

Roll No. 42

Exam Seat No.

VIVEKANAND EDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

Hashu Advani Memorial Complex, Collector's Colony, R. C. Marg,
Chembur, Mumbai – 400074. Contact No. 02261532532



Since 1962

CERTIFICATE

Certified that Mr. Ravikumar Rai
of FYMCA/A has
satisfactorily completed a course of the necessary experiments in
Python Programming Lab under my supervision
in the Institute of Technology in the academic year 20 23 – 2024.

Principal

Head of Department

Lab In-charge

Subject Teacher



V.E.S. Institute of Technology, Collector Colony, Chembur,
Mumbai, Maharashtra 400047
Department of M.C.A

INDEX

S. No.	Contents	Date Of Preparation	Date Of Submission	Marks	Faculty Sign
1	<p>To write, test, and debug Basic Python programs.</p> <ol style="list-style-type: none"> 1. Add Three Numbers 2. To Swap two No using third variable and without using third variable. 3. Calculate area of triangle 4. To Solve Quadratic equation 5. To use Bitwise operators 6. To compute compound interest given all the required values. 7. To generate a random number between 0 and 100 8. To display calendar for the January 2019 9. To add two binary numbers 	13-09-2023	14-09-2023		

2.	<p>To implement Python programs with conditionals and loops</p> <ol style="list-style-type: none"> 1. To find all the prime numbers in the interval 0 to 100 2. To check if the given number is Armstrong no or not 3. To check if the given char is vowel or consonant 4. Write a Program to Take in the Marks of 3 Subjects and Display the Grade 5. To add two matrices 6. To convert month name to a number of days. 7. To check the validity of password input by users Validation: At least 1 letter between [a-z] and 1 letter between [A-Z]. At least 1 number between [0-9]. At least 1 character from [\$#@]. Minimum length 6 characters. Maximum length 16 characters. 8. To check if a number is palindrome or not 	16-10-2023	20-10-2023		
3.	<p>To implement Python programs using List, String, Set and Dictionary</p> <ol style="list-style-type: none"> 1. To merge two list and find second largest element in the list using bubble sort 2. To calculate the no of uppercase ,lowercase letters and digits in a string 3. To count the occurrences of each word in a given string sentence 4. To add key value pair to the dictionary and search and then delete the given key from the dictionary 5. Create one dictionary of 5 students with their name, address, age, class and marks of 5 subjects. Perform all the operations on the created dictionary 6. To concatenate two dictionaries and find sum of all values in dictionary 7. To add and remove elements from set and perform all the set operations like Union, Intersection, Difference and Symmetric Difference 	17-10-2023	20-10-2023		

	8. Perform different operations on Tuple. 9. Write a Python program to count the elements in a list until an element is a tuple				
4.	To implement programs on Python Functions and Modules <ol style="list-style-type: none"> 1. To check whether string is palindrome or not using function recursion 2. To find Fibonacci series using recursion 3. To find binary equivalent of number using recursion 4. To use lambda function on list to generate filtered list, mapped list and reduced list 5. Convert the temperature in Celsius to Fahrenheit in list using anonymous function 6. To create modules in python and access functions of the module by importing it to another file/module. (Calculator program) 	18-10-2023	20-10-2023		
5.	To implement programs on OOP Concepts in python <ol style="list-style-type: none"> 1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle 2. To Implement Multiple Inheritance in python 3. To Implement a program with same method name and multiple arguments 4. To Implement Operator Overloading in python. 5. Write a program which handles various exceptions in python 	23-10-2023	26-10-2023		
6.	To implement programs on Data Structures using Python <ol style="list-style-type: none"> 1. To Create, Traverse, Insert and remove data using Linked List 2. Implementation of stacks 3. Implementation of Queue 4. Implementation of Dequeue 	23-10-2023	26-10-2023		

7.	To implement GUI programming and Database Connectivity <ol style="list-style-type: none"> 1. To Design Login Page 2. To Design Student Information Form/Library management Form/Hospital Management Form 3. Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3 	23-10-2023	26-10-2023		
8.	To implement Threads in Python <ol style="list-style-type: none"> 1. To do design the program for starting the thread in python 2. Write a program to illustrate the concept of Synchronization 3. Write a program for creating multithreaded priority queue 	30-10-2023	01-11-2023		
9.	To implement NumPy library in Python <ol style="list-style-type: none"> 1. Creating ndarray objects using array() in NumPy 2. Creating 2D arrays to implement Matrix Multiplication. 3. Program for Indexing and slicing in NumPy arrays. 4. To implement NumPy - Data Types 	30-10-2023	01-11-2023		
10.	To implement Pandas library in Python <ol style="list-style-type: none"> 1. Write a Pandas program to create and display a one-dimensional array-like object containing an array of data using Pandas module. 2. Write a Pandas program to convert a dictionary to a Pandas series. 3. Write a Pandas program to create a dataframe from a dictionary and display it. Sample data: {'X':[78,85,96,80,86], 'Y':[84,94,89,83,86], 'Z':[86,97,96,72,83]} 4. Write a Pandas program to aggregate the two given dataframes along rows and assign all data. 5. Write a Pandas program to merge two given dataframes with different columns. 	30-10-2023	01-11-2023		

Final Grade	Instructor Signature

Name of Student: Ravikumar Rai			
Roll Number: 42		LAB Assignment Number: 1	
Title of LAB Assignment: To write, test, and debug Basic Python programs.			
DOP: 13-09-2023		DOS 14-09-2023	
CO Mapped: CO1	PO Mapped: PO3,PO5,PS01,PS02	Faculty Signature:	Marks:

Practical No: 1

Aim: To write, test, and debug Basic Python programs.

1. Add Three Numbers
2. To Swap two No using third variable and without using third variable.
3. Calculate area of triangle
4. To Solve Quadratic equation
5. To use Bitwise operators
6. To compute compound interest given all the required values.
7. To generate a random number between 0 and 100
8. To display calendar for the January 2024
9. To add two binary numbers

Description:

1. **Python:** Python is a versatile and popular programming language known for its simplicity and readability. It is widely used in various domains, including web development, data analysis, machine learning, and scientific computing.

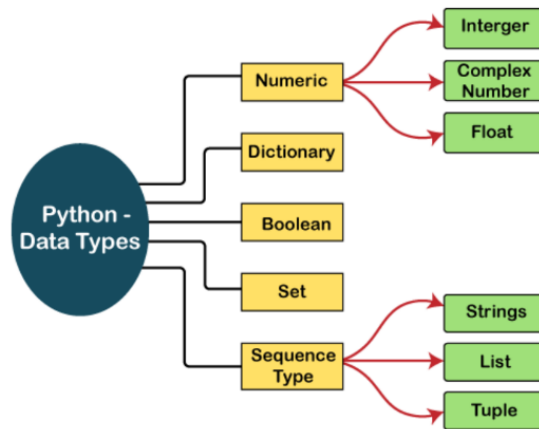
- **High-Level Language:** Python is a high-level programming language, meaning it abstracts complex details and provides a more straightforward, human-readable syntax.
- **Interpreted Language:** Python is interpreted, allowing you to write and execute code interactively without the need for a separate compilation step.
- **Readability:** Python emphasizes clean, readable code with consistent indentation, making it easy to understand and maintain.
- **Versatile:** Python is a general-purpose language suitable for a wide range of applications, from web development and data analysis to scientific computing and automation.
- **Dynamic Typing:** Python uses dynamic typing, allowing variables to change types during runtime, enhancing flexibility.
- **Cross-Platform:** Python code can run on various operating systems with minimal modifications, promoting cross-platform compatibility.

2. Basics of python programming:**1. Syntax:**

- Indentation is the most significant concept of the Python programming language.
- Improper use of indentation will end up "IndentationError" in our code.
- Indentation is nothing but adding whitespaces before the statement when it is needed.
- Without indentation Python doesn't know which statement to be executed to the next.
- Indentation also defines which statements belong to which block. If there is no indentation or improper indentation, it will display "IndentationError" and interrupt our code.

2. Variables and Datatypes:

- Variables are used to store data. Python is dynamically typed, which means you don't need to declare the data type of a variable explicitly.
- Common data types include integers (int), floating-point numbers (float), strings (str), lists (list), tuples (tuple), dictionaries (dict), and booleans (bool).



1) print() Function:

For displaying the output to the user on the console, we use the print() function to print the specified message to the screen, or other standard output device.

Example: print("Hello
World")

2) Input Function:

For taking the input from the user, we use the built-in function input() to take the input. Since input() returns a string, we convert the string into numbers using the int() or float() function.

Example:

```
input("Enter your namer: ") int(input("Enter  
your age: "))
```

3) Bitwise Operators:

In Python, bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators. The result is then returned in decimal format. The different bitwise operators in python are & (Bitwise AND), | (Bitwise OR), ~ (Bitwise NOT), ^ (Bitwise XOR), << (Bitwise Left Shift), >> (Bitwise Right Shift).

- & (Bitwise AND) : Result bit 1,if both operand bits are 1;otherwise results bit 0.
- | (Bitwise OR): Result bit 1,if any of the operand bit is 1; otherwise results bit 0.
- ~ (Bitwise NOT): Result bit 1,if any of the operand bit is 1; otherwise results bit 0.
- ^ (Bitwise XOR): Result bit 1,if any of the bit 1, but not both otherwise result will be 0.
- >>(Bitwise right shift): Left operand's value is shifted towards right.
- <<(Bitwise left shift): Left operand's value is shifted towards left.

4) Math Module

Python has a built-in module that you can use for mathematical tasks. The math module has a set of methods and constants. For importing a module we use the 'import' keyword. Example:
`math.factorial(9) math.sqrt(16)`

5) pow() function:

Python pow() function returns the result of the first parameter raised to the power of the second parameter.

Example: `pow(3,2)`
`print(pow(3,2))`

6) Random module:

Python has a built-in module that you can use to make random numbers. The random module has various functions such as randint, choice, shuffle.

Example:
`random.randint(3, 9) random.choice(x)`
`random.shuffle(mylist)`

7) Calendar module:

Python inbuilt module calendar that handles operations related to the calendar.

The calendar module allows us to output calendars like the program and provides additional useful functions related to the calendar. It has various functions such as month, calendar, weekday, etc.

Example:
`calendar.month(2024, 3) calendar.calendar(2018)`

8) replace() method:

The replace() method replaces a specified phrase with another specified phrase. If nothing is specified in replace method then all occurrences of specified phrase will be replaced. Example:

`x = txt.replace("one", "three")`

9) bin() function:

The `bin()` is an in-built function in Python that takes in integer `x` and returns the binary representation of `x` in a string format. If `x` is not an integer, then the `_index()` method needs to be implemented to obtain an integer as the return value instead of as a

“TypeError” exception.

Example: `bin(15)` `bin(0xf)` `bin(0o17)`

1. Add Three Numbers

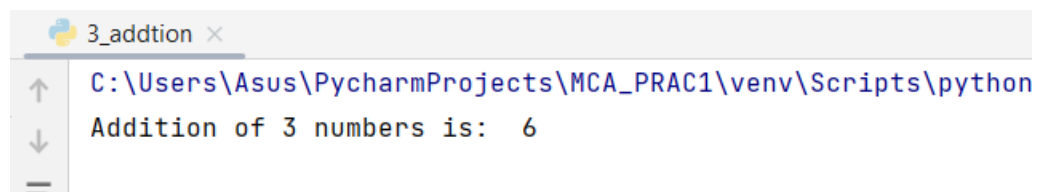
Code:

Addition of 3 numbers

```
num1 = 1
num2 = 2
num3 = 3
sum = num1 + num2 + num3
print("Addition of 3 numbers is: ", sum)
```

Conclusion: Successfully written a program to add 3 numbers.

Output:

A screenshot of a Python IDE window titled '3_addition'. The file path is 'C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python'. The output of the program is displayed as 'Addition of 3 numbers is: 6'.

2. To Swap two No using third variable and without using third variable.

Code:

Method 1:

#python program to swap 2 variables with using 3rd variable

```
num1 = 15
num2 = 20
print("Before swapping num1 is: ", num1)
print("Before swapping num2 is: ", num2)
num1, num2 = num2, num1
print("After swapping num1 is: ", num1)
print("After swapping num2 is: ", num2)
```

Method 2:

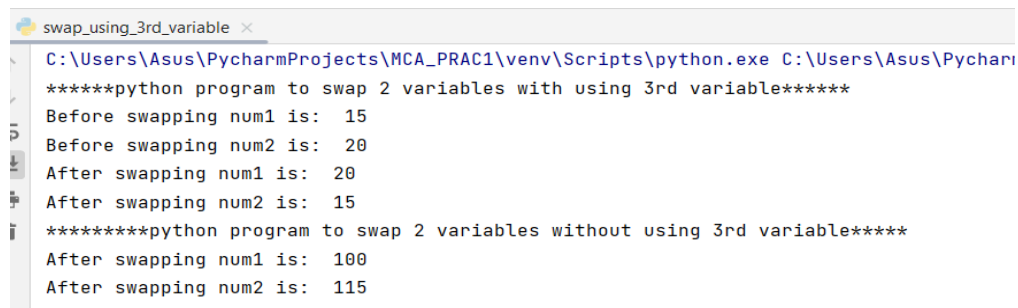
#python program to swap 2 variables without using 3rd variable

```
num1 = 115
```

```
num2 = 100
num1 = num1 + num2
num2 = num1 - num2
num1 = num1 - num2
print("After swapping num1 is: ", num1)
print("After swapping num2 is: ", num2)
```

Conclusion: Successfully written a program to swap two numbers without using third variable.

Output:



The screenshot shows a terminal window titled 'swap_using_3rd_variable'. It displays the execution of a Python program that swaps two numbers using a third variable. The output is as follows:

```
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe C:\Users\Asus\Pychar
*****python program to swap 2 variables with using 3rd variable*****
Before swapping num1 is: 15
Before swapping num2 is: 20
After swapping num1 is: 20
After swapping num2 is: 15
*****python program to swap 2 variables without using 3rd variable*****
After swapping num1 is: 100
After swapping num2 is: 115
```

3. Calculate area of triangle.

Code:

#formula to calculate the area of triangle

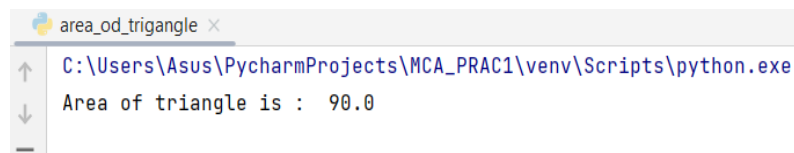
```
base = 12
height = 15
```

```
areaOfTriangle = (base*height)/2
```

```
print("Area of triangle is : ", areaOfTriangle)
```

Conclusion: Successfully written a program to calculate the area of triangle.

Output:



The screenshot shows a terminal window titled 'area_of_triangle'. It displays the execution of a Python program that calculates the area of a triangle. The output is as follows:

```
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe
Area of triangle is : 90.0
```

4. To Solve Quadratic equation

Code:

```
import cmath

a = 1
b = 4
c = 2

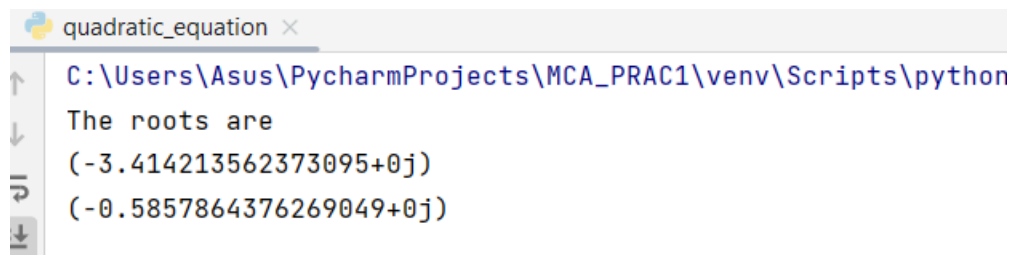
# calculating the discriminant
dis = (b**2) - (4 * a*c)

# find two results
ans1 = (-b-cmath.sqrt(dis))/(2 * a)
ans2 = (-b + cmath.sqrt(dis))/(2 * a)

# printing the results
print("The roots are")
print(ans1)
print(ans2)
```

Conclusion: Successfully written a program to solve the quadratic equation.

