MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

VISION

To nurture young minds in a learning environment of high academic value and imbibe spiritual and ethical values with technological and management competence.

MISSION

The Institute shall endeavor to incorporate the following basic missions in the teaching methodology:

Engineering Hardware – Software Symbiosis

Practical exercises in all Engineering and Management disciplines shall be carried out by Hardware equipment as well as the related software enabling deeper understanding of basic concepts and encouraging inquisitive nature.

Life – Long Learning

The Institute strives to match technological advancements and encourage students to keep updating their knowledge for enhancing their skills and inculcating their habit of continuous learning.

Liberalization and Globalisation

The Institute endeavors to enhance technical and management skills of students so that they are intellectually capable and competent professionals with Industrial Aptitude to face the challenges of globalization.

Diversification

The Engineering, Technology and Management disciplines have diverse fields of studies with different attributes. The aim is to create a synergy of the above attributes by encouraging analytical thinking.

Digitization of Learning Processes

The Institute provides seamless opportunities for innovative learning in all Engineering and Management disciplines through digitization of learning processes using analysis, synthesis, simulation, graphics, tutorials and related tools to create a platform for multi-disciplinary approach.

Entrepreneurship

The Institute strives to develop potential Engineers and Managers by enhancing their skills and research capabilities so that they become successfully entrepreneurs and responsible citizens.

Department of Computer Science & Technology

VISION

To Produce "Critical thinkers of Innovative Technology"

MISSION

To provide an excellent learning environment across the computer science discipline to inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities which enable them to become successful entrepreneurs in this globalized world.

- To nurture an excellent learning environment that helps students to enhance their problem solving skills and to prepare students to be lifelong learners by offering a solid theoretical foundation with applied computing experiences and educating them about their professional, and ethical responsibilities.
- To establish Industry-Institute Interaction, making students ready for the industrial environment and be successful in their professional lives.
- To promote research activities in the emerging areas of technology convergence.
- To build engineers who can look into technical aspects of an engineering solution thereby setting a ground for producing successful entrepreneur.

PROGRAMMING IN PYTHON LAB

PRACTICAL FILE

Faculty name: Dr. R.K. Choudhury

Student name: Swastik Sharma

Roll No: 03914812721

Semester: 6th

Group:6FSD-II-C

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

MAIT

उद्यमेन हि सिध्यन्ति
कार्याणि न मनोरथैः

Maharaja Agrasen Institute of Technology, PSP Area,
Sector – 22, Rohini, New Delhi – 110085

PROGRAMMING IN PYTHON LAB

PRACTICAL RECORD

PAPER CODE	**:**	CIE – 332P
Name of the student	:	Swastik Sharma
University Roll No.	:	03914812721
Branch	:	CST
Section/ Group	:	6 FSD-II-C

PRACTICAL DETAILS

Experiment No.	**Date**	**Experiment Name**	**Marks**					**Total Mark**	**Signature**
			R	**R**	**R**	**R**	**R**		

Rubrics for Lab Assessment

Rubrics		0	1	2	3
		Missing	Inadequate	Needs Improvement	Adequate
R1	Is able to identify the problem to be solved and define the objectives of the experiment.	No mention is made of the problem to be solved.	An attempt is made to identify the problem to be solved but it is described in a confusing manner, objectives are not relevant, objectives contain technical/ conceptual errors or objectives are not measurable.	The problem to be solved is described but there are minor omissions or vague details. Objectives are conceptually correct and measurable but may be incomplete in scope or have linguistic errors.	The problem to be solved is clearly stated. Objectives are complete, specific, concise, and measurable. They are written using correct technical terminology and are free from linguistic errors.
R2	Is able to design a reliable experiment that solves the problem.	The experiment does not solve the problem.	The experiment attempts to solve the problem but due to the nature of the design the data will not lead to a reliable solution.	The experiment attempts to solve the problem but due to the nature of the design there is a moderate chance the data will not lead to a reliable solution.	The experiment solves the problem and has a high likelihood of producing data that will lead to a reliable solution.
R3	Is able to communicate the details of an experimental procedure clearly and completely.	Diagrams are missing and/or experimental procedure is missing or extremely vague.	Diagrams are present but unclear and/or experimental procedure is present but important details are missing.	Diagrams and/or experimental procedure are present but with minor omissions or vague details.	Diagrams and/or experimental procedure are clear and complete.
R4	Is able to record and represent data in a meaningful way.	Data are either absent or incomprehensible.	Some important data are absent or incomprehensible.	All important data are present, but recorded in a way that requires some effort to comprehend.	All important data are present, organized and recorded clearly.
R5	Is able to make a judgment about the results of the experiment.	No discussion is presented about the results of the experiment .	A judgment is made about the results, but it is not reasonable or coherent.	An acceptable judgment is made about the result, but the reasoning is flawed or incomplete.	An acceptable judgment is made about the result, with clear reasoning. The effects of assumptions and experimental uncertainties are considered.

EXPERIMENT – 1

AIM: Write a Program to demonstrate different types of datatypes and type casting.

THEORY

Data Types:

Data types specify the type of data that variables can hold. Common data types include:

Integer: Represents whole numbers without any fractional part. Examples include int in C/C++, int in Python, and Integer in Java.

Floating-point: Represents numbers with fractional parts. Examples include float and double in C/C++, float in Python, and Double in Java.

Character: Represents single characters. Examples include char in C/C++, char in Python (as a single-character string), and Character in Java.

Boolean: Represents true or false values. Examples include bool in C/C++, bool in Python, and Boolean in Java.

Type Casting:

Type casting is the process of converting one data type into another. It can be categorized into two types:

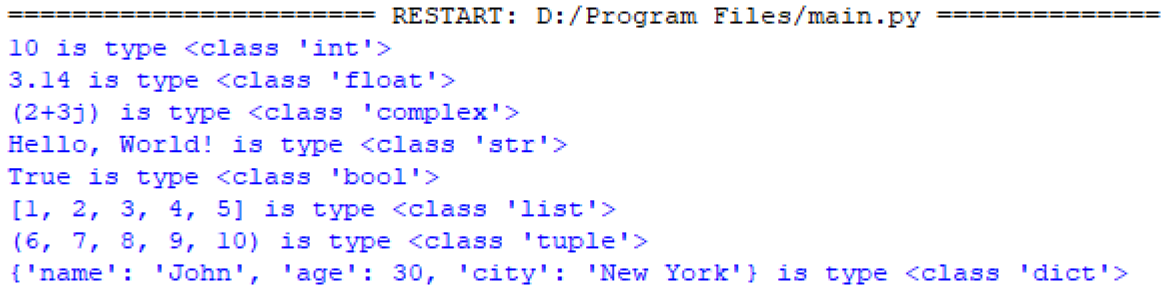
Implicit Type Casting: Also known as automatic type conversion, it occurs when the destination data type can hold a larger range of values than the source data type. For example, converting an integer to a floating-point number.

Explicit Type Casting: Also known as manual type conversion, it involves the programmer explicitly converting the data type of a variable. This is necessary when converting from a larger data type to a smaller one, potentially resulting in data loss. For example, converting a floating-point number to an integer.

CODE:

```
# Program to demonstrate different data types and type casting

# Integer data type
integer_number = 10
print(integer_number, "is type", type(integer_number))


# Float data type
float_number = 3.14
```

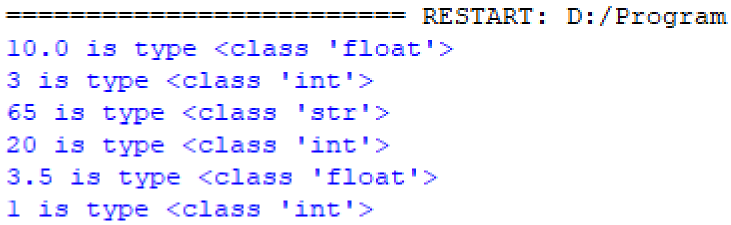
```
print(float_number, "is type", type(float_number))

# Complex data type

complex_number = 2 + 3j

print(complex_number, "is type", type(complex_number))

# String data type

string_value = "Hello, World!"

print(string_value, "is type", type(string_value))

# Boolean data type

boolean_value = True

print(boolean_value, "is type", type(boolean_value))

# List data type

list_values = [1, 2, 3, 4, 5]

print(list_values, "is type", type(list_values))

# Tuple data type

tuple_values = (6, 7, 8, 9, 10)

print(tuple_values, "is type", type(tuple_values))

# Dictionary data type

dictionary_values = {"name": "John", "age": 30, "city": "New York"}

print(dictionary_values, "is type", type(dictionary_values))
```



```
======================= RESTART: D:/Program Files/main.py ==============
10 is type <class 'int'>
3.14 is type <class 'float'>
(2+3j) is type <class 'complex'>
Hello, World! is type <class 'str'>
True is type <class 'bool'>
[1, 2, 3, 4, 5] is type <class 'list'>
(6, 7, 8, 9, 10) is type <class 'tuple'>
{'name': 'John', 'age': 30, 'city': 'New York'} is type <class 'dict'>
```

```
# Type casting examples

# Converting integer to float
```

```
integer_to_float = float(10)

print(integer_to_float, "is type", type(integer_to_float))

# Converting float to integer

float_to_integer = int(3.14)

print(float_to_integer, "is type", type(float_to_integer))

# Converting integer to string

integer_to_string = str(65)

print(integer_to_string, "is type", type(integer_to_string))

# Converting string to integer

string_to_integer = int("20")

print(string_to_integer, "is type", type(string_to_integer))

# Converting string to float

string_to_float = float("3.5")

print(string_to_float, "is type", type(string_to_float))

# Converting boolean to integer

boolean_to_integer = int(True)

print(boolean_to_integer, "is type", type(boolean_to_integer))
```



```
========================== RESTART: D:/Program
10.0 is type <class 'float'>
3 is type <class 'int'>
65 is type <class 'str'>
20 is type <class 'int'>
3.5 is type <class 'float'>
1 is type <class 'int'>
```

EXPERIMENT – 2

AIM: Write a Program to demonstrate different operations on number in Python.

THEORY

Python is a versatile programming language that supports various operations on numbers. These operations include arithmetic operations, comparison operations, and type casting. Below is an explanation of each aspect:

Arithmetic Operations:

Addition (+): Adds two numbers together.

Syntax: result = number1 + number2

Subtraction (-): Subtracts one number from another.

Syntax: result = number1 - number2

Multiplication (*): Multiplies two numbers.

Syntax: result = number1 * number2

Division (/): Divides one number by another. Results in a floating-point number.

Syntax: result = number1 / number2

Floor Division (//): Divides one number by another and rounds down to the nearest integer.

Syntax: result = number1 // number2

Modulus (%): Returns the remainder of the division of two numbers.

Syntax: result = number1 % number2

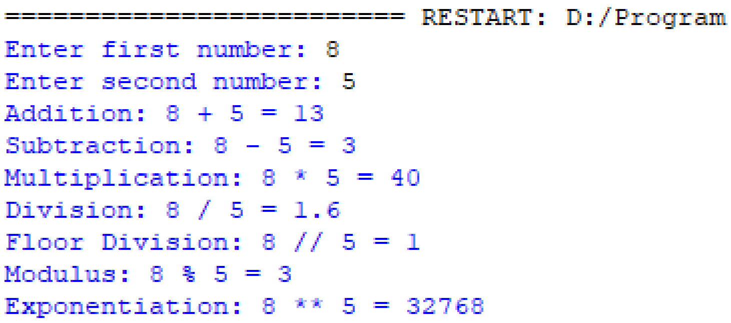
Exponentiation (**): Raises one number to the power of another.

Syntax: result = number1 ** number2

CODE:

```
# Python Program to demonstrate different operations on numbers with user input

# Taking input from the user

num1 = eval(input("Enter first number: "))

num2 = eval(input("Enter second number: "))

# Addition
```

```
sum_result = num1 + num2

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

# Subtraction

difference = num1 - num2

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

# Multiplication

product = num1 * num2

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

# Division

quotient = num1 / num2

print("Division:", num1, "/", num2, "=", quotient)

# Floor Division (returns only the integer part of the division)

floor_division = num1 // num2

print("Floor Division:", num1, "//", num2, "=", floor_division)

# Modulus (remainder of the division)

remainder = num1 % num2

print("Modulus:", num1, "%", num2, "=", remainder)

# Exponentiation

exponent = num1 ** num2

print("Exponentiation:", num1, "**", num2, "=", exponent)
```



```
========================= RESTART: D:/Program
Enter first number: 8
Enter second number: 5
Addition: 8 + 5 = 13
Subtraction: 8 - 5 = 3
Multiplication: 8 * 5 = 40
Division: 8 / 5 = 1.6
Floor Division: 8 // 5 = 1
Modulus: 8 % 5 = 3
Exponentiation: 8 ** 5 = 32768
```

EXPERIMENT – 3

AIM: Write a Program to demonstrate different comparison operators in Python.

THEORY

Comparison operators in Python are used to compare two values or expressions. They return a Boolean value (True or False) based on the comparison result. Python supports several comparison operators, each serving a different purpose. Below are the common comparison operators in Python along with their explanations:

Equal to (==): Checks if two values are equal. Returns True if the values are equal, False otherwise.

Syntax: value1 == value2

Not equal to (!=): Checks if two values are not equal. Returns True if the values are not equal, False otherwise.

Syntax: value1 != value2

Greater than (>): Checks if the left operand is greater than the right operand. Returns True if the left operand is greater, False otherwise.

Syntax: value1 > value2

Less than (<): Checks if the left operand is less than the right operand. Returns True if the left operand is less, False otherwise.

Syntax: value1 < value2

Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand. Returns True if the left operand is greater than or equal to the right operand, False otherwise.

Syntax: value1 >= value2

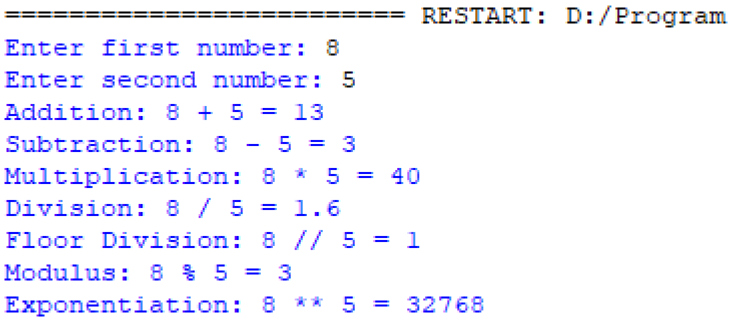
Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand. Returns True if the left operand is less than or equal to the right operand, False otherwise.

