ITM SKILLS UNIVERSITY

INSTITUTE OF TECHNOLOGY AND MANAGEMENT SKILLS UNIVERSITY
KHARGHAR, NAVI MUMBAI

PYTHON PROGRAMMING LAB

Prepared by:
Name of Student: Ashlin Lee George
Roll No: 11
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. 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 aprogram 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, 4.8 Grandfather→Father-→Child to show property inheritance from grandfather to child.
	4.9 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: Ashlin Lee George
Roll Number: **11**

Experiment No: 1.1

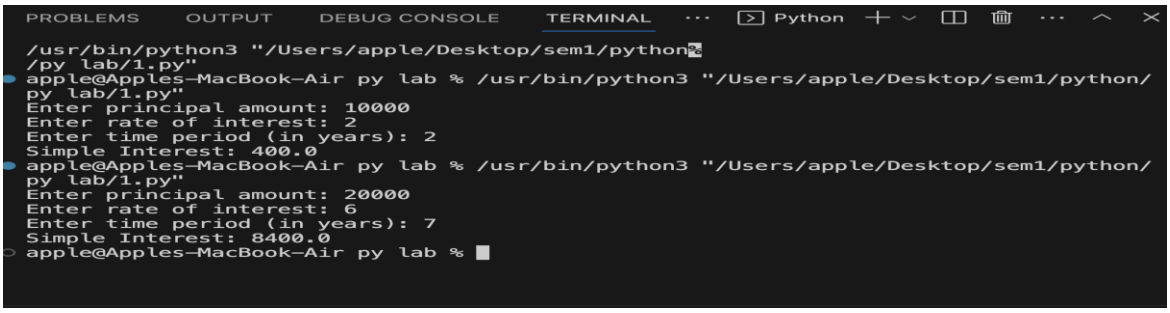
Title:
Write a program to compute Simple Interest.

Theory:
The Python program calculates Simple Interest 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×r×t) /100

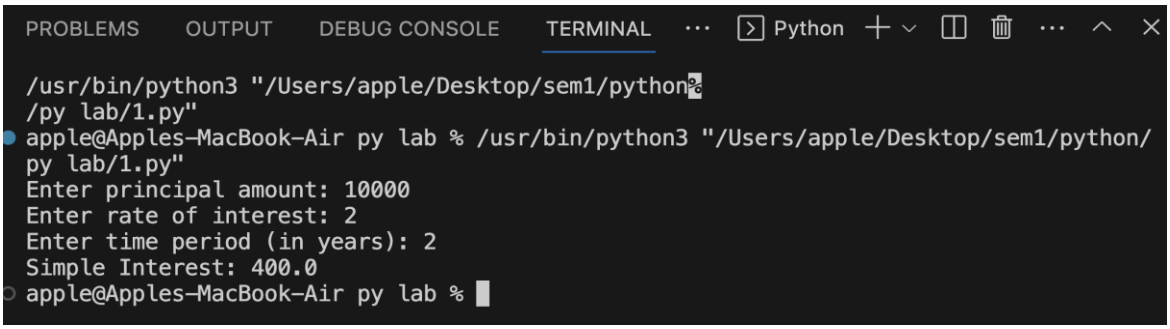
Code:

```
p= float(input("Enter principal amount: "))
r= float(input("Enter rate of interest: "))
t= float(input("Enter time period (in years): "))
simple_interest = (p * r * t) / 100
print("Simple Interest:",simple_interest)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:
The Python program efficiently calculates Simple Interest based on user input, adhering to the formula and providing clear output.

Experiment No: 1.2

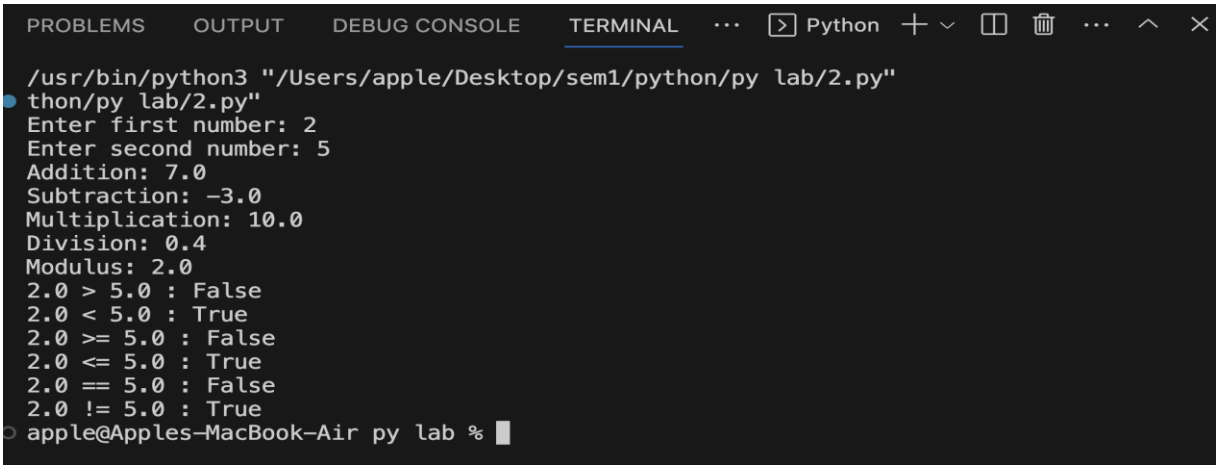
Title: Write a program to perform arithmetic, Relational operators.

Theory:
Arithmetic operators (+, -, *, /, %) manipulate numerical values for addition, subtraction, multiplication, division, and modulus (remainder). Relational operators (==, !=, >, <, >=, <=) compare values, determining equality, inequality, or order. Both sets of operators facilitate numerical computations and logical comparisons in programming.

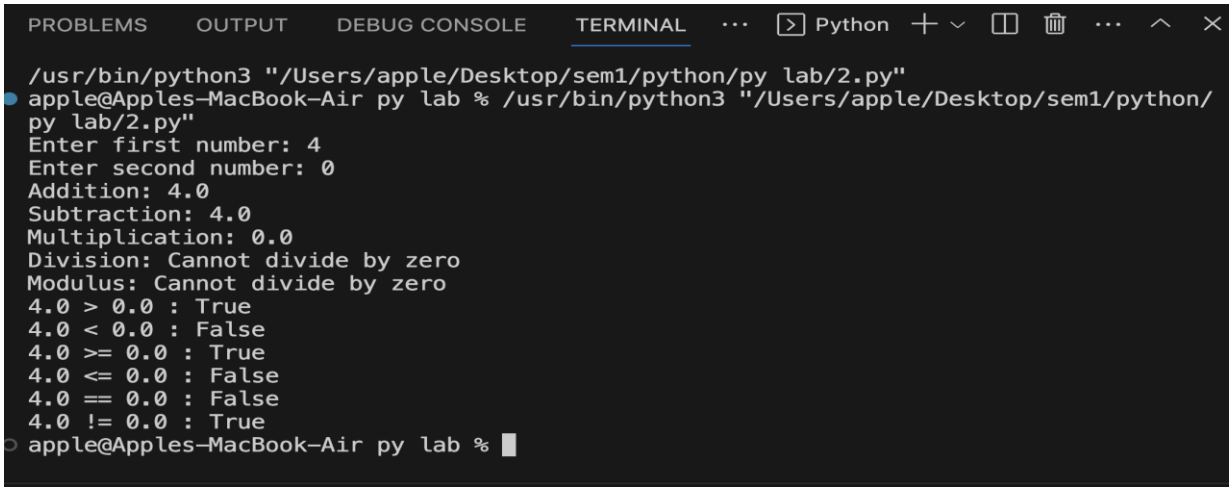
Code:

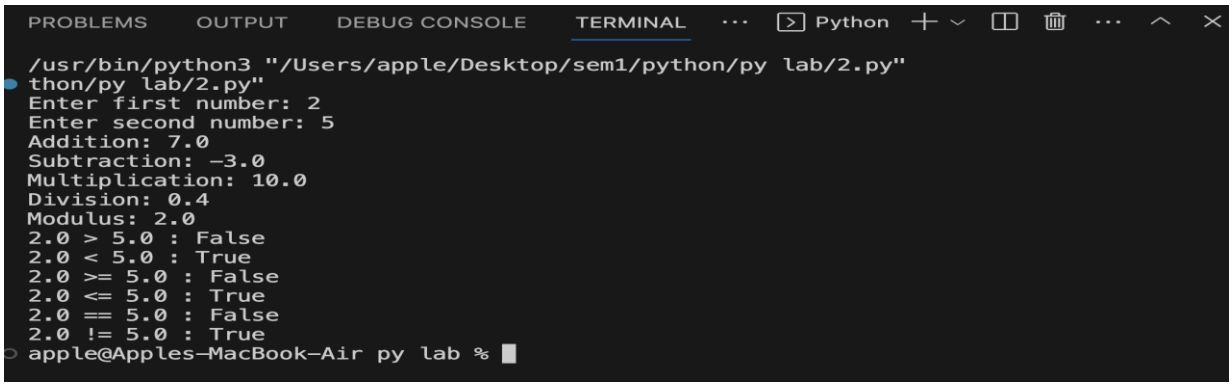
```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

add = num1 + num2
print("Addition:",add)

sub = num1 - num2
print("Subtraction:",sub)

mul = num1 * num2
print("Multiplication:",mul)

div = num1 / num2 if num2 != 0 else "Cannot divide by zero"
print("Division:",div)

mod = num1 % num2 if num2 != 0 else "Cannot divide by zero"
print("Modulus:",mod)

greater = num1 > num2
print(num1, ">", num2, ":", greater)

less = num1 < num2
print(num1, "<", num2, ":", less)

greater_equal = num1 >= num2
print(num1, ">=", num2, ":", greater_equal)

less_equal = num1 <= num2
print(num1, "<=", num2, ":", less_equal)

equal_to = num1 == num2
print(num1, "==", num2, ":", equal_to)

