INDEX

Sl. No	Date	Name of Experiment	Page No.	Marks	Signature of the faculty

INDEX

Sl. No	Date	Name of Experiment	Page No.	Marks	Signature of the faculty

EX:NO: 1.1	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
IMPLEMENTATION OF A PYTHON PROGRAM THAT CHECKS IF THE GIVEN INTEGER IS POSITIVE OR NEGATIVE OR ZERO.	

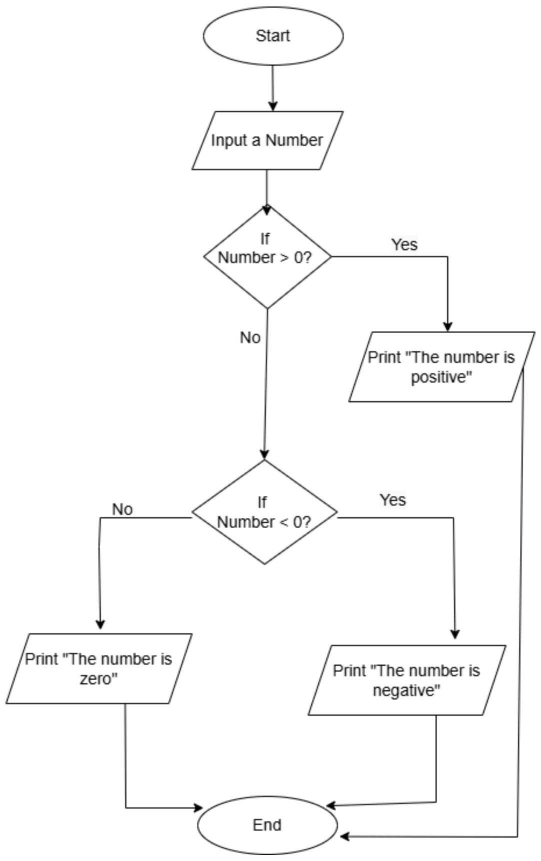
AIM:

To write a Python program prints whether the given integer is positive, negative, or zero.

PSEUDO CODE:

```
START
   PROMPT user to enter a number and store it in num
   IF num is greater than 0:
      PRINT "The number is positive."
   ELSE IF num is less than 0:
      PRINT "The number is negative."
   ELSE:
      PRINT "The number is zero."
END
```

FLOW CHART:


Start
Input a Number
If Number > 0?
Yes
No
Print "The number is positive"
If Number < 0?
No
Yes
Print "The number is zero"
Print "The number is negative"
End

SOURCE CODE:

```
num = int(input("Enter a number: "))
if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
  print("The number is zero.")
```

OUTPUT:

```
Enter a number: 10
The number is positive.
```

RESULT:

Thus the python program to print whether the given integer is positive, negative, or zero has been successfully executed and the output was verified.

EX:NO: 1.2	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
WRITE A PROGRAM TO CHECK WHETHER THE YEAR IS LEAP YEAR OR NOT	

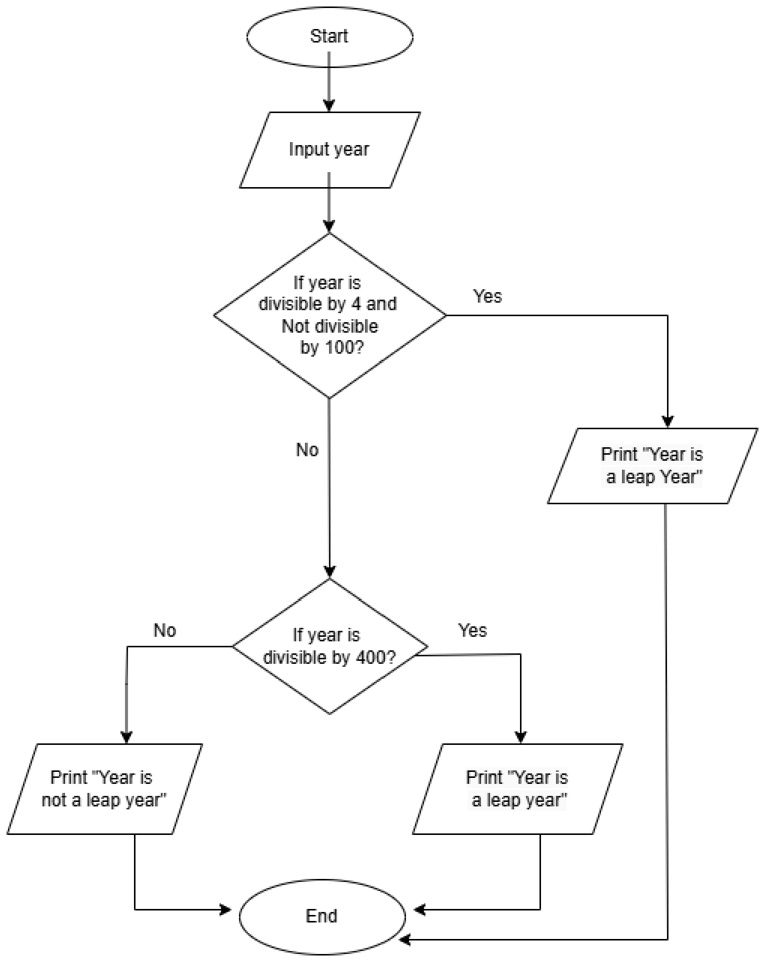
AIM:

To Implement a python program that checks whether a given year is a leap year or not.

PSEUDO CODE:

```
START
   PROMPT user to enter a year and store it in year
   IF (year is divisible by 4 AND year is NOT divisible by 100) OR (year is divisible by
400):
      PRINT "year is a leap year."
   ELSE:
      PRINT "year is not a leap year."
END
```

FLOW CHART:


Start
Input year
If year is divisible by 4 and Not divisible by 100?
Yes
No
Print "Year is a leap Year"
If year is divisible by 400?
No
Yes
Print "Year is not a leap year"
Print "Year is a leap year"
End

SOURCE CODE:

```
year = int(input("Enter a year: "))
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

OUTPUT:

```
Enter a year: 2024
2024 is a leap year.
```

RESULT:

Thus the python program that checks whether a given year is a leap year or not has been successfully executed and the output was verified.

EX:NO: 1.3	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
WRITE A PYTHON PROGRAM TO FIND THE SQUARE ROOT OF A NUMBER	

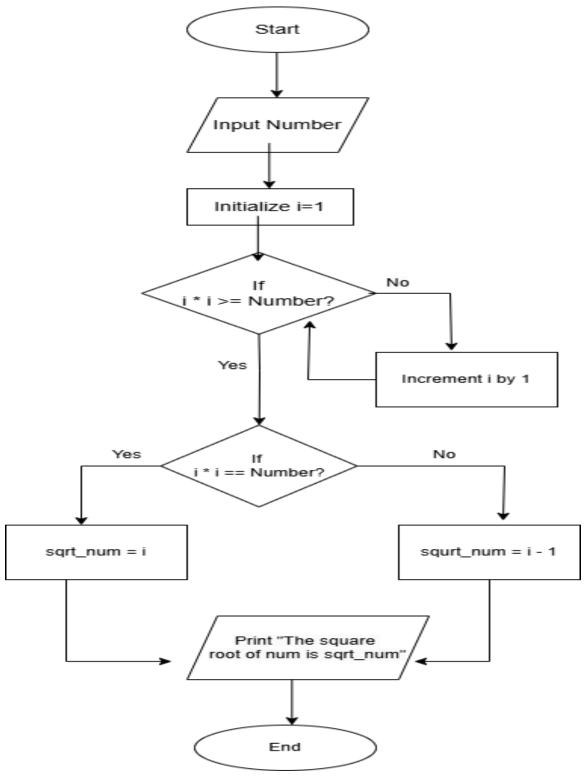
AIM:

To write a Python program to print the square root of the given number.

PSEUDO CODE:

