Ex.No. 1.a)

SUM OF EVEN NUMBERS FROM 1 TO 50

Aim:

To write a Python program to calculate and display the sum of all even numbers in the range 1 to 50.

Algorithm:

1. Start
2. Initialize a variable `sum` to 0
3. Use a `for` loop to iterate from 1 to 50
4. In each iteration, check if the number is even (i.e., divisible by 2)
5. If it is even, add it to the `sum`
6. After the loop, display the final `sum`
7. Stop

Program:

```
sum = 0

for num in range(1, 51):

    if num % 2 == 0:

        sum += num

print("Sum of even numbers from 1 to 50 is:", sum)
```

Output:

```
Sum of even numbers from 1 to 50 is: 650
```

Result:

The Python program successfully calculated and displayed the sum of all even numbers from 1 to 50.

Ex.No. 1.b) **PYTHON FUNCTION THAT RETURNS MULTIPLE VALUES**

Aim:

To develop a Python function that returns multiple values and demonstrates its usage.

Algorithm:

1. Start
2. Define a function that takes two numbers as input
3. Inside the function, calculate:
    - Sum
    - Difference
4. Return both results using a tuple
5. Call the function and store the returned values
6. Display the returned values
7. Stop

Program:

```
def calculate(a, b):

    sum_result = a + b

    diff_result = a - b

    return sum_result, diff_result

# Function usage

x = 10

y = 5

sum_val, diff_val = calculate(x, y)

print("Sum:", sum_val)

print("Difference:", diff_val)
```

Output:

```
Sum: 15
Difference: 5
```

Result:

The program successfully defined a Python function that returned multiple values (sum and difference) and demonstrated its usage.

Ex.no. 2

PATTERN GENERATION USING LOOPS

Aim:

To create a Python program that generates a specified pattern using loops.

Algorithm:

1. Start
2. Take input for number of rows (or set a fixed number, e.g., 5)
3. Use a nested loop:
   - Outer loop controls the number of rows
   - Inner loop prints stars in each row
4. Print the pattern
5. Stop

Program:

```
rows = 5  # You can change this value



for i in range(1, rows + 1):

    for j in range(i):

        print("*", end=" ")

    print()
```

Output:

```
*

* *

* * *

* * * *

* * * * *
```

Result:

The program successfully generated a right-angled triangle pattern using nested loops.

Ex.No.2.b

PYTHON FUNCTION WITH DEFAULT ARGUMENTS

Aim:

To design a Python function incorporating default arguments and demonstrate its functionality.

Algorithm:

1. Start
2. Define a function `area()` with two parameters: `length` and `width`, where `width` has a default value
3. If the function is called with one argument, use default value for `width`
4. If called with two arguments, override the default
5. Calculate and return the area
6. Print the result
7. Stop

Program:

```
# Function with default argument for width

def area(length, width=10):
        return length * width

# Function calls
print("Area 1:", area(5, 4))  # Both arguments provided
print("Area 2:", area(7))     # Only length provided, width uses default
```

Output:

```
Area 1: 20

Area 2: 70
```

Result:

The program successfully demonstrated the use of default arguments in a function that calculates area.

Ex.no.3a

FIND LENGTH OF A STRING WITHOUT USING BUILT-IN FUNCTIONS

Aim:

To develop a Python program to determine the length of a string without using built-in library functions.

Algorithm:

1. Start
2. Take a string as input (or use a predefined string)
3. Initialize a counter variable to 0
4. Use a loop to iterate through each character of the string
5. For each character, increase the counter by 1
6. After the loop ends, print the counter as the string length
7. Stop

Program:

```
text = "Hello Python"

count = 0

for char in text:

    count += 1

print("The length of the string is:", count)
```

Output:

```
The length of the string is: 12
```

Result:

The program successfully determined the length of a string without using built-in functions like `len()`.

Ex.no.3b

CHECK WHETHER A SUBSTRING EXISTS WITHIN A STRING

Aim:

To construct a Python program to check whether a given substring exists within a string.

Algorithm:

1. Start
2. Input the main string
3. Input the substring to search
4. Use `in` operator to check if the substring is present
5. Display the result
6. Stop

Program:

```
# Simple program to check if a substring exists in a string

main_string = input("Enter the main string: ")
substring = input("Enter the substring to search: ")

if substring in main_string:
    print("Substring found in the main string.")
else:
    print("Substring not found in the main string.")
```

Output:

```
Enter the main string: Hello Python

Enter the substring to search: Python

Substring found in the main string.
```

Result:

The program successfully checked and confirmed whether the given substring exists in the main string using the `in` operator.