not_equal_to = num1 != num2
print(num1, "!=", num2, ":", not_equal_to)
```

Output: (screenshot)



Test Case: Any two (screenshot)





Conclusion:

Arithmetic operators enable fundamental mathematical computations, while relational operators empower logical evaluations in Python programs. Their effective application ensures precise calculations and accurate decision-making.

Experiment No:1.3

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

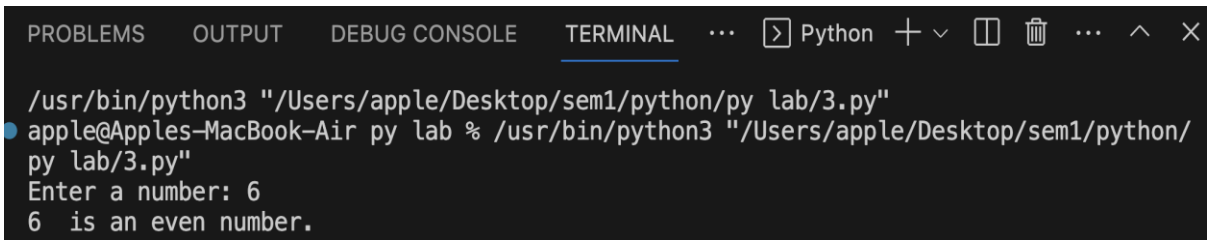
Theory:

In Python, the modulus operator (%) helps determine whether a number is even or odd. When a number is divided by 2, if the remainder is 0, it's even; otherwise, it's odd. The program takes user input and utilizes this logic to check if the number is even or odd.

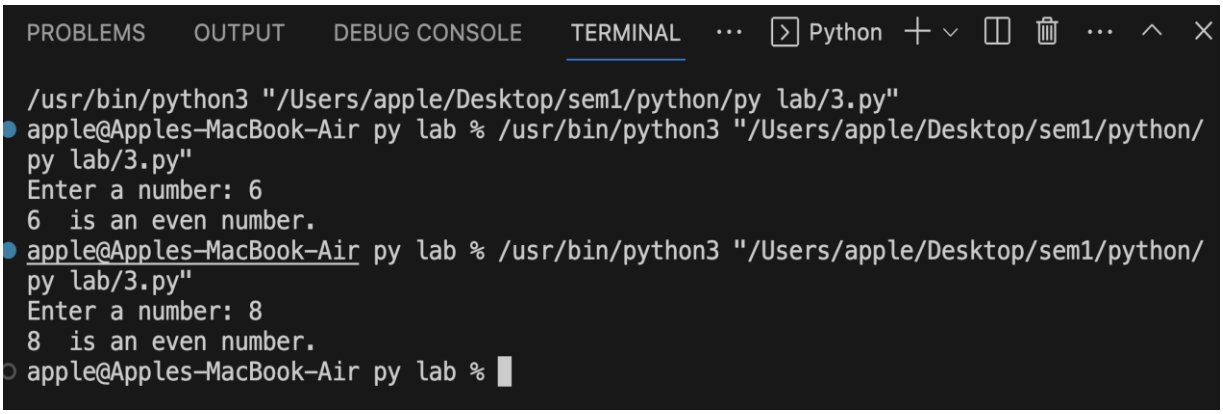
Code:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
   print(num," is an even number.")
else:
   print(num," is an odd number.")
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The Python program identifies whether a given number is even or odd by utilizing the modulus operator. By assessing the remainder when dividing by 2, it accurately categorises the number.

Experiment No:1.4

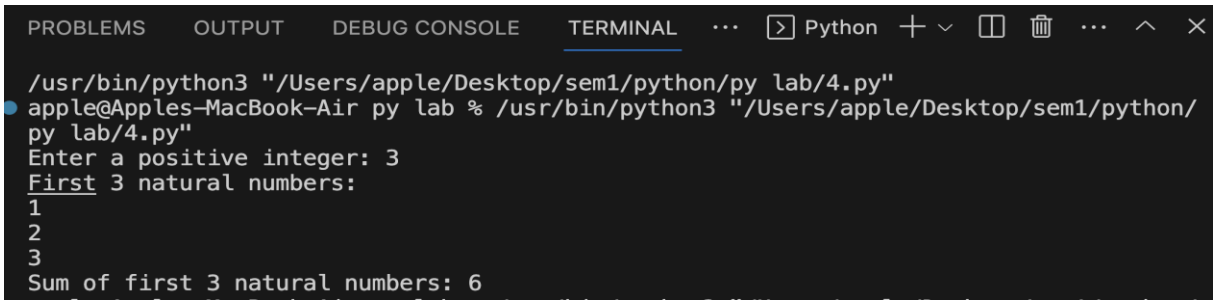
Title:
Write a program to print first n natural number & their sum.

Theory:
The program utilizes a loop to print the first 'n' natural numbers and computes their sum using a simple formula, displaying both outputs.

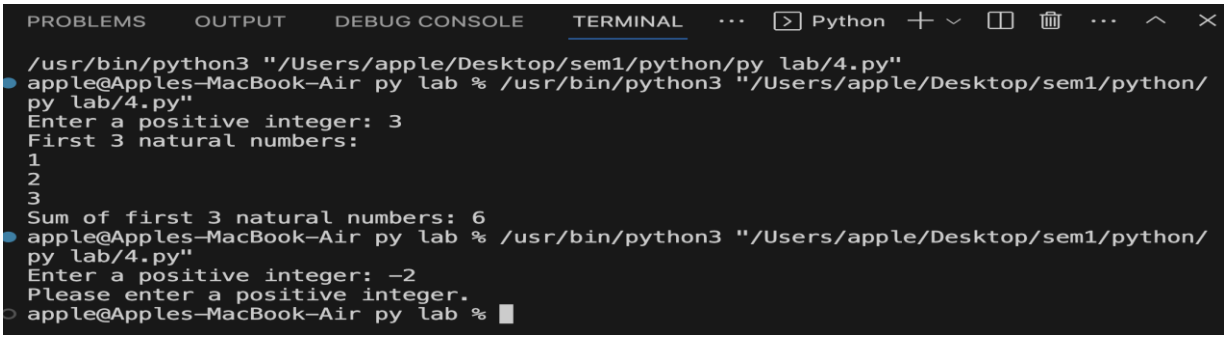
Code:

```
n = int(input("Enter a positive integer: "))
if n <= 0:
   print("Please enter a positive integer.")
else:
   sum = 0
   print("First",n,"natural numbers:")
   for i in range(1, n + 1):
      print(i)
      sum+= i
   print("Sum of first",n, "natural numbers:",sum)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

This Python program efficiently displays the first 'n' natural numbers and their sum, showcasing the simplicity of loops and basic mathematical operations for such tasks.

Experiment No:1.5

Title:
Write a program to determine whether the character entered is a Vowel or not

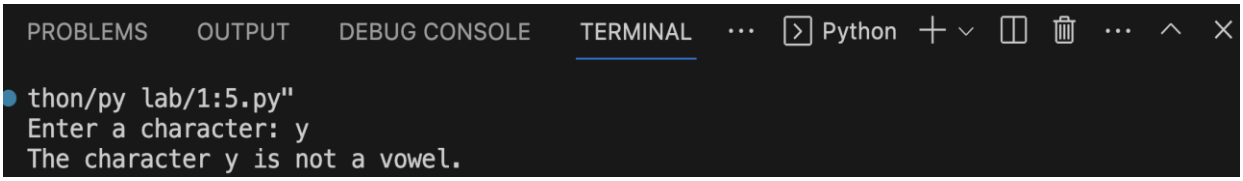
Theory:

The program checks if the input character is a vowel by comparing it to a predefined set of vowels using conditional statements.

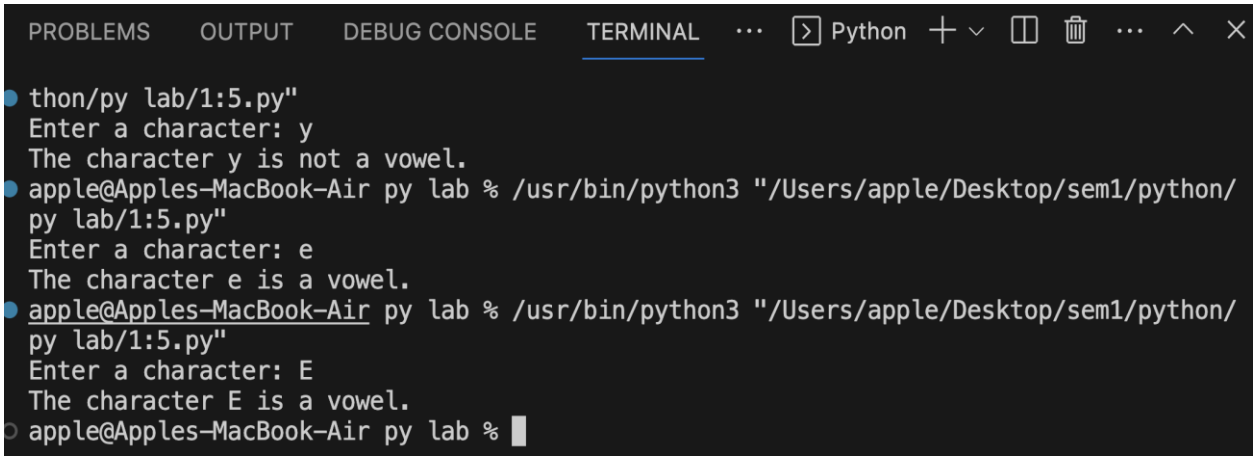
Code:

```
char = input("Enter a character: ")
if char.lower() in 'aeiou':
   print("The character",char, "is a vowel.")
else:
   print("The character",char,"is not a vowel.")
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

This Python program identifies if a character is a vowel, employing conditionals to assess user input against a set of defined vowels.

Experiment No:1.6

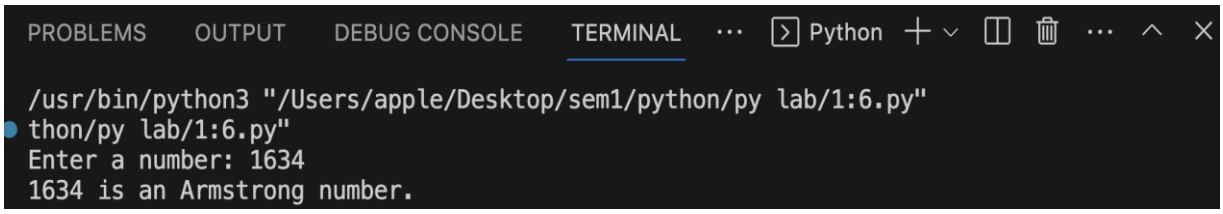
Title:
Write a program to find whether given number is an Armstrong Number.

Theory:
An Armstrong number is a number that equals the sum of its own digits each raised to the power of the number of digits.

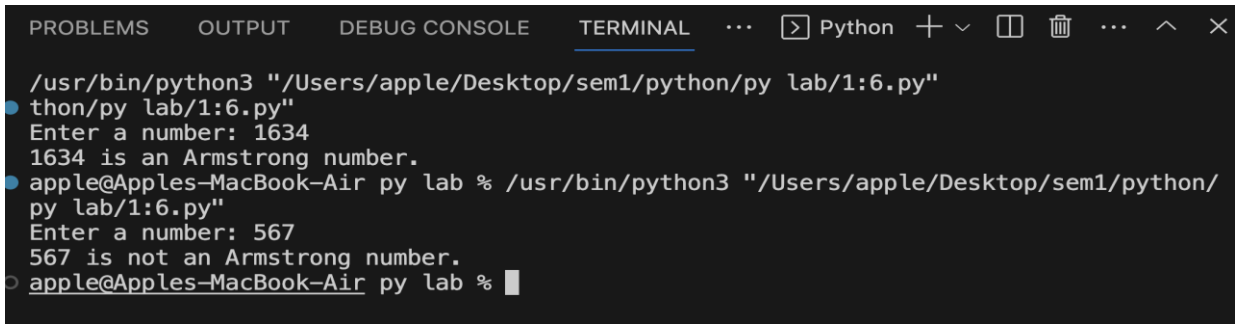
Code:

```
def armstrong_no(num):

    num_str = str(num)
    num_digits = len(num_str)

    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num

numo = int(input("Enter a number: "))

if armstrong_no(numo):
    print(f"{numo} is an Armstrong number.")
else:
    print(f"{numo} is not an Armstrong number.")
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The Python program verifies if a number is an Armstrong number by calculating the sum of its digits raised to the power of the digit count.