```
START
    PROMPT user to input a number and store it in variable num
    SET i to 1
    WHILE True:
        IF i *i  greater than or equals to  num:
            SET sqrt_num to i
            BREAK the loop
        INCREMENT i by 1
    IF i*i == num:
sqr_num = i
   ELSE:
           sqr_num = i-1
   PRINT "The square root of", num, "is", sqrt_num

END
```

FLOW CHART:



SOURCE CODE:

```
num = int(input("Enter the number: "))
i = 1
while True:
    if i*i >= num:
        break
    i += 1
if i*i == num:
    sqrt_num=i
else:
    sqrt_num=i-1
print(f"The square root of {num} is {sqrt_num}")
```

OUTPUT:

```
Enter the number: 16
The square root of 16 is 4
```

RESULT:

Thus the python program to print the square root of the given number has been successfully executed and the output was verified.

EX:NO: 1.4	WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON
DATE:	
IMPLEMENTATION OF A PYTHON PROGRAM TO PRINT THE GCD OF THE GIVEN NUMBERS.	

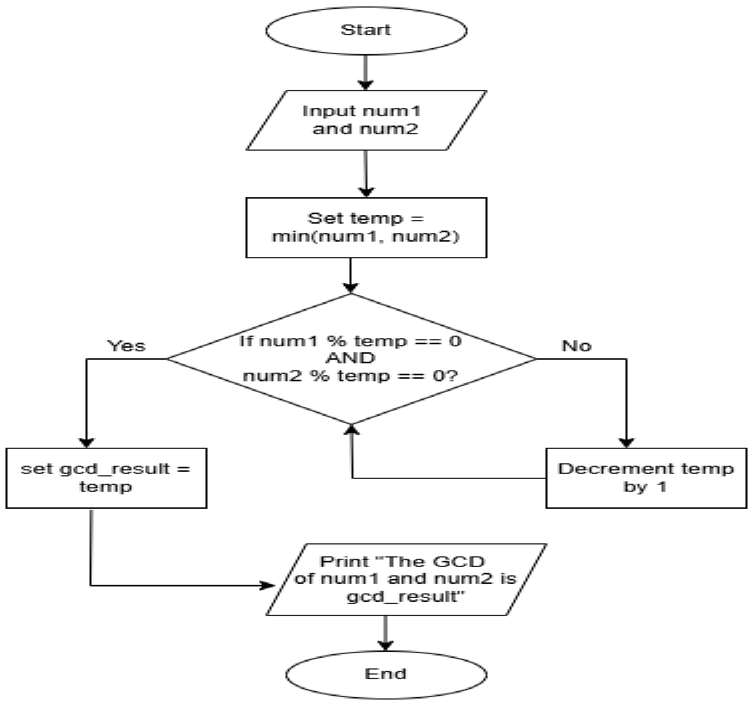
AIM:

To write a Python program to print the GCD of the given numbers.

PSEUDO CODE:

```
START

  PROMPT user to enter num1 and store it in num1
  PROMPT user to enter num2 and store it in num2

  SET temp to the smaller of num1 and num2

  WHILE True:
    IF num1 is divisible by temp AND num2 is divisible by temp:
      SET gcd_result to temp
      BREAK the loop

    DECREMENT temp by 1

  PRINT "The GCD of", num1, "and", num2, "is", gcd_result

END
```

FLOW CHART:


Start
Input num1 and num2
Set temp = min(num1, num2)
If num1 % temp == 0 AND num2 % temp == 0?
Yes
No
set gcd_result = temp
Decrement temp by 1
Print "The GCD of num1 and num2 is gcd_result"
End

SOURCE CODE:

```
num1 = int(input("Enter num1: "))
num2 = int(input("Enter num2: "))
temp = min(num1, num2)
while True:
    if num1 % temp == 0 and num2 % temp == 0:
        gcd_result = temp
        break
    temp -= 1

print(f"The GCD of {num1} and {num2} is {gcd_result}")
```

OUTPUT:

```
Enter num1: 15
Enter num2: 20
The GCD of 15 and 20 is 5
```

RESULT:

Thus the python program to print the GCD of the given number has been successfully executed and the output was verified.

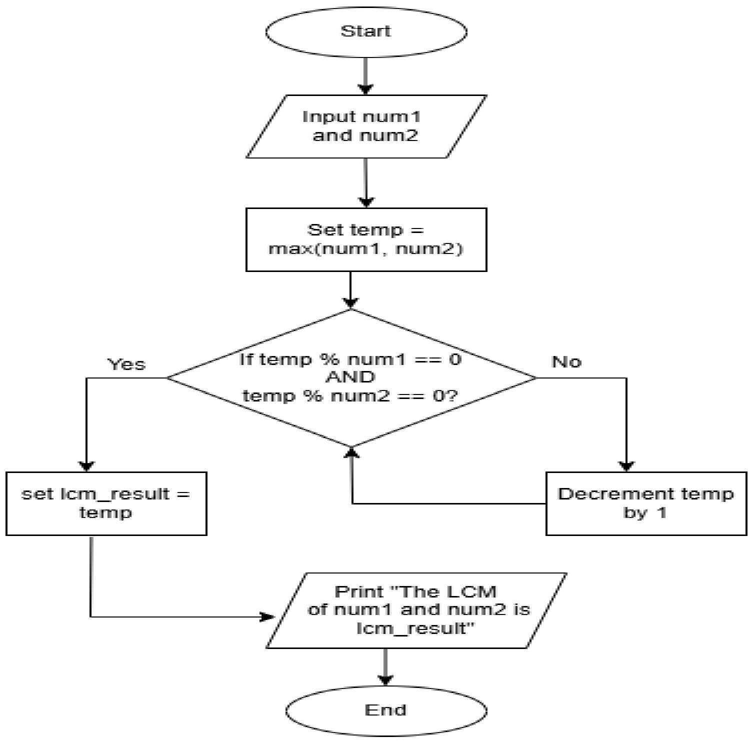
EX:NO: 1.5	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO PRINT THE LCM OF THE GIVEN NUMBERS.	

AIM:
To write a Python program to print the LCM of the given numbers

PSEUDO CODE:

```
START

  PROMPT user to enter num1 and store it in num1
  PROMPT user to enter num2 and store it in num2
  SET temp to the larger of num1 and num2
  WHILE True:
    IF num1 is divisible by temp AND num2 is divisible by temp:
      SET lcm_result to temp
      BREAK the loop