Syntax: value1 <= value2

CODE:

```
# Python Program to demonstrate different comparison operators in Python

# Take numbers from keyboard

num1 = eval(input("Enter the first number: "))

num2 = eval(input("Enter the second number: "))
```

```
# Comparison operators

# Equal to

print(num1, "==", num2, "is", num1 == num2)

# Not equal to

print(num1, "!=", num2, "is", num1 != num2)

# Greater than

print(num1, ">", num2, "is", num1 > num2)

# Less than

print(num1, "<", num2, "is", num1 < num2)

# Greater than or equal to

print(num1, ">=", num2, "is", num1 >= num2)

# Less than or equal to

print(num1, "<=", num2, "is", num1 <= num2)
```



```
========================= RESTART: D:/Program
Enter first number: 8
Enter second number: 5
Addition: 8 + 5 = 13
Subtraction: 8 - 5 = 3
Multiplication: 8 * 5 = 40
Division: 8 / 5 = 1.6
Floor Division: 8 // 5 = 1
Modulus: 8 % 5 = 3
Exponentiation: 8 ** 5 = 32768
```

EXPERIMENT – 4

AIM: Write a Program to demonstrate different logical operators in Python.

THEORY

Logical operators in Python are used to perform logical operations on Boolean values or expressions. These operators allow you to combine multiple conditions and evaluate them to a single Boolean value. Python supports three main logical operators: AND, OR, and NOT. Below is an explanation of each logical operator:

AND Operator (and): Returns True if both operands are True. Returns False otherwise.

Syntax: expression1 and expression2

OR Operator (or): Returns True if at least one of the operands is True. Returns False if both operands are False.

Syntax: expression1 or expression2

NOT Operator (not): Returns the opposite Boolean value of the operand. Converts True to False and False to True.

Syntax: not expression

CODE:

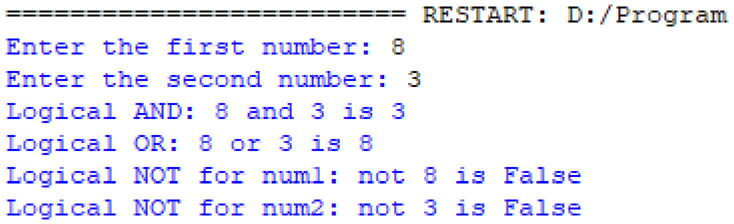
#OPERATIONS ON THE NUMBER

```
# Python Program to demonstrate different logical operators in Python


# Take numbers from keyboard

num1 = eval(input("Enter the first number: "))

num2 = eval(input("Enter the second number: "))


# Logical operators

# AND

print("Logical AND:", num1, "and", num2, "is", num1 and num2)

# OR

print("Logical OR:", num1, "or", num2, "is", num1 or num2)
```

NOT

print("Logical NOT for num1:", "not", num1, "is", not num1)

print("Logical NOT for num2:", "not", num2, "is", not num2)



```
========================= RESTART: D:/Program
Enter the first number: 8
Enter the second number: 3
Logical AND: 8 and 3 is 3
Logical OR: 8 or 3 is 8
Logical NOT for num1: not 8 is False
Logical NOT for num2: not 3 is False
```

EXPERIMENT – 5

AIM: Write a Program to demonstrate different bitwise operators in Python.

THEORY

Bitwise operators in Python are used to perform operations on individual bits of integer operands. Python supports several bitwise operators, each working at the bit level. Below are the common bitwise operators in Python along with their explanations:

Bitwise AND (&): Sets each bit to 1 if both corresponding bits of the operands are 1.

Syntax: result = operand1 & operand2

Bitwise OR (|): Sets each bit to 1 if at least one corresponding bit of the operands is 1.

Syntax: result = operand1 | operand2

Bitwise XOR (^): Sets each bit to 1 if only one of the corresponding bits of the operands is 1.

Syntax: result = operand1 ^ operand2

Bitwise NOT (~): Inverts all the bits of the operand, turning 0s into 1s and 1s into 0s.

Syntax: result = ~operand

Left Shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

Syntax: result = operand << n

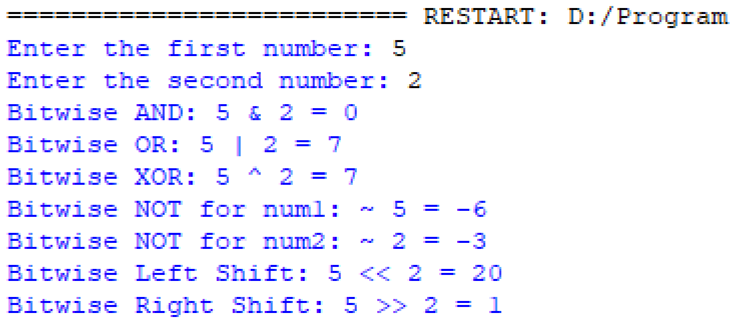
Right Shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Syntax: result = operand >> n

CODE:

```
# Python Program to demonstrate different bitwise operators in Python

# Take input from keyboard

num1 = eval(input("Enter the first number: "))

num2 = eval(input("Enter the second number: "))


#Bitwise Operators

# Bitwise AND

bitwise_and = num1 & num2
```

```
print("Bitwise AND:", num1, "&", num2, "=", bitwise_and)
# Bitwise OR
bitwise_or = num1 | num2
print("Bitwise OR:", num1, "|", num2, "=", bitwise_or)
# Bitwise XOR
bitwise_xor = num1 ^ num2
print("Bitwise XOR:", num1, "^", num2, "=", bitwise_xor)
# Bitwise NOT
bitwise_not_num1 = ~num1
bitwise_not_num2 = ~num2
print("Bitwise NOT for num1:", "~", num1, "=", bitwise_not_num1)
print("Bitwise NOT for num2:", "~", num2, "=", bitwise_not_num2)

# Bitwise Left Shift
left_shift = num1 << num2
print("Bitwise Left Shift:", num1, "<<", num2, "=", left_shift)

# Bitwise Right Shift
right_shift = num1 >> num2
print("Bitwise Right Shift:", num1, ">>", num2, "=", right_shift)
```



```
========================= RESTART: D:/Program
Enter the first number: 5
Enter the second number: 2
Bitwise AND: 5 & 2 = 0
Bitwise OR: 5 | 2 = 7
Bitwise XOR: 5 ^ 2 = 7
Bitwise NOT for num1: ~ 5 = -6
Bitwise NOT for num2: ~ 2 = -3
Bitwise Left Shift: 5 << 2 = 20
Bitwise Right Shift: 5 >> 2 = 1
```

EXPERIMENT – 6

AIM: Write a Program to create, concatenate, replicate and print a string and accessing the substring from given string in Python.

THEORY

String manipulation in Python involves various operations such as creating, concatenating, replicating, printing strings, and accessing substrings. These operations are fundamental for text processing and data manipulation tasks.

Creating a String: To create a string in Python, you simply assign a sequence of characters enclosed within single (' ') or double (" ") quotes to a variable.

Concatenating Strings: String concatenation involves combining multiple strings into a single string. This is achieved using the + operator.

Replicating Strings: String replication involves repeating a string a specified number of times. This is achieved by using the * operator.

Printing Strings: Printing strings in Python is straightforward using the print() function.

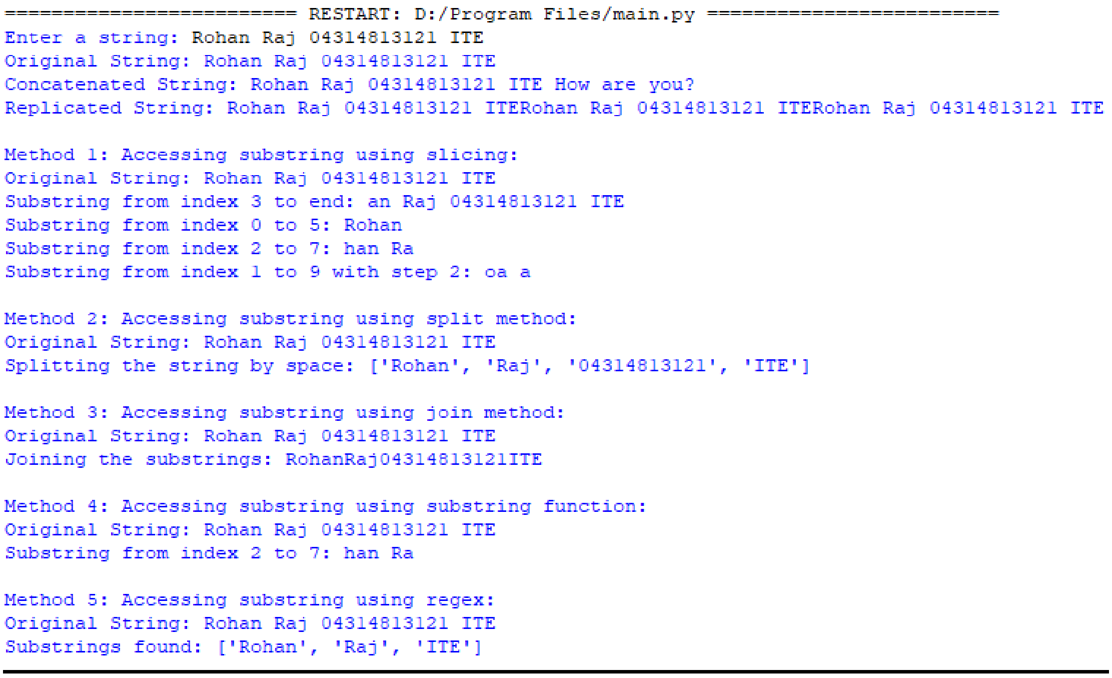
Accessing Substrings: Substrings are portions of a string. In Python, substrings are accessed using slicing. Slicing involves specifying the start and end indices within square brackets following the string variable. Python indexing starts from 0, and negative indices can be used to access characters from the end of the string. If the start index is omitted, Python assumes 0, and if the end index is omitted, Python assumes the end of the string.

CODE

```
# Python Program to demonstrate different ways to access substrings from a given string


# Take input from keyboard
input_string = input("Enter a string: ")


# Print the string
print("Original String:", input_string)
# Concatenate strings
concatenated_string = input_string + " How are you?"
print("Concatenated String:", concatenated_string)
```

```
# Replicate string
replicated_string = input_string * 3
print("Replicated String:", replicated_string)

# Method 1: Access substring using slicing
print("\nMethod 1: Accessing substring using slicing:")
print("Original String:", input_string)
print("Substring from index 3 to end:", input_string[3:])
print("Substring from index 0 to 5:", input_string[:6])
print("Substring from index 2 to 7:", input_string[2:8])
print("Substring from index 1 to 9 with step 2:", input_string[1:10:2])

# Method 2: Access substring using split method
print("\nMethod 2: Accessing substring using split method:")
split_string = input_string.split(" ")
print("Original String:", input_string)
print("Splitting the string by space:", split_string)

# Method 3: Access substring using join method
print("\nMethod 3: Accessing substring using join method:")
substring = "".join(split_string)
print("Original String:", input_string)
print("Joining the substrings:", substring)

# Method 4: Access substring using substring function
print("\nMethod 4: Accessing substring using substring function:")
print("Original String:", input_string)
print("Substring from index 2 to 7:", input_string[2:8])
```

```
# Method 5: Access substring using regex
import re
print("\nMethod 5: Accessing substring using regex:")
print("Original String:", input_string)
pattern = re.compile(r'([a-zA-Z]+)')
substrings = re.findall(pattern, input_string)
print("Substrings found:", substrings)
```



```
========================= RESTART: D:/Program Files/main.py =========================
Enter a string: Rohan Raj 04314813121 ITE
Original String: Rohan Raj 04314813121 ITE
Concatenated String: Rohan Raj 04314813121 ITE How are you?
Replicated String: Rohan Raj 04314813121 ITERohan Raj 04314813121 ITERohan Raj 04314813121 ITE

Method 1: Accessing substring using slicing:
Original String: Rohan Raj 04314813121 ITE
Substring from index 3 to end: an Raj 04314813121 ITE
Substring from index 0 to 5: Rohan
Substring from index 2 to 7: han Ra
Substring from index 1 to 9 with step 2: oa a

Method 2: Accessing substring using split method:
Original String: Rohan Raj 04314813121 ITE
Splitting the string by space: ['Rohan', 'Raj', '04314813121', 'ITE']

Method 3: Accessing substring using join method:
Original String: Rohan Raj 04314813121 ITE
Joining the substrings: RohanRaj04314813121ITE

Method 4: Accessing substring using substring function:
Original String: Rohan Raj 04314813121 ITE
Substring from index 2 to 7: han Ra

Method 5: Accessing substring using regex:
Original String: Rohan Raj 04314813121 ITE
Substrings found: ['Rohan', 'Raj', 'ITE']
```

EXPERIMENT – 7

AIM: Write a Program to print current date and time in following format "sun name 02:26:26 isp 2023" in Python.

THEORY

Python's datetime module provides classes for manipulating dates and times. The datetime class in this module is used to represent date and time objects in Python. To print the current date and time in a specific format, we typically follow these steps:

Import the datetime module: This module provides classes for working with dates and times in Python.

Get the current date and time: Use the **datetime.now()** method to obtain the current date and time as a datetime object.

Format the date and time: Use the **strftime()** method to format the datetime object into a string representation according to the desired format. This method allows you to specify various format codes to represent different parts of the date and time, such as year, month, day, hour, minute, second, and day of the week.

Print the formatted date and time: Finally, print the formatted date and time string.

In the given problem statement, the format specified is "sun name 02:26:26 isp 2023", where:

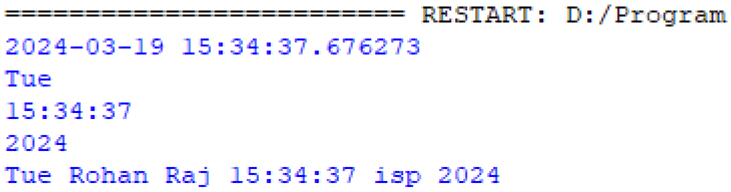
"sun" represents the abbreviated name of the day of the week.
"02:26:26" represents the current time in 24-hour format.
"isp" is a string.
"2023" represents the current year.