Experiment No:1.7

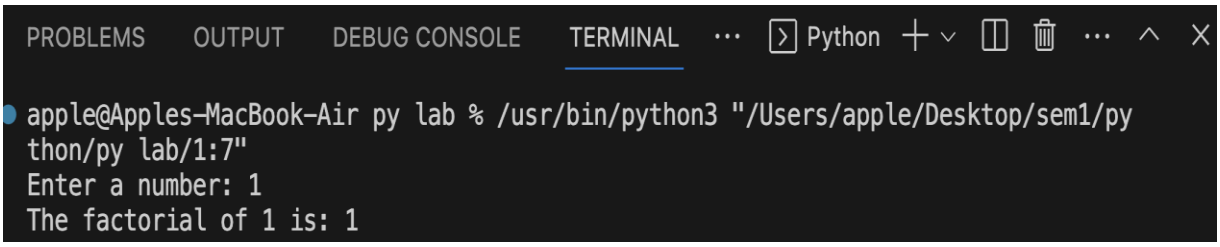
Title:
Write a program using for loop to calculate factorial of a No.

Theory:
Factorial represents the product of all positive integers up to a given number. A for loop iterates to multiply numbers from 1 to the given number.

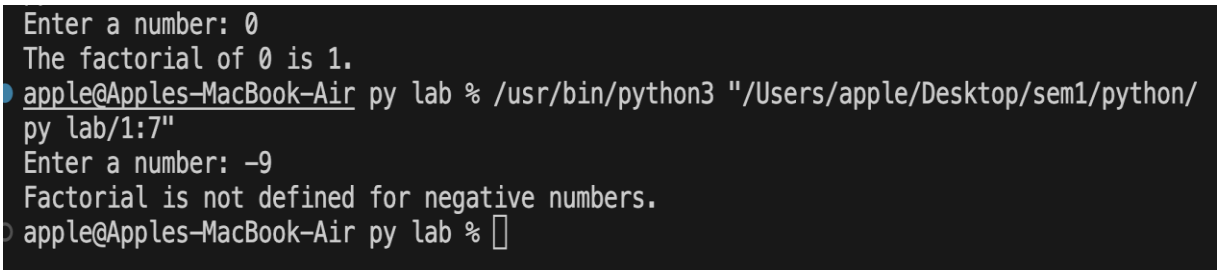
Code:

```
num = int(input("Enter a number: "))
factorial = 1
if num < 0:
   print("Factorial is not defined for negative numbers.")
elif num == 0:
   print("The factorial of 0 is 1.")
else:
   for i in range(1, num + 1):
      factorial *= i
   print("The factorial of",num,"is:",factorial)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

Utilizing a for loop, the program efficiently computes factorial, multiplying numbers sequentially. This simple and effective approach accurately determines factorials in Python.

Experiment No:1.8

Title:
Write a program to print the following pattern

Theory:
Nested loops generate rows and columns, creating a pattern. Loop control and print statements create the desired structure, iterating through rows and columns systematically

Code:

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

```
rows = 5
for i in range(1, rows + 1):
    for j in range(i):
        print(i, end=" ")
    print()
```

```
rows = 5
for i in range(1, rows + 1):
    print(" " * (rows - i), end="")
    for j in range(1, 2 * i):
        print("*", end="")
    print()
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

This Python program uses nested loops to print a specified pattern. It showcases the power of loops in creating structured outputs based on controlled iterations through rows and columns.

Experiment No:2.1

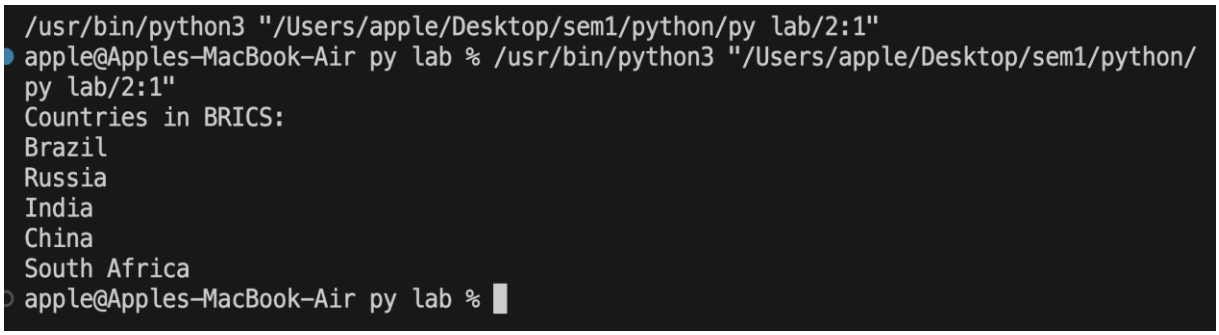
Title:
Write a program that define the list of defines the list of define countries that are in BRICS.

Theory:
BRICS comprises Brazil, Russia, India, China, and South Africa, prominent emerging economies collaborating in economic and political spheres. A Python program can define a list encapsulating these member countries.

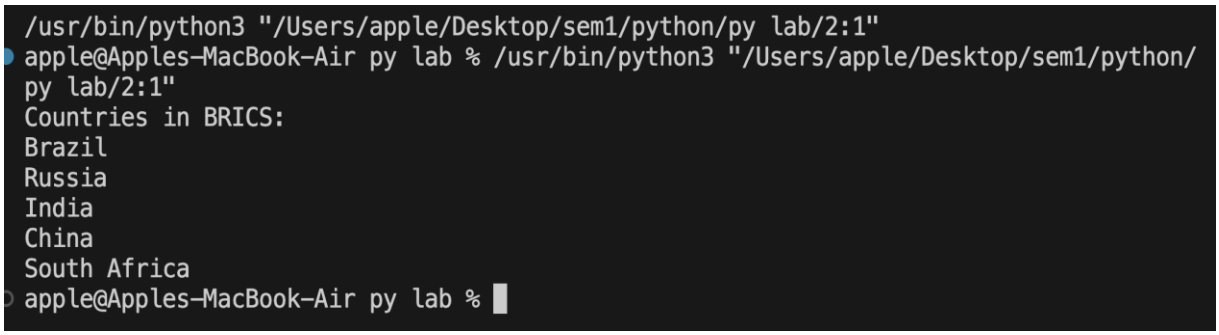
Code:

```
brics_countries = ["Brazil", "Russia", "India", "China", "South Africa"]

print("Countries in BRICS:")
for country in brics_countries:
    print(country)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The Python program efficiently defines a list encompassing BRICS member countries—Brazil, Russia, India, China, and South Africa

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:

Traversing a list in reverse order can be done in Python using the reverse() method to invert the list in place or by employing slicing with [::-1] to create a reversed copy.

Code:

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print("Traversing list in reverse using reverse() method:")
for item in my_list:
    print(item)
```

```
my_list = [1, 2, 3, 4, 5]
print("Traversing list in reverse using slicing:")
for item in my_list[::-1]:
    print(item)
```

Output: (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/2:2.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/2:2.py"
Traversing list in reverse using reverse() method:
5
4
3
2
1
Traversing list in reverse using slicing:
5
4
3
2
1
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/2:2.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/2:2.py"
Traversing list in reverse using reverse() method:
5
4
3
2
1
Traversing list in reverse using slicing:
5
4
3
2
1
apple@Apples-MacBook-Air py lab %
```

Conclusion:

Python allows list reversal using the reverse() method for in-place modification or slicing to create a reversed copy, offering diverse approaches for efficiently traversing lists in reverse order.

Experiment No:2.3

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

Theory:
The program extracts the username and domain from an email address by splitting it at the '@' symbol, creating a tuple with both components for further use.

Code:

```
email = input("Enter an email address: ")
username, _, domain = email.partition("@")
email_tuple = (username, domain)
print("Tuple of username and domain:", email_tuple)
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

This Python program swiftly dissects an email address into its username and domain components, encapsulating them into a tuple for convenient retrieval and processing in various applications.

Experiment No:2.4

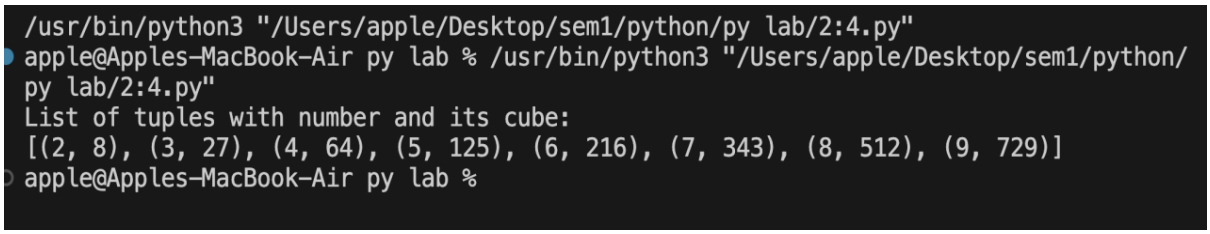
Title:
Write a program to create a list of tuples from given list having number and add its cube in tuple.
i/p: c= [2,3,4,5,6,7,8,9]

Theory:
The program generates a list of tuples by iterating through a given list, creating tuples with each number and its cube.

