ASSIGNMENT – 7.3

2303A51525

Batch-10

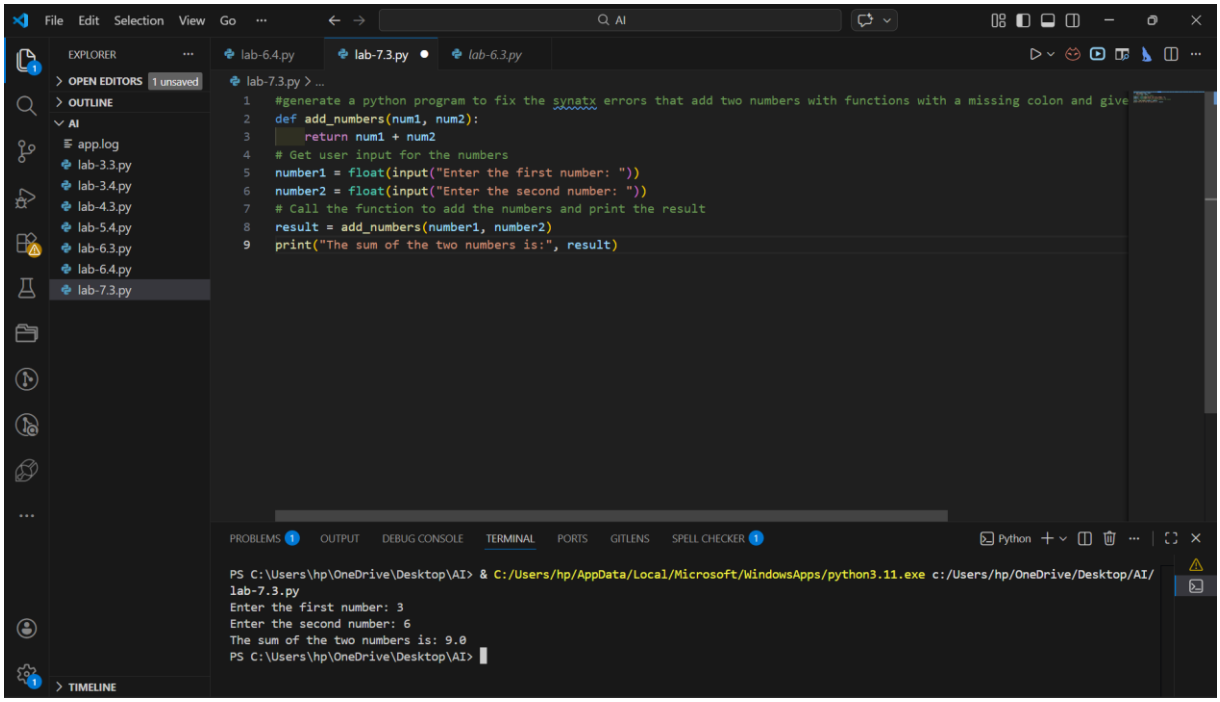
Task-1

Prompt: generate a python program to fix the synatx errors that add two numbers with functions with a missing colon and give user input for the numbers.

code :

```
def add_numbers(num1, num2):     return num1 +
num2 # Get user input for the numbers number1 =
float(input("Enter the first number: ")) number2 =
float(input("Enter the second number: "))
# Call the function and display the result result =
add_numbers(number1, number2) print("The sum of",
number1, "and", number2, "is:", result)
```

Output :



Code Analysis:

- The function add_numbers() takes two parameters and returns their sum.
- The missing colon after the function definition is corrected.
- User inputs are converted to float to allow decimal values.
- The function is called with user inputs and the result is printed.
- Using functions improves reusability and modular programming.

Task-2

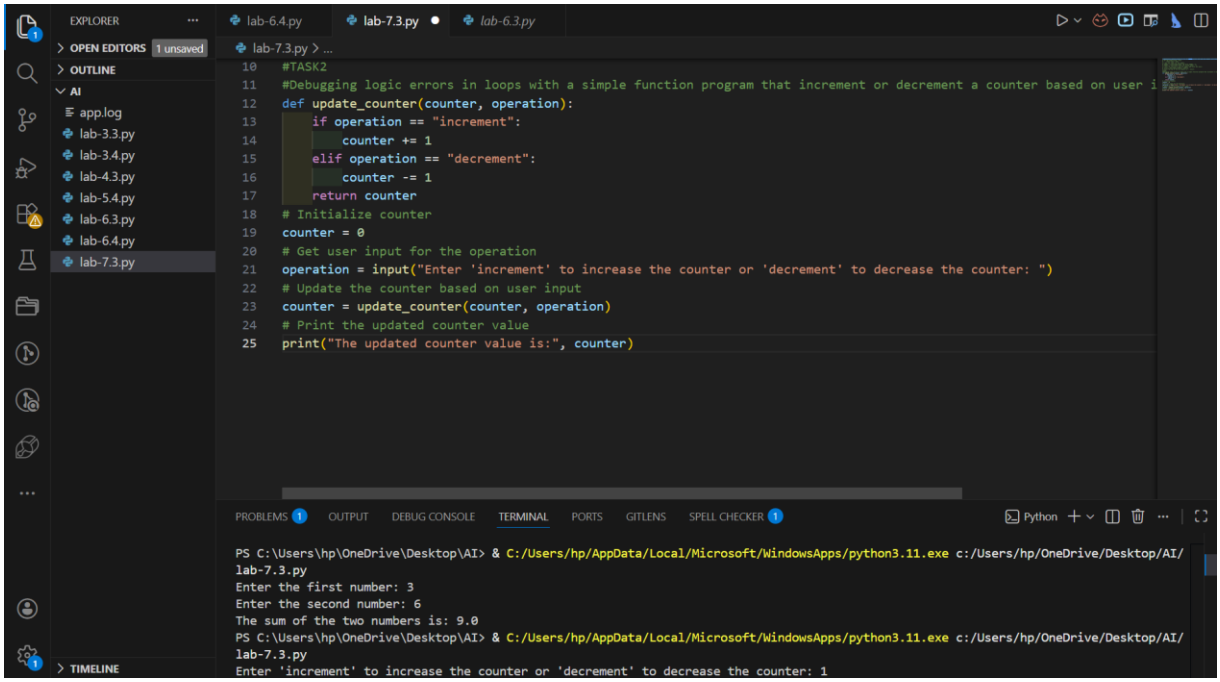
Prompt: Debugging logic errors in loops with a simple function program that increment or decrement a counter based on user input.