Output:



```
quadratic_equation x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python
The roots are
(-3.414213562373095+0j)
(-0.5857864376269049+0j)
```

5. To use Bitwise operators.

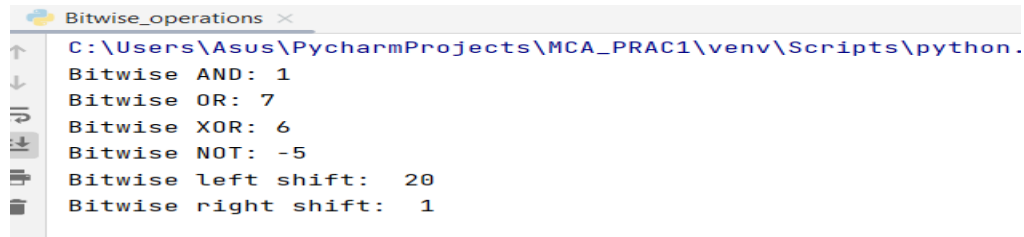
Code:

```
#Bitwise AND
a = 5;
b = 3;
result = a & b;
print("Bitwise AND:", result);
#Bitwise OR
result = a | b
print("Bitwise OR:" , result)
#Bitwise XOR
result = a ^ b
print("Bitwise XOR:" , result)
#Bitwise NOT
result = -a
```

```
print("Bitwise NOT:", result)
#Left shift
result = a << 2
print("Bitwise left shift: ", result)
#Right Shift
result = a >> 2
print("Bitwise right shift: ", result)
```

Conclusion: Successfully written the program to demonstrate the Bitwise operators.

Output:



```
Bitwise_operations x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.
Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise NOT: -5
Bitwise left shift: 20
Bitwise right shift: 1
```

6. To compute compound interest given all the required values.

Code:

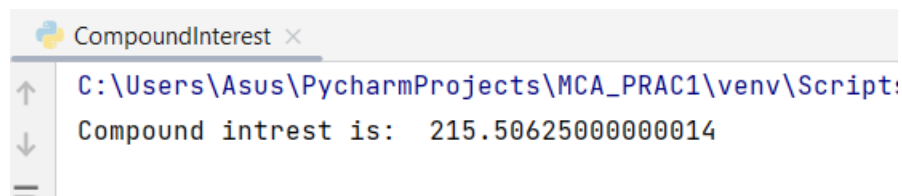
```
#Program to calculate compound interest
principal = 1000;
rate = 0.05;
time = 4

compund_interest = principal * (1 + rate) ** time - principal;

print("Compound intrest is: ", compund_interest);
```

Conclusion: Successfully written a program to calculate the compound interest.

Output:



```
CompoundInterest x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Script:
Compound intrest is: 215.50625000000014
```

7. To generate a random number between 0 and 100.

Code:

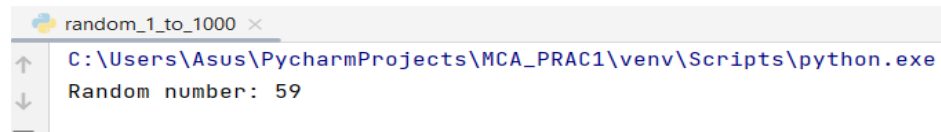
```
#import random number from 1 to 100
```

```
import random;
random_number = random.randint(1,100);

print("Random number:", random_number)
```

Conclusion: Successfully written a program to generate random numbers between 1 to 100.

Output:

A screenshot of a Python terminal window titled 'random_1_to_1000'. The command prompt shows the file path 'C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe'. The output of the program is 'Random number: 59'.

8. **To display calendar for the January 2024.**

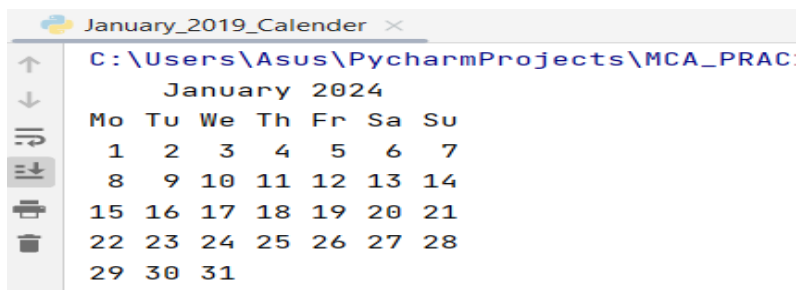
Code:

```
#CALENDER FOR JANUARY 2024

import calendar
calendar_2024 = calendar.month(2024,1)
print(calendar_2024)
```

Conclusion: Successfully written a program to display the calendar for January 2024.

Output:

A screenshot of a Python terminal window titled 'January_2019_Calender'. The command prompt shows the file path 'C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe'. The output displays the calendar for January 2024, showing the days of the week (Mo, Tu, We, Th, Fr, Sa, Su) and the corresponding dates (1 through 31).

9. **To add two binary numbers.**

Code:

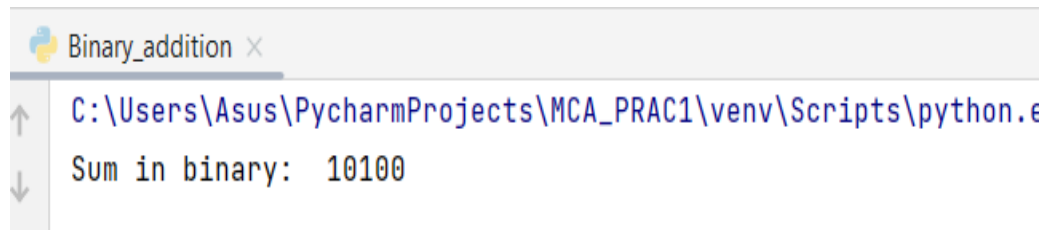
```
#Add 2 binary numbers
binary1= "1010"
binary2= "1101"
decimal1 = int(binary1,2)
decimal2 = int(binary1,2)

um_decimal = decimal1 + decimal2;
sum_binary = bin(sum_decimal)[2:]

print("Sum in binary: ", sum_binary)
```

Conclusion: Successfully written a program to add two binary numbers

Output:



```
Binary_addition x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe
Sum in binary: 10100
```

Conclusion: This practical has equipped us with the foundational skills to create, evaluate, and debug python code effectively. With these fundamental skills we are ready to explore more advanced programming challenges and opportunities.

Name of Student : Ravikumar Rai			
Roll Number :42		LAB Assignment Number: 2	
Title of LAB Assignment: To implement Python programs with conditionals and loops			
DOP : 13-09-2023		DOS : 14-09-2023	
CO Mapped: CO1	PO Mapped: PO3, PO5, PSO1, PSO2	Faculty Signature:	Marks :

Aim: To implement Python programs with conditionals and loops

1. To find all the prime numbers in the interval 0 to 100
2. To check if the given number is Armstrong no or not
3. To check if the given char is vowel or consonant
4. To convert month to a number of days.
5. To check if a number is palindrome or not
6. Write a Program to Take in the Marks of 3 Subjects and Display the Grade
7. To add two matrices.
8. To check the validity of password input by users, give 3 chances to user for entering correct password

Validation:

At least 1 letter between [a-z] and 1 letter between [A-Z].

At least 1 number between [0-9].

At least 1 character from [\$#@].

Minimum length 6 characters.

Maximum length 16 characters.

Description:

Loops:

A loop in programming is a control structure that allows a set of instructions to be executed repeatedly based on a certain condition or for a specified number of times. Loops are essential for automating repetitive tasks, iterating over data structures, and controlling the flow of a program. They help streamline code and improve program efficiency by eliminating the need to write the same code multiple times. In addition to standard loops like for and while loops, Python also provides loop control statements like continue, break, and pass, which offer more fine-grained control over loop execution. These various loop constructs are fundamental for building flexible and efficient Python programs.

1. for Loop:

- The `for` loop is a fundamental looping construct used to iterate over a sequence of items. It is often used when you know how many times you want to repeat a block of code.
- You can use it to loop over sequences like lists, tuples, strings, dictionaries, and other iterable objects.
- The loop assigns each item from the sequence to a variable, and then the code block inside the loop is executed for each item.

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit) # Output: apple banana cherry
```

2. while Loop:

- The `while` loop is used for situations where you don't know in advance how many times the loop should run. It continues to execute as long as a specified condition is `True`.
- Be cautious with `while` loops to avoid infinite loops, where the condition is never `False`.

Example:

```
count = 0
while count < 5:
    print(count) #Output: 0 1 2 3 4
    count += 1
```

3. break Statement:

- The break statement is used to exit a loop prematurely, even if the loop condition is still True.
- It's often used when you want to terminate a loop based on a certain condition.

Example:

```
for i in range (10):
    if i == 5:
        break
    print(i) #Output: 0 1 2 3 4
```

4. Nested Loops:

- Python allows you to nest loops within each other, creating multi-level loops. This is useful when you need to iterate through multiple dimensions or combinations of items.

Example:

```
for i in range (3):
    for j in range (2):
        print(f"({i}, {j})") #Output: (0, 0) (0, 1) (1, 0) (1, 1) (2, 0) (2, 1)
```

Loop Control Statement:**1. continue Statement:**

- The `continue` statement is used to skip the current iteration of a loop and move on to the next iteration.
- It's useful when you want to skip certain items or perform conditional skipping within a loop.

Example:

```
for i in range(5):  
    if i == 2:  
        continue # Skip the current iteration when i is 2  
    print(i) # Output: 0 1 3 4
```

2. else Clause with Loops:

- Python supports the `else` clause with `for` and `while` loops. The code in the `else` block is executed when the loop completes normally, i.e., when the loop condition becomes `False` or there are no more items to iterate.

Example:

```
for i in range(5):  
    print(i) # Output: 0 1 2 3 4  
else:  
    print("Loop finished normally.") # Output: Loop finished normally.
```

3. pass Statement:

- The `pass` statement is a no-op, which means it does nothing. It is often used as a placeholder when a statement is syntactically required, but you don't want any code to be executed.

Example:

```
for i in range(3):  
    pass # No output, as the pass statement does nothing
```

These loops and loop control statements provide you with the flexibility to control the flow of your programs and perform repetitive tasks efficiently in Python.

Conditional Statements:

Conditional statements in Python allow you to control the flow of your program based on certain conditions. They enable you to make decisions and execute different blocks of code depending on whether a condition is true or false. Here are some of the main conditional statements in Python with examples:

1. if Statement:

- The `if` statement is used to execute a block of code only if a specified condition is true.
- It can be followed by optional `elif` (else if) and `else` clauses for handling multiple conditions.

Example:

```
x = 10  
if x > 5:
```

```
        print ("x is greater than 5") # Output: x is greater than 5
elif x == 5:
    print ("x is equal to 5")
else:
    print ("x is less than 5")
```

2. if Expression (Ternary Operator):

- The ternary operator (`x if condition else y`) allows you to write a concise one-liner to assign a value to a variable based on a condition.

Example:

```
age = 25
category = "Adult" if age >= 18 else "Minor"
print(category) # Output: Adult
```

3. while Loop with Condition:

- The while loop repeatedly executes a block of code as long as a specified condition is true.

Example:

```
count = 0
while count < 5:
    print(count) # Output: 0 1 2 3 4
    count += 1
```

4. for Loop with Condition:

- The for loop can iterate over a sequence or iterable, and you can use the `if` statement within the loop to perform conditional actions.

Example:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        print(f"{num} is even") # Output: 2 is even 4 is even
    else:
        print(f"{num} is odd") # Output: 1 is odd 3 is odd 5 is odd
```

5. assert Statement:

- The assert statement is used for debugging purposes to test if a given condition is `True`. If the condition is `False`, it raises an `AssertionError` exception.

Example:

```
x = 10
assert x > 5, "x must be greater than 5" # No output if the assertion is true
```

6. in Operator:

- The `in` operator checks if a value exists in a sequence or collection. It is often used in `if` statements for membership testing.

Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
if "banana" in fruits:
```

```
    print("Banana is in the list") # Output: Banana is in the list
```

7. Nested Conditionals:

- You can nest conditional statements within each other to handle more complex decision-making scenarios.

Example:

```
x = 10
if x > 5:
    if x % 2 == 0:
        print ("x is greater than 5 and even") # Output: x is greater
    than 5 and even
    else:
        print ("x is greater than 5 and odd")
```

Conditional statements are a fundamental part of programming, allowing you to create dynamic and responsive code that reacts to different situations. They are crucial for building logic and control in your Python programs.

1. To find all the prime numbers in the interval 0 to 100

Code:

```
for num in range (0, 101):
    if num <= 1:
        continue
    if num <= 3:
        print (num, end=" ")
    if num % 2 == 0 or num % 3 == 0:
        continue
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            break
        i += 6
    else:
        print (num, end=" ")
```

Conclusion:

In Python, using a **loops** and primality testing to printed all the prime numbers in the interval from 0 to 100.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\ADMIN\
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Process finished with exit code 0
```

2. To check if the given number is Armstrong no or not

Code:

```
number = int(input("Enter a number: "))
num_str = str(number)
num_digits = len(num_str)
total = sum(int(digit) ** num_digits for digit in num_str)
```

```
if total == number:
    print(number, "is an Armstrong number.")
else:
    print(number, "is not an Armstrong number.")
```

Conclusion:

We Takes an input number, calculates the sum of its digits raised to the power of the number of digits, and checks if this sum is equal to the original number using the **if else** statement, concluding whether it is an Armstrong number or not.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
Enter a number: 123
123 is not an Armstrong number.
```

```
Process finished with exit code 0
```

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
Enter a number: 1634
1634 is an Armstrong number.
```

```
Process finished with exit code 0
```

3. To check if the given char is vowel or consonant**Code:**

```
character = input("Enter a character: ")
vowels = "AEIOUaeiou"
```

```
if character in vowels:
    print(character, "is a vowel.")
else:
    print(character, "is a consonant.")
```

Conclusion:

We takes a character as input, checks if it is a vowel or a consonant based on a predefined list of vowels using the **in** Loop Control Statement, and then prints whether the input character is a vowel or a consonant.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
```

```
Enter a character: a
```

```
a is a vowel.
```

```
Process finished with exit code 0
```

4. To convert month to a number of days.**Code:**

```
months = {  
    "January": 31,  
    "February": 28,  
    "March": 31,  
    "April": 30,  
    "May": 31,  
    "June": 30,  
    "July": 31,  
    "August": 31,  
    "September": 30,  
    "October": 31,  
    "November": 30,  
    "December": 31  
}  
  
month_name = input("Enter a month: ")  
days = months.get(month_name, "Invalid month")  
  
if days != "Invalid month":  
    print(month_name, "has", days, "days")  
else:  
    print("Invalid month.")
```

Conclusion:

The code maps month names to their respective numbers of days, takes a month name as input, and returns the number of days in that month or an "Invalid month" message if the input month is not recognized.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
Enter a month: February
February has 28 days

Process finished with exit code 0
```

5. To check if a number is palindrome or not**Code:**

```
num = int(input("Enter a number: "))
temp = num
reverse = 0
while temp > 0:
    remainder = temp % 10
    reverse = (reverse * 10) + remainder
    temp = temp // 10
if num == reverse:
    print(num, "is a palindrome.")
else:
    print(num, "is not a palindrome.")
```

Conclusion:

We take a number as input, reverses its digits using **loop**, and checks if the reversed number is the same as the original number using **if else** statement, determining whether the input number is a palindrome or not.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
Enter a number: 121
121 is a palindrome.