Ex.no.4a

PERFORM OPERATIONS ON A LIST

Aim:

To develop a Python program to perform operations on a list including adding elements, inserting elements, and slicing.

Algorithm:

1. Start
2. Create an empty or predefined list
3. Add elements using `append()`
4. Insert elements at specific positions using `insert()`
5. Use slicing to display portions of the list
6. Print the updated list and sliced parts
7. Stop

Program:

```
# Program to perform operations on a list

# Initial list
my_list = [10, 20, 30]
print("Initial List:", my_list)

# Adding elements using append()
my_list.append(40)
my_list.append(50)
print("After Appending:", my_list)

# Inserting element at specific position
my_list.insert(2, 25)  # Inserts 25 at index 2
print("After Inserting 25 at index 2:", my_list)

# Slicing the list
print("Sliced List [1:4]:", my_list[1:4])
print("Sliced List [:-1]:", my_list[:-1])
```

Output:

```
Initial List: [10, 20, 30]
After Appending: [10, 20, 30, 40, 50]
After Inserting 25 at index 2: [10, 20, 25, 30, 40, 50]
Sliced List [1:4]: [20, 25, 30]
Sliced List [:-1]: [10, 20, 25, 30, 40]
```

Result:

The program successfully performed list operations including adding, inserting, and slicing elements.

Ex.No.4b APPLY BUILT-IN FUNCTIONS ON A LIST

Aim:

To design a Python program that applies five built-in functions on a given list and displays the results.

Algorithm:

1. Start
2. Create a list of numbers
3. Apply the following built-in functions:
    1. `len()` to get the number of elements
    2. `max()` to find the maximum value
    3. `min()` to find the minimum value
    4. `sum()` to calculate total
    5. `sorted()` to sort the list
4. Display the results
5. Stop

Program:

```
numbers = [15, 42, 7, 29, 3]

print("Original List:", numbers)

print("Length of the list:", len(numbers))

print("Maximum value:", max(numbers))

print("Minimum value:", min(numbers))

print("Sum of elements:", sum(numbers))

print("Sorted list:", sorted(numbers))
```

Output:

```
Original List: [15, 42, 7, 29, 3]

Length of the list: 5

Maximum value: 42

Minimum value: 3

Sum of elements: 96

Sorted list: [3, 7, 15, 29, 42]
```

Result:

The program successfully applied five built-in functions (`len()`, `max()`, `min()`, `sum()`, and `sorted()`) on a given list and displayed the results.

Ex.no.5

REPRESENT AND CONCATENATE TUPLES

Aim:

To create a Python program to represent two members' details (name, age, address, college) as tuples, concatenate these tuples, and display the combined information.

Algorithm:

1. Start
2. Create two tuples, each containing details: name, age, address, and college
3. Concatenate the tuples using + operator
4. Display the individual and combined information in a readable format
5. Stop

Program:

```
# Member 1 details

member1 = ("Alice", 20, "Chennai", "ABC College")

# Member 2 details

member2 = ("Bob", 21, "Coimbatore", "XYZ College")

# Concatenating the two tuples

combined_info = member1 + member2

# Displaying combined details in a readable way

print("Combined Member Details:\n")

print("Member 1:")

print("Name:", member1[0])

print("Age:", member1[1])

print("Address:", member1[2])

print("College:", member1[3])

print()

print("Member 2:")
print("Name:", member2[0])

print("Age:", member2[1])

print("Address:", member2[2])

print("College:", member2[3])

print()

print("Concatenated Tuple:")

print(combined_info)
```

Output:

```
Combined Member Details:

Member 1:

Name: Alice

Age: 20

Address: Chennai

College: ABC College

Member 2:

Name: Bob

Age: 21

Address: Coimbatore

College: XYZ College

Concatenated Tuple:

('Alice', 20, 'Chennai', 'ABC College', 'Bob', 21, 'Coimbatore', 'XYZ College')
```

Result:

The program successfully created two tuples for members' details, concatenated them, and displayed the combined information in both formatted and raw tuple form.

Ex.No.6 SORT LIST OF STRINGS BASED ON VOWEL COUNT

Aim:

To design a Python program that sorts a given list of strings based on the number of vowels each string contains and outputs the sorted list.