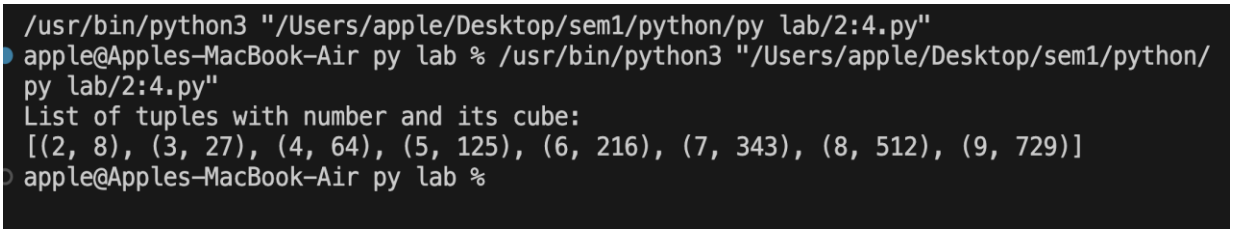
Code:

```
c = [2, 3, 4, 5, 6, 7, 8, 9]
tuple_list = [(num, num ** 3) for num in c]
print("List of tuples with number and its cube:")
print(tuple_list)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

This Python program efficiently constructs tuples, pairing numbers from a list with their respective cubes, creating a new list of tuples.

Experiment No:2.5

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

Theory:
Python's == operator compares if two dictionaries hold the same key-value pairs, highlighting similarities and differences in their contents.

Code:

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'a': 1, 'b': 2, 'c': 3}
if dict1 == dict2:
   print("The dictionaries are equal.")
else:
   print("The dictionaries are not equal.")
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

The == operator efficiently evaluates dictionary equality, determining if both dictionaries contain identical key-value pairs, offering a straightforward method for comparative analysis in Python.

Experiment No:2.6

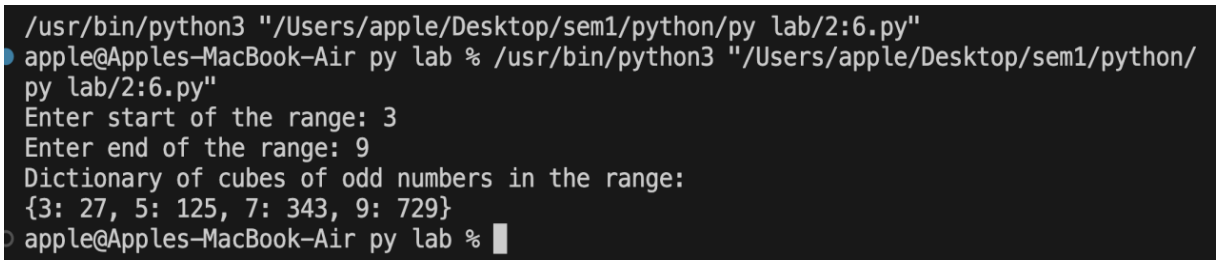
Title:
Write a program that creates dictionary of cube of odd numbers in the range.

Theory:
A Python program generates a dictionary storing cubes of odd numbers within a specified range, using a dictionary comprehension to map odd numbers to their respective cubes.

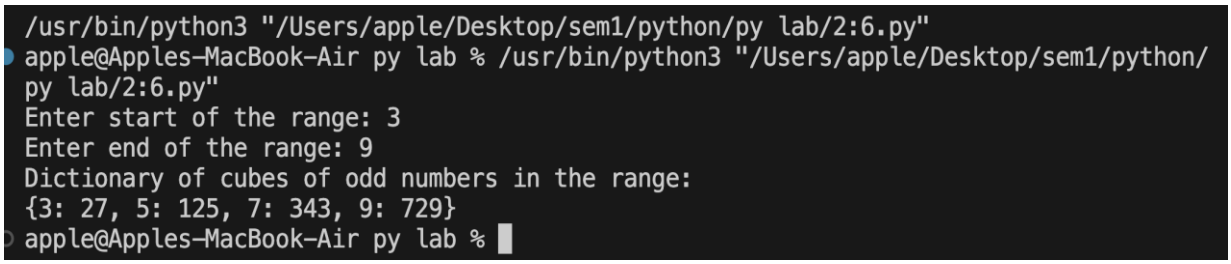
Code:

```
start = int(input("Enter start of the range: "))
end = int(input("Enter end of the range: "))
odd_cubes = {num: num ** 3 for num in range(start, end+ 1) if num % 2 != 0}
print("Dictionary of cubes of odd numbers in the range:")
print(odd_cubes)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The program effectively creates a dictionary containing cubes of odd numbers, showcasing concise implementation through dictionary comprehension for efficient mapping and storage.

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:

List slicing in Python involves extracting specific elements or sublists from a list using indexing and slicing operations. It allows manipulation, appending, sorting, and modification of lists.

Code:

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

# i. Print Complete list
print("Complete List:", a)

# ii. Print 4th element of list
print("4th Element of List:", a[3])

# iii. Print list from 0th to 4th index
print("List from 0th to 4th index:", a[:5])

# iv. Print list -7th to 3rd element
print("List from -7th to 3rd element:", a[-7:4])

# v. Appending an element to list
a.append(110)
print("List after Appending 110:", a)

# vi. Sorting the element of list
a.sort()
print("Sorted List:", a)
```

```python
# vii. Popping an element
popped = a.pop()
print("Popped Element from List:", popped)
print("List after Popping:", a)

# viii. Removing Specified element
a.remove(40)
print("List after Removing 40:", a)

# ix. Entering an element at specified index
a.insert(2, 35)
print("List after Inserting 35 at index 2:", a)

# x. Counting the occurrence of a specified element
count_50 = a.count(50)
print("Count of 50 in List:", count_50)

# xi. Extending list
a.extend([120, 130])
print("Extended List:", a)

# xii. Reversing the list
a.reverse()
print("Reversed List:", a)
```

Output: (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/2:7.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/2:7.py"
Complete List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
4th Element of List: 40
List from 0th to 4th index: [10, 20, 30, 40, 50]
List from -7th to 3rd element: [40]
List after Appending 110: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
Sorted List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
Popped Element from List: 110
List after Popping: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
List after Removing 40: [10, 20, 30, 50, 60, 70, 80, 90, 100]
List after Inserting 35 at index 2: [10, 20, 35, 30, 50, 60, 70, 80, 90, 100]
Count of 50 in List: 1
Extended List: [10, 20, 35, 30, 50, 60, 70, 80, 90, 100, 120, 130]
Reversed List: [130, 120, 100, 90, 80, 70, 60, 50, 30, 35, 20, 10]
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/2:7.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/2:7.py"
Complete List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
4th Element of List: 40
List from 0th to 4th index: [10, 20, 30, 40, 50]
List from -7th to 3rd element: [40]
List after Appending 110: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
Sorted List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
Popped Element from List: 110
List after Popping: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
List after Removing 40: [10, 20, 30, 50, 60, 70, 80, 90, 100]
List after Inserting 35 at index 2: [10, 20, 35, 30, 50, 60, 70, 80, 90, 100]
Count of 50 in List: 1
Extended List: [10, 20, 35, 30, 50, 60, 70, 80, 90, 100, 120, 130]
Reversed List: [130, 120, 100, 90, 80, 70, 60, 50, 30, 35, 20, 10]
apple@Apples-MacBook-Air py lab %
```

Conclusion:

The Python program demonstrates diverse list slicing operations. It showcases the versatility of manipulating lists: accessing elements by index, slicing sublists, modifying, sorting, inserting, counting occurrences, extending, and reversing. This serves as a comprehensive guide for utilizing Python's list functionalities in diverse scenarios.

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:

Lists in Python can be extended using the + operator to concatenate lists, the 'append()' method to add individual elements, or the 'extend()' method to add elements from an iterable.

Code:

```
og_list = [1, 2, 3]
print("Original List:", og_list)
# i. Using + operator to extend a list
extended_with_plus = og_list + [4, 5]
print("Extended with + operator:", extended_with_plus)
# ii. Using append() method to add a single element to the list
og_list.append(4)
og_list.append(5)
print("Extended with append():", og_list)
# iii. Using extend() method to add multiple elements to the list
og_list = [1, 2, 3]
og_list.extend([4, 5])
print("Extended with extend():", og_list)
```

Output: (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:1.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:1.py"
Original List: [1, 2, 3]
Extended with + operator: [1, 2, 3, 4, 5]
Extended with append(): [1, 2, 3, 4, 5]
Extended with extend(): [1, 2, 3, 4, 5]
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:1.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:1.py"
Original List: [1, 2, 3]
Extended with + operator: [1, 2, 3, 4, 5]
Extended with append(): [1, 2, 3, 4, 5]
Extended with extend(): [1, 2, 3, 4, 5]
apple@Apples-MacBook-Air py lab %
```

Conclusion:

Python's list operations offer multiple ways to extend lists. The + operator concatenates, 'append()' adds single elements, and 'extend()' appends iterable elements, providing flexibility in list manipulation

Experiment No:3.2

Title:
Write a program to add two matrices.

Theory:

Matrix addition involves adding corresponding elements of two matrices of the same dimensions. The sum matrix preserves the structure, adding values at each position.

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)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:2.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:2.py"
Matrix after Addition:
[10, 10, 10]
[10, 10, 10]
[10, 10, 10]
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:2.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:2.py"
Matrix after Addition:
[10, 10, 10]
[10, 10, 10]
[10, 10, 10]
apple@Apples-MacBook-Air py lab %
```

Conclusion:

The Python program adeptly adds matrices, ensuring they have identical dimensions. This essential operation finds application in various fields, from mathematics to computer science, facilitating combined data manipulation.

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 function distinct_elements() in Python uses set operations to extract unique elements from a list, ensuring the returned list contains only distinct values.

Code:

```
def get_unique_elements(input_list):
    return list(set(input_list))
# Example list
og_list = [1, 2, 2, 3, 4, 4, 5, 5, 5]
unique_elements = get_unique_elements(og_list)
print("Original List:", og_list)
print("List with Distinct Elements:", unique_elements)
```

Output: (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:3.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:3.py"
Original List: [1, 2, 2, 3, 4, 4, 5, 5, 5]
List with Distinct Elements: [1, 2, 3, 4, 5]
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/3:3.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/
py lab/3:3.py"
Original List: [1, 2, 2, 3, 4, 4, 5, 5, 5]
List with Distinct Elements: [1, 2, 3, 4, 5]
apple@Apples-MacBook-Air py lab %
```

Conclusion:

The Python function distinct_elements() efficiently generates a new list with unique elements, employing set operations for distinct element extraction, enhancing list manipulation capabilities.

Experiment No:3.4

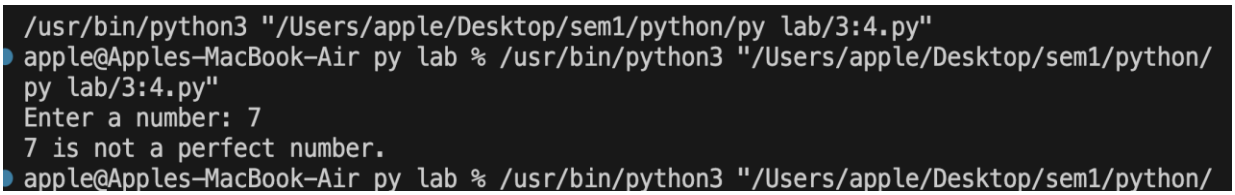
Title:
Write a program to Check whether a number is perfect or not.

Theory:
A perfect number's sum of its divisors (excluding itself) equals the number. This Python code checks divisibility to determine if a given number is perfect.

Code:

```
def per_num(num):
    if num <= 0:
        return False
    sum_of_divisors = sum(i for i in range(1, num) if num % i == 0)
    return sum_of_divisors == num
number = int(input("Enter a number: "))
if per_num(number):
    print(number, "is a perfect number.")
else:
    print(number,"is not a perfect number.")
```

Output: (screenshot)



Test Case: Any two (screenshot)

Conclusion:

The code efficiently identifies perfect numbers by verifying their divisor sum against the number itself, providing accurate results for inputted integers.

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 Python function uses loops to iterate through characters, counting upper and lower-case letters separately using isupper() and islower() functions, returning the counts.

Code:

```
def count_upper_lower(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'
upper, lower = count_upper_lower(string_test)
print("Number of uppercase letters:", upper)
print("Number of lowercase letters:", lower)
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

This Python function accurately tallies upper and lower-case letter counts within a given string, facilitating efficient analysis of text data for case distribution in a concise manner.