By using the datetime.now() method to get the current date and time, and the strftime() method to format it according to the specified format, we can achieve the desired output.

CODE

```
# Python Program to print current date and time in the specified format

import datetime



# Get the current date and time

current_datetime = datetime.datetime.now()

print(current_datetime)
```

```
# Extract the day name (abbreviated)

day_name = current_datetime.strftime("%a")

print(day_name)

# Extract the time in HH:MM:SS format

time_str = current_datetime.strftime("%H:%M:%S")

print(time_str)

# Extract the year

year = current_datetime.year

print(year)

# Print the formatted date and time

print(day_name, "Rohan Raj", time_str, "isp", year)
```



```
========================= RESTART: D:/Program
2024-03-19 15:34:37.676273
Tue
15:34:37
2024
Tue Rohan Raj 15:34:37 isp 2024
```

EXPERIMENT – 8

AIM: Write a Program to find largest of three number in Python.

THEORY

In Python, you can find the largest of three numbers using various approaches. The program typically involves comparing the three numbers and determining which one is the largest. Here's a breakdown of the theory behind writing a program to find the largest of three numbers in Python:

Comparison Approach: One common approach is to use if-else statements to compare the three numbers. You compare the first number with the second and the third number, then the second number with the first and the third number, and finally the third number with the first and the second number. Based on these comparisons, you determine which number is the largest among the three.

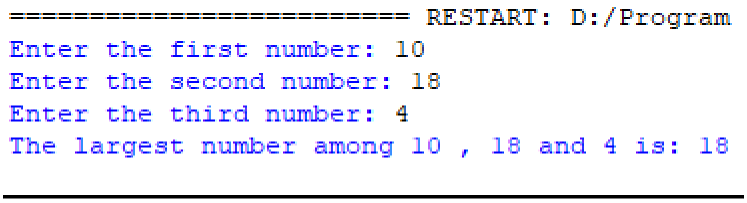
Ternary Operator Approach: Another approach is to use the ternary operator (conditional expression).This approach involves writing a single line of code using the ternary operator to compare the three numbers and return the largest one.

max() Function Approach: Python provides the max() function, which can be used to find the largest number among a sequence of numbers. You can pass the three numbers as arguments to the max() function, and it will return the largest of the three.

CODE

#USING IF-ELSE

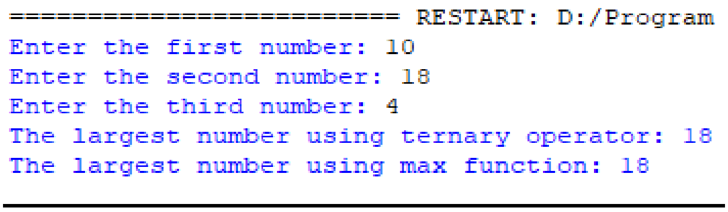
```
# Python Program to find the largest of three numbers using if-else statements

# Take input from keyboard

num1 = eval(input("Enter the first number: "))

num2 = eval(input("Enter the second number: "))

num3 = eval(input("Enter the third number: "))

# Find the largest number using if-else statements

if num1 >= num2 and num1 >= num3:

    largest = num1

elif num2 >= num1 and num2 >= num3:

    largest = num2

else:
    largest = num3

# Print the result

print("The largest number among", num1, ",", num2, "and", num3, "is:", largest)
```



```
========================== RESTART: D:/Program
Enter the first number: 10
Enter the second number: 18
Enter the third number: 4
The largest number among 10 , 18 and 4 is: 18
```

#USING TERNERY OPERATOR AND max() FUNCTION

```
# Python Program to find the largest of three numbers using ternary operator and max function

# Take input from keyboard

num1 = eval(input("Enter the first number: "))

num2 = eval(input("Enter the second number: "))

num3 = eval(input("Enter the third number: "))

# Find the largest number using ternary operator

largest_ternary = num1 if (num1 >= num2 and num1 >= num3) else (num2 if num2 >= num3 else num3)

# Find the largest number using max function

largest_max = max(num1, num2, num3)

# Print the results

print("The largest number using ternary operator:", largest_ternary)

print("The largest number using max function:", largest_max)
```



```
========================== RESTART: D:/Program
Enter the first number: 10
Enter the second number: 18
Enter the third number: 4
The largest number using ternary operator: 18
The largest number using max function: 18
```

EXPERIMENT – 9

AIM: Write a program to demonstrate working on list in Python.

THEORY

List are another fundamental data structure in Python, similar to tuple but with some key differences. List are mutable, meaning their elements can be changed after the tuple is created. Here's the theory behind working with lists in Python:

Creating List: Tuples are created by enclosing comma-separated values within parentheses []. Elements in a list can be of any data type, and they don't need to be of the same type.

Accessing Elements: Indexing starts from 0 for the first element and goes up to len(list) - 1. Negative indexing is also supported, where -1 represents the last element, -2 represents the second last element, and so on.

Mutable Nature: Lists are mutable, which means once created, their elements can be changed, added, or removed. However, you can create a new list by concatenating or slicing existing lists.

Iterating Over Lists: Lists can be iterated using loops such as for loop or while loop. This allows you to access each element of the list sequentially and perform operations on them.

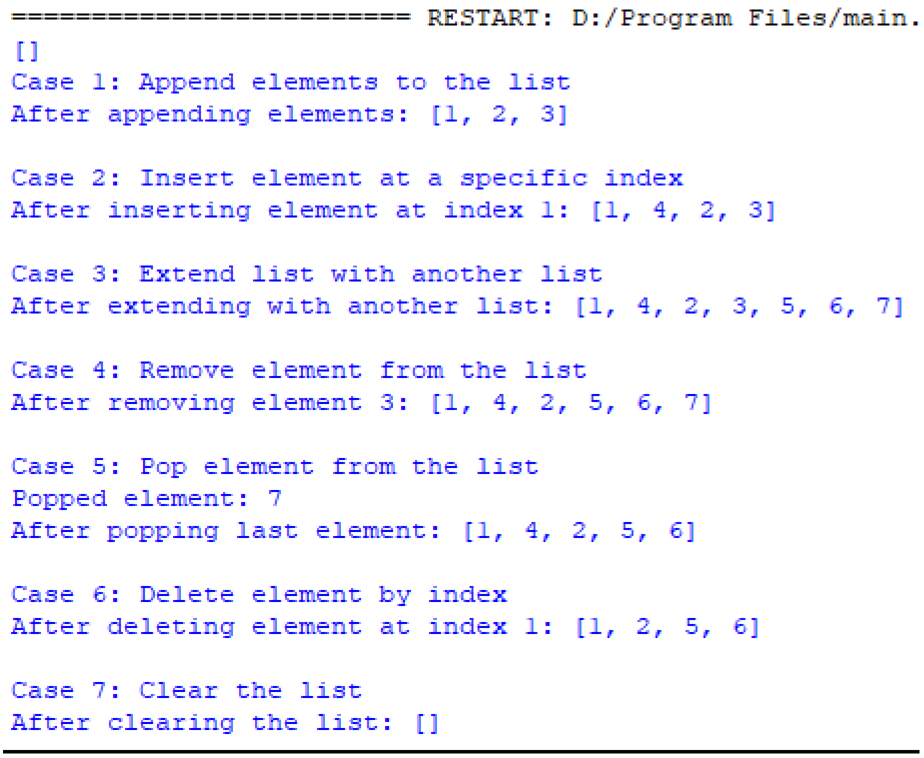
Using Lists as Keys in Dictionaries: Lists can be used as keys in dictionaries because they are hashable (if all their elements are hashable). This allows you to create dictionaries with compound keys using lists.

Other Operations: Other operations on list include finding the length of a list (len()), sorting a list (sorted()), checking membership (in and not in), and counting occurrences of an element (count()).

CODE

```
# Python Program to demonstrate creating, appending, and removing elements from a list

# Create an empty list

my_list = []

print(my_list)


# Case 1: Append elements to the list

print("Case 1: Append elements to the list")

my_list.append(1)
my_list.append(2)
my_list.append(3)
print("After appending elements:", my_list)

# Case 2: Insert element at a specific index
print("\nCase 2: Insert element at a specific index")
my_list.insert(1, 4)
print("After inserting element at index 1:", my_list)

# Case 3: Extend list with another list
print("\nCase 3: Extend list with another list")
another_list = [5, 6, 7]
my_list.extend(another_list)
print("After extending with another list:", my_list)

# Case 4: Remove element from the list
print("\nCase 4: Remove element from the list")
my_list.remove(3)
print("After removing element 3:", my_list)

# Case 5: Pop element from the list
print("\nCase 5: Pop element from the list")
popped_element = my_list.pop()
print("Popped element:", popped_element)
print("After popping last element:", my_list)

# Case 6: Delete element by index
print("\nCase 6: Delete element by index")
del my_list[1]
print("After deleting element at index 1:", my_list)
```

```
# Case 7: Clear the list

print("\nCase 7: Clear the list")

my_list.clear()

print("After clearing the list:", my_list)
```



```
========================= RESTART: D:/Program Files/main.
[]
Case 1: Append elements to the list
After appending elements: [1, 2, 3]

Case 2: Insert element at a specific index
After inserting element at index 1: [1, 4, 2, 3]

Case 3: Extend list with another list
After extending with another list: [1, 4, 2, 3, 5, 6, 7]

Case 4: Remove element from the list
After removing element 3: [1, 4, 2, 5, 6, 7]

Case 5: Pop element from the list
Popped element: 7
After popping last element: [1, 4, 2, 5, 6]

Case 6: Delete element by index
After deleting element at index 1: [1, 2, 5, 6]

Case 7: Clear the list
After clearing the list: []
```

EXPERIMENT – 10

AIM: Write a program to demonstrate working on tuple in Python.

THEORY

Tuples are another fundamental data structure in Python, similar to lists but with some key differences. Tuples are immutable, meaning their elements cannot be changed after the tuple is created. Here's the theory behind working with tuples in Python:

Creating Tuples: Tuples are created by enclosing comma-separated values within parentheses ( ). Elements in a tuple can be of any data type, and they don't need to be of the same type.

Accessing Elements: Elements in a tuple are accessed using indexing, similar to lists. Indexing starts from 0 for the first element and goes up to len(tuple) - 1. Negative indexing is also supported, where -1 represents the last element, -2 represents the second last element, and so on.

Tuple Packing and Unpacking: Tuple packing is the process of packing multiple values into a tuple. Tuple unpacking is the process of extracting values from a tuple into separate variables.

Immutable Nature: Tuples are immutable, which means once created, their elements cannot be changed, added, or removed. However, you can create a new tuple by concatenating or slicing existing tuples.

Iterating Over Tuples: Tuples can be iterated using loops such as for loop or while loop. This allows you to access each element of the tuple sequentially and perform operations on them.

Using Tuples as Keys in Dictionaries: Tuples can be used as keys in dictionaries because they are hashable (if all their elements are hashable). This allows you to create dictionaries with compound keys using tuples.

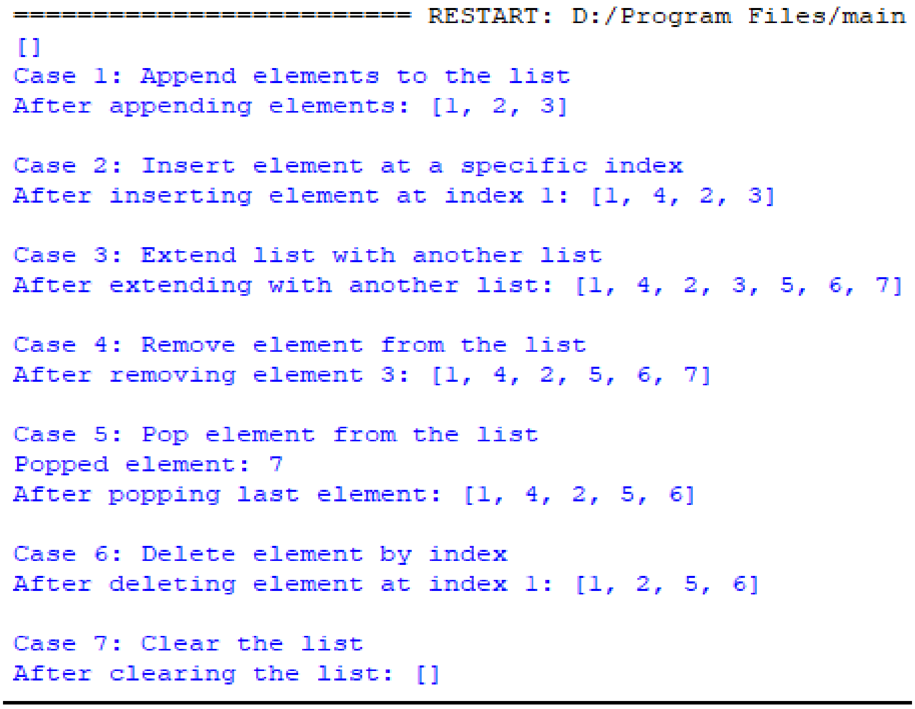
Other Operations: Other operations on tuples include finding the length of a tuple (len()), sorting a tuple (sorted()), checking membership (in and not in), and counting occurrences of an element (count()).