Algorithm:

1. Start
2. Define a function to count vowels in a string
3. Create a list of strings
4. Sort the list using the `sorted()` function with a custom key based on vowel count
5. Display the sorted list
6. Stop

Program:

```
# Function to count vowels in a string
def count_vowels(word):
    vowels = "aeiouAEIOU"
    count = 0
    for char in word:
        if char in vowels:
            count += 1
    return count


# List of strings
words = ["banana", "apple", "grape", "orange", "kiwi"]


# Sorting based on number of vowels
```

```
sorted_words = sorted(words, key=count_vowels)


# Display the result
print("Sorted list based on vowel count:")
print(sorted_words)
```

Output:

```
Sorted list based on vowel count:
['apple', 'grape', 'kiwi', 'banana', 'orange']
```

Result:

The program successfully sorted the list of strings based on the number of vowels in each string and displayed the sorted list.

Ex.No.7a CHECK IF A KEY EXISTS IN A DICTIONARY

Aim:

To examine a dictionary to determine whether a given key exists.

Algorithm:

1. Start
2. Create a dictionary with some key-value pairs
3. Input the key to search
4. Use the `in` operator to check if the key exists
5. Display whether the key is found or not

Program:

```
# Program to check if a key exists in a dictionary


# Sample dictionary
student = {
   "name": "John",
   "age": 21,
   "course": "B.Sc",
   "year": 3
}


# Input key to search
key_to_check = input("Enter the key to check: ")


# Check and display result
if key_to_check in student:
    print("Key exists in the dictionary.")
else:
    print("Key does not exist in the dictionary.")
```

Output:

```
Enter the key to check: age
Key exists in the dictionary.
```

Result:

The program successfully checked whether a given key exists in a dictionary.

Ex.no. 7b

ADD A NEW KEY-VALUE PAIR TO A DICTIONARY

Aim:

To analyze the process of adding a new key-value pair to an existing dictionary and understand its impact on the dictionary's structure.

Algorithm:

1. Start
2. Create a dictionary with some initial key-value pairs
3. Add a new key-value pair using assignment (`dict[key] = value`)
4. Display the dictionary before and after adding
5. Observe how the new pair is added
6. Stop

Program:

```
# Initial dictionary

student = {

    "name": "John",

    "age": 21,

    "course": "B.Sc"

}



# Display before adding

print("Before adding new key-value pair:")

print(student)



# Add new key-value pair

student["year"] = 3

# Display after adding
```

```
print("\nAfter adding new key-value pair:")

print(student)
```

Output:

```
Before adding new key-value pair:

{'name': 'John', 'age': 21, 'course': 'B.Sc'}


After adding new key-value pair:

{'name': 'John', 'age': 21, 'course': 'B.Sc', 'year': 3}
```

Result:

The program successfully added a new key-value pair to the dictionary.

Ex.no.8 IDENTIFY MOST FREQUENT WORDS IN A TEXT FILE

Aim:

To write a Python program that reads a text file and finds the most frequent words.

Algorithm:

1. Start
2. Open and read the file
3. Split the text into words
4. Count the frequency using a dictionary
5. Find the word(s) with the highest count
6. Display the most frequent word(s) and count
7. Stop

Program:

```
# Simple program to find the most frequent word in a text file

# Open the file
file = open("sample.txt", "r")
text = file.read()
file.close()

# Split into words
words = text.split()

# Count word frequencies
freq = {}

for word in words:
    if word in freq:
        freq[word] += 1
    else:
        freq[word] = 1

# Find the most frequent word
max_count = 0
most_frequent = ""

for word in freq:
    if freq[word] > max_count:
        max_count = freq[word]
        most_frequent = word

# Display result
print("Most frequent word:", most_frequent)
print("Frequency:", max_count)

Output:
Most frequent word: the
Frequency: 12
```

Result:

The program successfully read the file and displayed the most frequently occurring word along with its count.

Ex.no.9 CLASS WITH BMI CALCULATION

Aim:

To design a Python class with attributes `name`, `age`, `weight` (in kg), and `height` (in feet), and implement a method `get_bmi_result()` that returns the BMI category.