Experiment No:4.1

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

Theory:
The Employee Class defines attributes like name, ID, and salary. Methods within the class retrieve and display employee details, facilitating easy information access and printing.

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("Ashlin", "11", "Btech")
print(employee1.details())
```

Output: (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/4:1.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/4:1.py"
Name: Ashlin
Employee ID: 11
Department: Btech
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
/usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/4:1.py"
apple@Apples-MacBook-Air py lab % /usr/bin/python3 "/Users/apple/Desktop/sem1/python/py lab/4:1.py"
Name: Ashlin
Employee ID: 11
Department: Btech
apple@Apples-MacBook-Air py lab %
```

Conclusion:

This Employee Class program offers a structured way to manage employee information. It allows simple retrieval and display of employee details, enhancing accessibility and organization in employee data handling.

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 andkwargs) using function,**

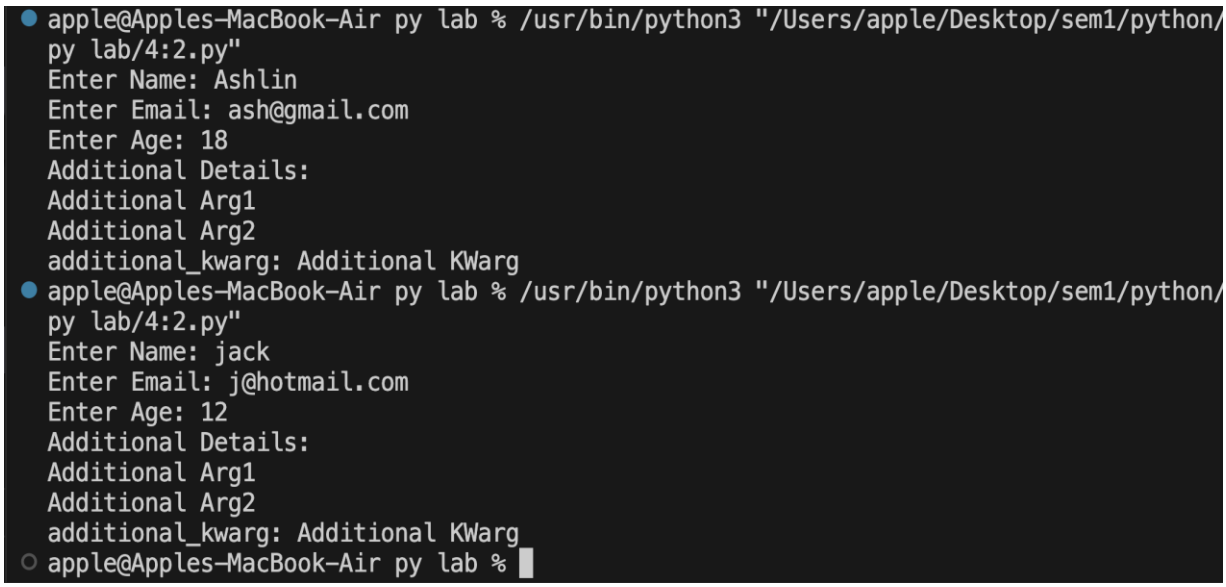
Theory:
*args allows variable positional arguments, **kwargs handles variable keyword arguments. Combining both captures diverse inputs effectively, enhancing flexibility in function parameters for user input**.**

Code:

```
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)

Test Case: Any two (screenshot)



Conclusion:

The program adeptly captures user details (name, email, age) using a blend of *args and **kwargs, showcasing their versatility in handling diverse input scenarios within a function.

Experiment No:4.3

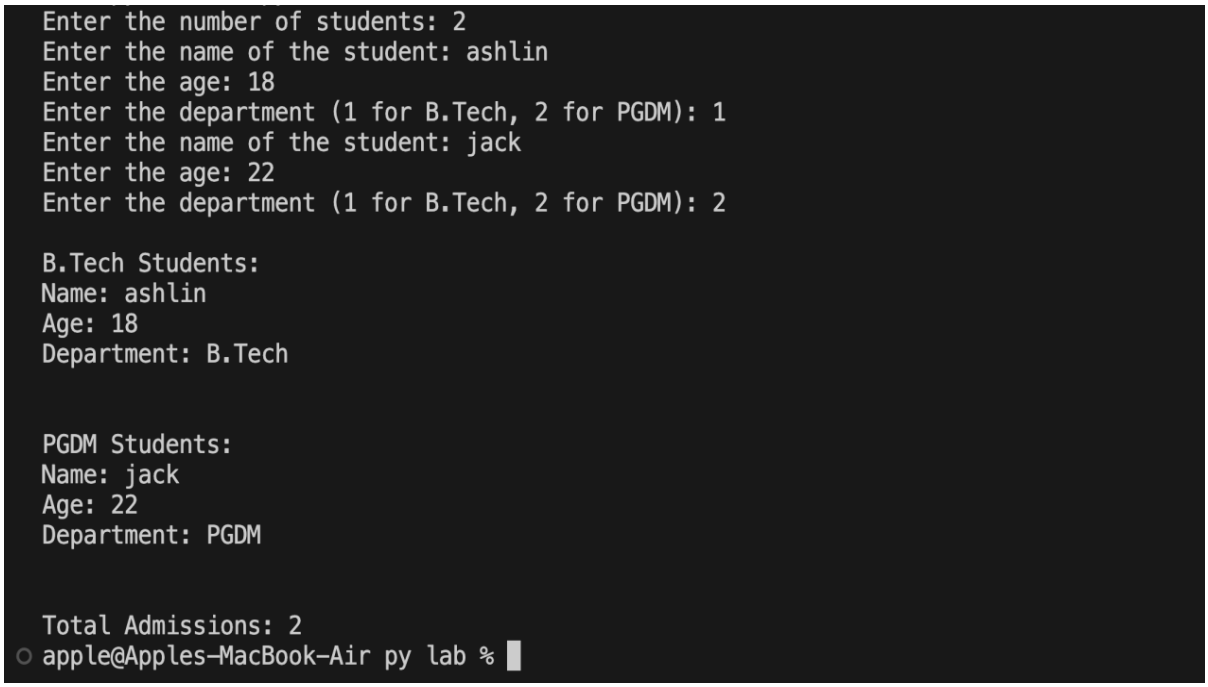
Title:
Write a program to admit the students in the different Departments(pgdm/btech)and count the students. (Class, Object and Constructor).

Theory:
In Python, classes represent objects. A constructor initializes class attributes. Creating a class for student admission allows counting students in different departments (PGDM/BTech) accurately.

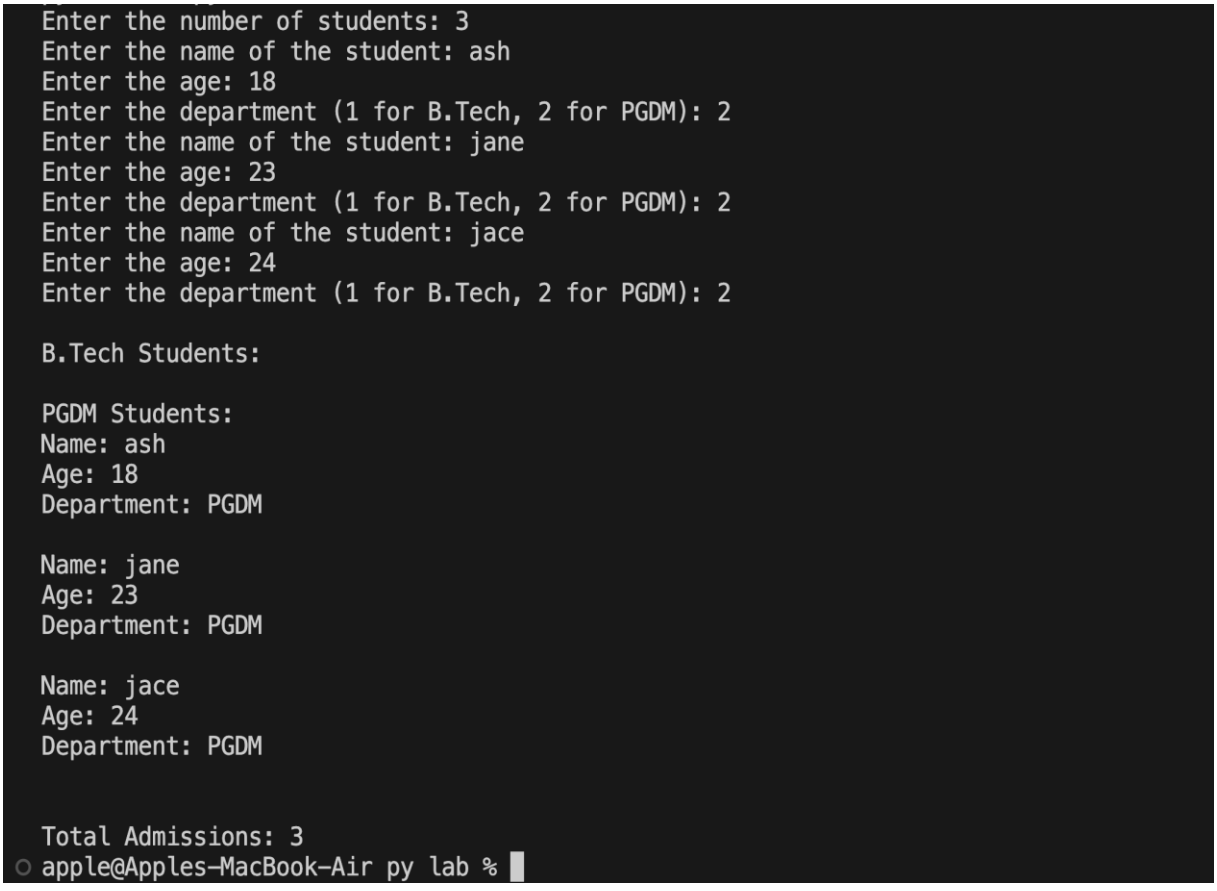
Code:

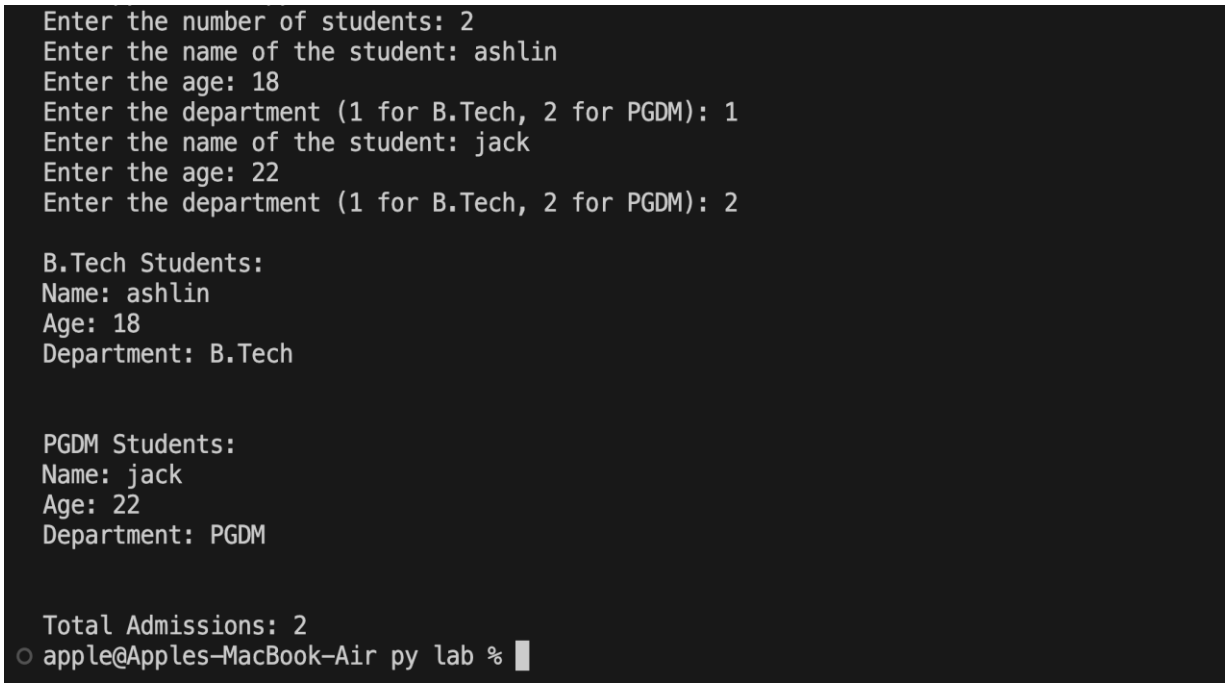
```
class ITM:
    count = 0
    def get(self):
        self.name = input("Enter the name of the student: ")
        self.age = int(input("Enter the age: "))
        self.dep = int(input("Enter the department (1 for B.Tech, 2 for PGDM): "))
        ITM.count += 1