    INCREMENT temp by 1

  PRINT "The LCM of", num1, "and", num2, "is", lcm_result

END
```

FLOW CHART:



Start
Input num1 and num2
Set temp = max(num1, num2)
If temp % num1 == 0 AND temp % num2 == 0?
Yes
No
set lcm_result = temp
Decrement temp by 1
Print "The LCM of num1 and num2 is lcm_result"
End

SOURCE CODE:

```
num1 = int(input("Enter num1: "))
num2 = int(input("Enter num2: "))
temp = max(num1, num2)

while True:
    if temp % num1 == 0 and temp % num2 == 0:
        lcm_result = temp
        break
    temp += 1

print(f"The LCM of {num1} and {num2} is {lcm_result}")
```

OUTPUT:

```
Enter num1: 15
Enter num2: 20
The LCM of 15 and 20 is 60
```

RESULT:

Thus the python program to print the LCM of the given number has been successfully executed and the output was verified.

EX:NO: 1.6	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
IMPLEMENTATION OF A PYTHON PROGRAM TO FIND THE FACTORIAL OF A NUMBER USING A WHILE LOOP	

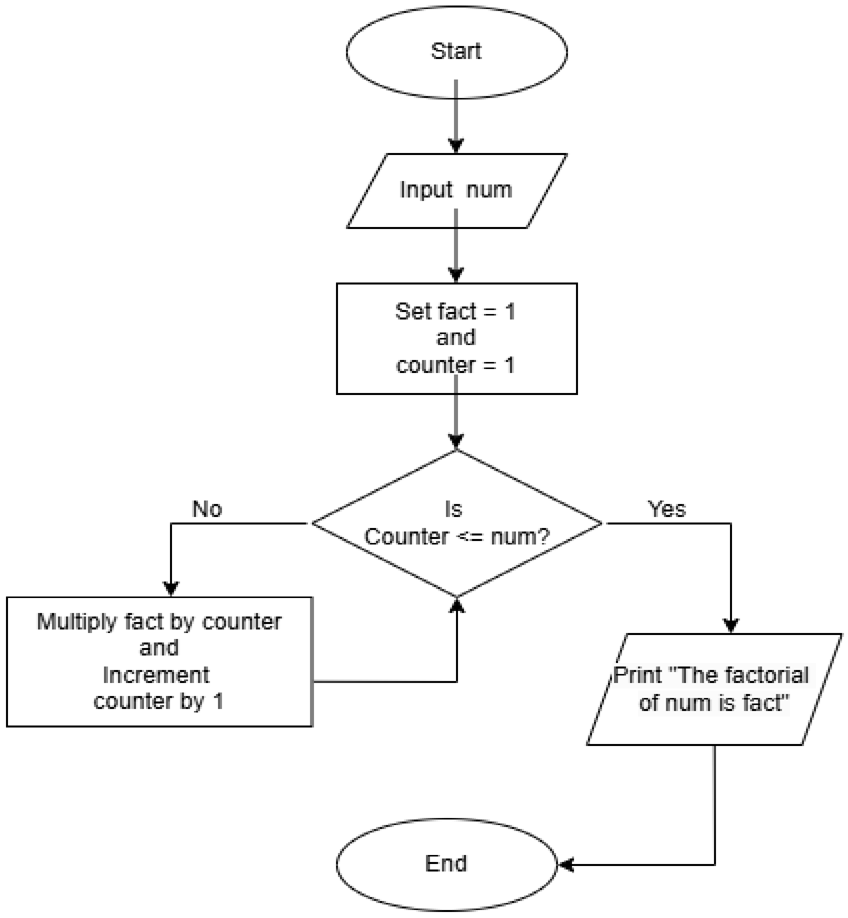
AIM:

To write a Python program to find the factorial of a number using a while loop.

PSEUDO CODE:

```
START
   PROMPT user to enter a number and store it in num
   SET factorial to 1
   SET counter to 1
   WHILE counter is less than or equal to num:
      MULTIPLY factorial by counter
      INCREMENT counter by 1
   PRINT "The factorial of num is factorial."
END
```

FLOW CHART:



SOURCE CODE:

```
num = int(input("Enter a number: "))
factorial = 1
counter = 1
while counter <= num:
    factorial *= counter
    counter += 1
print(f"The factorial of {num} is {factorial}.")
```

OUTPUT:

```
Enter a number: 5
The factorial of 5 is 120.
```

RESULT:

Thus the python program to find the factorial of a number using a while loop has been successfully executed and the output was verified.

EX:NO: 1.7	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO PRINT N FIBONACCI SERIES	

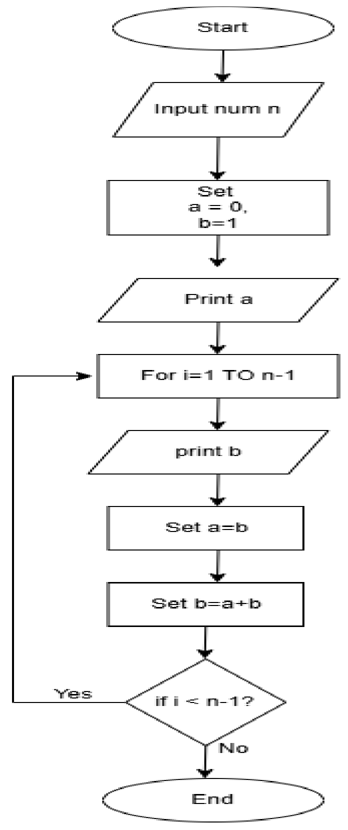
AIM:

To write a python program to print n Fibonacci series

PSEUDO CODE:

```
START
  START
    INPUT number 'n'
    SET a = 0
    SET b = 1
    PRINT a
    FOR i = 1 TO n-1:
        PRINT b
        SET a = b
        SET b = a + b
    END FOR
END
```

FLOW CHART:



SOURCE CODE:

```
n = int(input("Enter the value of N: "))
a, b = 0, 1
print(a,end=’ ‘)
for i in range(1,n):
    print(b, end=" ")
  a = b
  b=a+b
```

OUTPUT:

```
Enter the value of N: 4
0 1 1 2
```

RESULT:

Thus the python program to print n Fibonacci series has been successfully executed and the output was verified.

EX:NO: 1.8	**WRITE CONDITIONAL AND LOOPING STATEMENTS IN PYTHON**
DATE:	

IMPLEMENTATION OF A PYTHON PROGRAM TO PRINT A PATTERN OF STARS (*) IN A RIGHT-ANGLED TRIANGLE SHAPE WITH 5 ROWS USING A LOOP.

AIM:

To write a Python program to print a pattern of stars (*) in a right-angled triangle shape with 5 rows using a loop.

PSEUDO CODE:

```
START
   FOR i from 1 to 5 (inclusive):
      PRINT i number of "*" characters in a row
END
```

SOURCE CODE:

```
for i in range(1, 6):
    print("*" * i)
```

OUTPUT:

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

RESULT:

Thus the python program to print a pattern of stars (*) in a right-angled triangle shape with 5 rows using a loop has been successfully executed and the output was verified.

EX:NO: 2.1	CREATE AND MANIPULATE STRINGS USING INDEXING , SLICING AND VARIOUS STRING FUNCTIONS
DATE:	

CREATE AND MANIPULATE STRINGS USING INDEXING, SLICING, AND VARIOUS STRING FUNCTIONS

AIM:

To create and manipulate strings using indexing, slicing, and various string functions

PSEUDO CODE:

```
START

1. Initialize a string variable with the value "Hello, World!".
   Set my_string = "Hello, World!"
2. Indexing:
   - Access the first character of the string (index 0).
     PRINT the first character.
   - Access the last character of the string (index -1).
     PRINT the last character.
3. Slicing:
   - Extract the substring from index 7 to index 11 (inclusive).
     PRINT the substring.
4. Use string functions:
   - Calculate the length of the string using the length function.
     PRINT the length of the string.
   - Convert the string to uppercase and PRINT it.
   - Convert the string to lowercase and PRINT it.
END
```

SOURCE CODE:

```
my_string = "Hello, World!"
print("First character:", my_string[0])
print("Last character:", my_string[-1])
substring = my_string[7:12]
print("Substring (from index 7 to 11):", substring)
print("Length of the string:", len(my_string))
print("Uppercase string:", my_string.upper())
print("Lowercase string:", my_string.lower())
```

OUTPUT:

```
First character: H
Last character: !
Substring (from index 7 to 11): World
Length of the string: 13
Uppercase string: HELLO, WORLD!
Lowercase string: hello, world!
```

RESULT:

Thus the python program **to create and manipulate strings using indexing, slicing, and various string functions** has been successfully executed and the output was verified.

EX:NO: 2.2	CREATE AND MANIPULATE STRINGS USING INDEXING , SLICING AND VARIOUS STRING FUNCTIONS
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO CHECK IF ONE STRING CONTAINS ANOTHER STRING.	

AIM:

To write a Python program to check if one string contains another string.

PSEUDO CODE:

```
START
   PROMPT the user to enter the main string and store it in input_string
   PROMPT the user to enter the substring to check and store it in substring
   IF substring is found in input_string:
      PRINT "The substring 'substring' is found in the string."
   ELSE:
      PRINT "The substring 'substring' is not found in the string."
END
```

SOURCE CODE:

```
input_string = input("Enter the main string: ")
substring = input("Enter the substring to check: ")
if input_string.find(substring)>=0:
    print(f"The substring '{substring}' is found in the string.")
else:
    print(f"The substring '{substring}' is not found in the
string.")
```

OUTPUT:

```
Enter the main string: hello world
Enter the substring to check: world
The substring 'world' is found in the string.
```

RESULT:

Thus the **Python program to check if one string contains another string** has been successfully executed and the output was verified.

EX:NO: 2.3	CREATE AND MANIPULATE STRINGS USING INDEXING , SLICING AND VARIOUS STRING FUNCTIONS
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO CHECK IF ALL CHARACTERS IN A STRING ARE DIGITS	

AIM:

To write a Python program to Check if All Characters in a String Are Digits

PSEUDO CODE:

```
START
   PROMPT the user to enter a string and store it in input_string

   IF input_string contains only digits:
      PRINT "The string 'input_string' contains only digits."
   ELSE:
      PRINT "The string 'input_string' does not contain only digits."

END
```

SOURCE CODE:

```
input_string = input("Enter a string: ")
if input_string.isdigit():
    print(f"The string '{input_string}' contains only digits.")
else:
    print(f"The string '{input_string}' does not contain only
digits.")
```

OUTPUT:

```
Enter a string: 123
The string '123' contains only digits.
```

RESULT:

Thus the Python program to Remove Duplicates from a String has been successfully executed and the output was verified.

EX:NO: 2.4	CREATE AND MANIPULATE STRINGS USING INDEXING , SLICING AND VARIOUS STRING FUNCTIONS
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO REMOVE ALL VOWELS FROM THE STRING	

AIM:
To write a Python program to remove all vowels from the string

PSEUDO CODE:

```
START
   PROMPT the user to enter a string and store it in input_string
   SET vowels to the string "aeiouAEIOU" (all vowels in both lowercase and uppercase)
   CREATE an empty string result
   FOR each character in input_string:
      IF the character is NOT in vowels:
         ADD the character to result
   PRINT the message: "String after removing vowels: result"
END
```

SOURCE CODE:

```
input_string = input("Enter a string: ")
vowels = "aeiouAEIOU"
result = ''.join([char for char in input_string if char not in
vowels])
print(f"String after removing vowels: {result}")
```

OUTPUT:

```
Enter a string: hello world
String after removing vowels: hll wrld
```

RESULT:

Thus the python program to remove all vowels from the string has been successfully executed and the output was verified.

EX:NO: 2.5	CREATE AND MANIPULATE STRINGS USING INDEXING , SLICING AND VARIOUS STRING FUNCTIONS
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO COUNT OCCURRENCES OF A CHARACTER IN A STRING	

AIM:

To write a Python program to count occurrences of a character in a String

PSEUDO CODE:

```
START
   PROMPT the user to enter a string and store it in input_string
   PROMPT the user to enter the character to count and store it in char
   CALL the count() function on input_string to count the occurrences of char
   STORE the result in variable count
   PRINT the message: "The character 'char' appears 'count' times in the string."
END
```

SOURCE CODE:

```
input_string = input("Enter a string: ")
char = input("Enter the character to count: ")
count = input_string.count(char)
print(f"The character '{char}' appears {count} times in the
string.")
```

OUTPUT:

```
Enter a string: hello world
Enter the character to count: o
The character 'o' appears 2 times in the string.
```

RESULT:

Thus the python program to count occurrences of a character in a String has been successfully executed and the output was verified.

EX:NO: 3.1	CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING
DATE:	

CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING

AIM:

To create and manipulate lists using operations, slices, methods, list comprehension, and looping.

PSEUDO CODE:

```
START
1. Initialize a list called "numbers" with elements [5, 10, 15, 20, 25, 30].
2. Add the number 35 to the end of the list.
  PRINT the updated list.
3. Remove the number 15 from the list.
  PRINT the updated list.
4. Modify the element at index 2 of the list to be 100.
  PRINT the updated list.
5. Extract a sublist from index 1 to index 4 (inclusive).
  PRINT the sublist.
6. Create a new list containing all numbers greater than 20 from the "numbers" list using
list comprehension.
  PRINT the new list.
7. Loop through each number in the "numbers" list and print the number multiplied by 2.
8. Check if the number 25 exists in the list.
  IF 25 is in the list:
    PRINT "25 is in the list."
  ELSE:
    PRINT "25 is not in the list."
END
```

SOURCE CODE:

```
numbers = [5, 10, 15, 20, 25, 30]
numbers.append(35)
print("List after appending 35:", numbers)
numbers.remove(15)
print("List after removing 15:", numbers)
numbers[2] = 100
print("List after modifying the element at index 2:", numbers)
sublist = numbers[1:5]
print("Sliced list from index 1 to 4:", sublist)
greater_than_20 = [num for num in numbers if num > 20]
print("List of numbers greater than 20:", greater_than_20)
print("Each number doubled:")
for num in numbers:
    print(num * 2)
if 25 in numbers:
    print("25 is in the list.")
else:
    print("25 is not in the list.")
```

OUTPUT:

```
List after appending 35: [5, 10, 15, 20, 25, 30, 35]
List after removing 15: [5, 10, 20, 25, 30, 35]
List after modifying the element at index 2: [5, 10, 100, 25, 30,
35]
Sliced list from index 1 to 4: [10, 100, 25, 30]
List of numbers greater than 20: [100, 25, 30, 35]
Each number doubled:
10
20
200
50
60
70
25 is in the list.
```

RESULT:

Thus the python program to create and manipulate lists using operations, slices, methods, list comprehension, and looping has been successfully executed and the output was verified.

EX:NO: 3.2	CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO SEARCH AN ELEMENT FROM THE GIVEN LIST USING LINEAR SEARCH.	

AIM:

To write a Python program to search an element from the given list using Linear search.

PSEUDO CODE:

```
START
   PROMPT user to input a list of numbers separated by space
   CONVERT the input string to a list of integers and store it in arr
   PROMPT user to input the number to search for, store it in target
   SET found to False
   FOR each index i from 0 to length of arr - 1:
      IF arr[i] equals target:
         SET found to True
         PRINT "Number target found at index i"
         BREAK the loop
   IF found is False:
      PRINT "Number target not found in the list."
END
```

SOURCE CODE:

```
arr = list(map(int, input("Enter a list of numbers separated by
space: ").split()))
target = int(input("Enter the number you want to search for: "))
found = False
for i in range(len(arr)):
    if arr[i] == target:
        found = True
        print(f"Number {target} found at index {i}")
        break
if not found:
    print(f"Number {target} not found in the list.")
```

OUTPUT:

```
Enter a list of numbers separated by space: 2 5 8 9 10
Enter the number you want to search for: 5
Number 5 found at index 1
```

RESULT:

Thus the python program to search an element from the given list using Linear search has been successfully executed and the output was verified.

EX:NO: 3.3	CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING
DATE:	

IMPLEMENTATION OF PYTHON PROGRAM TO SEARCH AN ELEMENT FROM THE GIVEN LIST USING BINARY SEARCH.

AIM:

To write a Python program to search an element from the given list using Binary search.

PSEUDO CODE:

```
START
   PROMPT user to input a list of numbers separated by space
   CONVERT the input string to a list of integers and store it in arr
   PROMPT user to input the number to search for, store it in target
   SET found to False
   SET low to 0 and high to len-1
   WHILE low <=high:
      CALCULATE mid = (low+high)//2
       IF arr[mid] equals target:
           SET found to True
           PRINT "Number target found at index i"
           BREAK the loop
     ELSE IF arr[mid] is less than target:
          SET low to mid+1
    ELSE:
         SET high to mid-1
   IF found is False:
       PRINT "Number target not found in the list."
END
```

SOURCE CODE:

```
arr = list(map(int, input("Enter a list of numbers separated by
space (sorted): ").split()))
target = int(input("Enter the number you want to search for: "))
low = 0
high = len(arr) - 1
found = False
while low <= high:
    mid = (low + high) // 2
    if arr[mid] == target:
        found = True
        print(f"Number {target} found at index {mid}")
        break
    elif arr[mid] < target:
        low = mid + 1  # Search the right half
    else:
        high = mid - 1  # Search the left half
if not found:
    print(f"Number {target} not found in the list.")
```

OUTPUT:

```
Enter a list of numbers separated by space (sorted): 2 5 8 9 10
Enter the number you want to search for: 5
Number 5 found at index 1
```

RESULT:

Thus the python program to search an element from the given list using Binary search has been successfully executed and the output was verified.

EX:NO: 3.4	CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM SUM ALL THE NUMBERS IN THE GIVEN LIST	

AIM:

To write a Python program to sum all the numbers in the given list

PSEUDO CODE:

```
START
   PROMPT the user to enter a list of numbers and store it in a list called numbers
   SET total_sum = 0
   FOR each num in numbers:
      ADD num to total_sum
   PRINT total_sum
END
```

SOURCE CODE:

```python
numbers = list(map(int, input("Enter a list of numbers separated
by space: ").split()))
total_sum = 0
for num in numbers:
    total_sum += num
print("The sum of all numbers in the list is:", total_sum)
```

OUTPUT:

```
Enter a list of numbers separated by space: 1 8 4 2 6 10
The sum of all numbers in the list is: 31
```

RESULT:

Thus the python program to sum all the numbers in the given list has been successfully executed and the output was verified.

EX:NO: 3.5	**CREATE AND MANIPULATE LISTS USING OPERATIONS , SLICES, METHODS , LIST COMPREHENSION AND LOOPING**
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO SORT THE ELEMENTS IN THE LIST USING BUBBLE SORT	

AIM:

To write a Python program to sort the elements in the list using Bubble sort

PSEUDO CODE:

```
START
   PROMPT the user to enter a list of numbers and store it in a list called numbers
   SET n = length of numbers
   FOR i from 0 to n-1:
      FOR j from 0 to n-i-2:
         IF numbers[j] > numbers[j+1]:
            SWAP numbers[j] and numbers[j+1]
   PRINT numbers
END
```

SOURCE CODE:

```
numbers = list(map(int, input("Enter a list of numbers separated
by space: ").split()))
n = len(numbers)
for i in range(n):
    for j in range(0, n-i-1):
        if numbers[j] > numbers[j+1]:
            numbers[j], numbers[j+1] = numbers[j+1], numbers[j]
print("Sorted list:", numbers)
```

OUTPUT:

```
Enter a list of numbers separated by space: 4 2 8 1 4 7 4 0
Sorted list: [0, 1, 2, 4, 4, 4, 7, 8]
```

RESULT:

Thus the python program to sort the elements in the list using Bubble sort has been successfully executed and the output was verified.

EX:NO: 4.1	CREATE AND MANIPULATE TUPLES, DICTIONARIES, AND SETS, AND UNDERSTAND THE DIFFERENCES BETWEEN MUTABLE AND IMMUTABLE TYPES.
DATE:	
IMPLEMENTATION OF A PYTHON PROGRAM TO CREATE AND MANIPULATE TUPLES.	

AIM:

To write a python program to create and manipulate tuples.

PSEUDO CODE:

```
START
    Create a tuple of 'fruits' with the values: ("apple", "banana", "cherry", "orange",
    "mango").
    Print the original 'fruits' tuple.

    Access and print the first element of 'fruits'.
    Access and print the last element of 'fruits'.

    Slice the 'fruits' tuple from index 1 to index 4 (excluding 4), and print the sliced
    tuple.

    Create another tuple 'veggies' with values: ("carrot", "broccoli", "spinach").
    Concatenate 'fruits' and 'veggies' and store the result in 'all_food'. Print 'all_food'.

    Repeat the 'fruits' tuple two times and store the result in 'repeat_fruits'. Print
    'repeat_fruits'.