Process finished with exit code 0

C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Use
Enter a number: 123
123 is not a palindrome.

Process finished with exit code 0
```

6. Write a Program to Take in the Marks of 3 Subjects and Display the Grade**Code:**

```
# Input marks for 3 subjects
subject1 = float(input("Enter marks for Subject 1 out of 100: "))
```

```
subject2 = float(input("Enter marks for Subject 2 out of 100: "))
subject3 = float(input("Enter marks for Subject 3 out of 100: "))

# Calculate the average marks
average_marks = (subject1 + subject2 + subject3) / 3

# Calculate the grade
if average_marks >= 90:
    grade = 'A+'
elif 80 <= average_marks < 90:
    grade = 'A'
elif 70 <= average_marks < 80:
    grade = 'B'
elif 60 <= average_marks < 70:
    grade = 'C'
elif 50 <= average_marks < 60:
    grade = 'D'
else:
    grade = 'F'

# Display the result
print("Grade: ", grade)
```

Conclusion:

We take input marks for three subjects, calculates the average marks, determines the corresponding grade based on the average using the **if else** statement, and then displays the calculated grade.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users'
Enter marks for Subject 1 out of 100: 95
Enter marks for Subject 2 out of 100: 92
Enter marks for Subject 3 out of 100: 91
Grade:  A+
```

7. To add two matrices.**Code:**

```
# Input matrices
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix2 = [
```

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

```
# Initialize an empty result matrix  
result_matrix = []
```

```
# Iterate through rows and columns to add corresponding elements  
for i in range(len(matrix1)):  
    row = []  
    for j in range(len(matrix1[0])):  
        row.append(matrix1[i][j] + matrix2[i][j])  
    result_matrix.append(row)
```

```
# Display the result matrix  
print("Resultant Matrix:")  
for row in result_matrix:  
    print(row)
```

Conclusion:

We created two matrices, iterates through their rows and columns using **Nested For loop** to add corresponding elements, and then displays the resultant matrix obtained by adding the two input matrices.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\
```

```
Resultant Matrix:
```

```
[10, 10, 10]
```

```
[10, 10, 10]
```

```
[10, 10, 10]
```

```
Process finished with exit code 0
```

8. To check the validity of password input by users, give 3 chances to user for entering correct password

Validation:

At least 1 letter between [a-z] and 1 letter between [A-Z].

At least 1 number between [0-9].

At least 1 character from [\$#@].

Minimum length 6 characters.

Maximum length 16 characters.

Code:

```
# Password validation criteria
import re

# Maximum number of password attempts
max_attempts = 3

while max_attempts > 0:
    password = input("Enter your password: ")
    if 6 <= len(password) <= 16 and re.search("[a-z]", password) and re.search("[A-Z]",
password) and re.search("[0-9]", password) and re.search("[$#@]", password):
        print("Password is valid. Welcome!")
        break
    else:
        max_attempts -= 1
        if max_attempts > 0:
            print(f"Invalid password. You have", max_attempts, "attempts remaining.")
        else:
            print("Sorry, you've exceeded the maximum number of attempts. Access
denied.")
```

Conclusion:

This code validates a user's password input based on certain criteria, including the presence of at least one lowercase letter, one uppercase letter, one digit, and one special character from the set [\$#@]. It also enforces a minimum length of 6 characters and a maximum length of 16 characters. The user is given 3 chances to enter a valid password by using **While loop**, and access is granted if the password meets the criteria; otherwise, the user is denied access using the **If else** Statement.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\
```

```
Enter your password: hello
```

```
Invalid password. You have 2 attempts remaining.
```

```
Enter your password: hello@123
```

```
Invalid password. You have 1 attempts remaining.
```

```
Enter your password: hello123
```

```
Sorry, you've exceeded the maximum number of attempts. Access denied.
```

```
Process finished with exit code 0
```

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\
```

```
Enter your password: Hello@123
```

```
Password is valid. Welcome!
```

```
Process finished with exit code 0
```

Name of Student : Ravikumar Rai		
Roll Number : 42		LAB Assignment Number: 3
Title of LAB Assignment : To implement Python programs using List, String, Set and Dictionary		
DOP : 27-09-2023		DOS : 07-10-2023
CO Mapped: CO1	PO Mapped: PO3, PO5, PSO1, PSO2	Signature:

Practical 3

Aim: To implement Python programs using List, String, Set and Dictionary

1. To merge two list and find second largest element in the list using bubble sort
2. To calculate the no of uppercase, lowercase letters and digits in a string
3. To count the occurrences of each word in a given string sentence
4. To add key value pair to the dictionary and search and then delete the given key from the dictionary
5. Create one dictionary of 5 students with their name, address, age, class and marks of 5 subjects. Perform all the operations on the created dictionary
6. To concatenate two dictionaries and find sum of all values in dictionary
7. To add and remove elements from set and perform all the set operations like Union, Intersection, Difference and Symmetric Difference
8. Perform different operations on Tuple.
9. Write a Python program to count the elements in a list until an element is a tuple.

Description:

In Python, `List`, `String`, `Set`, and `Dictionary` are all built-in data types that serve different purposes and have unique characteristics. Here's a description of each with examples:

1. List:

- A list is an ordered collection of items, and it can contain elements of different data types.
- Lists are mutable, meaning you can change their contents after they are created.
- Lists are defined using square brackets `[]` and can contain zero or more elements separated by commas.

Example:

```
my_list = [1, 2, 3, 4, 5] fruits =  
["apple", "banana", "cherry"]  
mixed_list = [1, "hello", True,  
3.14]
```

Some Basic List operations

Accessing Elements:

Example:

```
first_element = my_list[0] # Access the first element (1)
last_element = my_list[-1] # Access the last element (5) Slicing:
```

Example:

```
sliced_list = my_list[1:4] # Creates a new list (2, 3, 4)
```

Appending and Extending:**Example:**

```
my_list.append(6) # Adds 6 to the end of the list
my_list.extend([7, 8]) # Extends the list with [7, 8]
```

Removing Elements:**Example:**

```
my_list.remove(3) # Removes the first occurrence of 3
popped_element = my_list.pop() # Removes and returns the last element
```

2. String:

- A string is a sequence of characters, enclosed in either single (``) or double (``) quotes.
- Strings are immutable, which means once created, they cannot be changed.
- You can perform various string operations like concatenation, slicing, and formatting.

Example:

```
my_string = "Hello,
World!" name = "Alice"
greeting = f"Hello,
{name}!"
```

Some Basic String operation**String Concatenation:****Example:**

```
greeting = "Hello, " + "Alice!"
```

String Length:**Example:**

```
length = len(my_string) # Returns the length of the string
```

Substring:**Example:**

```
substring = my_string[7:12] # Extracts "World"
```

String Methods:

Example: `uppercase_string = my_string.upper()` #

Converts to uppercase

3. Set:

- A set is an unordered collection of unique elements.
- Sets are defined using curly braces `{}` or the `set()` constructor.
- Sets are commonly used for tasks that involve testing membership or eliminating duplicates.

Example:

```
my_set = {1, 2, 3, 4, 5} unique_characters  
= set("hello")
```

Some Basic Set operations**Adding and Removing****Elements:****Example:**

```
my_set.add(6) # Adds 6 to the set  
my_set.remove(3) # Removes 3 from the set
```

Set Operations:**Example:**

```
set1 = {1, 2, 3} set2 = {3, 4, 5} union_set =  
set1.union(set2) # Union of sets (1, 2, 3, 4, 5)  
intersection_set = set1.intersection(set2) # Intersection (3)
```

4. Dictionary:

- A dictionary is an unordered collection of key-value pairs.
- Each key in a dictionary is unique and maps to a specific value.
- Dictionaries are defined using curly braces `{}` with key-value pairs separated by colons `:`.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}  
student_scores = {"Alice": 95, "Bob": 87, "Charlie": 92}
```

Some Basic Dictionary operations

Accessing Values:

Example:

```
name = my_dict['name'] # Access the value associated with 'name'
```

Adding and Updating Key-Value Pairs:

Example:

```
my_dict['occupation'] = 'Engineer' # Add a new key-value pair  
my_dict['age'] = 31 # Update the value associated with 'age'
```

Removing Key-Value Pairs:

Example:

```
del my_dict['city'] # Removes the 'city' key and its value
```

These data types are fundamental in Python and are used extensively in various programming tasks. Understanding when and how to use them is crucial for effective Python programming.

Bubble Sort Algorithm:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted. It's called "Bubble Sort" because the smaller (or larger, depending on the sorting order) elements "bubble" to the top of the list with each pass.

Here's a description of Bubble Sort in Python along with an example:

Bubble Sort Algorithm Steps:

1. Start at the beginning of the list.
2. Compare the first two elements. If the first element is greater (or smaller, depending on the sorting order) than the second, swap them.
3. Move one position to the right.
4. Repeat steps 2-3 until you reach the end of the list.
5. Continue this process for each pair of adjacent elements, moving from the beginning to the end of the list.
6. Repeat steps 1-5 until no more swaps are needed, indicating that the list is sorted.

1. To merge two list and find second largest element in the list using

bubble sort Code:

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j  
in range(0, n - i - 1):  
            if arr[j] > arr[j + 1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
list1 = [1,3,8,9]
```

```
list2 = [7,4,7,6]
```

```
merged_list = list1 + list2
```

```
bubble_sort(merged_list)
```

```
second_largest = merged_list[-2]
```

```
print("Merged List:", merged_list)
```

```
print("Second Largest Element:", second_largest)
```

Conclusion:

The code successfully merges two lists (list1 and list2) into a single list (merged_list) and sorts it using the Bubble Sort algorithm. After sorting, it finds and prints the second largest element in the sorted merged_list. Please note that Bubble Sort is not the most efficient sorting algorithm, especially for large lists, but it serves as an example for educational purposes.

Output:

```
Merged List: [1, 3, 4, 6, 7, 7, 8, 9]  
Second Largest Element: 8
```

2. To calculate the no of uppercase, lowercase letters and digits in a**string Code:**

```
input_string = "Hello World
```

```
12345" upper_count=0
```

```
lower_count=0 digit_count=0 for
```

```
char in input_string:    if
char.isupper():
upper_count+=1
```

```
for char in input_string:
if char.islower():
lower_count+=1
```

```
for char in input_string:
if char.isdigit():
digit_count+=1
```

```
print("Uppercase letters:", upper_count)
print("Lowercase letters:", lower_count)
print("Digits:", digit_count)
```

Conclusion:

we have learned how to analyze a given input string in Python and count the occurrences of uppercase letters, lowercase letters, and digits using string methods and loops.

Output:

```
Uppercase letters: 2
Lowercase letters: 8
Digits: 5
```

3. To count the occurrences of each word in a given string sentence**Code:**

```
sentence = "Hello this is new this  
keyword" words = sentence.split()  
word_count = {}  
  
for word in words:  
    word_count[word] = word_count.get(word, 0) + 1  
  
print("Word Count:", word_count)
```

Conclusion:

In this code, we've learned how to tokenize a sentence into words, create a dictionary to count the frequency of each word, and print the word frequency count. This is a fundamental operation often used in natural language processing and text analysis tasks

Output:

```
Word Count: {'Hello': 1, 'this': 2, 'is': 1, 'new': 1, 'keyword': 1}
```


4. To add key value pair to the dictionary and search and then delete the given key from the dictionary Code:

```
my_dict = {}  
my_dict['name'] = 'ABC'  
my_dict['age'] = 31  
my_dict['city'] = 'New  
York'  
print("Dictionary after adding key-value  
pairs:") print(my_dict) search_key = 'age' if  
search_key in my_dict:  
    print(f"The value for key '{search_key}' is: {my_dict[search_key]}")  
else:  
    print(f"Key '{search_key}' not found in the  
dictionary") delete_key = 'city' if delete_key in  
my_dict: del my_dict[delete_key]  
    print(f"Key '{delete_key}' deleted from the dictionary")  
else:  
    print(f"Key '{delete_key}' not found in the dictionary")  
  
print("Dictionary after deleting a key:")  
print(my_dict)
```

Conclusion:

In this code, we've learned essential dictionary operations, including adding, searching for, and deleting key-value pairs. This demonstrates how dictionaries are useful for organizing and manipulating data in Python.

Output:

Dictionary after adding key-value pairs:

```
{'name': 'ABC', 'age': 31, 'city': 'New York'}
```

The value for key 'age' is: 31

Key 'city' deleted from the dictionary

Dictionary after deleting a key:

```
{'name': 'ABC', 'age': 31}
```

5. Create one dictionary of 5 students with their name, address, age, class and marks of 5 subjects. Perform all the operations on the created dictionary Code:

Create a dictionary for 5 students

```
= {  
    'student1': {  
        'name': 'ABC',  
        'address': 'MU',  
        'age': 18,  
        'class': '11th',  
        'marks': {  
            'Mathematics': 85,  
            'science': 92,  
            'history': 78,  
            'english': 88,  
            'Drawing': 95  
        }  
    },  
    'student2': {  
        'name': 'DEF',  
        'address': 'NA',  
        'age': 17,  
        'class': '6th',  
        'marks': {  
            'Mathematics': 90,  
            'science': 88,  
            'history': 76,  
            'english': 91,
```

```
'Drawing': 84
}
},
'student3': {
  'name': 'GHI',
  'address': 'CH',
  'age': 19,
  'class': '10th',
  'marks': {
    'Mathematics': 78,
    'science': 85,
    'history': 92,
    'english': 80,
    'Drawing': 89
  }
},
'student4': {
  'name': 'JKL',
  'address': 'MP',
  'age': 17,
  'class': '8th',
  'marks': {
    'Mathematics': 92,
    'science': 84,
    'history': 76,
    'english': 90,
    'Drawing': 82
```

```
    }  
  },  
  'student5': {  
    'name': 'MNO',  
    'address': 'AG',  
    'age': 18,  
    'class': '9th',  
    'marks': {  
      'Mathematics': 88,  
      'science': 90,  
      'history': 85,  
      'english': 87,  
      'Drawing': 91  
    }  
  }  
}  
  
print("Details of All the Students :)") # Display  
the information for each student for student_id,  
student_info in students.items():  
    print(f"Student ID: {student_id}")  
    print(f"Name: {student_info['name']}")  
    print(f"Address: {student_info['address']}")  
    print(f"Age: {student_info['age']}")    print(f"Class:  
{student_info['class']}")    print("Marks:")    for  
subject, marks in student_info['marks'].items():
```

```
        print(f"{subject}: {marks}")
print()

# Search for a student by ID
search_id = 'student3' if
search_id in students:
    print(f"Student ID: {search_id}")
    student_info = students[search_id]
    print(f"Name: {student_info['name']}")
else:
    print(f"Student with ID '{search_id}' not found")