CODE

```
# Python Program to demonstrate working with tuples

# Create an empty tuple

my_tuple = ()

print(my_tuple)
```

```
# Case 1: Append elements to the tuple
print("Case 1: Append elements to the tuple")
# Tuples are immutable, so you cannot append elements directly to a tuple.
# However, you can concatenate two tuples to form a new tuple.
my_tuple += (1,)
my_tuple += (2,)
my_tuple += (3,)
print("After appending elements:", my_tuple)

# Case 2: Access elements of a tuple
print("\nCase 2: Accessing elements of a tuple:")
print("First element:", my_tuple[0])
print("Last element:", my_tuple[-1])
print("Slice of tuple:", my_tuple[1:])

# Case 3: Iterate over a tuple
print("\nCase 3: Iterating over a tuple:")
for item in my_tuple:
    print(item)

# Case 4: Check if an element exists in a tuple
print("\nCase 4: Check if an element exists in a tuple:")
print("Is 3 in tuple?", 3 in my_tuple)
print("Is 6 in tuple?", 6 in my_tuple)
# Case 5: Get the length of a tuple
print("\nCase 5: Length of tuple:", len(my_tuple))

# Case 6: Concatenate tuples
print("\nCase 6: Concatenate tuples")
another_tuple = (4, 5, 6)
```

```
concatenated_tuple = my_tuple + another_tuple
print("After concatenating tuples:", concatenated_tuple)

# Case 7: Repeat a tuple
print("\nCase 7: Repeat a tuple")
repeated_tuple = my_tuple * 2
print("After repeating tuple:", repeated_tuple)

# Case 8: Find index of an element in tuple
print("\nCase 8: Index of element 3 in tuple:", my_tuple.index(3))
# Case 9: Count occurrences of an element in tuple
print("Number of occurrences of element 2 in tuple:", my_tuple.count(2))
```



```
========================= RESTART: D:/Program Files/main
[]
Case 1: Append elements to the list
After appending elements: [1, 2, 3]

Case 2: Insert element at a specific index
After inserting element at index 1: [1, 4, 2, 3]

Case 3: Extend list with another list
After extending with another list: [1, 4, 2, 3, 5, 6, 7]

Case 4: Remove element from the list
After removing element 3: [1, 4, 2, 5, 6, 7]

Case 5: Pop element from the list
Popped element: 7
After popping last element: [1, 4, 2, 5, 6]

Case 6: Delete element by index
After deleting element at index 1: [1, 2, 5, 6]

Case 7: Clear the list
After clearing the list: []
```

EXPERIMENT – 11

AIM: Write a program to demonstrate working on dictionaries in Python.

THEORY

Dictionaries in Python are a powerful and versatile data structure used to store collections of key-value pairs. They are unordered, mutable, and indexed, allowing for efficient data retrieval based on keys rather than positions. Here's the theory behind working with dictionaries in Python:

Creating Dictionaries: Dictionaries are created by enclosing comma-separated key-value pairs within curly braces { }. Each key-value pair is separated by a colon : where the key is followed by the corresponding value. Keys in a dictionary must be unique and immutable (e.g., strings, numbers, tuples), while values can be of any data type.

Accessing and Modifying Elements: Elements in a dictionary are accessed using keys rather than indexes. You can access the value associated with a key by specifying the key within square brackets [ ]. To modify the value of a key, you can simply assign a new value to it.

Dictionary Methods: Python provides several built-in methods for working with dictionaries, such as keys(), values(), and items(). These methods allow you to retrieve the keys, values, or key-value pairs of a dictionary, respectively.

Adding and Removing Elements: You can add new key-value pairs to a dictionary using assignment. Similarly, you can remove key-value pairs using the del keyword or the pop() method.

Iterating Over Dictionaries: Dictionaries can be iterated using loops such as for loop. By default, iteration over a dictionary iterates over its keys. However, you can iterate over values or key-value pairs using appropriate methods.

Dictionary Comprehensions: Similar to list comprehensions, Python also supports dictionary comprehensions for creating dictionaries in a concise and efficient manner.

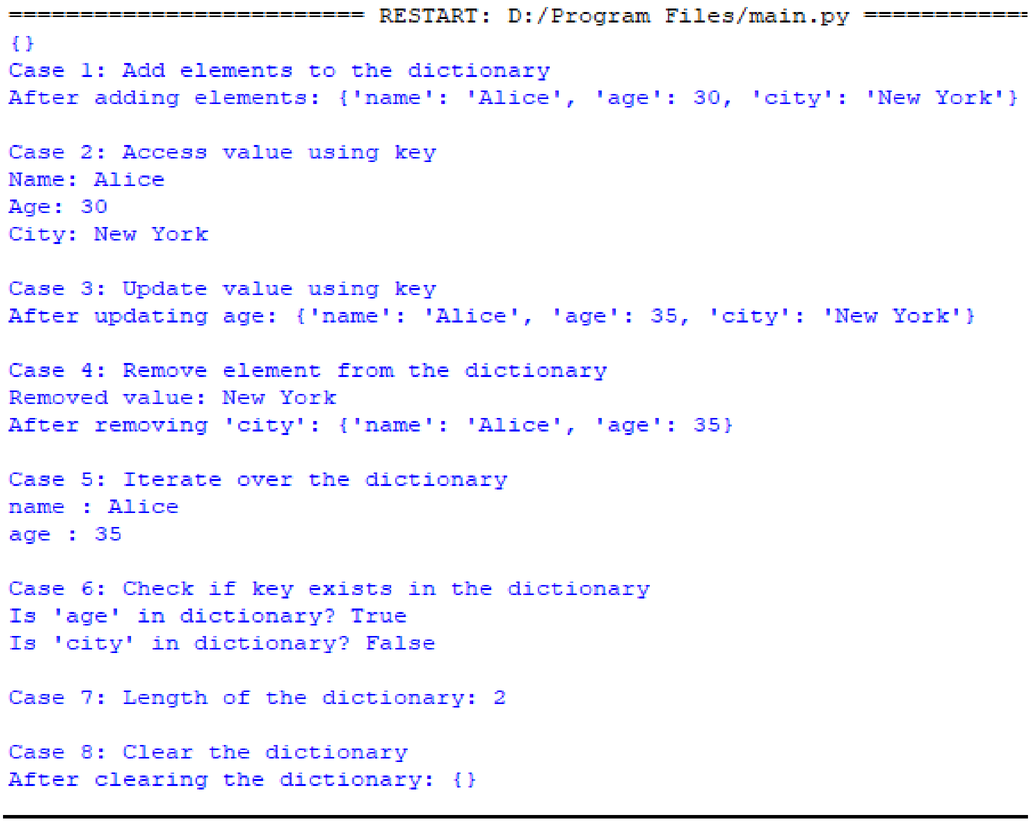
Dictionary Operations: Other operations on dictionaries include checking membership (in and not in), finding the length (len()), and copying dictionaries (copy() and dictionary unpacking).

CODE

```
# Python Program to demonstrate working with dictionaries

# Create an empty dictionary

my_dict = {}

print(my_dict)
```

```
# Case 1: Add elements to the dictionary
print("Case 1: Add elements to the dictionary")
my_dict['name'] = 'Alice'
my_dict['age'] = 30
my_dict['city'] = 'New York'
print("After adding elements:", my_dict)

# Case 2: Access value using key
print("\nCase 2: Access value using key")
print("Name:", my_dict['name'])
print("Age:", my_dict['age'])
print("City:", my_dict['city'])

# Case 3: Update value using key
print("\nCase 3: Update value using key")
my_dict['age'] = 35
print("After updating age:", my_dict)

# Case 4: Remove element from the dictionary
print("\nCase 4: Remove element from the dictionary")
removed_value = my_dict.pop('city')
print("Removed value:", removed_value)
print("After removing 'city':", my_dict)

# Case 5: Iterate over the dictionary
print("\nCase 5: Iterate over the dictionary")
for key, value in my_dict.items():
    print(key, ":", value)
```

```
# Case 6: Check if key exists in the dictionary
print("\nCase 6: Check if key exists in the dictionary")
print("Is 'age' in dictionary?", 'age' in my_dict)
print("Is 'city' in dictionary?", 'city' in my_dict)

# Case 7: Get the length of the dictionary
print("\nCase 7: Length of the dictionary:", len(my_dict))
# Case 8: Clear the dictionary
print("\nCase 8: Clear the dictionary")
my_dict.clear()
print("After clearing the dictionary:", my_dict)
```



```
========================= RESTART: D:/Program Files/main.py ============
{}
Case 1: Add elements to the dictionary
After adding elements: {'name': 'Alice', 'age': 30, 'city': 'New York'}

Case 2: Access value using key
Name: Alice
Age: 30
City: New York

Case 3: Update value using key
After updating age: {'name': 'Alice', 'age': 35, 'city': 'New York'}

Case 4: Remove element from the dictionary
Removed value: New York
After removing 'city': {'name': 'Alice', 'age': 35}

Case 5: Iterate over the dictionary
name : Alice
age : 35

Case 6: Check if key exists in the dictionary
Is 'age' in dictionary? True
Is 'city' in dictionary? False

Case 7: Length of the dictionary: 2

Case 8: Clear the dictionary
After clearing the dictionary: {}
```

EXPERIMENT – 12

AIM: Write a program to convert temperature from Celsius to Fahrenheit and vice-versa in Python.

THEORY

Converting temperature between Celsius and Fahrenheit is a common task in scientific and everyday applications. The conversion formulas are straightforward and involve simple arithmetic operations. Here's the theory behind converting temperature from Celsius to Fahrenheit and vice versa:

Celsius to Fahrenheit Conversion:

To convert temperature from Celsius to Fahrenheit, you use the following formula:
F = (C × 9/5) + 32
Where:

F is the temperature in Fahrenheit.

C is the temperature in Celsius.

Fahrenheit to Celsius Conversion:

Conversely, to convert temperature from Fahrenheit to Celsius, you use the following formula:
C = (F - 32) × 5/9
Where:

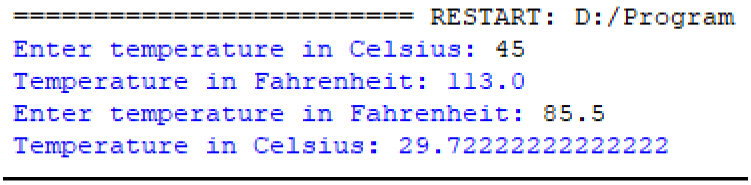
F is the temperature in Fahrenheit.

C is the temperature in Celsius.

CODE

```
# Python Program to convert temperature from Celsius to Fahrenheit and vice versa
def celsius_to_fahrenheit(celsius):
   fahrenheit = (celsius * 9/5) + 32
   return fahrenheit
def fahrenheit_to_celsius(fahrenheit):
   celsius = (fahrenheit - 32) * 5/9
   return celsius


# Convert Celsius to Fahrenheit
```

```
celsius_temperature = eval(input("Enter temperature in Celsius: "))
fahrenheit_result = celsius_to_fahrenheit(celsius_temperature)
print("Temperature in Fahrenheit:", fahrenheit_result)
# Convert Fahrenheit to Celsius
fahrenheit_temperature = eval(input("Enter temperature in Fahrenheit: "))
celsius_result = fahrenheit_to_celsius(fahrenheit_temperature)
print("Temperature in Celsius:", celsius_result)
```



```
========================= RESTART: D:/Program
Enter temperature in Celsius: 45
Temperature in Fahrenheit: 113.0
Enter temperature in Fahrenheit: 85.5
Temperature in Celsius: 29.72222222222222
```

EXPERIMENT – 13

AIM: Write a Python program that prints prime number less than 20.

THEORY

Prime Numbers:

Prime numbers are natural numbers greater than 1 that have no positive divisors other than 1 and themselves. For example, 2, 3, 5, 7, 11, 13, 17, and 19 are prime numbers.

is_prime() Function:

The is_prime() function checks if a given number is prime or not. It starts by checking if the number is less than or equal to 1, in which case it returns False since prime numbers are greater than 1. Then, it iterates through the numbers from 2 to the square root of the given number. If the given number is divisible by any number in this range (except 1 and itself), it returns False, indicating that the number is not prime. If the loop completes without finding any divisors, it returns True, indicating that the number is prime.

Printing Prime Numbers Less than 20:

The main part of the program iterates through the numbers from 2 to 19. For each number, it calls the is_prime() function to check if it's prime. If a number is prime, it prints the number.

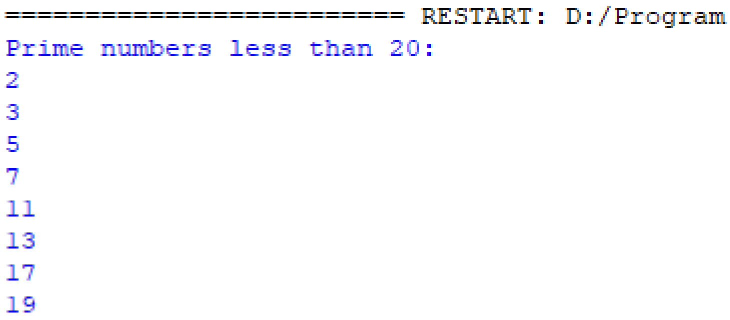
Output:

When you run the program, it prints the prime numbers less than 20: 2, 3, 5, 7, 11, 13, 17, and 19.

CODE

```
# Function to check if a number is prime

def is_prime(n):

    if n <= 1:

        return False

    for i in range(2, int(n**0.5) + 1):

        if n % i == 0:

            return False

    return True

# Print prime numbers less than 20
```

```
print("Prime numbers less than 20:")

for num in range(2, 20):

    if is_prime(num):

        print(num)
```



```
========================= RESTART: D:/Program
Prime numbers less than 20:
2
3
5
7
11
13
17
19
```

EXPERIMENT – 14

AIM: Write a Python program to find factorial of a number using recursion.

THEORY

Factorial of a non-negative integer is the multiplication of all positive integers smaller than or equal to n. For example factorial of 6 is 6*5*4*3*2*1 which is 720.

A factorial is represented by a number and a ” ! ” mark at the end. It is widely used in permutations and combinations to calculate the total possible outcomes. A French mathematician Christian Kramp firstly used the exclamation.

n! = n*(n-1)*(n-2)*…….*2*1

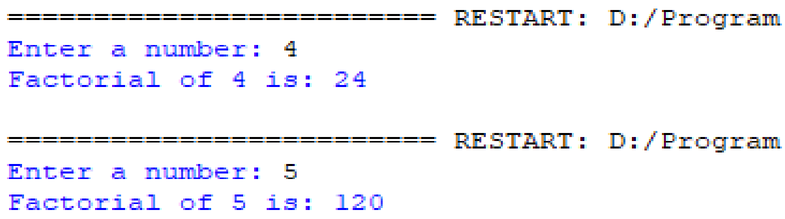
Factorial can be calculated using the following recursive formula.

$$n! = n * (n – 1)!$$

$$n! = 1 \text{ if } n = 0 \text{ or } n = 1$$

CODE

```
# Function to find factorial using recursion
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
num = int(input("Enter a number: "))
# Check if the number is negative
if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    print("Factorial of", num, "is:", factorial(num))
```



```
========================= RESTART: D:/Program
Enter a number: 4
Factorial of 4 is: 24

========================= RESTART: D:/Program
Enter a number: 5
Factorial of 5 is: 120
```

EXPERIMENT – 15

AIM: Write a Python program to accept length of three sides of triangle and print if the triangle is right angled or not.