    Check if "banana" exists in 'fruits'. Print the result ("True" or "False").
    Check if "grape" exists in 'fruits'. Print the result ("True" or "False").

    Calculate the length of the 'fruits' tuple and print the result.

END
```

SOURCE CODE:

```
fruits = ("apple", "banana", "cherry", "orange", "mango")
print("Original Tuple:", fruits)

print("\nFirst element:", fruits[0])
print("Last element:", fruits[-1])

print("\nSliced Tuple (2nd to 4th elements):", fruits[1:4])

veggies = ("carrot", "broccoli", "spinach")
all_food = fruits + veggies
print("\nConcatenated Tuple (Fruits + Veggies):", all_food)

repeat_fruits = fruits * 2
print("\nRepeated Tuple:", repeat_fruits)

print("\nIs 'banana' in the fruits tuple?", "banana" in fruits)
print("Is 'grape' in the fruits tuple?", "grape" in fruits)

print("\nLength of the tuple:", len(fruits))
```

OUTPUT:

```
Original Tuple: ('apple', 'banana', 'cherry', 'orange', 'mango')

First element: apple
Last element: mango

Sliced Tuple (2nd to 4th elements): ('banana', 'cherry',
'orange')

Concatenated Tuple (Fruits + Veggies): ('apple', 'banana',
'cherry', 'orange', 'mango', 'carrot', 'broccoli', 'spinach')

Repeated Tuple: ('apple', 'banana', 'cherry', 'orange', 'mango',
'apple', 'banana', 'cherry', 'orange', 'mango')

Is 'banana' in the fruits tuple? True
Is 'grape' in the fruits tuple? False

Length of the tuple: 5
```

RESULT:

Thus the python program to create and manipulate tuples has been successfully executed and the output was verified.

EX:NO: 4.2	CREATE AND MANIPULATE TUPLES, DICTIONARIES, AND SETS, AND UNDERSTAND THE DIFFERENCES BETWEEN MUTABLE AND IMMUTABLE TYPES.
DATE:	

IMPLEMENTATION OF PYTHON PROGRAM TO CREATE AND MANIPULATE DICTIONARIES.

AIM:

To write a python program to create and manipulate dictionaries.

PSEUDO CODE:

```
START
        Create a dictionary called "student" with keys "name", "age", and "subjects" and
        their corresponding values.
        Print the original "student" dictionary.
        Access the value associated with the key "name" and print it.
        Add a new key-value pair "grade" with the value "A" to the "student" dictionary.
        Print the updated dictionary after adding the "grade".
        Modify the value of the key "age" to 21.
        Print the updated dictionary after modifying the "age".
        Remove the key-value pair for "subjects" from the "student" dictionary using the
        `pop` method.
        Print the updated dictionary after removing "subjects".
        Check if the key "grade" exists in the dictionary.
           - If it exists, print the value associated with the key "grade".
        Iterate over the "student" dictionary.
           - For each key-value pair in the dictionary, print the key and its corresponding
        value.
END
```

SOURCE CODE:

```
student = {
    "name": "Alice",
    "age": 20,
    "subjects": ["Math", "Science"]
}

print("Original Dictionary:", student)

print("\nName:", student["name"])

student["grade"] = "A"
print("\nAfter adding grade:", student)

student.update({"age": 21})
print("\nAfter modifying age:", student)

student.pop("subjects")
print("\nAfter removing subjects:", student)

print("\nIterating through the dictionary:")
for key, value in student.items():
    print(f"{key}: {value}")
```

OUTPUT:

```
Original Dictionary: {'name': 'Alice', 'age': 20, 'subjects':
['Math', 'Science']}

Name: Alice

After adding grade: {'name': 'Alice', 'age': 20, 'subjects':
['Math', 'Science'], 'grade': 'A'}

After modifying age: {'name': 'Alice', 'age': 21, 'subjects':
['Math', 'Science'], 'grade': 'A'}

After removing subjects: {'name': 'Alice', 'age': 21, 'grade':
'A'}

Iterating through the dictionary:
name: Alice
age: 21
grade: A
```

RESULT:

Thus the python program to create and manipulate dictionaries has been successfully executed and the output was verified.

EX:NO: 4.3	**CREATE AND MANIPULATE TUPLES, DICTIONARIES, AND SETS, AND UNDERSTAND THE DIFFERENCES BETWEEN MUTABLE AND IMMUTABLE TYPES.**
DATE:	
IMPLEMENTATION OF PYTHON PROGRAM TO CREATE AND MANIPULATE SETS.	

AIM:
To write a python program to create and manipulate sets.

PSEUDO CODE:

```
START
   CREATE a set "fruits" with elements: "apple", "banana", "cherry", "orange", "mango"
   PRINT "Original Set" with "fruits"

   ADD "grapes" to "fruits"
   PRINT "After adding grapes" with "fruits"

   REMOVE "banana" from "fruits"
   PRINT "After removing banana" with "fruits"

   CHECK if "apple" is in "fruits"
   PRINT result for checking if "apple" exists in "fruits"

   CHECK if "pear" is in "fruits"
   PRINT result for checking if "pear" exists in "fruits"

   CREATE a set "vegetables" with elements: "carrot", "broccoli", "spinach"

   UNION "fruits" and "vegetables" into "all_food"
   PRINT "Union of fruits and vegetables" with "all_food"

   INTERSECTION of "fruits" and "vegetables" into "common_food"
   PRINT "Common food in fruits and vegetables" with "common_food"

   POP an arbitrary element from "fruits" into "removed_element"
   PRINT "Removed an arbitrary element" with "removed_element"
   PRINT "Set after popping an element" with "fruits"
   CLEAR "fruits"
   PRINT "Set after clearing all elements" with "fruits"

END
```

SOURCE CODE:

```
fruits = {"apple", "banana", "cherry", "orange", "mango"}

print("Original Set:", fruits)

fruits.add("grapes")
print("\nAfter adding grapes:", fruits)

fruits.remove("banana")
print("\nAfter removing banana:", fruits)

print("\nIs 'apple' in the set?", "apple" in fruits)
print("Is 'pear' in the set?", "pear" in fruits)

vegetables = {"carrot", "broccoli", "spinach"}
all_food = fruits.union(vegetables)
print("\nUnion of fruits and vegetables:", all_food)

common_food = fruits.intersection(vegetables)
print("\nCommon food in fruits and vegetables:", common_food)

removed_element = fruits.pop()
print("\nRemoved an arbitrary element:", removed_element)
print("Set after popping an element:", fruits)

fruits.clear()
print("\nSet after clearing all elements:", fruits)
```

OUTPUT:

```
Original Set: {'orange', 'banana', 'mango', 'apple', 'cherry'}

After adding grapes: {'grapes', 'orange', 'banana', 'mango',
'apple', 'cherry'}

After removing banana: {'grapes', 'orange', 'mango', 'apple',
'cherry'}

Is 'apple' in the set? True
Is 'pear' in the set? False

Union of fruits and vegetables: {'spinach', 'grapes', 'orange',
'carrot', 'mango', 'apple', 'cherry', 'broccoli'}