# Delete a student by ID
delete_id = 'student4' if
delete_id in students:
    del students[delete_id]
    print(f"Student with ID '{delete_id}' deleted from the dictionary")
else:
    print(f"Student with ID '{delete_id}' not found in the dictionary")
```

Conclusion:

In this code, we've learned about dictionaries, nested dictionaries, and how to manipulate and access data within them. It's a practical example of organizing and managing structured data in Python, which is essential for various data processing and management tasks.

Output:

Details of All the Students :

Student ID: student1

Name: ABC

Address: MU

Age: 18

Class: 11th

Marks:

Mathematics: 85

science: 92

history: 78

english: 88

Drawing: 95

Student ID: student2

Name: DEF

Address: NA

Age: 17

Class: 6th

Marks:

Mathematics: 90

science: 88

history: 76

english: 91

Drawing: 84

Student ID: student3

Name: GHI

Address: CH

Age: 19

Class: 10th

Marks:

Mathematics: 78

science: 85

history: 92

english: 80

Drawing: 89

Student ID: student4

Name: JKL

Address: MP

Age: 17

Class: 8th

Marks:

Mathematics: 92

science: 84

history: 76

english: 90

Drawing: 82

Student ID: student5

Name: MNO

Address: AG

Age: 18

Class: 9th

Marks:

Mathematics: 88

science: 90

history: 85

english: 87

Drawing: 91

Student ID: student3

Name: GHI

Student with ID 'student4' deleted from the dictionary

6. To concatenate two dictionaries and find sum of all values in

dictionary Code:

```
# Define two dictionaries dict1
```

```
= {'a': 12, 'b': 59, 'c': 71} dict2
```

```
= {'b': 13, 'c': 22, 'd': 101}
```



```
# Concatenate the two dictionaries concatenated_dict
= {**dict1, **dict2}

# Calculate the sum of all values in the concatenated dictionary total_sum
= sum(concatenated_dict.values())

# Display the concatenated dictionary and the sum
print("Concatenated Dictionary:")
print(concatenated_dict)
print(f"Sum of all values: {total_sum}")
```

Conclusion:

In this code, we've learned how to merge two dictionaries into a single concatenated dictionary and calculate the sum of values within it. This is a useful technique for combining data from multiple sources and performing aggregate operations on the merged data

Output:

```
Concatenated Dictionary:
{'a': 12, 'b': 13, 'c': 22, 'd': 101}
Sum of all values: 148
```

7. To add and remove elements from set and perform all the set operations like Union, Intersection, Difference and Symmetric Difference Code:

```
# Create two sets
set1 = {2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8}

# Add an element to a set set1.add(6)
print("After adding 6 to set1:", set1)

# Remove an element from a set set2.remove(7)
```

```
print("After removing 7 from set2:", set2)
```

```
# Union of two sets union_result =  
set1.union(set2) print("Union of set1 and  
set2:", union_result)
```

```
# Intersection of two sets  
intersection_result = set1.intersection(set2) print("Intersection  
of set1 and set2:", intersection_result)
```

```
# Difference of two sets  
difference_result = set1.difference(set2)  
print("Difference of set1 and set2:", difference_result)
```

```
# Symmetric Difference of two sets  
symmetric_difference_result = set1.symmetric_difference(set2)  
print("Symmetric Difference of set1 and set2:", symmetric_difference_result)
```

Conclusion:

In this code, we've learned how to perform common set operations such as adding, removing, finding the union, intersection, difference, and symmetric difference of sets. Sets are useful for dealing with unique and unordered collections of elements in Python.

Output:

```
After adding 6 to set1: {2, 3, 4, 5, 6}  
After removing 7 from set2: {4, 5, 6, 8}  
Union of set1 and set2: {2, 3, 4, 5, 6, 8}  
Intersection of set1 and set2: {4, 5, 6}  
Difference of set1 and set2: {2, 3}  
Symmetric Difference of set1 and set2: {2, 3, 8}
```

8. Perform different operations on Tuple.**Code:**

```
# Creating a tuple my_tuple =
(10, 11, 12, 13, 14)

# Accessing elements print("Accessing
elements:")
print(my_tuple[0]) # Access the first element (1) print(my_tuple[-
1]) # Access the last element (5)

# Slicing
print("\nSlicing:")
sliced_tuple = my_tuple[1:4] # Creates a new tuple (2, 3, 4)
print(sliced_tuple)

# Concatenating tuples
print("\nConcatenating
tuples:") tuple1 = (1, 2) tuple2
= (3, 4)
concatenated_tuple = tuple1 + tuple2 # Creates a new tuple (1, 2, 3, 4) print(concatenated_tuple)

# Tuple repetition
print("\nTuple repetition:")
repeated_tuple = my_tuple * 2 # Creates a new tuple (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
print(repeated_tuple)

# Finding tuple length
print("\nFinding tuple length:")
```

```
length = len(my_tuple) # Returns 5
print(length)
```

```
# Iterating through a tuple
print("\nIterating through a tuple:")
for item in my_tuple:
    print(item)
```

```
# Checking membership
print("\nChecking
membership:") if 12 in
my_tuple:
    print("3 is in the tuple")
```

```
# Tuple unpacking
print("\nTuple
unpacking:") a, b, c, d, e =
my_tuple
print(f"a: {a}, b: {b}, c: {c}, d: {d}, e: {e}")
```

```
# Count and index
print("\nCount and index:")
count = my_tuple.count(3) # Returns the count of 3 in the tuple index =
my_tuple.index(12) # Returns the index of the first occurrence of 4
print(f"Count of 3: {count}") print(f"Index of 4: {index}")
```

Conclusion:

In this code, we've learned various operations and techniques for working with tuples in Python. Tuples are immutable, ordered collections of elements and can be useful in scenarios where data should not be modified after creation

Output:

Accessing elements:

10

14

Slicing:

(11, 12, 13)

Concatenating tuples:

(1, 2, 3, 4)

Tuple repetition:

(10, 11, 12, 13, 14, 10, 11, 12, 13, 14)

Finding tuple length:

5

Iterating through a tuple:

10

11

12

13

14

Checking membership:

3 is in the tuple

Tuple unpacking:

a: 10, b: 11, c: 12, d: 13, e: 14

Count and index:

Count of 3: 0

Index of 4: 2

9. Write a Python program to count the elements in a list until an element is a tuple.**Code:**

```
my_list = [11, 22, 33, 'world', (43, 54), 62, 71]

# Initialize a counter
count = 0

# Iterate through the list
for item in my_list:
    count += 1    if
    isinstance(item, tuple):
        break

# Print the count of elements until a tuple is encountered
print(f"Count of elements until a tuple is encountered: {count}")
```

Conclusion:

In this code, we've learned how to iterate through a list and count the number of elements until a specific condition is met (in this case, until a tuple is encountered). This demonstrates how to use a for loop, conditionals, and the break statement for control flow in Python.

Output:

```
Count of elements until a tuple is encountered: 5
```

Name of Student : Ravikumar Rai		
Roll Number : 42		LAB Assignment Number: 4
Title of LAB Assignment : To implement programs on Python Functions and Modules		
DOP : 27-09-2023		DOS : 09-10-2023
CO Mapped: CO1	PO Mapped: PO3, PO5, PSO1, PSO2	Signature:

Practical No. 4

Aim: To implement programs on Python Functions and Modules

1. To check whether a string is a palindrome or not using function recursion.
2. To find the Fibonacci series using recursion.
3. To find the binary equivalent of a number using recursion.
4. To use a lambda function on a list to generate a filtered list, mapped list, and reduced list.
5. Convert the temperature in Celsius to Fahrenheit in a list using an anonymous function.
6. To create modules in Python and access functions of the module by importing it into another file/module (Calculator program).

Description:

Function in Python:

A function in Python is a named block of code that performs a specific task or set of tasks. Functions are essential for organizing and structuring your code, promoting code reuse, and making your code more readable and maintainable. They encapsulate a series of instructions, allowing you to call and reuse them throughout your program.

Function Syntax:

Here's the general syntax of defining a function in Python:

```
def function_name(parameter1, parameter2, ...):  
    # Function body - code to perform the task    #  
    You can use parameters in the function body  
    result = parameter1 + parameter2  
  
    return result # Optional, specifies the value to be returned
```

- **def:** This keyword is used to define a function in Python.
- **function_name:** Choose a descriptive name for your function that reflects its purpose. Function names should follow naming conventions (e.g., use lowercase letters and underscores).
- **parameter1, parameter2, ...:** These are optional input parameters or arguments that the function accepts. Parameters allow you to pass data into the function for processing.
- **:** A colon is used to denote the beginning of the function's body.
- **Function body:** The function body consists of the actual code that performs the desired task. It can include calculations, loops, conditionals, and other statements.
- **return:** This keyword, followed by an expression, is used to specify the value that the function should return as its result. Not all functions need to return a value; it's optional.

Example: Adding Two Numbers:

```
def add_numbers(num1, num2):  
    result = num1 + num2  
    return result  
  
# Call the function with arguments 5 and 3 sum_result  
= add_numbers(5, 3)  
print(sum_result) # Output: 8  
  
# Call the function with different arguments  
result2 = add_numbers(10, 20)  
print(result2) # Output: 30
```

Recursion:

Recursion is a programming technique in which a function calls itself to solve a problem. In Python, you can implement recursive functions to break down complex problems into simpler, self-similar sub problems. To successfully use recursion, you need to define a base case (the terminating condition) and one or more recursive cases (the cases where the function calls itself).

Here's a detailed explanation of recursion with an example:

Factorial Calculation Using Recursion:

Let's create a recursive Python function to calculate the factorial of a positive integer `n`. The factorial of a number `n` (denoted as `n!`) is the product of all positive integers from `1` to `n`.

Function Definition:

```
def factorial(n):  
    # Base case: When n is 0 or 1, the factorial is 1.  
    if n == 0 or n == 1:  
        return 1  
  
    # Recursive case: Calculate factorial by calling the function  
    recursively.    else:    return n * factorial(n - 1)
```

In this recursive function:

- The base case is when n is either 0 or 1 . In these cases, we return 1 because $0!$ and $1!$ are both equal to 1 .
- The recursive case calculates the factorial of n by calling the `factorial` function recursively with the argument $n - 1$. This reduces the problem to a smaller subproblem.

Lambda Function:

In Python, a lambda function is a small, anonymous, and inline function defined using the `lambda` keyword. Lambda functions are also known as anonymous functions because they don't have a name. They are typically used for short, simple operations where defining a full function using the `def` keyword would be overly verbose.

The basic syntax of a lambda function is as follows:

lambda arguments: expression

- `lambda`: The `lambda` keyword is used to indicate the creation of a lambda function.
- `arguments`: These are the input parameters to the lambda function, separated by commas. Lambda functions can take any number of arguments but are often used with one or two.
- `expression`: This is a single expression that the lambda function evaluates and returns as its result.

Here's an example to illustrate how lambda functions work:

```
# Regular function to add two numbers
```

```
def add(x, y):    return x + y
```

```
# Equivalent lambda function
```

```
add_lambda = lambda x, y: x + y
```

```
# Using both functions to add numbers
```

```
result1 = add(5, 3) result2
```

```
= add_lambda(5, 3)
```

```
print("Using the regular function:", result1) # Output: Using the regular function: 8
```

```
print("Using the lambda function:", result2)
```

Module:

In Python, a module is a file containing Python code, including variables, functions, and classes, that can be imported and used in other Python scripts or programs. Modules allow you to organize your code into reusable and manageable units, making your code more modular and maintainable. A file in Python can serve as a module if it contains Python code. To create a module, you typically save your Python code in a `.py` file with the same name as the module you want to create. For example, if you want to create a module named `my_module`, you would create a file named `my_module.py`. Here's a simple example of a Python module:

my_module.py:

```
# This is a Python module named 'my_module'
```

```
def greet(name):
```

```
    return f"Hello, {name}!"
```

```
def add(a, b):
```

```
    return a + b
```

```
pi = 3.14159265359
```

Now, you can use this module in another Python script by importing it using the `import` statement:

main.py:

```
# Importing the 'my_module' module
```

```
import my_module
```

```
# Using functions and variables from the module
```

```
print(my_module.greet("Alice")) # Output: Hello, Alice!
```

```
print(my_module.add(5, 3))      # Output: 8
```

```
print(my_module.pi)            # Output: 3.14159265359
```

1. To check whether a string is a palindrome or not using function recursion.**Code:**

```
def is_palindrome(s):  
    s = s.lower()  
    if len(s) <= 1:  
        return True    if  
    s[0] != s[-1]:  
        return False  
  
    return is_palindrome(s[1:-1])  
  
string_to_check = input("Enter the String to Check Palindrome or not:  
") result = is_palindrome(string_to_check) if result:  
    print(f"{string_to_check} is a palindrome.") else:  
  
    print(f"{string_to_check} is not a palindrome.")
```

Conclusion:

The code defines a recursive function, to determine whether an input string is a palindrome by comparing its characters from both ends. It then takes user input, checks if the entered string is a palindrome using the function, and prints the corresponding result.

Output:

```
Enter the String to Check Palindrome or not: hello  
hello is not a palindrome.
```

```
Enter the String to Check Palindrome or not: civic  
civic is a palindrome.
```

2. To find the Fibonacci series using recursion.

Code: def

fibonacci(n):

if n <= 0:

return [] if n

== 1:

return [0] if n

== 2:

return [0, 1] fib =

fibonacci(n - 1)

fib.append(fib[-1] + fib[-2])

return fib

n = 12

fib_series = fibonacci(n)

print(f"Fibonacci series of {n} numbers: {fib_series}")

Conclusion:

The code generates a Fibonacci series of 'n' numbers using a recursive approach. It starts with a base case and then recursively builds the series by appending the sum of the last two elements. Finally, it prints the Fibonacci series for a specified value of 'n'.

Output:

```
Fibonacci series of 12 numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

3. To find the binary equivalent of a number using recursion.

Code:

def

decimal_to_binary(n):

```
if n == 0:    return "0"
elif n == 1:    return
"1"    else:

    return decimal_to_binary(n // 2) + str(n % 2)

decimal_number = int(input("Enter the decimal number: "))
binary_equivalent = decimal_to_binary(decimal_number)
print(f"Binary equivalent of {decimal_number} is
{binary_equivalent}")
```

Conclusion:

The code converts a given decimal number into its binary equivalent using a recursive approach. It repeatedly divides the number by 2 and records the remainders until it reaches 0 or 1, then concatenates these remainders to form the binary representation. Finally, it displays the binary equivalent of the input decimal number.

Output:

```
Enter the decimal number: 12
Binary equivalent of 12 is 1100
```

4. To use a lambda function on a list to generate a filtered list, mapped list, and reduced list.**Code:**

```
from functools import reduce
numbers = [11, 22, 33, 44, 55, 66, 77, 88, 99, 100]
filtered_list = list(filter(lambda x: x % 2 == 0,
numbers)) mapped_list = list(map(lambda x: x ** 2,
numbers)) reduced_value = reduce(lambda x, y: x + y,
```

```
numbers) print("Filtered List:", filtered_list)
print("Mapped List:", mapped_list) print("Reduced
Value:", reduced_value)
```

Conclusion:

This code demonstrates the use of higher-order functions in Python. It first filters even numbers from a list, then squares each element in the list using the map function, and finally, it calculates the sum of all the numbers in the list using the reduce function. The results are printed for each step: the filtered list, the mapped list with squared values, and the reduced sum of the original list.