    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)



Test Case: Any two (screenshot)





Conclusion:

Utilizing Python's class, object, and constructor features, the program efficiently manages student admission by tracking counts for various departments, facilitating streamlined enrollment management.

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 program creates a 'Store' class managing product records. It displays a menu, prompts for quantities, generates a bill, and shows the total amount based on input.

Code:

```
class Store:
    def __init__(self):
        self.products = {"Product1": 100, "Product2": 50, "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)

```
py lab/4:4.py"
Product Menu:
Product1: Rs.100
Product2: Rs.50
Product3: Rs.30
2 Product1(s) added to your bill.
1 Product3(s) added to your bill.
Your Bill:
Product1 x 2: Rs.200
Product3 x 1: Rs.30
Total Amount: Rs.230
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
py lab/4:4.py"
Product Menu:
Product1: Rs.100
Product2: Rs.50
Product3: Rs.30
2 Product1(s) added to your bill.
1 Product3(s) added to your bill.
Your Bill:
Product1 x 2: Rs.200
Product3 x 1: Rs.30
Total Amount: Rs.230
apple@Apples-MacBook-Air py lab %
```

Conclusion:

The program efficiently manages product data, calculates bills, and displays the total amount. However, it lacks error handling and advanced functionalities for a more comprehensive store management system.

Experiment No:4.5

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

Theory:
Single inheritance in Python involves deriving a new class from a single base class. It allows the new class to inherit attributes and methods from the base class.

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)

```
py lab/4:5.py"
Enter the first number: 5
Enter the second number: 4
The sum of 5 and 4 is: 9
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
Enter the first number: 7
Enter the second number: 8
The sum of 7 and 8 is: 15
apple@Apples-MacBook-Air py lab %
```

```
py lab/4:5.py"
Enter the first number: 5
Enter the second number: 4
The sum of 5 and 4 is: 9
apple@Apples-MacBook-Air py lab %
```

Conclusion:

This Python program, using single inheritance, prompts user input for two numbers, performing their addition. Demonstrates inheritance for code reusability.

Experiment No:4.6

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

Theory:
Multiple inheritance in Python allows a class to inherit attributes and methods from multiple base classes. The derived class can access functionalities from both parent classes.

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)

Test Case: Any two (screenshot)

Conclusion:

Utilizing multiple inheritance, the program creates two base classes, LU and ITM, and a derived class that inherits features from both, demonstrating the versatility of Python's inheritance mechanisms.

Experiment No:4.7

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

Theory:
Multilevel inheritance involves a hierarchy where a child class inherits properties from its immediate parent and grandparent classes, showcasing a hierarchical relationship in object-oriented programming.

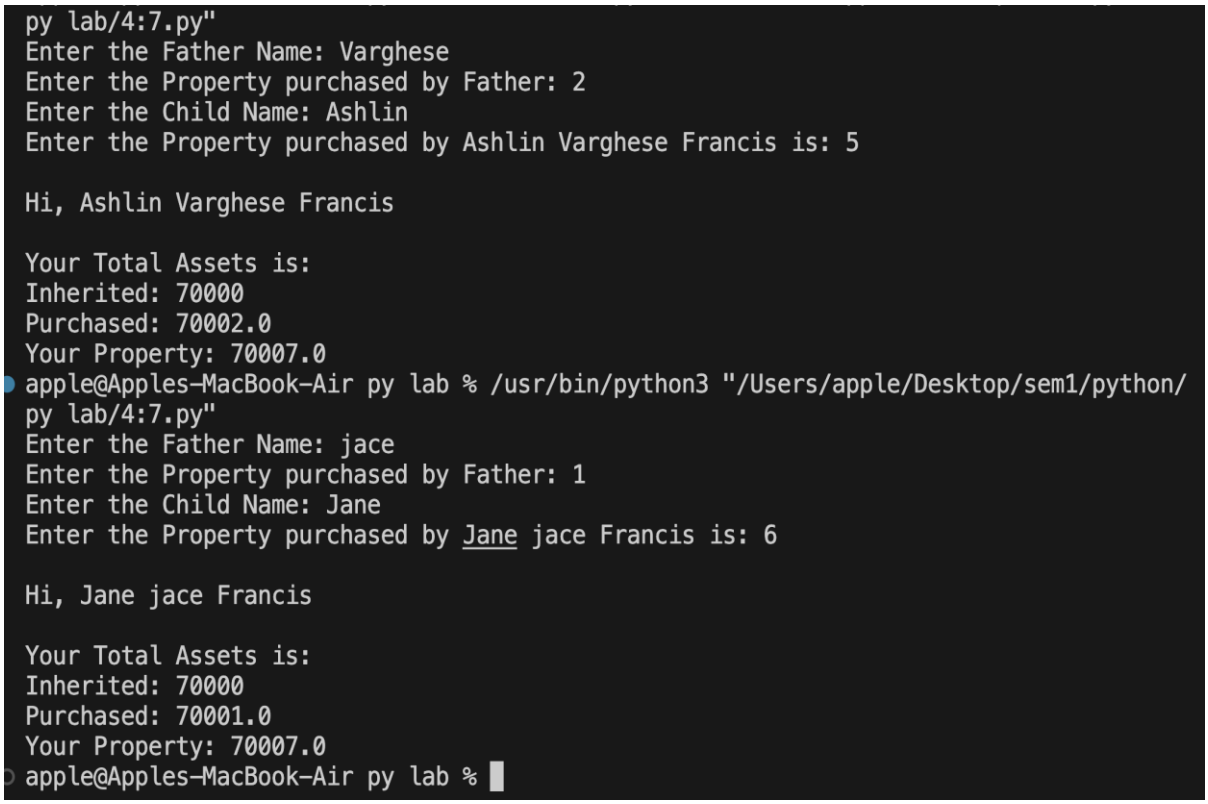
Code:

```
class Grandfather:
    def __init__(self):
        self.gfname = " Francis"
        self.inheritance = 70000