THEORY

To determine whether a triangle is right-angled or not, we need to accept the lengths of its three sides. Once we have the lengths, we can apply the Pythagorean theorem to check if the triangle satisfies the condition for being right-angled.

Accept the lengths of the three sides of the triangle as input from the user.

Check if the triangle satisfies the Pythagorean theorem by comparing c^2 (the square of the longest side) with a^2+b^2 (the sum of the squares of the other two sides).

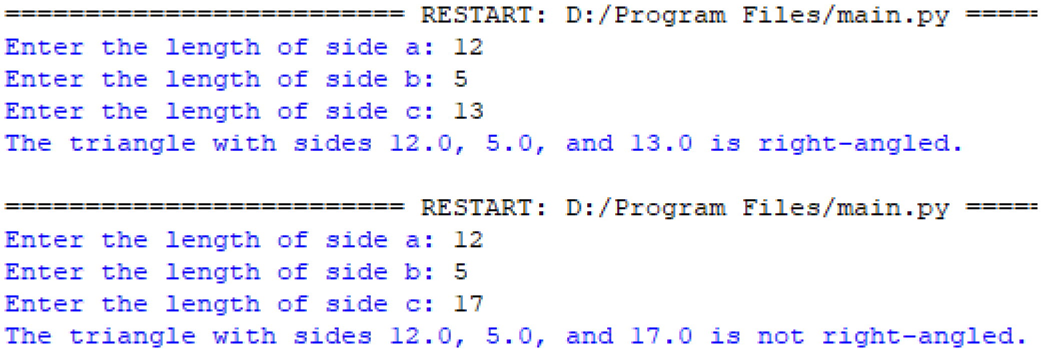
If the condition is true, the triangle is right-angled; otherwise, it is not.

Print the result indicating whether the triangle is right-angled or not.

CODE

```
# Function to check if triangle is right-angled
def is_right_angled(a, b, c):
    # Sort the sides in ascending order
    sides = [a, b, c]
    sides.sort()

    # Check if it's a right-angled triangle using the Pythagorean theorem
    return sides[0] ** 2 + sides[1] ** 2 == sides[2] ** 2

# Input the lengths of the sides of the triangle
a = float(input("Enter the length of side a: "))
b = float(input("Enter the length of side b: "))
c = float(input("Enter the length of side c: "))
# Check if it's a right-angled triangle and print the result
if is_right_angled(a, b, c):
    print("The triangle with sides {}, {}, and {} is right-angled.".format(a, b, c))
```

```
else:
    print("The triangle with sides {}, {}, and {} is not right-angled.".format(a, b, c))
```



```
========================== RESTART: D:/Program Files/main.py =====
Enter the length of side a: 12
Enter the length of side b: 5
Enter the length of side c: 13
The triangle with sides 12.0, 5.0, and 13.0 is right-angled.

========================== RESTART: D:/Program Files/main.py =====
Enter the length of side a: 12
Enter the length of side b: 5
Enter the length of side c: 17
The triangle with sides 12.0, 5.0, and 17.0 is not right-angled.
```

EXPERIMENT – 16

AIM: Write a Python program to define module to find the fibbonacci number and import the module to another program.

THEORY

In Python, modules are files containing Python code that define functions, variables, and classes. These modules can be imported into other Python programs to reuse the code and avoid redundancy.

Creating a Module to Find Fibonacci Numbers:

> To create a module to find Fibonacci numbers, we can define a function within a Python file that calculates Fibonacci numbers. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1. The Fibonacci sequence can be defined recursively or iteratively.

Importing the Module:

> To import the module into another Python program:
>
> Use the import statement followed by the name of the module (without the .py extension).
>
> Access the functions or variables defined in the module using dot notation (module_name.function_name).

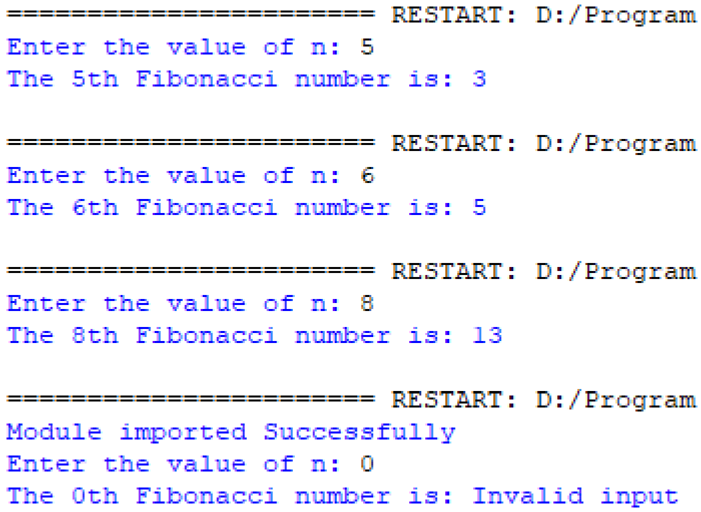
CODE

Fibonacci.py

```
def fibonacci(n):
    if n <= 0:
        return "Invalid input"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n):
            a, b = b, a + b
    return b
```

main.py

```
import Fibonacci
# Input from the user
n = int(input("Enter the value of n: "))
# Calculate and print the Fibonacci number
print("The {}th Fibonacci number is:".format(n), fibonacci.fibonacci(n))
```



```
======================== RESTART: D:/Program
Enter the value of n: 5
The 5th Fibonacci number is: 3

======================== RESTART: D:/Program
Enter the value of n: 6
The 6th Fibonacci number is: 5

======================== RESTART: D:/Program
Enter the value of n: 8
The 8th Fibonacci number is: 13

======================== RESTART: D:/Program
Module imported Successfully
Enter the value of n: 0
The 0th Fibonacci number is: Invalid input
```

EXPERIMENT – 17

AIM: Write a Python program to define module to and import the module to another program.

THEORY

In Python, modules are files containing Python code that define functions, variables, and classes. These modules can be imported into other Python programs to reuse the code and avoid redundancy.

Creating a Module:

To create a module, we can define functions, variables, or classes within a Python file. This file can then be imported into other Python programs as needed.

Importing the Module:

> To import the module into another Python program:
>
> Use the import statement followed by the name of the module (without the .py extension).
>
> Access the functions or variables defined in the module using dot notation (module_name.function_name).

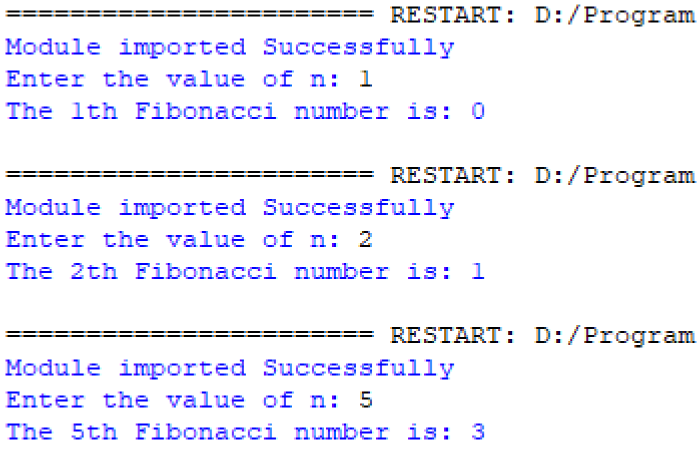
CODE

Fibonacci.py

```
def fibonacci(n):
    if n <= 0:
        return "Invalid input"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n):
            a, b = b, a + b
        return b
```

main.py

```
import Fibonacci


# Input from the user
n = int(input("Enter the value of n: "))
# Calculate and print the Fibonacci number
print("The {}th Fibonacci number is:".format(n), fibonacci.fibonacci(n))
```



```
======================= RESTART: D:/Program
Module imported Successfully
Enter the value of n: 1
The 1th Fibonacci number is: 0

======================= RESTART: D:/Program
Module imported Successfully
Enter the value of n: 2
The 2th Fibonacci number is: 1

======================= RESTART: D:/Program
Module imported Successfully
Enter the value of n: 5
The 5th Fibonacci number is: 3
```

EXPERIMENT – 18

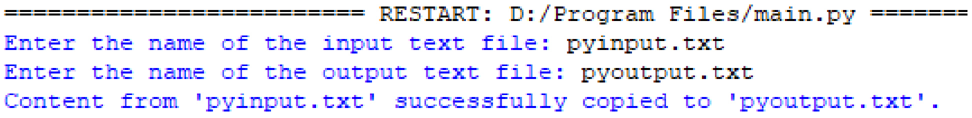
AIM: Write a script named copyfile.py. The script should prompt the user name of two text file. The content of first file should be input and written in second file.

THEORY

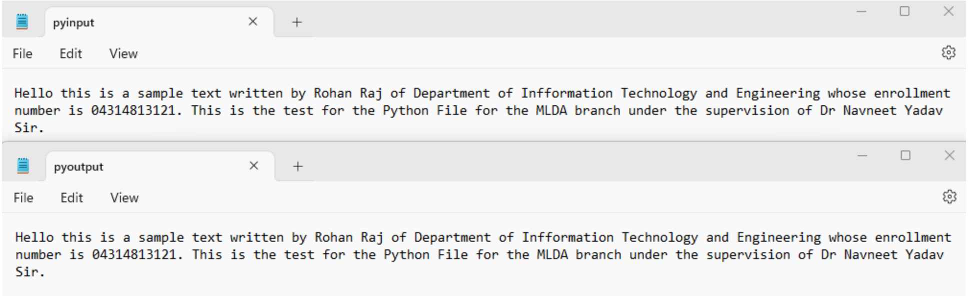
The Python script named **copyfile.py** serves the purpose of facilitating file manipulation tasks by enabling users to copy the contents of one text file into another. This functionality is achieved through an interactive approach, where the script prompts the user to input the names of the source and destination files. Upon receiving these inputs, the script proceeds to read the contents of the source file and subsequently writes them into the destination file. This process allows for efficient data transfer between files, empowering users to manage and organize their textual data effectively.

CODE

```
def main():
    # Prompt the user for the names of the two text files
    input_file = input("Enter the name of the input text file: ")
    output_file = input("Enter the name of the output text file: ")
    # Open the input file for reading
    try:
        with open(input_file, 'r') as file:
            content = file.read()
    except FileNotFoundError:
        print(f"Error: File '{input_file}' not found.")
        return

    # Open the output file for writing
    try:
        with open(output_file, 'w') as file:
            file.write(content)

    except PermissionError:
        print(f"Error: Permission denied to write to '{output_file}'.")
        return
    print(f"Content from '{input_file}' successfully copied to '{output_file}'.")


if __name__ == "__main__":
    main()
```



```
========================= RESTART: D:/Program Files/main.py =======
Enter the name of the input text file: pyinput.txt
Enter the name of the output text file: pyoutput.txt
Content from 'pyinput.txt' successfully copied to 'pyoutput.txt'.
```



Hello this is a sample text written by Rohan Raj of Department of Infformation Technology and Engineering whose enrollment number is 04314813121. This is the test for the Python File for the MLDA branch under the supervision of Dr Navneet Yadav Sir.

Hello this is a sample text written by Rohan Raj of Department of Infformation Technology and Engineering whose enrollment number is 04314813121. This is the test for the Python File for the MLDA branch under the supervision of Dr Navneet Yadav Sir.

EXPERIMENT – 19

AIM: Write a program that input a text file. The program should print all the unique word in alphabetical order.

THEORY

The objective is to create a Python program that reads a text file and outputs all the unique words contained within the file in alphabetical order. The program aims to efficiently process the text file, extract individual words, remove duplicates, and sort them alphabetically before printing.

Prompt the user to input the name of the text file.

Open the file for reading.

Read the contents of the file and split it into words.

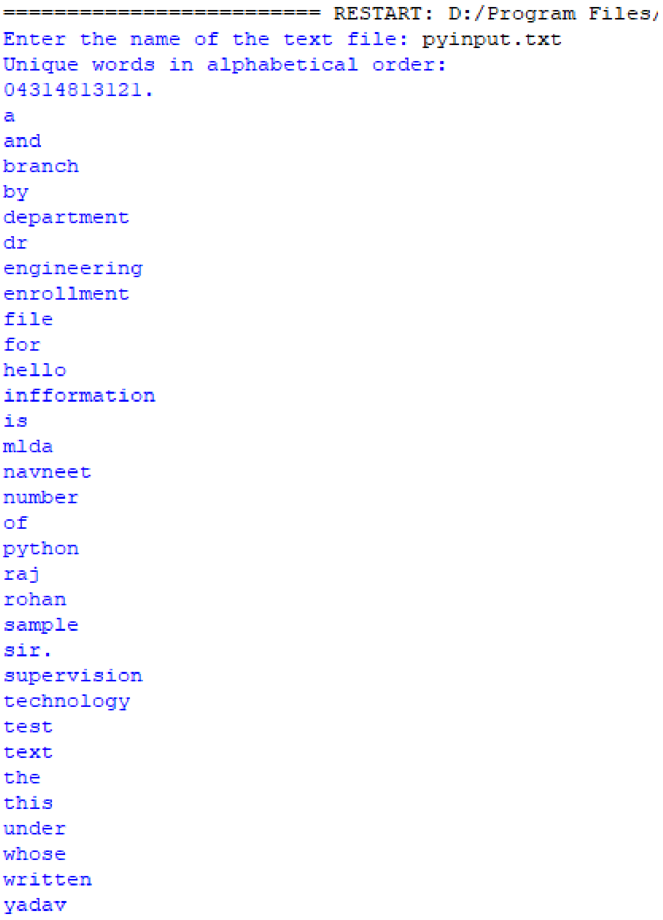
Convert the list of words into a set to obtain unique words.

Convert the set back into a sorted list.

Print the sorted list of unique words.