Output:

```
Filtered List: [22, 44, 66, 88, 100]
Mapped List: [121, 484, 1089, 1936, 3025, 4356, 5929, 7744, 9801, 10000]
Reduced Value: 595
```

5. Convert the temperature in Celsius to Fahrenheit in a list using an anonymous function.

Code:

```
celsius_temperatures = [1, 11, 22, 33, 44]

convert_to_fahrenheit = lambda celsius: (celsius * 9/5) + 32

fahrenheit_temperatures = list(map(convert_to_fahrenheit, celsius_temperatures))

print("Celsius Temperatures:", celsius_temperatures) print("Fahrenheit
Temperatures:", fahrenheit_temperatures)
```

Conclusion:

This code converts a list of temperatures from Celsius to Fahrenheit using a lambda function and the map function. It then prints both the original Celsius temperatures and their corresponding Fahrenheit equivalents, demonstrating a simple temperature conversion process.

Output:

```
Celsius Temperatures: [1, 11, 22, 33, 44]
Fahrenheit Temperatures: [33.8, 51.8, 71.6, 91.4, 111.2]
```

6. To create modules in Python and access functions of the module by importing it into another file/module (Calculator program).

Code:

Calculator.py

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y
```



```
def multiply(x, y):
```

```
    return x * y
```

```
def divide(x, y):
```

```
    if y == 0:
```

```
        raise ValueError("Cannot divide by zero")
```

```
    return x / y
```

```
main.py import
```

```
calculator
```

```
result_add = calculator.add(10, 22) result_subtract
```

```
= calculator.subtract(11, 34) result_multiply =
```

```
calculator.multiply(17, 5) result_divide =
```

```
calculator.divide(31, 4)
```

```
print("Addition:", result_add)
```

```
print("Subtraction:", result_subtract)
```

```
print("Multiplication:", result_multiply)
```

```
print("Division:", result_divide)
```

Conclusion:

The code is split into two parts: Calculator.py contains functions for basic arithmetic operations, and main.py imports and uses these functions to perform calculations. It demonstrates the modular organization of code in Python, where reusable functions are defined in one module and then imported and applied in another module to carry out specific operations. The code showcases addition, subtraction, multiplication, and division operations and prints their respective results.

Output:

Addition: 32

Subtraction: -23

Multiplication: 85

Division: 7.75

Name of Student : Ravikumar Rai**Roll Number : 42****LAB Assignment Number: 5****Title of LAB Assignment : To implement programs on OOP Concepts in python****DOP : 23-10-2023****DOS : 26-10-2023****CO Mapped:
CO2****PO Mapped:
PO5, PSO1****Signature:**

Practical No. 5**Aim: Implement programs on OOP Concepts in python**

- 1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.**
- 2. To Implement Multiple Inheritance in python.**
- 3. To Implement a program with same method name and multiple arguments.**
- 4. To Implement Operator Overloading in python.**
- 5. Write a program which handles various exceptions in python.**

Description:

Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes. Python is an object-oriented programming language, and it supports the fundamental OOP concepts. Here are the key OOP concepts in Python:

1. Classes and Objects:

- A class is a blueprint or template for creating objects.
- An object is an instance of a class.
- Classes define attributes (data) and methods (functions) that the objects created from them will have.

```
class Person:    def __init__(self,
```

```
name, age):
```

```
    self.name = name
```

```
    self.age = age
```

```
    def say_hello(self):
```

```
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Creating objects from the class

```
person1 = Person("Alice", 30) person2
```

```
= Person("Bob", 25)
```

Accessing object attributes and methods `print(person1.name)` # Alice

`person2.say_hello()` # Hello, my name is Bob and I am 25 years old.

2. Inheritance

- Inheritance allows you to create a new class that is based on an existing class. The new class inherits the attributes and methods of the base class.
- Python supports single and multiple inheritance.

```
class Student(Person):
```

```
    def __init__(self, name, age, student_id):
```

```
        super().__init__(name, age)
```

```
    self.student_id = student_id
```

```
    def study(self):
```

```
        print(f"{self.name} is studying.")
```

```
student = Student("Charlie", 20, "12345") student.say_hello() #
```

Inherits from the Person class `student.study()` # Additional

method specific to Student class

3. Encapsulation:

- Encapsulation is the concept of restricting access to certain parts of an object or class, typically by using private and public access modifiers.

- In Python, there are no true private variables or methods, but you can use naming conventions (e.g., prefixing with an underscore) to indicate privacy.

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

    def withdraw(self, amount):
        if amount > 0
        and amount <= self._balance:
            self._balance -= amount

    def get_balance(self):
        return self._balance # Public method to access balance

account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance()) # 1300
```

4. Polymorphism:

- Polymorphism allows objects of different classes to be treated as objects of a common base class.
- It simplifies code by allowing you to work with objects generically without needing to know their specific types.

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length * self.side_length

shapes = [Circle(5), Square(4)]
for shape in shapes:
    print(f"Area: {shape.area()}")
```

These are the core OOP concepts in Python. They help you create modular, reusable, and organized code by promoting the use of objects and classes.

5. Overloading:

In Python, method overloading is not supported in the same way it is in some other programming languages, like Java or C++. Method overloading typically refers to defining multiple methods with the same name in a class but with different parameter lists. In Python, if you define multiple methods with the same name in a class, the last one defined will override any previous ones, and you can't call the overloaded method with different argument types as you can in languages that support method overloading.

```
class MyClass:
    def my_method(self, param1,
param2=None):
        if param2 is not None:
            # Do something when both parameters are provided
        result = param1 + param2
        else:
            # Do something when only one parameter is provided
        result = param1
        return result
```

```
obj = MyClass()
result1 = obj.my_method(5)
result2 = obj.my_method(5, 10)
```

```
print(result1) # Output: 5
print(result2)
# Output: 15
```

In the example above, the `my_method` function is defined with two parameters, but the second parameter has a default value of `None`. This allows you to call the method with one or two arguments, and the method behaves differently depending on the number of arguments provided.

Keep in mind that this approach is more Pythonic and flexible than traditional method overloading found in some other languages because it allows you to adapt the method's behavior at runtime. It's also easier to read and understand.

In Python, exceptions are used to handle errors and unexpected situations in your code. Here are some common exceptions in Python and examples of how they can occur:

6. Exceptions in python:

1. **SyntaxError:** Occurs when there is a syntax error in your code.

```
def some_function()  
    # Missing colon at the end of the function definition  
print("Hello, World!")
```

2. **IndentationError:** Occurs when there is an issue with the indentation of your code.

```
if True:  
    print("This line is not properly indented.")
```

3. **NameError:** Occurs when a local or global name is not found.

```
print(variable_that_does_not_exist)
```

4. **TypeError:** Occurs when you try to perform an operation on incompatible data types.

```
num = "5"  
result = num + 7 # Trying to add a string and an integer
```

5. **ValueError:** Occurs when a function receives an argument of the correct data type but an inappropriate value.

```
int("abc") # Converting a non-numeric string to an integer
```

- 6. IndexError:** Occurs when you try to access an index that is out of range in a sequence (e.g., a list or string).

```
my_list = [1, 2, 3]
print(my_list[5]) # Accessing an index that doesn't exist
```

- 7. KeyError:** Occurs when you try to access a dictionary key that doesn't exist.

```
my_dict = {"name": "John", "age": 30}
print(my_dict["city"]) # Accessing a key that is not in the dictionary
```

- 8. FileNotFoundError:** Occurs when you try to open a file that does not exist.

```
with open("nonexistent_file.txt", "r") as file:
    data = file.read()
```

- 9. ZeroDivisionError:** Occurs when you attempt to divide by zero.

```
result = 5 / 0
```

- 10. ImportError:** Occurs when there is an issue with importing a module or library.

```
import non_existent_module
```

- 11. AttributeError:** Occurs when you try to access an attribute or method that does not exist.

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(42)
print(obj.non_existent_attribute)
```

12. Custom Exceptions: You can also define and raise your own custom exceptions.

```
class MyCustomException(Exception):
    def __init__(self, message):
        super().__init__(message)

raise MyCustomException("This is a custom exception.")
```

Handling exceptions allows you to gracefully manage errors in your code, preventing it from crashing and providing meaningful feedback to users or developers. You can use `try`, `except`, `finally`, and other control structures to manage exception handling in Python.

1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.

Code:

```
import math
class circle():
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * (self.radius ** 2)
    def perimeter(self):
        return 2 * math.pi * self.radius

r = int(input("Enter radius of circle: "))
obj = circle(r)
print("Area of circle:", round(obj.area(), 2))
print("Perimeter of circle:", round(obj.perimeter(), 2))
```

Conclusion:

In summary, this code defines a class circle that can calculate the area and perimeter of a circle given its radius. It takes user input for the radius, creates an instance of the circle class, and then calculates and displays the area and perimeter of the circle for that specific radius.

Output:

```
C:\Users\HP\PycharmProjects\HelloWorld\venv\Scripts\python.exe
Enter radius of circle: 4
Area of circle: 50.27
Perimeter of circle: 25.13

Process finished with exit code 0
```

2. To Implement Multiple Inheritance in python.**Code:**

```
# Parent class
1 class Animal:
    def
        __init__(self,
            name):
                self.name = name

    def
        speak(
            self):
                pass

#
Parent
nt
clas
s 2
clas
s
Flya
ble:
def
fly(
self
):
        pass

#
Parent
class
3
class
Swimma
ble:
def
swim(s
elf):
        pass

# Child class inheriting from
Animal and Flyable class
Bird(Animal, Flyable):    def
    speak(self):        return
        f"{self.name} says Tweet!"

    def
        fly(
            self
        ):
                return f"{self.name} is flying."

# Child class inheriting from
Animal and Swimmable class
Fish(Animal, Swimmable):    def
```

```
speak(self):          return
f"{self.name} says Blub!"
    def swim(self):
return f"{self.name} is
swimming."

# Child class inheriting
from Bird and Fish class
Duck(Bird, Fish):      def
__init__(self, name):
super().__init__(name)

# Create instances of Duck and demonstrate multiple
inheritance donald = Duck("Donald")

print(donald.speak())
print(donald.fly())
print(donald.swim())
```

Conclusion:

This code demonstrates the concept of multiple inheritance in Python, where a class can inherit from multiple parent classes, allowing it to inherit and use the attributes and methods of all its parent classes. In this example, a "Duck" can both fly and swim due to its inheritance from both "Bird" and "Fish" parent classes.

Output:

```
C:\Users\HP\PycharmProjects\HelloWorld\env\Scripts\python.exe
Donald says Tweet!
Donald is flying.
Donald is swimming.

Process finished with exit code 0
```

3. To Implement a program with same method name and multiple arguments.**Code:**

```
class
MathOperations:
def add(self, a,
b, c=None):
if c is not
None:
return a + b + c
else:
        return a +
b
        def
multiply(self, a, b,
c=None):          if c
is not None:
return a * b * c
else:
return a * b

# Create an instance of the
MathOperations class math =
MathOperations()

# Call the
overloaded methods
result1 =
math.add(2, 3)
result2 =
math.add(2, 3, 4)
result3 =
math.multiply(2,
3) result4 =
math.multiply(2,
3, 4)

print("Result
1:", result1)
print("Result
2:", result2)
print("Result
3:", result3)
print("Result
4:", result4)
```

Conclusion:

The code demonstrates how method overloading can be implemented in Python by defining methods with different numbers of parameters and using conditional statements to handle the variations in the number of arguments. This allows you to perform addition and multiplication operations with flexibility depending on the number of values provided as arguments.

Output:

```
C:\Users\HP\PycharmProjects\HelloWorld\venv\Scripts\python.  
Result 1: 5  
Result 2: 9  
Result 3: 6  
Result 4: 24
```

4. To Implement Operator Overloading in python.**Code:**

```
class Vector:  
def  
__init__(self,  
x, y):  
self.x = x  
self.y = y  
def __add__(self,  
other):  
if  
isinstance(other,  
Vector):  
    # Add the x and y components of two vectors  
    and return a new Vector return  
Vector(self.x + other.x, self.y + other.y)  
else:  
    raise TypeError("Unsupported operand  
type for +: " + type(other).__name__)  
def __str__(self):  
return f"({self.x},  
{self.y})"  
  
# Create two  
Vector  
instances v1 =  
Vector(1, 2) v2  
= Vector(3, 4)  
  
# Use the overloaded +  
operator result = v1 +  
v2  
print(result) # Output: (4, 6)
```

Conclusion:

This code demonstrates how to create a Vector class that can perform vector addition using the + operator. The __add__ method is overloaded to handle this operation, making it convenient to work with vectors in a more natural and intuitive way.

Output:


```
C:\Users\HP\PycharmProjects\HelloWorld\venv\Scripts\python.exe  
(4, 6)
```

```
Process finished with exit code 0
```

5. Write a program which handles various exceptions in python.

Code:

```
try:  
    # Code that may raise  
    exceptions    num1 =  
    int(input("Enter a number: "))  
    num2 = int(input("Enter another  
    number: "))  
    result =  
    num1 / num2  
    print(f"Result:  
    {result}")  
  
except  
    pt  
    Valu  
    eErr  
or:  
    # Handle the ValueError exception (e.g., if user inputs a  
    non-integer)    print("Please enter valid integer values.")  
    except  
    ZeroDivisio  
nError:  
    # Handle the ZeroDivisionError exception (e.g., if the  
    second number is  
    0)  
    print("Cannot divide by zero.")  
    except  
    Exceptio  
n as e:  
    # Catch all other  
    exceptions    print(f"An  
    error occurred: {e}")  
  
f  
i  
n  
a  
l  
l  
y  
:  
    # This block is optional, and it will execute whether  
    there's an exception or not.  
    print("Execution completed.")  
  