Algorithm:

1. Start
2. Define a class `Person` with `__init__` constructor to initialize name, age, weight, and height
3. Convert height from feet to meters
4. Calculate BMI using formula:

   BMI=weight (kg)height (m)2\text{BMI} = \frac{\text{weight (kg)}}{\text{height (m)}^2}BMI=height (m)2weight (kg)

5. Return category based on BMI:
   1. BMI < 18.5 → Underweight
   2. 18.5 ≤ BMI < 25 → Healthy
   3. BMI ≥ 25 → Obesity
6. Stop

Program:

```
class Person:
    def __init__(self, name, age, weight, height):
        self.name = name
        self.age = age
        self.weight = weight  # in kg
        self.height = height  # in feet

    def get_bmi_result(self):
        height_m = self.height * 0.3048  # convert feet to meters
        bmi = self.weight / (height_m ** 2)

        if bmi < 18.5:
            return "Underweight"
        elif 18.5 <= bmi < 25:
            return "Healthy"
        else:
            return "Obesity"

# Example usage
person1 = Person("John", 25, 70, 5.5)
```

```
print("BMI Category:", person1.get_bmi_result())
```

Output:

```
BMI Category: Healthy
```

Result:

The program successfully created a class with a BMI calculation method and returned the correct BMI category.

Ex.No.10a) CREATING NUMPY ARRAYS USING ARRAY()

Aim:

To develop a Python program to demonstrate the creation of NumPy arrays using the `array()` function.

Algorithm:

1. Start
2. Import the NumPy library
3. Create arrays using `numpy.array()` with:
   A list (1D array)
   A list of lists (2D array)

4. Display the arrays
5. Stop

Program:

```
# Program to create NumPy arrays using array() function

import numpy as np

# Creating a 1D array
arr1 = np.array([10, 20, 30, 40])
print("1D Array:")
print(arr1)

# Creating a 2D array
arr2 = np.array([[1, 2], [3, 4]])
print("\n2D Array:")
print(arr2)
```

Output:

```
1D Array:
[10 20 30 40]

2D Array:
[[1 2]
```

[3 4]]

Result:

The program successfully demonstrated the creation of 1D and 2D NumPy arrays using the `array()` function.

Ex.No.10b) DICTIONARY TO PANDAS DATAFRAME AND DATA EXPLORATION

Aim:

To create a dictionary with at least five keys (each containing a list of 10 values), convert it into a Pandas DataFrame, and explore the data using `head()` and data selection operations.

Algorithm:

1. Start
2. Import the Pandas library
3. Create a dictionary with five keys and list of values
4. Convert the dictionary into a DataFrame using `pd.DataFrame()`
5. Use `head()` to display the top rows
6. Perform column, row, and specific cell selection
7. Stop

Program:

```
import pandas as pd

# Step 1: Create dictionary
data = {
   "Name": ["John", "Emma", "Amit", "Sara", "Ravi", "Nina", "Kiran", "Lina", "Arun", "Priya"],
   "Age": [22, 23, 21, 22, 24, 23, 22, 21, 24, 23],
   "Maths": [78, 85, 90, 88, 76, 80, 92, 89, 75, 84],
   "Science": [80, 89, 85, 87, 77, 90, 86, 88, 76, 82],
   "English": [75, 80, 78, 82, 79, 77, 85, 81, 74, 83]
}

# Step 2: Convert dictionary to DataFrame
df = pd.DataFrame(data)

# Step 3: Display top 5 rows
print("Head of the DataFrame:")
print(df.head())

# Step 4: Data selection operations
print("\nSelect 'Name' and 'Maths' columns:")
print(df[["Name", "Maths"]])

print("\nSelect rows 0 to 2:")
print(df[0:3])

print("\nSelect value at row 2 and column 'English':")
print(df.at[2, "English"])
```

Output:

```
Head of the DataFrame:

   Name  Age  Maths  Science  English

0   John   22     78       80       75

1   Emma   23     85       89       80

2   Amit   21     90       85       78

3   Sara   22     88       87       82

4   Ravi   24     76       77       79


Select 'Name' and 'Maths' columns:

   Name  Maths

0   John     78

1   Emma     85

2   Amit     90

...

Select rows 0 to 2:

   Name  Age  Maths  Science  English

0  John   22     78       80       75

1  Emma   23     85       89       80

2  Amit   21     90       85       78

Select value at row 2 and column 'English':

78
```

Result:

The program successfully created a Pandas DataFrame from a dictionary and demonstrated data exploration using `head()`, column selection, row selection, and single value access.