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)

```
py lab/4:7.py"
Enter the Father Name: Varghese
Enter the Property purchased by Father: 2
Enter the Child Name: Ashlin
Enter the Property purchased by Ashlin Varghese Francis is: 5

Hi, Ashlin Varghese Francis

Your Total Assets is:
Inherited: 70000
Purchased: 70002.0
Your Property: 70007.0
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)



Conclusion:

The implemented program demonstrates property inheritance across multiple levels, showcasing how attributes and behaviors pass from the grandfather class through the father to the child class, exemplifying multilevel inheritance in Python.

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:

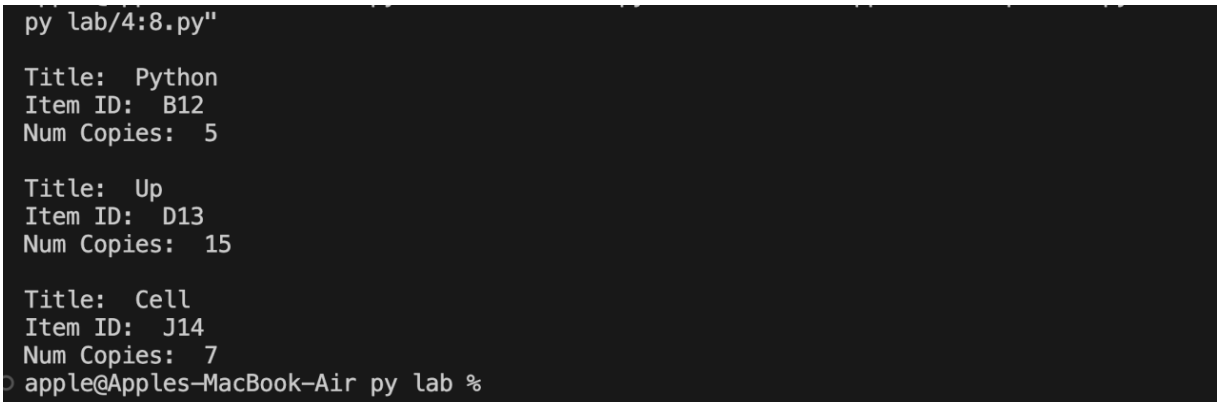
Inheritance in Python facilitates a Library Catalogue System. Base class "LibraryItem" extends to derived classes (Book, DVD, Journal), each with distinct attributes and methods, supporting check-in and check-out via method overriding.

Code:

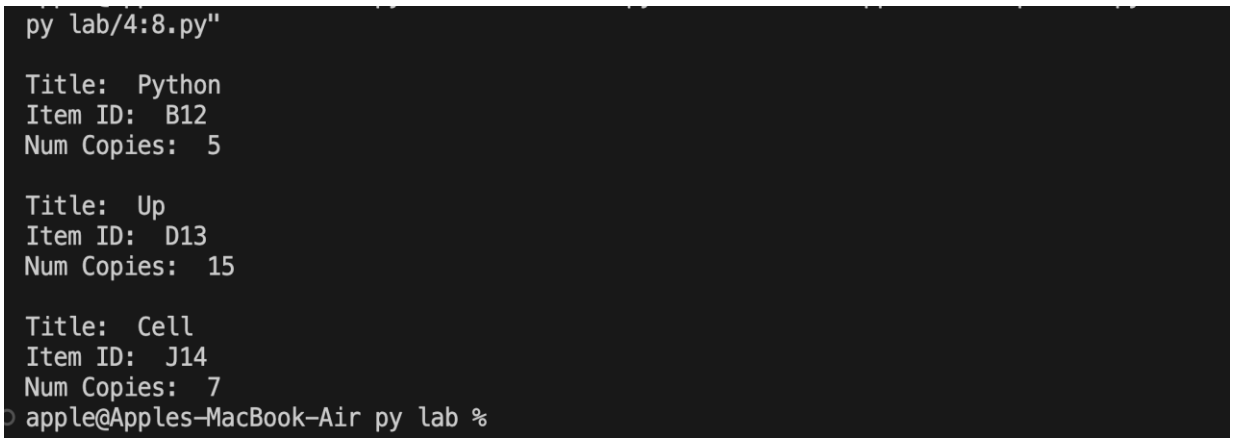
```
class LibraryItem:
    def __init__(self, title, item_id, no_copies):
        self.title = title
        self.item_id = item_id
        self.no_copies = no_copies
    def display(self):
        print("\nTitle: ",self.title)
        print("Item ID: ",self.item_id)
        print("Num Copies: ",self.no_copies)
    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("Python", "B12", 5)
dvd = DVD("Up", "D13", 15)
journal = Journal("Cell", "J14",7)

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

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The implemented Library Catalogue System in Python showcases the power of inheritance, allowing unique attributes and methods for various items while enabling efficient check-in and check-out functionalities.

Experiment No:5.1

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

Theory:
Modules encapsulate reusable code. Create my_module.py with a function adding two numbers. In the main script, import my_module to access its functions.

Code:

In my_module.py file:

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

In main file:

```
import my_module

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

result = my_module.add_numbers(num1, num2)
print("Sum of the numbers:", result)
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

Modular programming enhances code reusability. The program separates addition logic into a module, promoting cleaner code organization, easier maintenance, and efficient collaboration in larger projects.

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 Bank Module manages bank account operations: checking balance, withdrawing, and depositing money. It encapsulates functions for financial transactions, enhancing modularity and reusability.

Code:

In bank_module.py file:

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

    def check_balance(self):
        return f"Your current balance is: {self.balance}"

    def deposit(self, amount):
        self.balance += amount
        return f"Deposited {amount}. Your new balance is: {self.balance}"

    def withdraw(self, amount):
        if amount > self.balance:
            return "Insufficient funds"
        self.balance -= amount
        return f"Withdrawn {amount}. Your new balance is: {self.balance}"
```

In main file:

```
from bank_module import BankAccount

account = BankAccount(10000)

print(account.check_balance())

print(account.deposit(5000))
print(account.withdraw(2000))

print(account.check_balance())
```

Output: (screenshot)

```
py lab/5:2.py"
Your current balance is: 10000
Deposited 5000. Your new balance is: 15000
Withdrawn 2000. Your new balance is: 13000
Your current balance is: 13000
apple@Apples-MacBook-Air py lab %
```

Test Case: Any two (screenshot)

```
py lab/5:2.py"
Your current balance is: 10000
Deposited 5000. Your new balance is: 15000
Withdrawn 2000. Your new balance is: 13000
Your current balance is: 13000
apple@Apples-MacBook-Air py lab %
```

Conclusion:

The Bank Module efficiently handles account transactions, promoting modular coding practices for reusability. Importing this module streamlines banking operations in the main file, fostering code clarity and maintenance.

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:
A package 'cars' is created in Python to encapsulate modules like BMW, Audi, Nissan, each containing classes and functionalities. These modules are imported and accessed within the main 'cars' file.

Code:

In car packet: BMW module:

```
class BMW:
    def __init__(self, model, year_start,year_stop):
        self.model = model
        self.year_start = year_start
        self.year_stop = year_stop

    def start(self):
        return f"Starting the {self.model} {self.year_start }"

    def stop(self):
        return f"Stopping the {self.model} {self.year_stop}"
```

AUDI module:

```
class AUDI:
    def __init__(self, model, year_start,year_stop):
        self.model = model
        self.year_start = year_start
        self.year_stop = year_stop

    def start(self):
        return f"Starting the {self.model} {self.year_start }"

    def stop(self):
        return f"Stopping the {self.model} {self.year_stop}"
```

NISSAN module:

```
class NISSAN:
    def __init__(self, model, year_start,year_stop):
        self.model = model
        self.year_start = year_start
        self.year_stop = year_stop

    def start(self):
        return f"Starting the {self.model} {self.year_start }"

    def stop(self):
        return f"Stopping the {self.model} {self.year_stop}"
```

In main file:

```
from Cars import bmw, audi, nissan

user = input("Enter a BMW model:")
start = int(input("Enter the Starting year:"))
end = int(input("Enter the Ending year:"))

bmw_car = bmw.BMW(user, start, end)
print(bmw_car.start())
print(bmw_car.stop())

user = input("Enter an AUDI model:")
start = int(input("Enter the Starting year:"))
end = int(input("Enter the Ending year:"))

audi_car = audi.AUDI(user, start, end)
print(audi_car.start())
print(audi_car.stop())

user = input("Enter a NISSAN model:")
start = int(input("Enter the Starting year:"))
end = int(input("Enter the Ending year:"))

nissan_car = nissan.NISSAN(user, start, end)
print(nissan_car.start())
print(nissan_car.stop())
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

The 'cars' package organises car-related modules effectively, allowing modular design and easy access to functionalities. It enhances code readability and maintainability by compartmentalising car-specific operations.

Experiment No:6.1

Title:

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

Theory:

Multithreading in Python allows concurrent execution. The program utilizes two threads, each printing either "Hello" or "Hi" independently, demonstrating simultaneous thread execution.

Code:

```
import threading

def print_hello():
    for _ in range(5):
        print("Hello")

def print_hi():
    for _ in range(5):
        print("Hi")

thread_hello = threading.Thread(target=print_hello)
thread_hi = threading.Thread(target=print_hi)

thread_hello.start()
thread_hi.start()

thread_hello.join()
thread_hi.join()