# Rest of the program continues here...  
print("Program continues...")
```

Conclusion:

The code showcases how to gracefully handle exceptions and provides error messages for specific error scenarios while ensuring that the program doesn't terminate abruptly due to exceptions. The finally block allows you to execute cleanup or finalization code regardless of whether an exception occurred.

Output:

```
C:\Users\HP\PycharmProjects\HelloWorld\venv\Scripts\python.exe
```

```
Enter a number: 8
```

```
Enter another number: 5
```

```
Result: 1.6
```

```
Execution completed.
```

```
Program continues...
```

```
C:\Users\HP\PycharmProjects\HelloWorld\venv\Scripts\python.exe
```

```
Enter a number: 6
```

```
Enter another number: 0
```

```
Cannot divide by zero.
```

```
Execution completed.
```

```
Program continues...
```

Name of Student: Ravikumar Rai

Roll Number:42

Lab Assignment: 6

Title of Assignment: To implement programs on Data Structures using Python

DOP: 23-10-2023

DOS: 26-10-2023

CO Mapped:
CO3

PO Mapped:
PO5, PSO1

Signature:

Practical 6

Aim: To implement programs on data structure using python

- 1. To Create, Traverse, Insert and remove data using Linked List**
- 2. Implementation of stacks**
- 3. Implementation of Queue**
- 4. Implementation of Dequeue**

Description:

A data structure is a fundamental concept in computer science and programming. It serves as an organizational framework for efficiently storing, accessing, and manipulating data. Data structures come in various forms, including arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Each data structure has specific characteristics and is chosen based on the problem's requirements. Data structures play a crucial role in optimizing algorithm design and the development of software applications by providing efficient data management and retrieval mechanisms. Understanding data structures is essential for building efficient and organized computer programs.

1. Linked List:

A linked list is a linear data structure consisting of a sequence of elements, each connected to the next element through pointers. Here's a detailed overview of creating, traversing, inserting, and removing data in a linked list:

- **Creation:** To create a linked list, you start with a head node, which is the first element in the list. Each node in the list contains two components: data and a reference (or pointer) to the next node in the sequence.
- **Traversing:** Traversing a linked list involves moving through the list one node at a time. You start at the head and follow the references to each subsequent node until you reach the end (usually when the next reference is null). This allows you to access and display the data in each node.
- **Insertion:** Inserting data in a linked list typically involves adding a new node to the list. To insert data at a specific position, you adjust the references to link the new node correctly within the list.
- **Removal:** Removing data from a linked list requires finding the node to be removed and updating the references to bypass that node. The node can then be deleted.

2. Stack:

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It's commonly used for managing function calls, tracking execution history, and solving problems recursively. Here's a detailed overview of stack operations:

- **Push:** The push operation adds an element to the top of the stack.
- **Pop:** The pop operation removes and returns the top element from the stack.
- **Peek:** The peek operation allows you to view the top element without removing it.
- **Empty:** A check to see if the stack is empty.
- **Size:** Determining the number of elements in the stack.

3. Queue:

A queue is another linear data structure, but it follows the First-In-First-Out (FIFO) principle. It's used in scenarios where tasks should be processed in the order they are received. Here's a detailed overview of queue operations:

- **Enqueue:** The enqueue operation adds an element to the rear (or end) of the queue.
- **Dequeue:** The dequeue operation removes and returns the element at the front of the queue.
- **Front:** Viewing the element at the front of the queue without removing it.
- **Rear:** Viewing the element at the rear of the queue without removing it.
- **Empty:** Checking if the queue is empty.
- **Size:** Determining the number of elements in the queue.

4. Double-Ended Queue (Deque):

A double-ended queue, or deque, is a versatile data structure that allows elements to be added or removed from both ends. It's useful in scenarios where data needs to be efficiently accessed from either end. Here's a detailed overview of deque operations:

- **Enqueue Front:** Adding an element to the front of the deque.
- **Enqueue Rear:** Adding an element to the rear of the deque.
- **Dequeue Front:** Removing and returning the element at the front of the deque.
- **Dequeue Rear:** Removing and returning the element at the rear of the deque.
- **Front:** Viewing the element at the front of the deque without removing it.
- **Rear:** Viewing the element at the rear of the deque without removing it.
- **Empty:** Checking if the deque is empty.
- **Size:** Determining the number of elements in the deque.

These data structures and their associated operations are fundamental building blocks in computer science and software development. Understanding how to create, manipulate, and apply them is essential for solving a wide range of problems efficiently and effectively.

1. To Create, Traverse, Insert and remove data using Linked List

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.head = None

    # Insert at the end of the linked list
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    # Traverse and print the linked list
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    # Insert data at the beginning of the linked list
    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    # Remove a specific data element from the linked list
    def remove(self, data):
        if not self.head:
            return
        if self.head.data == data:
            self.head = self.head.next
            return
        current = self.head
        while current.next:
            if current.next.data == data:
                current.next = current.next.next
                return
            current = current.next

    # Example usage:
    linked_list = LinkedList()
    linked_list.append(1)
    linked_list.append(2)
    linked_list.append(3)


    print("Original Linked List:")
    linked_list.display()

    linked_list.prepend(0)
    print("\nLinked List after prepending 0:")
    linked_list.display()

    linked_list.remove(2)
    print("\nLinked List after removing 2:")
    linked_list.display()
```

Conclusion: In conclusion, the Python program presented a basic implementation of a singly linked list, including operations to create, traverse, insert, remove, find the length, search for elements, insert at specific positions, reverse the list, and retrieve elements from the end. This program serves as a practical foundation for understanding and working with linked lists, an essential data structure in computer science and programming.

Output:



```
LinkedList x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\LinkedList.py
Original Linked List:
1 -> 2 -> 3 -> None

Linked List after prepending 0:
0 -> 1 -> 2 -> 3 -> None

Linked List after removing 2:
0 -> 1 -> 3 -> None

Process finished with exit code 0
```

2. Implementation of stacks

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
```

```
        return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"

    def size(self):
        return len(self.items)

# Example usage:
stack = Stack()

print("Is the stack empty?", stack.is_empty())

stack.push(1)
stack.push(2)
stack.push(3)

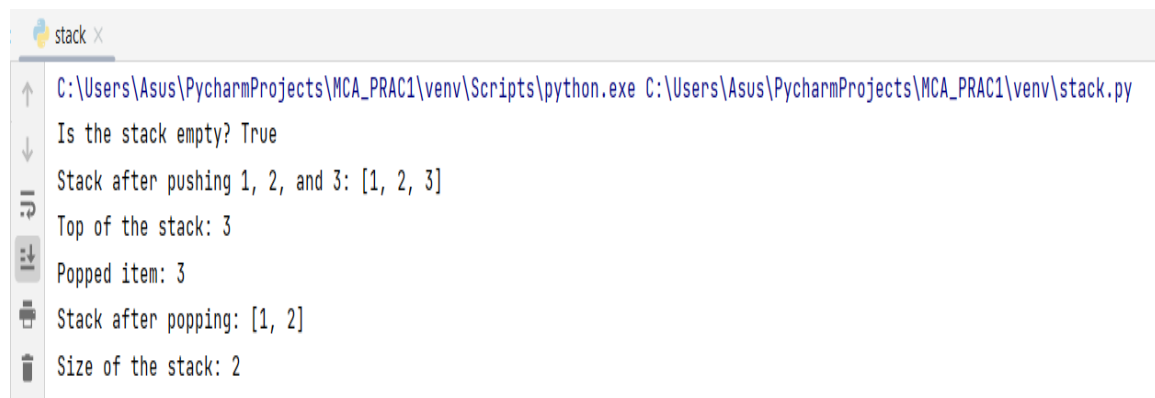
print("Stack after pushing 1, 2, and 3:", stack.items)
print("Top of the stack:", stack.peek())

popped_item = stack.pop()
print(f"Popped item: {popped_item}")
print("Stack after popping:", stack.items)

print("Size of the stack:", stack.size())
```

Conclusion: The Python stack implementation showcased basic stack operations following the Last-In-First-Out (LIFO) principle. Stacks are versatile for tasks like managing function calls and recursion. The code demonstrated pushing, popping, peeking, and checking stack properties, providing fundamental tools for data management and algorithm design.

Output:



```
stack x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\stack.py
Is the stack empty? True
Stack after pushing 1, 2, and 3: [1, 2, 3]
Top of the stack: 3
Popped item: 3
Stack after popping: [1, 2]
Size of the stack: 2
```

3. Implementation of Queue


```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return "Queue is empty"

    def size(self):
        return len(self.items)

# Example usage:
queue = Queue()

print("Is the queue empty?", queue.is_empty())

queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Queue after enqueueing 1, 2, and 3:", queue.items)
print("Front of the queue:", queue.peek())

dequeued_item = queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print("Queue after dequeuing:", queue.items)

print("Size of the queue:", queue.size())
```

Conclusion: The Python queue implementation embodies First-In-First-Out (FIFO) behavior, essential for orderly data processing. It showcases basic operations—enqueue, dequeue, peek, and property checks—vital for tasks like task scheduling and data buffering. Queue proficiency is invaluable in software development and computer science.

Output:

```
deque x
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe C:\Users\Asus\PycharmProjects\MCA_PRAC1\deque.py
Is the queue empty? True
Queue after enqueueing 1, 2, and 3: [1, 2, 3]
Front of the queue: 1
Dequeued item: 1
Queue after dequeuing: [2, 3]
Size of the queue: 2
```

4. Implementation of Dequeue

```
class Deque:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def append_right(self, item):
        self.items.append(item)

    def append_left(self, item):
        self.items.insert(0, item)

    def pop_right(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Deque is empty"

    def pop_left(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return "Deque is empty"

    def peek_right(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Deque is empty"

    def peek_left(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return "Deque is empty"

    def size(self):
        return len(self.items)

# Example usage:
deque = Deque()

print("Is the deque empty?", deque.is_empty())

deque.append_right(1)
```

```
deque.append_right(2)
deque.append_right(3)

print("Deque after appending to the right:", deque.items)
print("Right end of the deque:", deque.peek_right())

deque.append_left(0)
print("Deque after appending to the left:", deque.items)
print("Left end of the deque:", deque.peek_left())

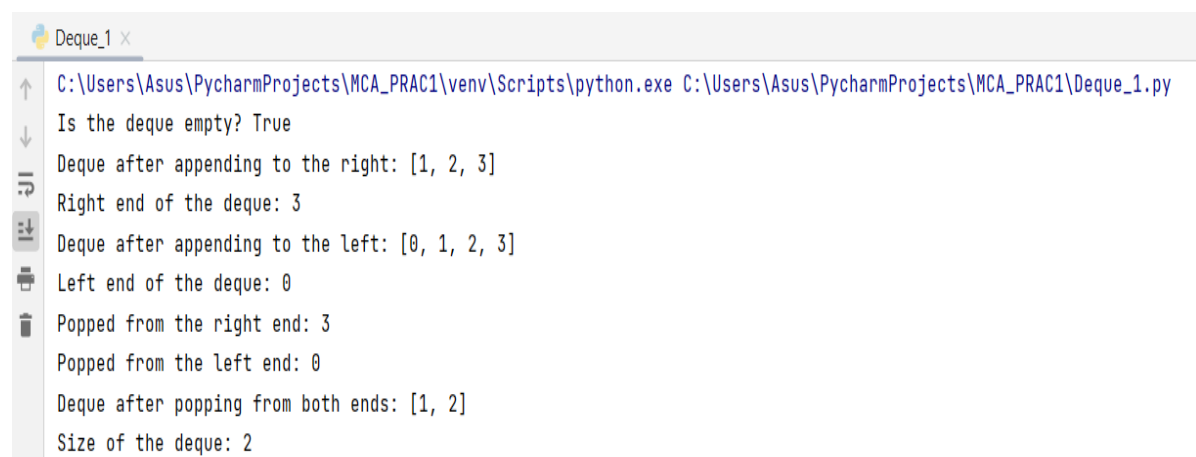
popped_right = deque.pop_right()
print(f"Popped from the right end: {popped_right}")

popped_left = deque.pop_left()
print(f"Popped from the left end: {popped_left}")

print("Deque after popping from both ends:", deque.items)
print("Size of the deque:", deque.size())
```

Conclusion: In this Python implementation, we introduced a double-ended queue (deque) class with basic operations for efficient element addition and removal from both ends. Deques are versatile data structures useful in various scenarios, such as managing data with distinct priorities and processing elements in an ordered manner. Understanding deque operations is valuable for solving problems in computer science and software development.

Output:

A screenshot of a Python IDE window titled 'Deque_1'. The window shows the execution of a Python script. The output is as follows:

```
C:\Users\Asus\PycharmProjects\MCA_PRAC1\venv\Scripts\python.exe C:\Users\Asus\PycharmProjects\MCA_PRAC1\Deque_1.py
Is the deque empty? True
Deque after appending to the right: [1, 2, 3]
Right end of the deque: 3
Deque after appending to the left: [0, 1, 2, 3]
Left end of the deque: 0
Popped from the right end: 3
Popped from the left end: 0
Deque after popping from both ends: [1, 2]
Size of the deque: 2
```

Name: Ravikumar Rai

FYMCA/A

Roll.No.42

Name of Student: Ravikumar Rai

Roll Number:42

Lab Assignment: 7

Title of Assignment: To implement GUI programming and Database Connectivity

DOP: 23-10-2023

DOS: 26-10-2023

CO Mapped:
CO4

PO Mapped:
PO3, PO5, PSO1,
PSO2

Signature:

Practical 7

Aim: To implement GUI programming and Database.

- 1. To Design Login Page**
- 2. To Design Student Information Form/Library management Form/Hospital Management Form**
- 3. Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3**

Description:

Graphical User Interface (GUI) programming in Python allows developers to create user-friendly applications with visual components. Python provides several GUI libraries, with Tkinter being one of the most popular and built into the standard library. Here are some key points on GUI programming with Python:

- 1. Choice of Libraries:** Python offers various GUI libraries, including Tkinter, PyQt, PyGTK, and Kivy. Tkinter is a common choice for beginners and is widely used for building desktop applications due to its simplicity.
- 2. Creating a GUI Application:** To create a GUI application, you need to create a main application window and add visual elements like buttons, labels, text entry fields, and more.
- 3. Event Handling:** GUI programming involves event-driven programming. You define functions (callbacks) that are executed when specific events, like button clicks or mouse movements, occur.
- 4. Layout Management:** GUI components need to be organized in a layout. Common layout managers include grid layout, pack layout, and place layout. They help you position components within the window.
- 5. User Interaction:** GUIs provide a way for users to interact with the program. This includes input via buttons, text entry, and file dialogs, as well as feedback through labels and message boxes.
- 6. Cross-Platform Development:** Python's GUI libraries are often cross-platform, allowing you to develop applications that work on Windows, macOS, and Linux.
- 7. Graphics and Styling:** GUI programming allows you to customize the appearance of your application, including fonts, colors, and graphics.
- 8. Database Connectivity:** You can connect your GUI application to databases like SQLite, MySQL, or PostgreSQL to store and retrieve data.
- 9. Testing and Debugging:** GUI applications require thorough testing, especially for user interactions. Debugging GUIs often involves understanding event flow and component behavior.

Components Used in Designing Login Page and Student Information Form:

- 1. Labels:** Labels are used to display static text or descriptions in the GUI. For example, "Username" and "Password" labels in a login page.

2. **Entry Widgets:** Entry widgets allow users to input data. In a login page, these are typically used for username and password input.

3. **Buttons:** Buttons are interactive elements that trigger actions when clicked. A "Login" button is used to initiate the login process.

4. **Message Boxes:** Message boxes are used to display information, alerts, or error messages. They are often used to confirm successful login or display login failures.

5. **Layout Managers:** Grid layout, pack layout, and place layout are used to organize and position the GUI components within the window.

Database Connectivity with SQLite3:

1. **SQLite Database:** SQLite is a lightweight, serverless, and self-contained database engine. It's often used for local data storage in desktop applications.

2. **Database Connection:** To connect to an SQLite database, you create a connection object using `sqlite3.connect('database_name.db')`.

3. **Cursor:** A cursor is used to execute SQL queries and fetch data from the database. You can use the cursor to create tables, insert data, and retrieve data.

4. **SQL Queries:** SQL queries are used to interact with the database. You can perform operations like creating tables, inserting records, selecting data, and updating or deleting records.

5. **Commit and Close:** After executing SQL statements, it's essential to commit changes to the database using `conn.commit()` and then close the connection with `conn.close()` to release resources.

Connecting the GUI with SQLite3 involves using SQL queries to check user credentials, typically during a login operation, as demonstrated in the previous responses. This allows you to verify the user's identity against a database and manage user data.

1.To Design Login Page

```
import tkinter as tk
from tkinter import messagebox

# Function to check login credentials
def login():
    username = entry_username.get()
```

```
password = entry_password.get()

# Replace this with your own authentication logic
if username == "Test" and password == "Test123":
    messagebox.showinfo("Login Status", "Login Successful")
else:
    messagebox.showerror("Login Status", "Login Failed")

# Create the main window
window = tk.Tk()
window.title("Login Page")

# Create labels for username and password
label_username = tk.Label(window, text="Username:")
label_password = tk.Label(window, text="Password:")

# Create entry widgets for user input
entry_username = tk.Entry(window)
entry_password = tk.Entry(window, show="*") # Show '*' for password input