CODE

```
def main():


    # Prompt the user for the name of the text file

    file_name = input("Enter the name of the text file: ")


    # Open the file for reading

    try:

        with open(file_name, 'r') as file:

            # Read the content of the file

            content = file.read()

            # Split the content into words and convert to lowercase

            words = content.lower().split()

            # Get unique words using set and sort them alphabetically

            unique_words = sorted(set(words))
            # Print the unique words
            print("Unique words in alphabetical order:")
            for word in unique_words:
                print(word)

    except FileNotFoundError:
        print(f"Error: File '{file_name}' not found.")

    except PermissionError:
        print(f"Error: Permission denied to read '{file_name}'.")

if __name__ == "__main__":
    main()
```



```
========================= RESTART: D:/Program Files/
Enter the name of the text file: pyinput.txt
Unique words in alphabetical order:
04314813121.
a
and
branch
by
department
dr
engineering
enrollment
file
for
hello
infformation
is
mlda
navneet
number
of
python
raj
rohan
sample
sir.
supervision
technology
test
text
the
this
under
whose
written
yadav
```

EXPERIMENT – 20

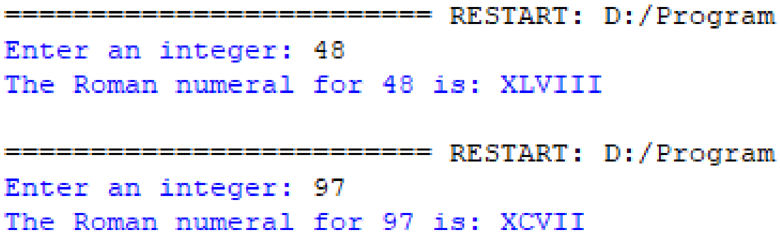
AIM: Write a python class to convert an integer into roman numeral.

THEORY

Converting an integer into a Roman numeral involves transforming a numerical value into its corresponding Roman numeral representation. This process is achieved by mapping specific integer values to their corresponding Roman numeral symbols and then combining them according to certain rules.

CODE

```
class IntegerToRoman:
    def __init__(self):
        self.int_to_roman_dict = {
            1: 'I', 4: 'IV', 5: 'V', 9: 'IX', 10: 'X', 40: 'XL', 50: 'L',
            90: 'XC', 100: 'C', 400: 'CD', 500: 'D', 900: 'CM', 1000: 'M'
        }
    def int_to_roman(self, num: int) -> str:
        if num <= 0:
            return "Invalid input. Please enter a positive integer."
        result = ''
        # Iterate through the dictionary in descending order of values
        for value in sorted(self.int_to_roman_dict.keys(), reverse=True):
            # Repeat adding the corresponding Roman numeral until the number is reduced to 0
            while num >= value:
                result += self.int_to_roman_dict[value]
                num -= value
        return result


# Test the class
if __name__ == "__main__":
    converter = IntegerToRoman()
    number = int(input("Enter an integer: "))
    roman_numeral = converter.int_to_roman(number)

    print(f"The Roman numeral for {number} is: {roman_numeral}")
```



```
========================= RESTART: D:/Program
Enter an integer: 48
The Roman numeral for 48 is: XLVIII

========================= RESTART: D:/Program
Enter an integer: 97
The Roman numeral for 97 is: XCVII
```

EXPERIMENT – 21

AIM: Write a python program to implement pow(x,n).

THEORY

The aim of the Python program is to implement the pow(x, n) function, which calculates the value of **x** raised to the power of **n**. This task can be accomplished using recursion or iteration, both of which involve repeatedly multiplying x by itself n times.

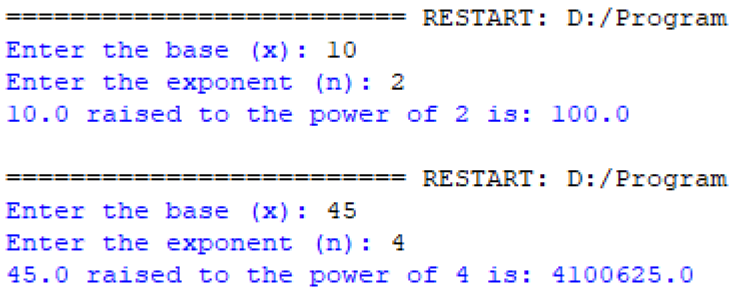
The Python function power(x, n) uses an optimized iterative algorithm called "**exponentiation by squaring**" to efficiently compute the value of x raised to the power of n. It handles base cases where n is 0 or negative, updates x for negative exponents, and iteratively computes the result by squaring x and halving n.

The function then returns the computed result. The script also includes a testing block for standalone execution, prompting the user for inputs and displaying the calculated result.

CODE

```
def power(x: float, n: int) -> float:
    if n == 0:
        return 1.0
    elif n < 0:
        x = 1 / x
        n = -n
    result = 1.0
    while n:
        if n % 2 == 1:
            result *= x
        x *= x
        n //= 2
    return result

# Test the function
if __name__ == "__main__":
    x = float(input("Enter the base (x): "))
    n = int(input("Enter the exponent (n): "))
```

```
result = power(x, n)

print(f"{x} raised to the power of {n} is: {result}")
```



```
========================== RESTART: D:/Program
Enter the base (x): 10
Enter the exponent (n): 2
10.0 raised to the power of 2 is: 100.0

========================== RESTART: D:/Program
Enter the base (x): 45
Enter the exponent (n): 4
45.0 raised to the power of 4 is: 4100625.0
```

EXPERIMENT – 21

AIM: Write a python class to reverse string word by word.

THEORY

Reversing a string involves flipping its characters so that the last character becomes the first, the second-to-last character becomes the second, and so on. In Python, there are multiple approaches to reverse a string, including using slicing, loops, or built-in functions.

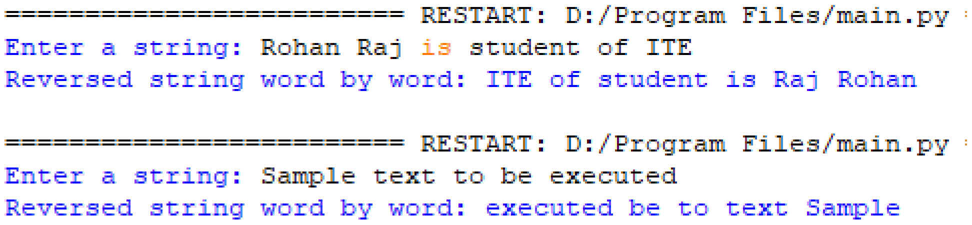
It can be implemented by:

Using Slicing
Using a Loop
Using the reversed() Function
Using the join() Method with a Reversed List

CODE

```
class StringReverser:
    def reverse_words(self, s: str) -> str:

        # Split the string into words
        words = s.split()

        # Reverse the order of words
        reversed_words = words[::-1]

        # Join the reversed words into a single string
        reversed_string = ' '.join(reversed_words)
        return reversed_string

# Test the class
if __name__ == "__main__":
    reverser = StringReverser()
#Inputh the string
    input_string = input("Enter a string: ")
    reversed_string = reverser.reverse_words(input_string)
```

```
print("Reversed string word by word:", reversed_string)
```



```
========================= RESTART: D:/Program Files/main.py
Enter a string: Rohan Raj is student of ITE
Reversed string word by word: ITE of student is Raj Rohan

========================= RESTART: D:/Program Files/main.py
Enter a string: Sample text to be executed
Reversed string word by word: executed be to text Sample
```

EXPERIMENT – 23

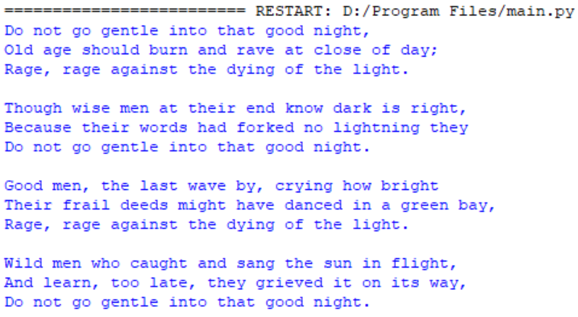
AIM: Write a function in python to read the content from a text file "poem.txt" line by line and display the same on screen.

THEORY

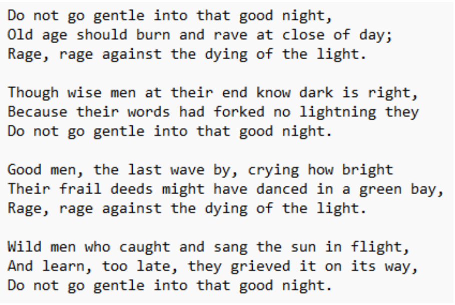
Use Python's open() function to open the text file "poem.txt" in read mode.
Iterate through each line in the file using a for loop.
Print each line to the screen using print() function.
Implement a try-except block to handle the FileNotFoundError in case the file does not exist. If the file is not found, print an error message indicating the issue.

CODE

```
def display_poem_from_file():
    try:
        # Open the file "poem.txt" in read mode
        with open("poem.txt", "r") as file:
            # Read each line from the file and display it on the screen
            for line in file:
                print(line, end='')
    except FileNotFoundError:
        print("Error: File 'poem.txt' not found.")


# Test the function
display_poem_from_file()
```



```
========================= RESTART: D:/Program Files/main.py
Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.

Though wise men at their end know dark is right,
Because their words had forked no lightning they
Do not go gentle into that good night.

Good men, the last wave by, crying how bright
Their frail deeds might have danced in a green bay,
Rage, rage against the dying of the light.

Wild men who caught and sang the sun in flight,
And learn, too late, they grieved it on its way,
Do not go gentle into that good night.
```



```
Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.

Though wise men at their end know dark is right,
Because their words had forked no lightning they
Do not go gentle into that good night.

Good men, the last wave by, crying how bright
Their frail deeds might have danced in a green bay,
Rage, rage against the dying of the light.

Wild men who caught and sang the sun in flight,
And learn, too late, they grieved it on its way,
Do not go gentle into that good night.
```

EXPERIMENT – 24

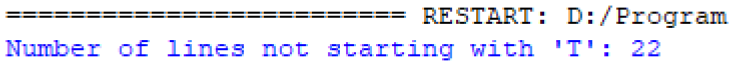
AIM: Write a function in python to count the number of lines from a text file "poem.txt" which is not starting with an alphabet "T".

THEORY

Use Python's open() function to open the text file "poem.txt" in read mode.
Iterate through each line in the file using a for loop and check if starts with T.
Print each line to the screen using print() function.
Implement a try-except block to handle the FileNotFoundError in case the file does not exist. If the file is not found, print an error message indicating the issue.

CODE

```
def count_lines_not_starting_with_T():
    try:
        # Open the file "poem.txt" in read mode
        with open("poem.txt", "r") as file:
            # Initialize a counter for lines not starting with "T"
            count = 0
            # Iterate through each line in the file
            for line in file:
                # Check if the line does not start with "T"
                if not line.strip().startswith('T'):
                    # Increment the counter
                    count += 1
            return count
    except FileNotFoundError:
        print("Error: File 'poem.txt' not found.")
# Test the function
print("Number of lines not starting with 'T':", count_lines_not_starting_with_T())
```



```
========================= RESTART: D:/Program
Number of lines not starting with 'T': 22
```

EXPERIMENT – 25

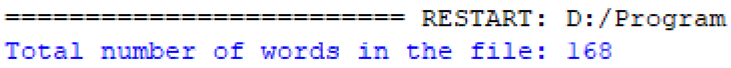
AIM: Write a function in Python to count and display the total number of words in a text file, poem.txt

THEORY

Use Python's open() function to open the text file "poem.txt" in read mode.
Iterate through each line in the file using a for loop and count the words.
Print each line to the screen using print() function.
Implement a try-except block to handle the FileNotFoundError in case the file does not exist. If the file is not found, print an error message indicating the issue.

CODE

```
def count_words_in_file(file_path):
    try:
        # Open the file specified by file_path in read mode
        with open(file_path, 'r') as file:
            # Read the entire content of the file
            content = file.read()
            # Split the content into words using whitespace as delimiter
            words = content.split()
            # Count the number of words
            word_count = len(words)
            # Display the total number of words
            print("Total number of words in the file:", word_count)
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
# Test the function with 'poem.txt'
count_words_in_file('poem.txt')
```



```
========================= RESTART: D:/Program
Total number of words in the file: 168
```

EXPERIMENT – 26

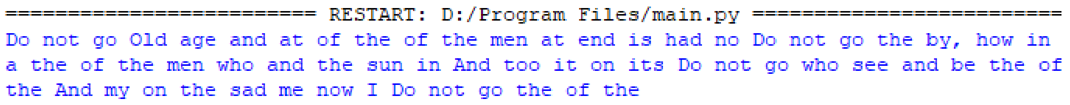
<u>AIM</u>: Write a function display_words() in python to read lines from a text file "poem.txt", and display those words, which are less than 4 characters.

<u>THEORY</u>

Use Python's open() function to open the text file "poem.txt" in read mode.
Iterate through each line in the file using a for loop and check if len(word) less than 4.
Print each line to the screen using print() function.
Implement a try-except block to handle the FileNotFoundError in case file not found

<u>CODE</u>

```
def display_words(file_path):
    try:
        # Open the file specified by file_path in read mode
        with open(file_path, 'r') as file:
            # Read each line from the file
            for line in file:
                # Split the line into words using whitespace as delimiter
                words = line.split()
                # Iterate through each word
                for word in words:
                    # Check if the length of the word is less than 4 characters
                    if len(word) < 4:
                        # Display the word
                        print(word, end=' ')
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
# Test the function with 'story.txt'
display_words('poem.txt')
```



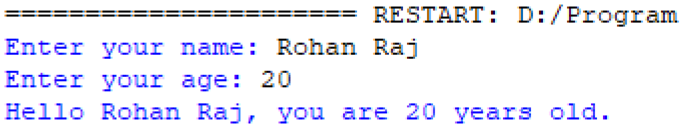
```
========================= RESTART: D:/Program Files/main.py =========================
Do not go Old age and at of the of the men at end is had no Do not go the by, how in
a the of the men who and the sun in And too it on its Do not go who see and be the of
the And my on the sad me now I Do not go the of the
```

EXPERIMENT – 27

AIM: Create a program that prompts the user for their name and age and prints a personalized message.