print("Execution completed.")
```

Output: (screenshot)

Test Case: Any two (screenshot)

Conclusion:

Multithreading facilitates concurrent operations, showcasing simultaneous "Hello" and "Hi" printing. This enhances performance by executing tasks concurrently, a valuable feature in various applications for efficient utilization of resources.

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.

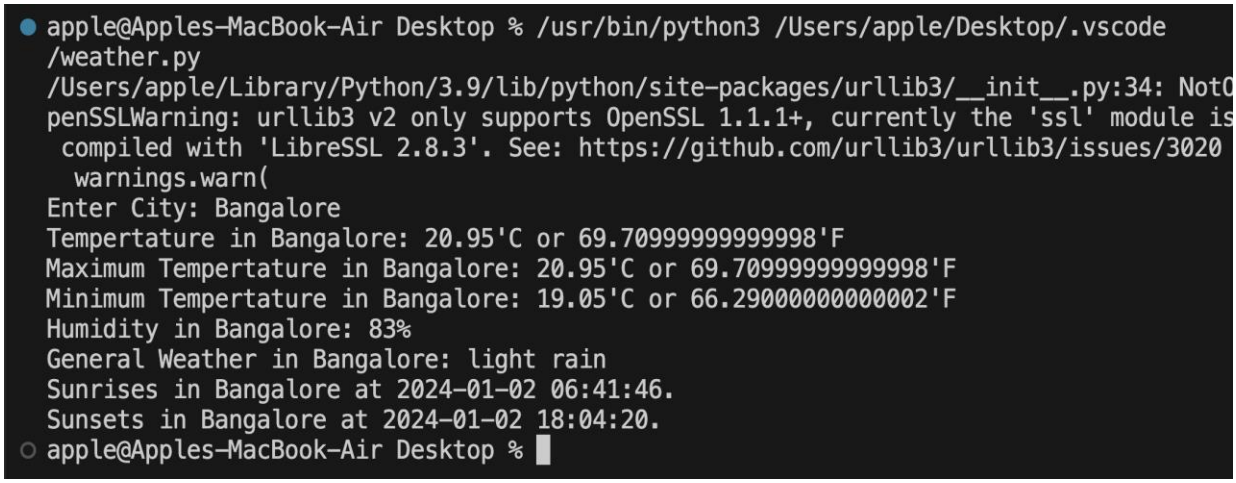
Theory:

Python accesses a weather API to fetch weather data for a specified city, extracting temperature and sunrise/sunset times. This utilizes API requests to display relevant weather information.

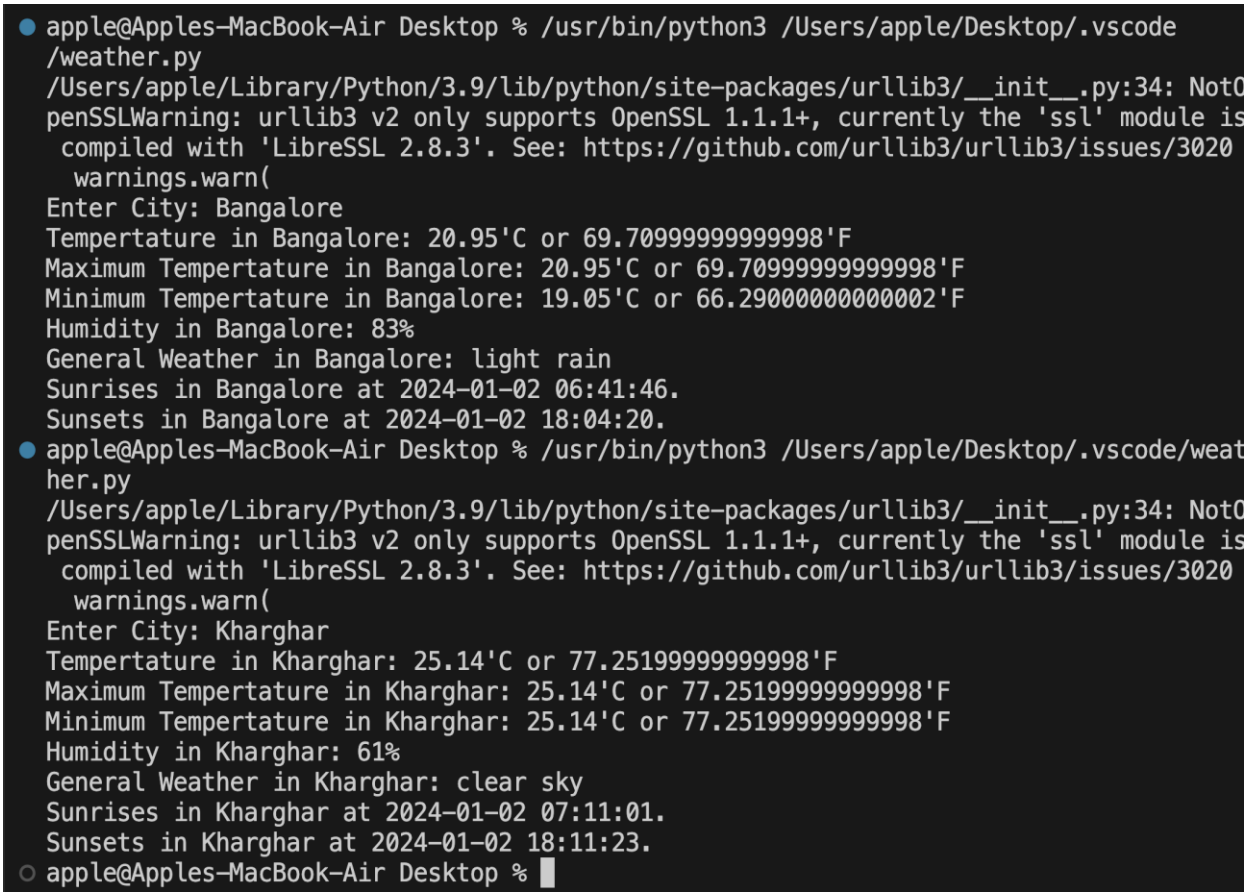
Code:

```
import datetime as dt
import requests

baseurl="http://api.openweathermap.org/data/2.5/weather?"
api_key="62d7a50e6f4b25ed70de07839c010508"
city=input('Enter City: ')

def kel_to_cel_fahren(kelvin):
    celsius=kelvin-273
    fahrenheit=celsius*(9/5)+32
    return celsius,fahrenheit
#  to form the complete API request URL.
url= baseurl + 'appid=' + api_key + '&q=' + city
response=requests.get(url).json()

temp_kelvin=response['main']['temp']
temp_celsius, temp_fahrenheit=kel_to_cel_fahren(temp_kelvin)
max_temp=response['main']['temp_max']
tempc,tempf=kel_to_cel_fahren(max_temp)
min_temp=response['main']['temp_min']
temc,temf=kel_to_cel_fahren(min_temp)
humidity=response['main']['humidity']
description=response['weather'][0]['description']
sunrise=dt.datetime.utcfromtimestamp(response['sys']['sunrise']+response['timezone'])
sunset=dt.datetime.utcfromtimestamp(response['sys']['sunset']+response['timezone'])

print(f"Tempertature in {city}: {temp_celsius:.2f}'C or {temp_fahrenheit}'F")
print(f"Maximum Tempertature in {city}: {tempc:.2f}'C or {tempf}'F")
print(f"Minimum Tempertature in {city}: {temc:.2f}'C or {temf}'F")
print(f"Humidity in {city}: {humidity}%")
print(f"General Weather in {city}: {description}")
print(f"Sunrises in {city} at {sunrise}.")
print(f"Sunsets in {city} at {sunset}.")
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

The program fetches real-time weather data through an API, displaying temperature and sunrise/sunset times. It showcases the ease of integrating external APIs for obtaining specific weather details.

Experiment No:7.2

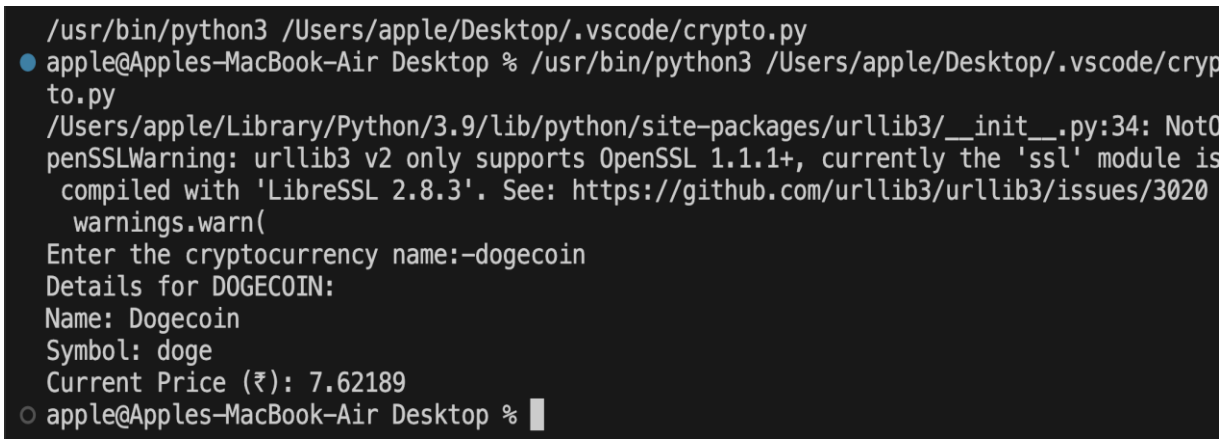
Title:
Write a program to use the 'API 'of crypto currency.

Theory:
Python utilizes cryptocurrency APIs to access real-time data for analysis, fetching information like prices enabling integration of crypto-related data into applications or analysis tools.

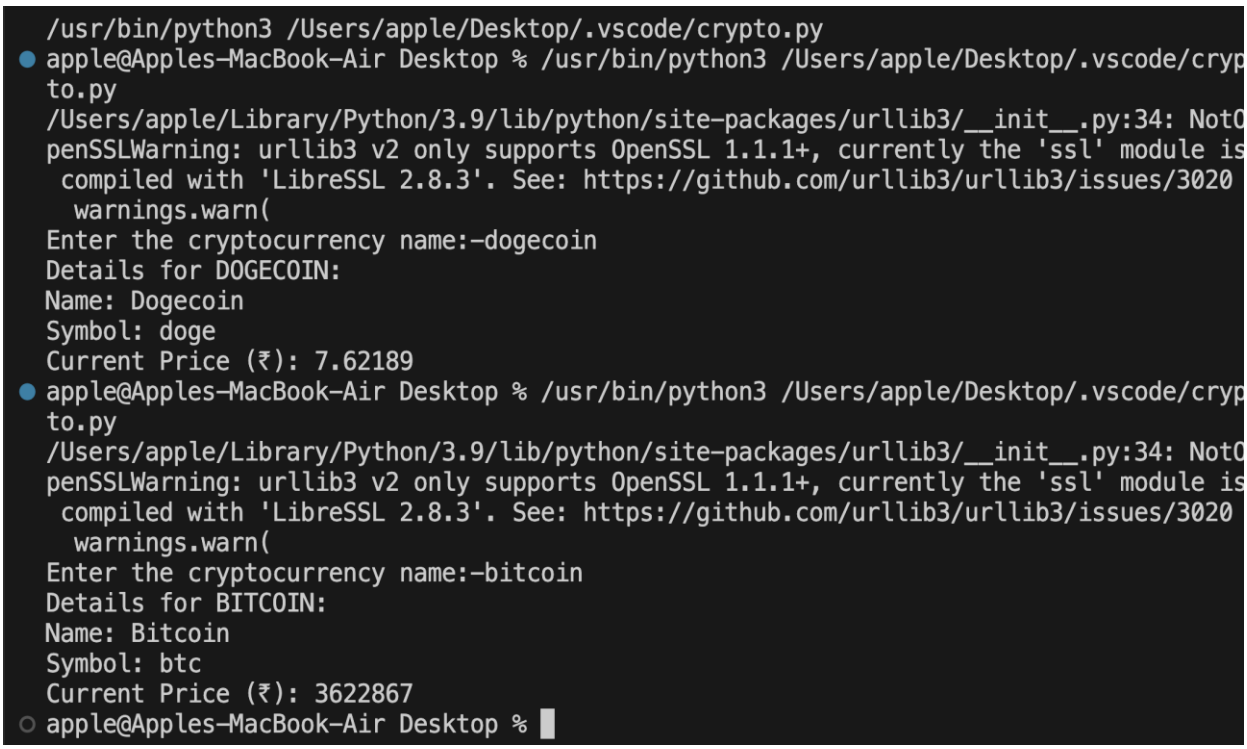
Code:

```
import requests
api_key="CG-hw5JqqeSPWRHKa2AdtbLLU3b"
base_url="https://api.coingecko.com/api/v3/coins/"

cryptocurrency = input("Enter the cryptocurrency name:-").lower()
url = f"{base_url}{cryptocurrency}"
headers = {
  "Content-Type": "application/json",
  "Authorization": f"Bearer {api_key}"
}
response = requests.get(url, headers=headers)

if response.status_code == 200:
  data = response.json()
  print(f"Details for {cryptocurrency.upper()}:")
  print("Name:", data['name'])
  print("Symbol:", data['symbol'])
  print("Current Price (₹):", data['market_data']['current_price']['usd']*83)
else:
  print(f"Failed to fetch data for {cryptocurrency}. Status code:", response.status_code)
```

Output: (screenshot)



Test Case: Any two (screenshot)



Conclusion:

Utilizing cryptocurrency APIs empowers developers to access live data for informed decision-making, fostering the integration of real-time crypto information into applications, trading tools, and analysis platforms.