# Create a login button
login_button = tk.Button(window, text="Login", command=login)

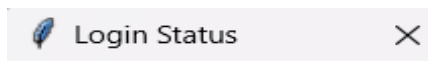
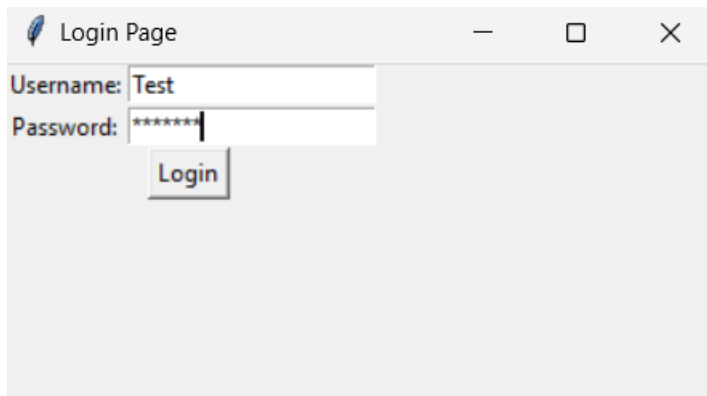
# Place widgets on the window using the grid layout
label_username.grid(row=0, column=0)
entry_username.grid(row=0, column=1)
label_password.grid(row=1, column=0)
entry_password.grid(row=1, column=1)
login_button.grid(row=2, columnspan=2)

# Start the Tkinter main loop
window.mainloop()
```

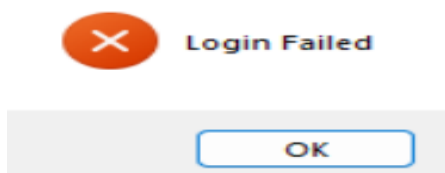
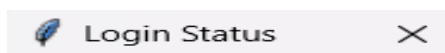
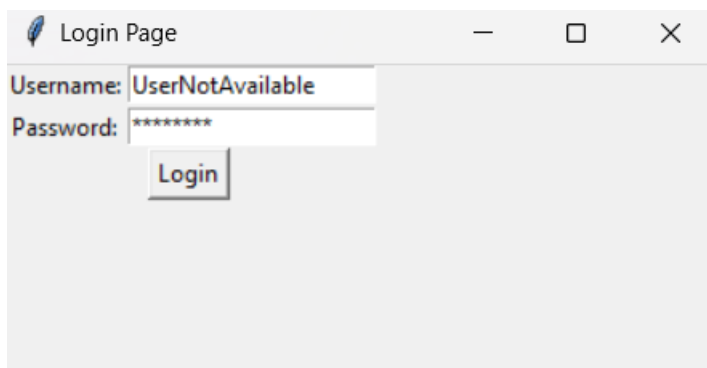
Conclusion: Designing a login page using a GUI in Python is a critical step in creating user-friendly applications. Key points include selecting a GUI library like Tkinter, using visual components (labels, entry fields, buttons), event-driven programming, layout management, user interaction elements, cross-platform compatibility, customization, database connectivity, and the importance of testing and security. A well-designed login page provides a secure and user-friendly entry point for applications.

Output:

Login Successful



Login Failed



2.To Design Student Information Form.

```
import tkinter as tk

def save_student_info():
    # Get data from input fields and save to a database or file
    name = entry_name.get()
    roll_number = entry_roll.get()
```



```
# Add data processing logic here
print(f"Saved Student: {name}, Roll: {roll_number}")

# Create the main window
window = tk.Tk()
window.title("Student Information Form")

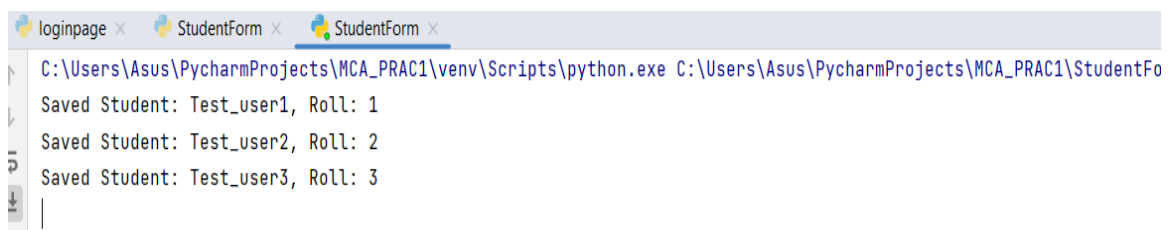
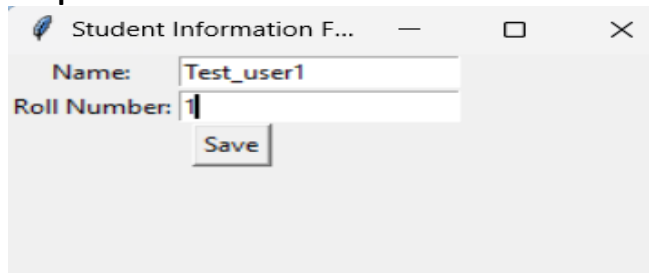
# Create labels and entry widgets for student data
label_name = tk.Label(window, text="Name:")
label_roll = tk.Label(window, text="Roll Number:")
entry_name = tk.Entry(window)
entry_roll = tk.Entry(window)
save_button = tk.Button(window, text="Save", command=save_student_info)

# Place widgets on the window using the grid layout
label_name.grid(row=0, column=0)
entry_name.grid(row=0, column=1)
label_roll.grid(row=1, column=0)
entry_roll.grid(row=1, column=1)
save_button.grid(row=2, columnspan=2)

window.mainloop()
```

Conclusion: Designing a Student Information Form using a GUI in Python allows for efficient data collection and management. It simplifies user input, streamlines data processing, and enhances user experience, making it an essential component for educational institutions and data-driven applications.

Output:



3.Implement Database connectivity For Login Page i.e. Connect Login GUI with Sqlite3

```
import tkinter as tk
import sqlite3
from tkinter import messagebox

# Function to check login credentials
def login():
    username = entry_username.get()
    password = entry_password.get()
```

```
# Connect to the SQLite database
conn = sqlite3.connect("user_credentials.db")
cursor = conn.cursor()

cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
("Test", "Test123"))
cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
("Prasad", "1234"))

cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY,
                username TEXT NOT NULL,
                password TEXT NOT NULL
            )''')

# Check if the user exists in the database
cursor.execute("SELECT * FROM users WHERE username=? AND password=?",
(username, password))
user = cursor.fetchone()

if user:
    messagebox.showinfo("Login Status", "Login Successful")
else:
    messagebox.showerror("Login Status", "Login Failed")

# Close the database connection
conn.close()

# Create the main window
window = tk.Tk()
window.title("Login Page")

# Create labels for username and password
label_username = tk.Label(window, text="Username:")
label_password = tk.Label(window, text="Password:")

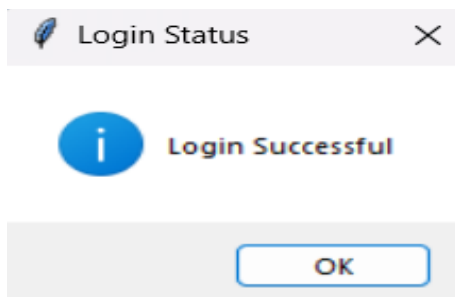
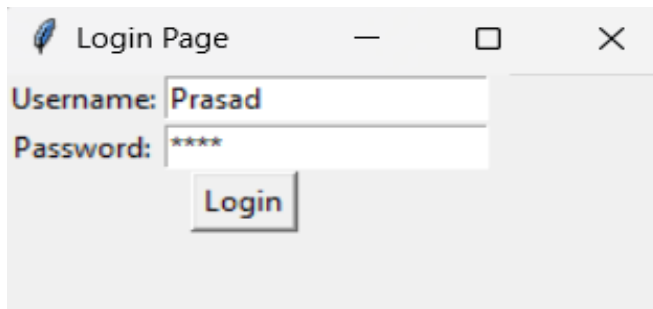
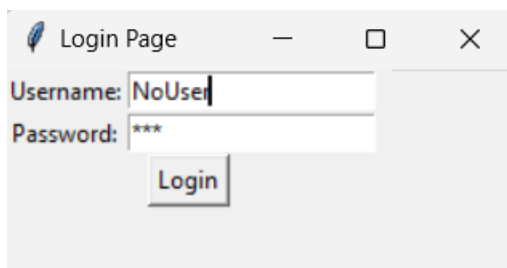
# Create entry widgets for user input
entry_username = tk.Entry(window)
entry_password = tk.Entry(window, show="*") # Show '*' for password input

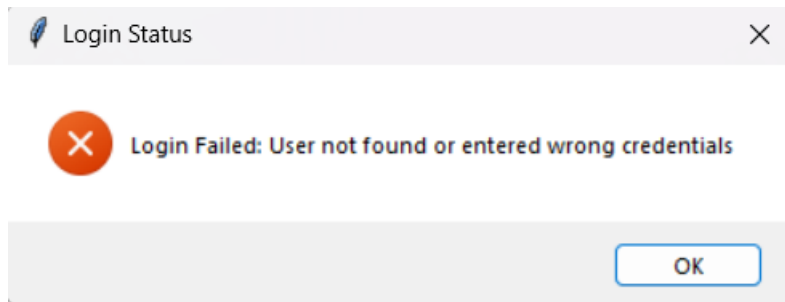
# Create a login button
login_button = tk.Button(window, text="Login", command=login)

# Place widgets on the window using the grid layout
label_username.grid(row=0, column=0)
entry_username.grid(row=0, column=1)
label_password.grid(row=1, column=0)
entry_password.grid(row=1, column=1)
login_button.grid(row=2, columnspan=2)

# Start the Tkinter main loop
window.mainloop()
```

Connecting a Login GUI with SQLite3 database enables secure user authentication. It enhances data storage and retrieval capabilities, ensuring user credentials are validated effectively, making it a crucial feature for login systems in Python applications.

Output:**Login Page****User available in sqllite3 database****User not available in sqllite3 database**



Name: Ravikumar Rai

FYMCA/A

Roll.No.42

Name of Student : Ravikumar Rai

Roll Number :42

Practical No: 8

Title of LAB Assignment: To implement Threads in Python

DOP : 31-10-2023

DOS : 01-11-2023

CO Mapped:

CO5

PO Mapped:

PO3, PO5, PSO1, PSO2

Signature:

Aim: To implement Threads in Python

1. To do design the program for starting the thread in python
2. Write a program to illustrate the concept of Synchronization
3. Write a program for creating multithreaded priority queue

Description:**Threading:**

Threading is a powerful and essential concept in computer programming that allows for concurrent execution of multiple threads within a single program. Threads are lightweight processes that share the same memory space and can run independently, enabling various functionalities and features. Here are some of the key functionality and features of threading:\

Concurrency: Threading provides a way to achieve concurrency in a program. Concurrency allows multiple tasks to make progress and execute independently, leading to improved performance in tasks like I/O-bound operations and responsiveness in applications.

Parallelism: While threading allows for concurrency, it's important to note that due to the Global Interpreter Lock (GIL) in Python, true parallelism in CPU-bound operations is limited. However, threading can still provide benefits when handling I/O-bound tasks, where threads can run in parallel and make better use of multi-core processors.

Creation and Management: Threading in Python is made straightforward through the built-in threading module. This module offers classes and functions for creating, starting, stopping, and managing threads.

Thread Safety: Threads can access shared resources, and this shared access can lead to race conditions and data corruption. Threading provides mechanisms like locks, semaphores, and conditions to ensure thread safety and proper synchronization.

Parallel Task Execution: Threading allows you to break down a larger task into smaller subtasks and execute them in parallel. This is particularly useful in scenarios where you have a set of similar tasks to perform concurrently.

Responsive User Interfaces: Threading is often used in graphical user interfaces (GUIs) to keep the user interface responsive while performing time-consuming operations in the background. This prevents the UI from freezing and improves the user experience.

Resource Sharing: Threads can share resources and data with each other, making it easier to pass information between different parts of your program. However, care must be taken to ensure data integrity through synchronization.

Deadlock Avoidance: Threading provides tools for managing potential deadlocks, a situation where threads wait indefinitely for each other to release resources.

Techniques like timeout-based locks and deadlock detection are available for deadlock avoidance.

Distributed Computing: Threading can be used to implement distributed computing and parallel processing. It is often used in scenarios like web scraping, where multiple threads can fetch data from multiple sources simultaneously.

Event Handling: Threads can be used to handle events, such as responding to user input or reacting to changes in data, in a responsive and timely manner.

Load Balancing: Threading can be employed in load balancing scenarios where multiple threads are used to distribute and process tasks among different resources, such as CPU cores.

Task Prioritization: Threading allows you to assign priorities to threads, ensuring that critical tasks are executed before less important ones.

Example:

```
import threading

def print_numbers():
    for i in range(1, 6):
        print(f"Number {i}")

def print_letters():
    for letter in 'abcde':
        print(f"Letter {letter}")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Both threads have finished.")
```

Synchronization:

Synchronization is the process of coordinating the activities of multiple threads to ensure that they access shared resources or data in a controlled and orderly manner. In a multithreaded environment, synchronization is essential to prevent issues like data corruption, race conditions, and deadlocks. Python offers several mechanisms for synchronization, including locks, semaphores, and condition variables.

Example:

```
import threading

total = 0
lock = threading.Lock()

def increment_total():
    global total
    for _ in range(1000000):
        with lock:
            total += 1

def decrement_total():
    global total
    for _ in range(1000000):
        with lock:
            total -= 1

thread1 = threading.Thread(target=increment_total)
thread2 = threading.Thread(target=decrement_total)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Final total:", total)
```

Multithreading in Python:

Multithreading in Python is the process of running multiple threads within a Python program. This enables concurrent execution and can significantly improve the performance of certain types of applications. Python's Global Interpreter Lock (GIL) can limit the true parallelism of threads, particularly in CPU-bound operations, but it's still beneficial for I/O-bound tasks.

1. To do design the program for starting the thread in python**Code:**

```
import threading

# Define a function that will be executed in the thread
def my_thread_function():
    print("Thread is running")

# Create a thread object and pass the function to it
my_thread = threading.Thread(target=my_thread_function)

# Start the thread
my_thread.start()

# You can do other work here in the main thread
print("Main thread continues to run")

# Wait for the thread to finish (if needed)
my_thread.join()

# The program continues to execute here
print("Main thread has finished")
```

Conclusion:

The provided code showcases the fundamental principles of threading in Python. A new thread is created, and the `my_thread_function` is executed concurrently with the main thread. Threading allows for parallel execution, enabling the main thread to continue its tasks while the new thread performs its designated function. Proper synchronization with `join()` ensures orderly program termination. This simple example illustrates the power of threading in enhancing program efficiency and responsiveness by executing tasks concurrently.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\ADMIN\
```

```
Thread is running
```

```
Main thread continues to run
```

```
Main thread has finished
```

2. Write a program to illustrate the concept of Synchronization**Code:**

```
import threading

# Shared resource (a counter)
counter = 0

# Define a lock for synchronization
counter_lock = threading.Lock()

# Function to increment the counter safely
def increment_counter():
    global counter
    with counter_lock:
        for _ in range(1000000):
            counter += 1

# Create two threads to increment the counter
thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)

# Start both threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()
```

```
# Print the final counter value
```

```
print("Final counter value:", counter)
```

Conclusion:

The provided code demonstrates the use of threading and synchronization to increment a shared counter safely. Two threads concurrently execute the `increment_counter` function, each adding a million to the counter. A lock (`counter_lock`) is used to ensure exclusive access to the shared resource, preventing race conditions. This example showcases the importance of synchronization in multithreaded applications to maintain data integrity. The final counter value reflects the successful coordination of the threads, emphasizing the power of threading and synchronization in concurrent programming.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users'  
Final counter value: 2000000
```

3. Write a program for creating multithreaded priority queue**Code:**