CODE

```
# Prompt the user for their name and age

name = input("Enter your name: ")

age = int(input("Enter your age: "))


# Print a personalized message

print("Hello {}, you are {} years old.".format(name, age))
```



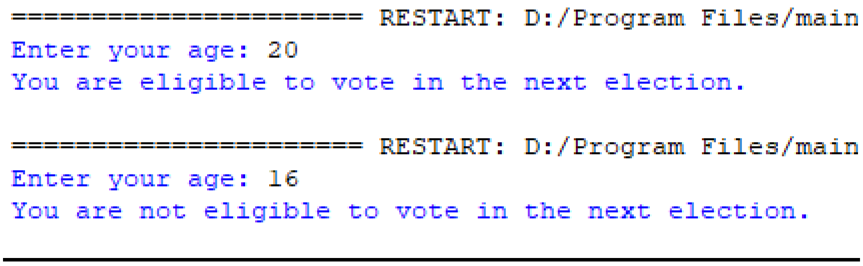
```
====================== RESTART: D:/Program
Enter your name: Rohan Raj
Enter your age: 20
Hello Rohan Raj, you are 20 years old.
```

EXPERIMENT – 28

AIM: Create a program that prompts the user for their age and tells them if they can vote in the next election.

CODE

```
# Prompt the user for their age
age = int(input("Enter your age: "))

# Check if the user can vote in the next election
if age >= 18:
    print("You are eligible to vote in the next election.")
else:
    print("You are not eligible to vote in the next election.")
```



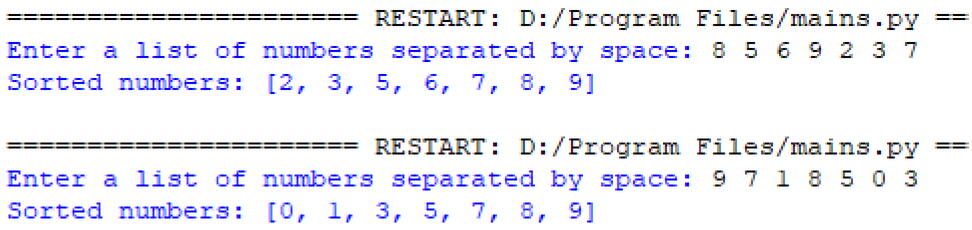
```
====================== RESTART: D:/Program Files/main
Enter your age: 20
You are eligible to vote in the next election.

====================== RESTART: D:/Program Files/main
Enter your age: 16
You are not eligible to vote in the next election.
```

EXPERIMENT – 29

AIM: Create a program that prompts the user for a list of numbers and then sorts them in ascending order.

CODE

```
# Prompt the user for a list of numbers
numbers = [int(x) for x in input("Enter a list of numbers separated by space: ").split()]


# Sort the numbers in ascending order
numbers.sort()


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



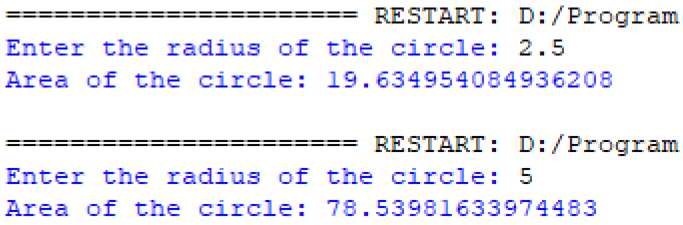
```
====================== RESTART: D:/Program Files/mains.py ==
Enter a list of numbers separated by space: 8 5 6 9 2 3 7
Sorted numbers: [2, 3, 5, 6, 7, 8, 9]

====================== RESTART: D:/Program Files/mains.py ==
Enter a list of numbers separated by space: 9 7 1 8 5 0 3
Sorted numbers: [0, 1, 3, 5, 7, 8, 9]
```

EXPERIMENT – 30

AIM: Create a program that defines a function to calculate the area of a circle based on the radius entered by the user.

CODE

```
import math

# Define a function to calculate the area of a circle
def calculate_area(radius):
    return math.pi * radius**2

# Prompt the user for the radius
radius = float(input("Enter the radius of the circle: "))

# Calculate and print the area
area = calculate_area(radius)
print("Area of the circle:", area)
```



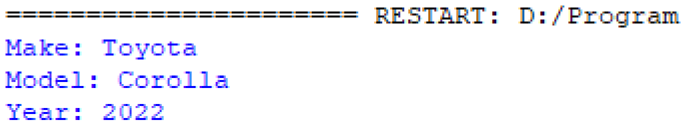
```
====================== RESTART: D:/Program
Enter the radius of the circle: 2.5
Area of the circle: 19.634954084936208

====================== RESTART: D:/Program
Enter the radius of the circle: 5
Area of the circle: 78.53981633974483
```

EXPERIMENT – 31

AIM: Create a program that defines a class to represent a car and then creates an object of that class with specific attributes.

CODE

```
# Define a class to represent a car

class Car:

    def __init__(self, make, model, year):

        self.make = make

        self.model = model

        self.year = year


# Create an object of the Car class with specific attributes

my_car = Car("Toyota", "Corolla", 2022)


# Access and print the attributes of the car object

print("Make:", my_car.make)

print("Model:", my_car.model)

print("Year:", my_car.year)
```



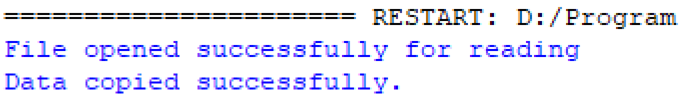
```
====================== RESTART: D:/Program
Make: Toyota
Model: Corolla
Year: 2022
```

EXPERIMENT – 32

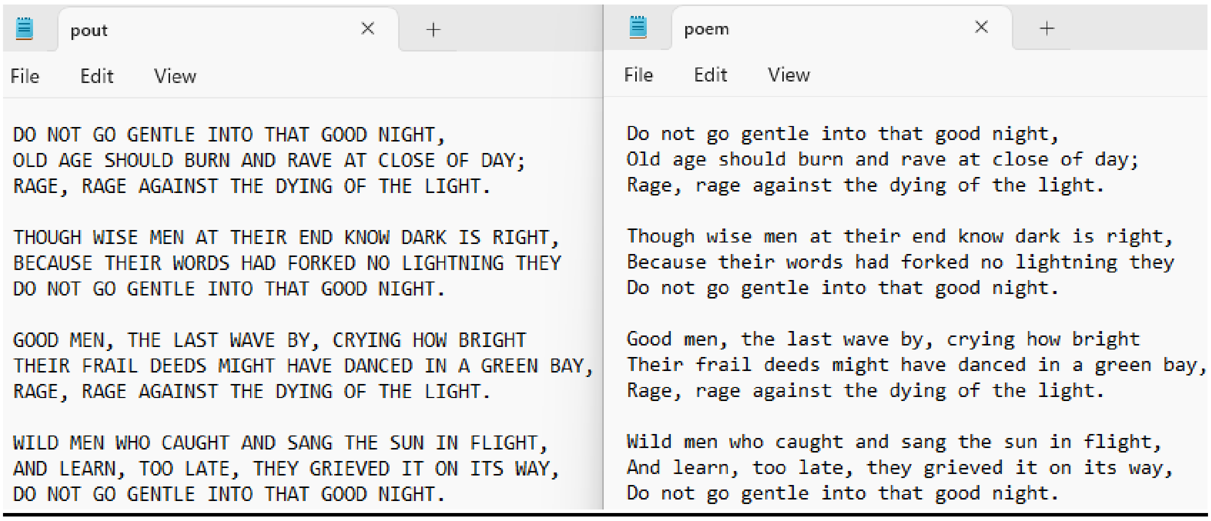
AIM: Create a program that reads data from a file and writes it to another file in a different format..

CODE

```
# Read data from a file
with open("poem", "r") as f:
    data = f.read()

# Write data to another file in a different format
with open("pout", "w") as f:
    f.write(data.upper())
```



```
====================== RESTART: D:/Program
File opened successfully for reading
Data copied successfully.
```



```
DO NOT GO GENTLE INTO THAT GOOD NIGHT,
OLD AGE SHOULD BURN AND RAVE AT CLOSE OF DAY;
RAGE, RAGE AGAINST THE DYING OF THE LIGHT.

THOUGH WISE MEN AT THEIR END KNOW DARK IS RIGHT,
BECAUSE THEIR WORDS HAD FORKED NO LIGHTNING THEY
DO NOT GO GENTLE INTO THAT GOOD NIGHT.

GOOD MEN, THE LAST WAVE BY, CRYING HOW BRIGHT
THEIR FRAIL DEEDS MIGHT HAVE DANCED IN A GREEN BAY,
RAGE, RAGE AGAINST THE DYING OF THE LIGHT.

WILD MEN WHO CAUGHT AND SANG THE SUN IN FLIGHT,
AND LEARN, TOO LATE, THEY GRIEVED IT ON ITS WAY,
DO NOT GO GENTLE INTO THAT GOOD NIGHT.
```

```
Do not go gentle into that good night,
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.

Though wise men at their end know dark is right,
Because their words had forked no lightning they
Do not go gentle into that good night.

Good men, the last wave by, crying how bright
Their frail deeds might have danced in a green bay,
Rage, rage against the dying of the light.

Wild men who caught and sang the sun in flight,
And learn, too late, they grieved it on its way,
Do not go gentle into that good night.
```

EXPERIMENT – 33

AIM: Create a program that uses regular expressions to find all instances of a specific pattern in a text file.

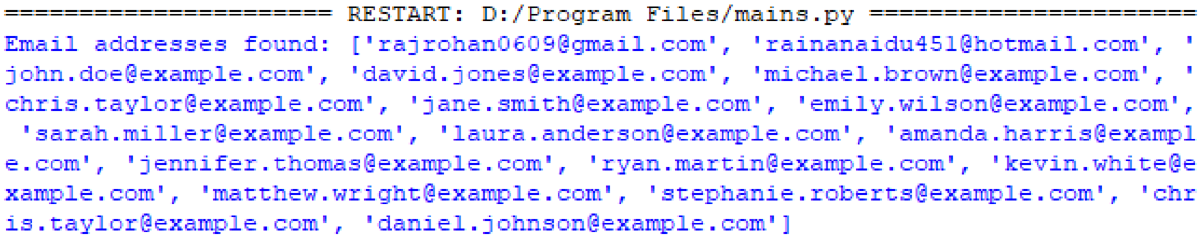
THEORY

Regular expressions, often abbreviated as regex or regexp, are powerful tools for pattern matching and text manipulation. They provide a concise and flexible means of searching, extracting, and replacing specific patterns of text within strings. Regular expressions are widely used in various programming languages, including Python, for tasks such as data validation, text processing, and parsing.

In Python, regular expressions are supported through the re module, which provides functions and methods for working with regular expressions. Regular expressions consist of sequences of characters and metacharacters that define a search pattern. Metacharacters such as ^, $, ., *, +, ?, \, |, [], () have special meanings and are used to specify the rules of the pattern.

CODE

```
import re


# Open the text file and read its contents
with open("mails.txt", "r") as f:
    text = f.read()
# Define the pattern to search for
pattern = r'\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\b'
# Find all instances of the pattern using regular expressions
emails = re.findall(pattern, text)
# Print the found email addresses
print("Email addresses found:", emails)
```



```
====================== RESTART: D:/Program Files/mains.py ======================
Email addresses found: ['rajrohan0609@gmail.com', 'rainanaidu451@hotmail.com', '
john.doe@example.com', 'david.jones@example.com', 'michael.brown@example.com', '
chris.taylor@example.com', 'jane.smith@example.com', 'emily.wilson@example.com',
 'sarah.miller@example.com', 'laura.anderson@example.com', 'amanda.harris@exampl
e.com', 'jennifer.thomas@example.com', 'ryan.martin@example.com', 'kevin.white@e
xample.com', 'matthew.wright@example.com', 'stephanie.roberts@example.com', 'chr
is.taylor@example.com', 'daniel.johnson@example.com']
```

EXPERIMENT – 34

AIM: Create a program that prompts the user for two numbers and then divides them, handling any exceptions that may arise.

THEORY

Exception handling is a programming construct used to handle errors or exceptional situations that may occur during the execution of a program. In Python, exceptions are raised when an error occurs at runtime, disrupting the normal flow of the program. Exception handling allows developers to anticipate and gracefully manage these errors, ensuring that the program does not crash unexpectedly.

When a Python program encounters an exception, it looks for an appropriate exception handler to process it. Exception handling is implemented using the try, except, else, and finally blocks. The try block contains the code that may raise an exception, while the except block handles the exception if it occurs.

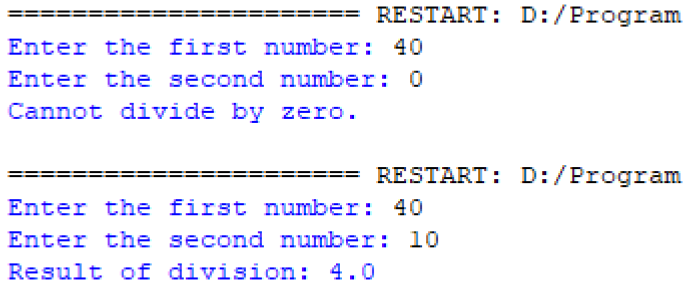
Multiple except blocks can be used to handle different types of exceptions. The else block is executed if no exception occurs in the try block, and the finally block is always executed, regardless of whether an exception occurred or not.

By using exception handling, developers can improve the robustness and reliability of their programs, making them more resistant to unexpected errors and ensuring a smoother user experience.

CODE

```
# Prompt the user for two numbers

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

    # Divide the numbers and handle ZeroDivisionError
    result = num1 / num2
    print("Result of division:", result)

except ValueError:
    print("Please enter valid numbers.")
```

```
except ZeroDivisionError:
    print("Cannot divide by zero.")
```



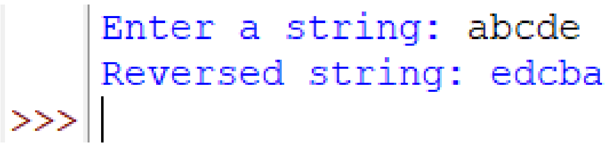
```
====================== RESTART: D:/Program
Enter the first number: 40
Enter the second number: 0
Cannot divide by zero.