Common food in fruits and vegetables: set()

Removed an arbitrary element: grapes
Set after popping an element: {'orange', 'mango', 'apple',
'cherry'}

Set after clearing all elements: set()
```

RESULT:

Thus the python program to create and manipulate sets has been successfully executed and the output was verified.

EX:NO: 4.4	**CREATE AND MANIPULATE TUPLES, DICTIONARIES, AND SETS, AND UNDERSTAND THE DIFFERENCES BETWEEN MUTABLE AND IMMUTABLE TYPES.**
DATE:	
ILLUSTRATE THE DIFFERENCES BETWEEN MUTABLE AND IMMUTABLE TYPES.	

AIM:

To write a Python program to Illustrate the differences between mutable and immutable types.

PSEUDO CODE:

```
START
1. Define an immutable object (String):
   - Set `str_example` to "Hello"
   - Print the original string
   - Modify the string by creating a new string: "h" + the substring of `str_example`
starting from the second character
   - Print the modified string

2. Define a mutable object (List):
   - Set `list_example` to [1, 2, 3]
   - Print the original list
   - Modify the first element of the list to 10
   - Print the modified list
   - Append 4 to the list
   - Print the list after adding an element
   - Remove element 2 from the list
   - Print the list after removing an element

3. Demonstrate the modification of a mutable object (List):
   - Set `mutable_list` to [1, 2, 3]
   - Print the original mutable list
   - Append 100 to the list
   - Print the modified mutable list

4. Demonstrate the modification of an immutable object (String):
   - Set `immutable_str` to "Hello"
   - Print the original immutable string
   - Modify the string by concatenating " World!" to `immutable_str` and assign it to a
new string
   - Print the modified immutable string
END
```

SOURCE CODE:

```
str_example = "Hello"
print("Original String:", str_example)

str_example = "h" + str_example[1:]
print("\nModified String (New String Created):", str_example)


list_example = [1, 2, 3]
print("\nOriginal List:", list_example)

list_example[0] = 10
print("\nModified List:", list_example)

list_example.append(4)
print("\nList after adding an element:", list_example)

list_example.remove(2)
print("\nList after removing an element:", list_example)


mutable_list = [1, 2, 3]
print("\nBefore modifying the mutable list:", mutable_list)
mutable_list.append(100)
print("After modifying the mutable list:", mutable_list)

immutable_str = "Hello"
print("\nBefore modifying the immutable string:", immutable_str)
immutable_str = immutable_str + " World!"
print("After modifying the immutable string:", immutable_str)
```

OUTPUT:

```
Original String: Hello

Modified String (New String Created): hello

Original List: [1, 2, 3]

Modified List: [10, 2, 3]

List after adding an element: [10, 2, 3, 4]

List after removing an element: [10, 3, 4]

Before modifying the mutable list: [1, 2, 3]
After modifying the mutable list: [1, 2, 3, 100]

Before modifying the immutable string: Hello
After modifying the immutable string: Hello World!
```

RESULT:

Thus the python program to Illustrate the differences between mutable and immutable types has been successfully executed and the output was verified.

EX:NO: 5.1	IMPLEMENT USER-DEFINED FUNCTIONS AND UNDERSTAND THE DIFFERENT TYPES OF FUNCTION ARGUMENTS, SUCH AS POSITIONAL, KEYWORD, AND DEFAULT ARGUMENTS.
DATE:	

IMPLEMENT USER-DEFINED FUNCTIONS AND UNDERSTAND THE DIFFERENT TYPES OF FUNCTION ARGUMENTS, SUCH AS POSITIONAL, KEYWORD, AND DEFAULT ARGUMENTS.

AIM:

To implement user-defined functions and understand the different types of function arguments, such as positional, keyword, and default arguments.

PSEUDO CODE:

```
START
    Define a function "add_numbers" that takes two parameters (a, b) and returns
    their sum.
    Define a function "greet" that takes two parameters (name, age) and prints a
    greeting message.
    Define a function "introduce" that takes two parameters (name, age) where 'age'
    has a default value of 30. It prints a message introducing the person.
    Call the "add_numbers" function with 10 and 20 as arguments and store the result
    in "result".
      PRINT the result.
    Call the "greet" function with "Alice" and 25 as keyword arguments.
    Call the "introduce" function with "Bob" as the only argument. Use the default
    value for age.
    Call the "introduce" function with "Charlie" and 40 as arguments to override the
    default age.
END
```

SOURCE CODE:

```python
def add_numbers(a, b):
    return a + b
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")
def introduce(name, age=30):
    print(f"My name is {name} and I am {age} years old.")
result = add_numbers(10, 20)
print("Sum using positional arguments:", result)
greet(name="Alice", age=25)
introduce(name="Bob")
introduce(name="Charlie", age=40)
```

OUTPUT:

```
Sum using positional arguments: 30
Hello, Alice! You are 25 years old.
My name is Bob and I am 30 years old.
My name is Charlie and I am 40 years old.
```

RESULT:

Thus the python program to implement user-defined functions and understand the different types of function arguments, such as positional, keyword, and default arguments has been successfully executed and the output was verified.

EX:NO: 6.1	**IIMPLEMENT INHERITANCE AND UNDERSTAND THE DIFFERENT TYPES OF INHERITANCE.**
DATE:	
IMPLEMENT A PYTHON PROGRAM TO ILLUSTRATE SINGLE INHERITANCE	

AIM:

To write a python program to illustrate Single inheritance

PSEUDO CODE:

```
START
1. Define Parent Class `Person`:
   - Define `__init__` method:
      - Initialize attributes `name` and `age` with provided values.
   - Define `display` method:
      - Print the values of `name` and `age`.

2. Define Child Class `Student` that inherits from `Person`:
   - Define `__init__` method:
      - Call the parent class's `__init__` method to initialize `name` and `age`.
      - Initialize attribute `grade` with provided value.
   - Define `display_student_info` method:
      - Call the `display` method from the parent class to print `name` and `age`.
      - Print the value of `grade`.

3. In Main Program:
   - Create an instance of `Student` with `name` as "Alice", `age` as 20, and `grade` as
"A".
   - Call the `display_student_info` method of the `student1` instance to display all
information (name, age, grade).
END
```

SOURCE CODE:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)
        self.grade = grade

    def display_student_info(self):
        self.display()
        print(f"Grade: {self.grade}")

student1 = Student("Alice", 20, "A")
student1.display_student_info()
```

OUTPUT:

```
Name: Alice
Age: 20
Grade: A
```

RESULT:

Thus the python program to illustrate Single level inheritance has been successfully executed and the output was verified.

