) having classes and functionality and import them in main file cars.

Theory:

The provided Python program demonstrates the creation of a package named "cars" that contains different modules (BMW, AUDI, NISSAN). Each module represents a car brand and includes a class with functionality. The main file (**cars**) imports these modules and uses their classes to display available car models.

Code:

Bmw.py > ...

```python
class Bmw_m:
    def __init__(self):
        self.models = ['i8', 'x1', 'x5', 'x6']

    def Models(self):
        print('These are the available models for BMW:')
        for model in self.models:
            print("BMW",model,"\n")

```

Audi.py > ...

```python
class Audi_m:

    def __init__(self):
        self.models = ['q7', 'a6', 'a8', 'a3']

    def Models(self):
        print('These are the available models for Audi')
        for model in self.models:
            print('AUDI',model,"\n")

```

Nissan.py > ...

```python
class Nissan_m:
    def __init__(self):
        self.models = ['altima', '370z', 'cube', 'rogue']

    def Models(self):
        print('These are the available models for Nissan')
        for model in self.models:
            print('NISSAN',model,"\n")

```

Q5C_main.py > ...

```python
# Write a program to create a package with name cars and add different modules
# (such as BMW, AUDI, NISSAN) having classes and functionality and import
# them in main file cars.

import Bmw as bmwpro
import Audi as audipro
import  Nissan as nissanpro

ModBMW = bmwpro.Bmw_m()
ModBMW.Models()

ModAudi = audipro.Audi_m()
ModAudi.Models()

ModNissan = nissanpro.Nissan_m()
ModNissan.Models()

```

Output (screenshot):

```
premthakare@2023premt_isu_ac_in Python_Lab % /usr/bin/python3 /Users/premthakare/Documents/Python_Lab/Q5C_main.py
These are the available models for BMW:
BMW i8

BMW x1

BMW x5

BMW x6

These are the available models for Audi
AUDI q7

AUDI a6

AUDI a8

AUDI a3

These are the available models for Nissan
NISSAN altima

NISSAN 370z

NISSAN cube

NISSAN rogue

premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

- **Car Modules (Bmw, Audi, Nissan):** Each module contains a class (**Bmw_m**, **Audi_m**, **Nissan_m**) with an **__init__** method initializing the available car models and a **Models** method to display the models for the respective brand.
- **Importing Modules:** In the main script (**cars**), the modules (**Bmw**, **Audi**, **Nissan**) are imported using the **import** statement with aliases (**as bmwpro**, **as audipro**, **as nissanpro**).
- **Creating Module Instances:** Instances of the classes from each module are created (**ModBMW**, **ModAudi**, **ModNissan**).
- **Displaying Available Car Models:** The **Models** method of each module instance is called to display the available car models for BMW, Audi, and Nissan.
- **Organization into a Package:** The modules are organized into a package named "cars" to facilitate better code organization and encapsulation of related functionality.

The program showcases the use of packages and modules to structure code related to different car brands. The modular approach enhances code organization and reusability, allowing for the easy addition of new car brands in the future.

In summary, the program demonstrates the creation of a package with multiple modules, each representing a car brand with its available models. The main script imports these modules and displays the available car models for each brand.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 6

Title:

Write a program to implement Multithreading. Printing “Hello” with one thread & printing “Hi” with another thread.

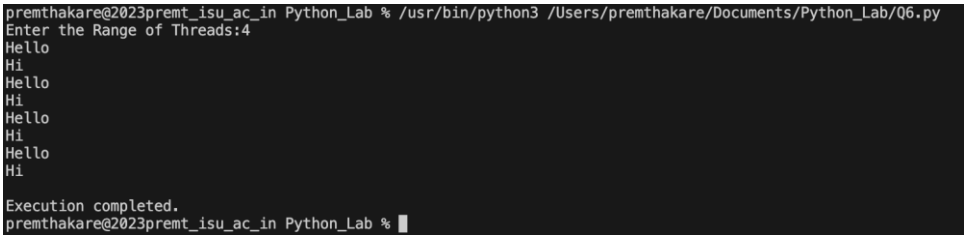
Theory:

The provided Python program demonstrates the implementation of multithreading using the **threading** module. Two threads are created, each printing a message (**"Hello"** and **"Hi"**) a specified number of times.

Code:

```
import threading
import time

ran=int(input("Enter the Range of Threads:"))

def hello():
    for _ in range(ran):
        time.sleep(1)
        print("Hello")

def hi():
    for _ in range(ran):
        time.sleep(1)
        print("Hi")

thread_hello = threading.Thread(target=hello)
thread_hi = threading.Thread(target=hi)

thread_hello.start()
thread_hi.start()

thread_hello.join()
thread_hi.join()

print("\nExecution completed.")
```

Output (screenshot):



Conclusion:

Multithreading is utilized to execute two functions concurrently. The **hello** thread prints "Hello," and the **hi** thread prints "Hi" a specified number of times. The use of threads allows these tasks to be performed concurrently, improving program efficiency and responsiveness.

The user-defined functions (**hello** and **hi**) represent the operations that can be performed concurrently. The program demonstrates the basic principles of multithreading and how it can be applied to execute tasks concurrently in a Python program.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 7.1

Title:

Write a program to use ‘whether API’ and print temperature of any city, also print the sunrise and sunset times for the same humidity of that area.

Theory:

The provided Python program uses the 'OpenWeatherMap API' to retrieve weather information for a specified city. It prints the temperature, humidity, sunrise time, and sunset time for the given city.

Code:

```python
import requests

api_key = "65a84ed61a806ba4e196b8be0ef37f89"
url = "http://api.openweathermap.org/data/2.5/weather"

city = str(input("Enter city to search: "))

params = {'q': city, 'appid': api_key}
response = requests.get(url, params=params)

if response.status_code == 200:
    data = response.json()

    temperature = data['main']['temp'] - 273.15
    humidity = data['main']['humidity']
    sunrise_time = data['sys']['sunrise']
    sunset_time = data['sys']['sunset']

    print(f"Temperature in {city}: {temperature:.2f}°C")
    print(f"Humidity in {city}: {humidity}%")
    print(f"Sunrise in {city}: {sunrise_time}")
    print(f"Sunset in {city}: {sunset_time}")
else:
    print(f"Error: Unable to fetch data. Status Code: {response.status_code}")
```

Output (screenshot):

```
Enter city to search: Nashik
Temperature in Nashik: 18.81°C
Humidity in Nashik: 57%
Sunrise in Nashik: 1704159592
Sunset in Nashik: 1704199000
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

The program demonstrates how to use the OpenWeatherMap API to fetch weather data for a specified city. It extracts and displays information such as temperature, humidity, sunrise time, and sunset time. The API key is crucial for authentication when making requests to the API. This program showcases the use of APIs to obtain real-time data and process it in a Python program.

Name of Student: Prem Anil Thakare

Roll Number: 02

Experiment No: 7.2

Title:

Write a program to use the 'API' of crypto currency.

Theory:

The provided Python program uses the 'CoinGecko API' to retrieve real-time cryptocurrency prices in USD and INR. It continuously prompts the user to input a cryptocurrency name and displays its current prices in both currencies. The user can choose to exit the program by entering 'n'.

Code:

```
import requests
api_id="CG-MVwfDPbzcds9onoar9oKSaqf"
while True:
    coin=input("Enter cryptocoin: ")
    response = requests.get(f"https://api.coingecko.com/api/v3/simple/price?ids={coin}&vs_currencies=usd,inr&x_cg_demo_api_key={api_id}")
    a=response.json()

    if(response.status_code == 200):
        while input("Press any key to see updates or \"N\" to exit") != "n":
            response = requests.get(f"https://api.coingecko.com/api/v3/simple/price?ids={coin}&vs_currencies=usd,inr&x_cg_demo_api_key={api_id}")
            a=response.json()
            for i in a:
                print("₹",a[i]['inr'])
        break
    else:
        print("Invalid currency name please retype")
```

Output (screenshot):

```
Enter cryptocoin: Bitcoin
Press any key to see updates or "N" to exit
₹ 3638504
Press any key to see updates or "N" to exit
₹ 3638504
Press any key to see updates or "N" to exitn
premthakare@2023premt_isu_ac_in Python_Lab %
```

Conclusion:

The program demonstrates how to use the CoinGecko API to fetch real-time cryptocurrency prices in USD and INR. It provides a continuous loop for the user to input cryptocurrency names and see updates. The API key is crucial for authentication when making requests to the API. This program showcases the use of APIs to obtain real-time financial data and interact with it in a Python program.