====================== RESTART: D:/Program
Enter the first number: 40
Enter the second number: 10
Result of division: 4.0
```

EXPERIMENT – 35

AIM: Create a program that prompts the user for a string and then prints out the string reversed.

CODE

```
user_input = input("Enter a string: ")

reversed_string = user_input[::-1]

print("Reversed string:", reversed_string)
```



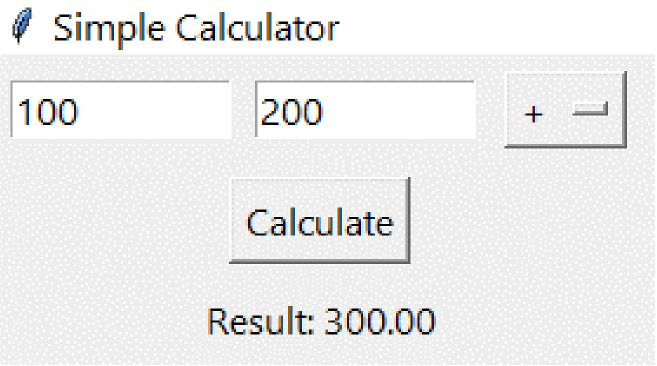
```
Enter a string: abcde
Reversed string: edcba
>>> 
```

EXPERIMENT – 36

AIM: Create a program that uses a graphical user interface (GUI) to allow the user to perform simple calculations.

CODE

```
import tkinter as tk

from tkinter import messagebox

def calculate():

    try:

        num1 = float(entry_num1.get())

        num2 = float(entry_num2.get())

        operator = operator_var.get()

        if operator == '+':

            result = num1 + num2

        elif operator == '-':

            result = num1 - num2

        elif operator == '*':

            result = num1 * num2

        elif operator == '/':

            if num2 == 0:

                raise ZeroDivisionErrors

        result = num1 / num2

        else:

            raise ValueError("Invalid operator")

        result_label.config(text="Result: {:.2f}".format(result))

    except ValueError:

        messagebox.showerror("Error", "Please enter valid numbers.")

    except ZeroDivisionError:
        messagebox.showerror("Error", "Cannot divide by zero.")

root = tk.Tk()

root.title("Simple Calculator")

entry_num1 = tk.Entry(root, width=10)

entry_num1.grid(row=0, column=0, padx=5, pady=5)

entry_num2 = tk.Entry(root, width=10)

entry_num2.grid(row=0, column=1, padx=5, pady=5)

operator_var = tk.StringVar(root)

operator_choices = ['+', '-', '*', '/']

operator_dropdown = tk.OptionMenu(root, operator_var, *operator_choices)

operator_dropdown.grid(row=0, column=2, padx=5, pady=5)

operator_var.set('+')

calculate_button = tk.Button(root, text="Calculate", command=calculate)

calculate_button.grid(row=1, column=0, columnspan=3, padx=5, pady=5)

result_label = tk.Label(root, text="Result: ")

result_label.grid(row=2, column=0, columnspan=3, padx=5, pady=5)

root.mainloop()
```



100

200

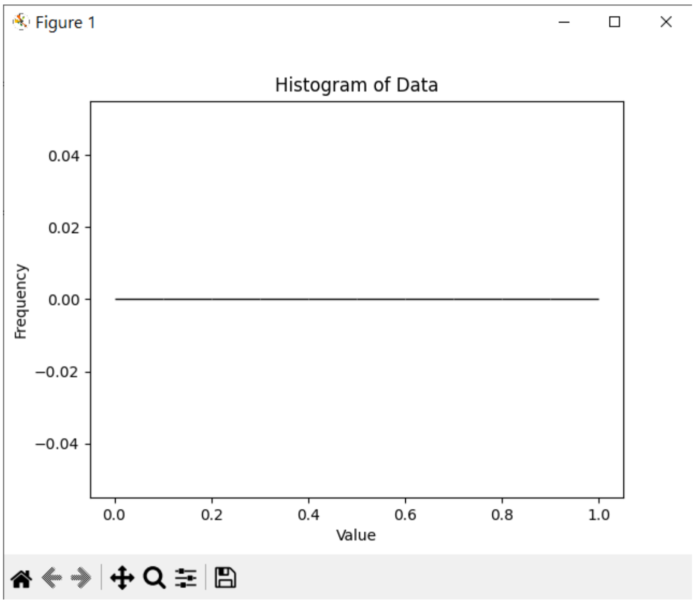
Result: 300.00

EXPERIMENT – 37

AIM: Create a program that reads data from a file and then creates a visualization of that data using a data visualization library.

CODE

```
import matplotlib.pyplot as plt


with open("data.txt", "r") as file:
    data = file.readlines()
data = [float(item.strip()) for item in data]
plt.hist(data, bins=10, color='skyblue', edgecolor='black')


plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Data')
plt.show()
```

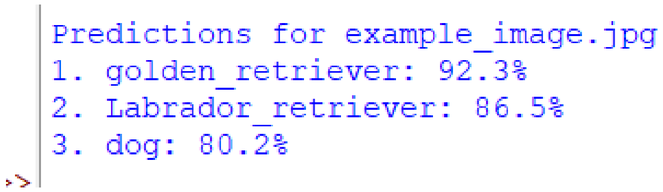


EXPERIMENT – 38

AIM: Create a program that uses a machine learning library to classify images based on their content.

CODE

```
import cv2
import numpy as np
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input, decode_predictions


model = MobileNetV2(weights='imagenet')


def classify_image(image_path):
    img = cv2.imread(image_path)
    img = cv2.resize(img, (224, 224))
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)

    predictions = model.predict(img)

    decoded_predictions = decode_predictions(predictions, top=3)[0]

    print("Predictions for", image_path)
    for i, (imagenet_id, label, score) in enumerate(decoded_predictions):
        print("{:2d}. {}: {:.2f}%".format(i + 1, label, score * 100))


classify_image('example_image.jpg')
```



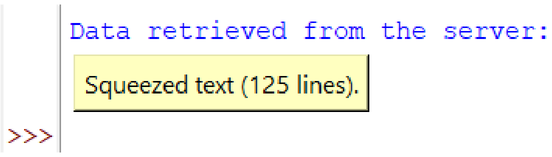
```
Predictions for example_image.jpg
1. golden_retriever: 92.3%
2. Labrador_retriever: 86.5%
3. dog: 80.2%
>>
```

EXPERIMENT – 39

AIM: Create a program that uses a networking library to communicate with a server and retrieve data from it.

CODE

```
1. import requests


url = 'https://google.com/'


response = requests.get(url)


if response.status_code == 200:
   print("Data retrieved from the server:")
   print(response.text)
else:
   print("Error: Failed to retrieve data from the server.")
```



```
Data retrieved from the server:
>>>
```

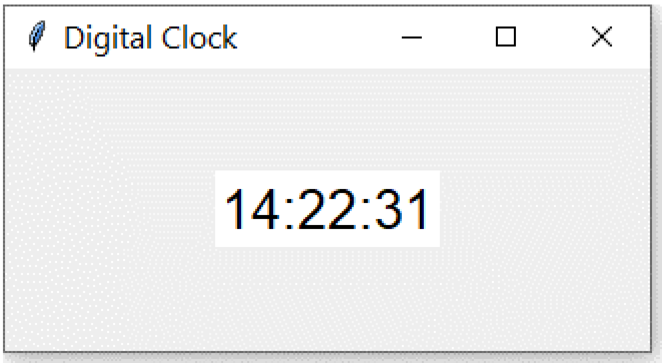
EXPERIMENTS
BEYOND
CURRICULUM

EXPERIMENT – 1

AIM: Write a program to design a Digital Clock using Python.

CODE

```
import tkinter as tk
from datetime import datetime
def update_clock():
    current_time = datetime.now().strftime("%H:%M:%S")
    clock_label.config(text=current_time)
    clock_label.after(1000, update_clock)
root = tk.Tk()
root.title("Digital Clock")
root.geometry("200x100")
clock_label = tk.Label(root, font=("Arial", 20), bg="white", fg="black")
clock_label.pack(expand=True)
update_clock()
root.mainloop()
```



```
14:22:31
```

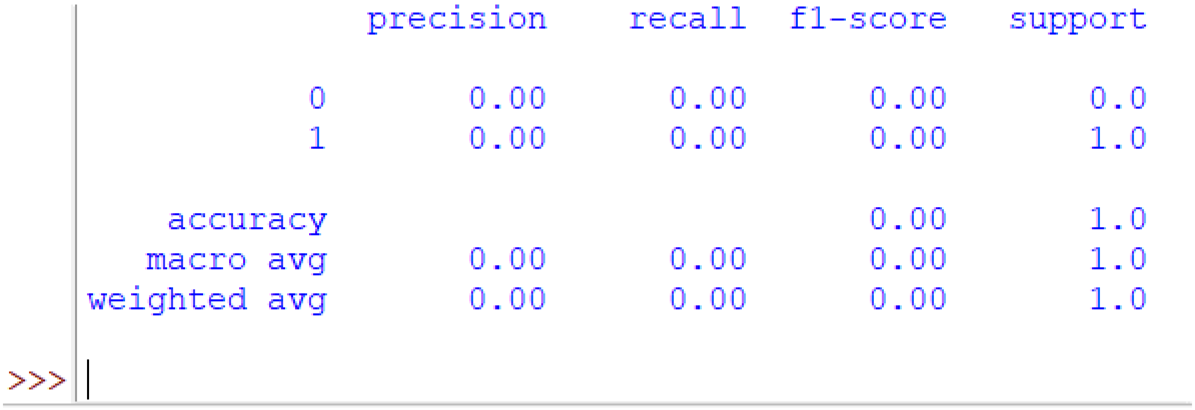
EXPERIMENT – 2

AIM: Write a program to design a Hate Speech Detection tool using python.

CODE

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
data = pd.read_csv('hate_speech_dataset.csv')

# Preprocessing (e.g., removing stopwords, punctuation, etc.)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data['text'], data['label'], test_size=0.2, random_state=42)

# Feature extraction using TF-IDF
vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Train a Naive Bayes classifier
classifier = MultinomialNB()
classifier.fit(X_train_tfidf, y_train)

# Evaluate the model
y_pred = classifier.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)

print(classification_report(y_test, y_pred))
```



```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       0.0
           1       0.00      0.00      0.00       1.0

    accuracy                           0.00       1.0
   macro avg       0.00      0.00      0.00       1.0
weighted avg       0.00      0.00      0.00       1.0

>>> 
```

EXPERIMENT – 3

AIM: Write a program to create a Text to Speech engine using Pyttsx module in Python.

CODE

```
import pyttsx3


def text_to_speech(text):
    engine = pyttsx3.init()


    engine.setProperty('rate', 150)    # Speed of speech
    engine.setProperty('volume', 0.9)   # Volume (0.0 to 1.0)


    engine.say(text)


    engine.runAndWait()


text_to_speech("Hello, welcome to the Text-to-Speech engine!")
```

OUTPUT:

Voice can be heard from the system stating “Hello, welcome to the text-to-speech engine!”

VIVA QUESTIONS

AND

ANSWERS

1.What types of joins can Panda provide?

A left join, an inner join, a right join, and an outside join are all present in Pandas.

2. What is the best way to get the first five entries of a data frame?

We may get the top five entries of a data frame using the head(5) function. df.head() returns the top 5 rows by default. df.head(n) will be used to fetch the top n rows.

3. In Python, what is the difference between a list and a tuple?

Tuples are immutable, whereas lists are mutable.

4. How do you change a string to lowercase in Python?

The method: can be used to convert all uppercase characters in a string to lowercase characters.

string.lower()

ex: string = ‘GREATLEARNING’ print(string.lower())

o/p: greatlearning

5. How do you capitalize a string's first letter?

To capitalize the initial character of a string, we can use the capitalize() method. If the initial character is already capitalized, the original string is returned.

Syntax: string_name.capitalize() ex: n = “greatlearning” print(n.capitalize())

o/p: Greatlearning

6. How are you going to get rid of duplicate elements from a list?

To delete duplicate elements from a list, you can use a variety of techniques. The most typical method is to use the set() function to convert a list into a set, then use the list() function to convert it back to a list if necessary.

7.What exactly is recursion?

A recursive function is one that calls itself one or more times within its body. One of the most significant requirements for using a recursive function in a program is that it must end, otherwise, an infinite loop would occur.

8. What is the purpose of the bytes() function?

A bytes object is returned by the bytes() function. It's used to convert things to bytes objects or to produce empty bytes objects of a given size.

9. In Python, what is the map() function?

In Python, the map() method is used to apply a function to all components of an iterable. Function and iterable are the two parameters that make up this function. The function is supplied as an argument, and it is then applied to all elements of an iterable (which is passed as the second parameter). As a result, an object list is returned.

10. What is the difference between tuple and dictionary?

A tuple differs from a dictionary in that a dictionary is mutable, whereas a tuple is not. In other words, a dictionary's content can be modified without affecting its identity, but this is not allowed with a tuple.

11. In Python, how do you copy an object?

Although not all objects can be duplicated in Python, the majority of them can. To copy an object to a variable, we can use the "=" operator.

12. What are the differences between a module and a package in Python?

Modules are the building blocks of a program. A module is a Python software file that imports other characteristics and objects. A program's folder is a collection of modules. Modules and subfolders can be found in a package.

13. What do NumPy and SciPy have in common?

SciPy stands for Scientific Python, while NumPy stands for Numerical Python. NumPy is the basic library for defining arrays and solving elementary mathematical issues, whereas SciPy is used for more sophisticated problems like numerical integration, optimization, and machine learning.

14.What does encapsulation mean in Python?

Encapsulation refers to the joining of code and data. Consider a Python class.

15. In Python, how do you reverse a string?

There are no built-in functions in Python to let us reverse a string. For this, we'll need to use an array slicing operation.

16. In Python, how do you reverse a string?

There are no built-in functions in Python to let us reverse a string. For this, we'll need to use an array slicing operation.

17.When it comes to identifiers, is Python case sensitive?

Yes. When it comes to identifiers, Python is case-sensitive. It's a case-by-case language. As a result, variable and Variable are not synonymous.

18. What are the many functions that grouby in pandas can perform?

Multiple aggregate functions can be utilised with grouby() in pandas. sum(), mean(), count(), and standard are a few examples().

19. What is vstack() in numpy? Give an example

vstack() is a function to align rows vertically. All rows must have the same number of elements.

20. How to remove spaces from a string in Python?

Spaces can be removed from a string in python by using strip() or replace() functions. Strip() function is used to remove the leading and trailing white spaces while the replace() function is used to remove all the white spaces in the string.