```
import queue
import threading
import time
import random

# Create a priority queue
priority_queue = queue.PriorityQueue()

# Create a flag to signal the retrieval thread to stop
stop_retrieval = threading.Event()

# Function to insert items into the priority queue
def insert_into_queue():
    for i in range(5):
        item = random.randint(1, 100)
        priority_queue.put((item, f"Item {item}"))
        time.sleep(1)

# Function to retrieve items from the priority queue
def retrieve_from_queue():
    while not stop_retrieval.is_set():
        try:
            item = priority_queue.get(timeout=1)
            print(f"Retrieved: {item[1]}")
            priority_queue.task_done()
        except queue.Empty:
            continue
```

```
# Create multiple threads for inserting and retrieving items
insert_thread = threading.Thread(target=insert_into_queue)
retrieve_thread = threading.Thread(target=retrieve_from_queue)

# Start the threads
insert_thread.start()
retrieve_thread.start()

# Wait for the insert_thread to finish
insert_thread.join()

# Set the stop_retrieval flag to signal the retrieve_thread to stop
stop_retrieval.set()

# Wait for the retrieve_thread to finish
retrieve_thread.join()
```

Conclusion:

The provided code demonstrates the use of the `queue.PriorityQueue` and `threading` in Python for concurrent data insertion and retrieval. An insertion thread adds random items with priorities to the priority queue, while a retrieval thread continuously retrieves and prints items. The code showcases the importance of synchronization and coordination in a multithreaded environment, as the insertion and retrieval threads work together. Additionally, the use of the `threading.Event` allows for graceful termination of the retrieval thread. This example highlights how Python's `threading` and `queue` modules can be used to manage shared resources and safely handle concurrent operations.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\A
```

```
Retrieved: Item 12
```

```
Retrieved: Item 2
```

```
Retrieved: Item 71
```

```
Retrieved: Item 30
```

```
Retrieved: Item 77
```

Name: Ravikumar Rai

FYMCA/A

Roll.No.42

Name of Student : Ravikumar Rai

Roll Number : 42

Practical No: 9

Title of LAB Assignment: To implement NumPy library in Python

DOP : 31-10-2023

DOS : 01-11-2023

CO Mapped:

CO6

PO Mapped:

PO3, PO5, PSO1, PSO2

Signature:

Aim: To implement NumPy library in Python

1. Creating ndarray objects using array() in NumPy
2. Creating 2D arrays to implement Matrix Multiplication.
3. Program for Indexing and slicing in NumPy arrays.
4. To implement NumPy - Data Types

Description:**NumPy:**

NumPy, short for "Numerical Python," is a powerful open-source Python library used for numerical and scientific computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is a fundamental library for data manipulation in various scientific and engineering applications and is an essential building block for many other Python libraries like SciPy, pandas, and scikit-learn.

Key Functionality of NumPy:

Multidimensional Arrays: NumPy's core data structure is the `numpy.ndarray`, which is an efficient, homogeneous N-dimensional array that can store elements of the same data type. These arrays are highly memory-efficient and enable fast element-wise operations.

Mathematical Operations: NumPy provides a wide range of mathematical functions for array manipulation. You can perform element-wise operations, vectorized calculations, and linear algebra operations easily. For example:

Example:

```
# Creating arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise addition
result = a + b
print(result) # [5 7 9]

# Dot product
dot_product = np.dot(a, b)
print(dot_product) # 32
```

Broadcasting: NumPy allows you to perform operations on arrays of different shapes. It automatically broadcasts the smaller array to match the shape of the larger array, making element-wise operations more flexible.

Example:

```
a = np.array([1, 2, 3])
b = 2
```

```
result = a + b # Broadcasting b to match the shape of a
print(result) # [3 4 5]
```

Array Slicing and Indexing: NumPy supports powerful slicing and indexing capabilities, making it easy to extract and manipulate subsets of arrays.

Example:

```
arr = np.array([0, 1, 2, 3, 4, 5])
```

```
# Slicing
subset = arr[2:5]
print(subset) # [2 3 4]
```

```
# Indexing
element = arr[1]
print(element) # 1
```

Random Number Generation: NumPy includes functions for generating random numbers from various probability distributions, which is useful for simulations and statistics.

Example

```
# Generate an array of random numbers from a uniform distribution
```

```
random_data = np.random.uniform(0, 1, size=(3, 3))
print(random_data)
```

Linear Algebra Operations: NumPy provides functions for linear algebra operations, making it a valuable tool for tasks like matrix multiplication, eigenvalue computation, and solving systems of linear equations.

Example

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
product = np.dot(A, B)
print(product)
```

Aggregation and Statistical Functions: NumPy provides numerous functions for aggregating data and performing statistical calculations, including mean, median, standard deviation, variance, and more.

Example:

```
data = np.array([1, 2, 3, 4, 5])
```

```
mean = np.mean(data)
```

```
std_dev = np.std(data)
```

Array Manipulation: NumPy allows you to reshape, transpose, concatenate, and split arrays easily. These operations are helpful for data preprocessing and manipulation.

Example:

```
array = np.array([[1, 2], [3, 4]])
```

```
# Reshape
```

```
reshaped = array.reshape((4, 1))
```

```
# Transpose
```

```
transposed = array.T
```

```
# Concatenate
```

```
concatenated = np.concatenate((array, array), axis=0)
```

NumPy's efficiency, versatility, and extensive mathematical capabilities make it an essential library for data analysis, machine learning, scientific research, and more in the Python ecosystem.

1. Creating ndarray objects using array() in NumPy

Code:

```
import numpy as np

# Creating a 1D array
arr1d = np.array([1, 2, 3, 4, 5])

# Creating a 2D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print ("One Dimensional Array\n", arr1d)
print ("Two Dimensional Array\n", arr2d)
```

Conclusion:

In this code snippet, we demonstrate the fundamental capabilities of NumPy, a powerful numerical library in Python. We start by creating both a one-dimensional array (arr1d) and a two-dimensional array or matrix (arr2d). NumPy's np.array function allows for efficient storage and manipulation of data, whether in a simple list or a more complex structure like a matrix. This code provides a basic illustration of how NumPy is used to work with arrays, which is just the tip of the iceberg when it comes to the extensive functionality that NumPy offers for numerical and scientific computing in Python.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\ADMIN\
One Dimensional Array
[1 2 3 4 5]
Two Dimensional Array
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

2. Creating 2D arrays to implement Matrix Multiplication.

Code:

```
import numpy as np

matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

result = np.dot(matrix_a, matrix_b)
print("np.dot(matrix_a, matrix_b): \n", result)

# or

result = matrix_a.dot(matrix_b)
print("matrix_a.dot(matrix_b): \n", result)
```

Conclusion:

In this code snippet, we delve into the power of NumPy for matrix operations. Two matrices, `matrix_a` and `matrix_b`, are created using NumPy arrays. We then utilize the `np.dot` function to perform matrix multiplication, which is a fundamental operation in linear algebra. The result is a new matrix that represents the product of the two original matrices. The code showcases the versatility of NumPy for matrix computations and provides two different ways to achieve the same result: using `np.dot(matrix_a, matrix_b)` or the `dot` method directly on the array, `matrix_a.dot(matrix_b)`. NumPy simplifies complex mathematical operations, making it an essential tool for scientific and engineering applications in Python.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\ADMIN\
np.dot(matrix_a, matrix_b):
[[19 22]
 [43 50]]
matrix_a.dot(matrix_b):
[[19 22]
 [43 50]]
```

3. Program for Indexing and slicing in NumPy arrays.**Code:**

```
import numpy as np

arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing
element = arr1d[2] # Access the element at index 2
print("Access the element at index 2 : ", element)

# Slicing
slice1 = arr1d[1:4] # Slice elements from index 1 to 3
print("Slice elements from index 1 to 3 : ", slice1)

slice2 = arr2d[0, 1:3] # Slice elements from the first row, columns 1 to 2
print("Slice elements from the first row, columns 1 to 2 : ", slice2)
```

Conclusion:

In this code snippet, we explore the basics of NumPy for array indexing and slicing. Two arrays, `arr1d` and `arr2d`, are created using NumPy. The code demonstrates how to access individual elements and slices of these arrays. Using index notation, we access the element at index 2 in the one-dimensional array and slice elements from index 1 to 3. Similarly, in the two-dimensional array, we perform a more complex slice operation by specifying both row and column indices. NumPy simplifies data manipulation by providing efficient ways to access and extract specific elements or subsets from arrays, which is fundamental for data analysis and scientific computing in Python.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\
Access the element at index 2 : 3
Slice elements from index 1 to 3 : [2 3 4]
Slice elements from the first row, columns 1 to 2 : [2 3]
```

4. To implement NumPy - Data Types

Code:

```
import numpy as np

# Specifying data types when creating arrays
arr_int = np.array([1, 2, 3], dtype=np.int32)
arr_float = np.array([1.0, 2.0, 3.0], dtype=np.float64)

# Check the data type of an array
print(arr_int.dtype)
print(arr_float.dtype)

# You can also specify the data type for individual elements
arr_custom = np.array([1, 2, 3], dtype=np.uint16)

print(arr_custom)
```

Conclusion:

This code snippet illustrates how to specify and work with different data types in NumPy arrays. By using the dtype parameter during array creation, you can control the data type of the elements within the array. In this example, we create arrays arr_int and arr_float with explicit data types of int32 and float64, respectively. We then check and print the data types of these arrays using the dtype attribute. Additionally, the code showcases that you can specify the data type for individual elements within an array, as demonstrated with the creation of the arr_custom array with a uint16 data type. NumPy's ability to handle various data types allows for precise control over memory usage and numerical accuracy in scientific and numerical computing tasks.

Output:

```
C:\Users\ADMIN\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\ADMIN\  
int32  
float64  
[1 2 3]
```


Name of Student: Ravikumar Rai**Roll Number** 42**LAB Assignment Number:** 10**Title of LAB Assignment:** To implement Pandas library in Python.**DOP:** 31/10/2023**DOS:** 1/11/2023**CO Mapped:**
CO6**PO Mapped:**
PO3,PO5,PSO1,PS
O2**Signature:****Marks:**

Practical 10

Aim:

1. Write a Pandas program to create and display a one dimensional array like object containing an array of data using pandas module.
2. Write a pandas program to convert a dictionary to a Pandas Series.
3. Write a pandas program to create a dataframe from a dictionary and display it. sample data: {'X': [78,85,96,80,86], 'Y': [84,94,89,83,86], 'Z': [86,97,96,72,83]}
4. Write a pandas program to aggregate the two given dataframes along rows and assign all data.
5. write a pandas program to merge two given dataframes with different columns.

Description:

Creating and displaying a one-dimensional array-like object using the Pandas module involves the following key concepts:

1. Pandas Library:

Pandas is a powerful open-source data manipulation and analysis library for Python.

It provides data structures and functions for working with structured data, including Series and DataFrame.

2. Pandas Series:

A Pandas Series is a one-dimensional labeled array that can hold data of any data type.

It is similar to a NumPy array, but with the added feature of having an index associated with each element.

3. Creating a Pandas Series:

To create a Pandas Series, you can use the `pd.Series()` constructor, which accepts a variety of data types, including lists, dictionaries, and NumPy arrays.

The Series constructor allows you to specify the data and optionally customize the index.

4. Displaying a Pandas Series:

You can display the contents of a Pandas Series simply by using the `print()` function or by calling the Series object directly.

When you display a Pandas Series, you'll see both the data values and the associated index (default is an integer index).

Here's a breakdown of the steps in the Pandas program to create and display a one-dimensional array-like object:

Import the Pandas library using `import pandas as pd`.

Create an array of data (e.g., a list) that you want to convert into a Pandas Series. Use the `pd.Series()` constructor to convert the array of data into a Pandas Series. Optionally, you can

customize the index by providing it as an argument when creating the SeriesDisplay the Pandas Series using the print() function or by calling the Series object directly.

converting a dictionary to a Pandas Series using the `pd.Series()` constructor is a straightforward process. This allows you to leverage the powerful features of Pandas, such as data analysis, filtering, and manipulation, with your structured data. The resulting Series is a labeled one-dimensional data structure that can be accessed, indexed, and modified to suit your specific data analysis needs.

A Pandas DataFrame can hold data of various types, including integers, floating-point numbers, strings, or more complex objects.

Each column in the DataFrame can have its own data type, depending on the data provided.

In summary, creating a Pandas DataFrame from a dictionary is a powerful way to work with structured data in Python. It allows you to organize and manipulate data in a tabular format, making it easier to perform data analysis, filtering, and other operations on your data. The resulting DataFrame can be used for various data analysis tasks and is a fundamental building block in data science and data engineering.

1. Write a Pandas program to create and display a one dimensional array like object containing an array of data using pandas module.

Code:

```
import pandas as pd
data = [10, 20, 30, 40, 50]
my_series = pd.Series(data)
print(my_series)
```

Conclusion: Here we have successfully created and displayed a one dimensional array like object containing an array using pandas

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

2. Write a pandas program to convert a dictionary to a Pandas Series.

Code:

```
import pandas as pd
```

```
# Sample dictionary
data_dict = {'A': 10, 'B': 20, 'C': 30, 'D': 40, 'E': 50}

# Convert the dictionary to a Pandas Series
my_series = pd.Series(data_dict)

# Display the Pandas Series
print(my_series)
```

Conclusion: Here we have successfully converted a dictionary to a Pandas Series.

Output:

```
py
A    10
B    20
C    30
D    40
E    50
dtype: int64
```

3. Write a pandas program to create a dataframe from a dictionary and display it. sample data: {'X': [78,85,96,80,86], 'Y': [84,94,89,83,86], 'Z': [86,97,96,72,83]}

Code:

```
import pandas as pd

# Sample data dictionary
data_dict = {'X': [78, 85, 96, 80, 86], 'Y': [84, 94, 89, 83, 86], 'Z': [86, 97, 96, 72, 83]}

# Create a Pandas DataFrame from the sample data
df = pd.DataFrame(data_dict)

# Display the DataFrame
print(df)
```

Conclusion: Here we have successfully created a dataframe from a dictionary for the given sample data.

Output:

```

X  Y  Z
0  78 84 86
1  85 94 97
2  96 89 96
3  80 83 72
4  86 86 83
```

4. Write a pandas program to aggregate the two given dataframes along rows and assign all data.

Code:

```
import pandas as pd

# Sample data for two DataFrames
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12]}

# Create the first DataFrame
df1 = pd.DataFrame(data1)

# Create the second DataFrame
df2 = pd.DataFrame(data2)

# Concatenate the two DataFrames along rows and assign all data
result = pd.concat([df1, df2])

# Display the aggregated DataFrame
print(result)
```

Conclusion: Here we have successfully i aggregated the two given dataframes along rows and assign all data.

Output:

```

A  B
0  1  4
1  2  5
2  3  6
0  7 10
1  8 11
2  9 12
```

5. write a pandas program to merge two given dataframes with different columns.

Code:

```
import pandas as pd

# Sample data for two DataFrames
data1 = {'ID': [1, 2, 3], 'Name': ['Sahil', 'Vira', 'Pranav']}
data2 = {'ID': [2, 3, 4], 'Age': [25, 30, 35]}

# Create the first DataFrame
df1 = pd.DataFrame(data1)

# Create the second DataFrame
df2 = pd.DataFrame(data2)

# Merge the two DataFrames on the 'ID' column
merged_df = pd.merge(df1, df2, on='ID', how='inner')

# Display the merged DataFrame
print(merged_df)
```

Conclusion: Here we have successfully merged two given dataframes with different columns.

Output:

```
py
   ID  Name  Age
0   2   Vira  25
1   3  Pranav  30
```