EX:NO: 6.2	**IIMPLEMENT INHERITANCE AND UNDERSTAND THE DIFFERENT TYPES OF INHERITANCE.**
DATE:	
IMPLEMENT PYTHON PROGRAM TO ILLUSTRATE MULTI-LEVEL INHERITANCE	

AIM:

To write a python program to illustrate multi-level inheritance

PSEUDO CODE:

```
START
1. Define the Grandparent class "Animal":
   - Initialize a variable "species" in the constructor.
   - Define a method "display_species" that prints the species.

2. Define the Parent class "Mammal" that inherits from "Animal":
   - Initialize variables "species" and "habitat" in the constructor.
   - Call the parent class constructor using `super()` to initialize "species".
   - Define a method "display_habitat" that prints the habitat.

3. Define the Child class "Dog" that inherits from "Mammal":
   - Initialize variables "species", "habitat", and "breed" in the constructor.
   - Call the parent class constructor using `super()` to initialize "species" and "habitat".
   - Define a method "display_breed" that prints the breed.

4. In the main program:
   - Create an instance of the "Dog" class with specific "species", "habitat", and "breed".
   - Call the "display_species" method from "Animal".
   - Call the "display_habitat" method from "Mammal".
   - Call the "display_breed" method from "Dog".
END
```

SOURCE CODE:

```python
class Animal:
    def __init__(self, species):
        self.species = species

    def display_species(self):
        print(f"Species: {self.species}")

class Mammal(Animal):
    def __init__(self, species, habitat):
        super().__init__(species)
        self.habitat = habitat

    def display_habitat(self):
        print(f"Habitat: {self.habitat}")

class Dog(Mammal):
    def __init__(self, species, habitat, breed):
        super().__init__(species, habitat)
        self.breed = breed

    def display_breed(self):
        print(f"Breed: {self.breed}")

dog1 = Dog("Dog", "Domestic", "Golden Retriever")
dog1.display_species()
dog1.display_habitat()
dog1.display_breed()
```

OUTPUT:

```
Species: Dog
Habitat: Domestic
Breed: Golden Retriever
```

RESULT:

Thus the python program to illustrate Multi-level inheritance has been successfully executed and the output was verified.

EX:NO: 6.3	**IIMPLEMENT INHERITANCE AND UNDERSTAND THE DIFFERENT TYPES OF INHERITANCE.**
DATE:	
IMPLEMENT A PYTHON PROGRAM TO ILLUSTRATE MULTIPLE INHERITANCE	

AIM:

To write a python program to illustrate multiple inheritance

PSEUDO CODE:

```
START
    Define Class Person:
        Constructor:
            Input: name
            Set self.name = name
        Method get_name:
            Return self.name
    Define Class Worker:
        Constructor:
            Input: job_title
            Set self.job_title = job_title
        Method get_job:
            Return self.job_title
    Define Class Manager (Inherits from Person and Worker):
        Constructor:
            Input: name, job_title, department
            Call Person constructor with name
            Call Worker constructor with job_title
            Set self.department = department
        Method get_department:
            Return self.department
    Create Instance of Manager:
        Input: "Alice", "Software Engineer", "IT"
        Assign to variable manager
    Call Methods on manager Object:
        Call get_name:
            Output: "Alice"
        Call get_job:
            Output: "Software Engineer"
        Call get_department:
            Output: "IT"
END
```

SOURCE CODE:
```
class Person:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

class Worker:
    def __init__(self, job_title):
        self.job_title = job_title

    def get_job(self):
        return self.job_title

class Manager(Person, Worker):  # Inherits from both Person and 
Worker
    def __init__(self, name, job_title, department):
        Person.__init__(self, name)
        Worker.__init__(self, job_title)
        self.department = department

    def get_department(self):
        return self.department


manager = Manager("Alice", "Software Engineer", "IT")
print(manager.get_name())
print(manager.get_job())
print(manager.get_department())
```

OUTPUT:
```
Alice
Software Engineer
IT
```

RESULT:

Thus the python program to illustrate multiple inheritance has been successfully executed and the output was verified.

EX:NO: 7.1	**IMPLEMENT POLYMORPHISM THROUGH METHOD OVERLOADING,**
DATE:	**OVERRIDING, AND OPERATOR OVERLOADING.**
IMPLEMENT POLYMORPHISM THROUGH METHOD OVERLOADING BY PROVIDING DIFFERENT LOGIC FOR DIFFERENT INPUT	

AIM:

To implement polymorphism through method overloading by writing the method's logic so that different code executes inside the function depending on the parameter passed.

PSEUDO CODE:

```
START
Create Shape class
Define a method area
create Square object and call the area method
create rectangle object and call the area method
END
```

SOURCE CODE

```python
class Shape:
    # function with two default parameters
    def area(self, a, b=0):
        if b > 0:
            print('Area of Rectangle is:', a * b)
        else:
            print('Area of Square is:', a ** 2)

square = Shape()
square.area(5) # if no.of arg is 1 then it is square

rectangle = Shape()
rectangle.area(5, 3) # if  no.of arg is 2 then it is rectangle
```

OUTPUT:

```
Area of Square is:25
Area of Rectangle is:15
```

RESULT :

Thus the python program for method overloading has been successfully executed and the output was verified.

EX:NO: 7.2	IMPLEMENT POLYMORPHISM THROUGH METHOD OVERLOADING, OVERRIDING, AND OPERATOR OVERLOADING.
DATE:	
IMPLEMENT A PYTHON PROGRAM TO IMPLEMENT METHOD OVERRIDING	

AIM:

To write a python program to implement method overriding

PSEUDO CODE:

```
START
    Define a class "Animal":
        - Define a method "sound" that prints "Animal makes a sound".
    Define a class "Dog" that inherits from "Animal":
        - Override the "sound" method to print "Dog barks".

    Create an object "animal" from the "Animal" class:
        - Call the "sound" method on "animal".
    Create an object "dog" from the "Dog" class:
        - Call the "sound" method on "dog".
END
```

SOURCE CODE

```python
class Animal:
    def sound(self):
        print("Animal makes a sound")
class Dog(Animal):
    def sound(self):
        print("Dog barks")
animal = Animal()
animal.sound()
dog = Dog()
dog.sound()
```

OUTPUT

```
Animal makes a sound
Dog barks
```

RESULT:

Thus the python program for method overriding has been successfully executed and the output was verified.

EX:NO: 7.3	**IMPLEMENT POLYMORPHISM THROUGH METHOD OVERLOADING, OVERRIDING, AND OPERATOR OVERLOADING.**
DATE:	
IMPLEMENT A PYTHON PROGRAM TO IMPLEMENT OPERATOR OVERLOADING	

AIM:

To write a python program to implement operator overloading

PSEUDO CODE:

```
START
 Define a class "ComplexNumber":
    - Define an initializer method that takes two parameters: "real" and "imag".
    - Define the "_add_" method to add two complex numbers by adding their real and
imaginary parts.
    - Define the "_str_" method to represent the complex number as a string in the form
"real + imag i".

 Create two objects "c1" and "c2" from the "ComplexNumber" class with real and
imaginary parts:
    - Add "c1" and "c2" using the overloaded "+" operator and store the result in "c3".
    - Print the result of adding the complex numbers.
END
```

SOURCE CODE

```
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag +
other.imag)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

c1 = ComplexNumber(3, 2)
c2 = ComplexNumber(1, 7)
c3 = c1 + c2
print("Sum of complex numbers:", c3)
```

OUTPUT:

```
Sum of complex numbers: 4 + 9i
```

RESULT:

Thus the python program for operator overloading has been successfully executed and the output was verified.