Code :

```
def update_counter(counter, action):
if action == 'increment':        return
counter + 1     elif action ==
'decrement':
```

```
        return counter - 1
    else:
        return counter #
Initialize counter
counter = 0
# Taking user input for action action = input("Enter
action (increment/decrement): ")
# Updating counter based on user input and printing
the result counter = update_counter(counter, action)
print(f"Counter value after {action}: {counter}")
```

Output :



Code Analysis:

- The function modifies the counter based on user action.
- action.lower() avoids case-sensitivity issues.
- If invalid input is entered, the counter remains unchanged.
- The logic ensures proper increment/decrement functionality.

- This demonstrates basic debugging of logical conditions.

Task-3

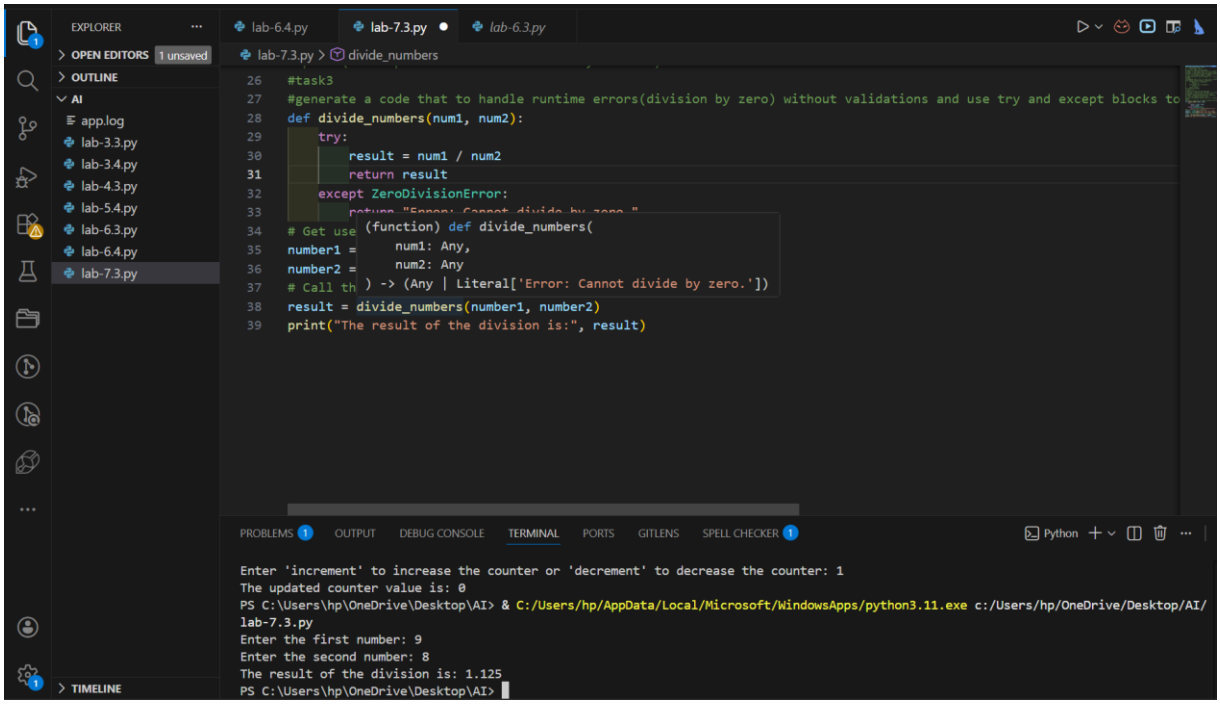
Prompt: generate a code that to handle runtime errors(division by zero) without validations and use try and except blocks to catch the error. take user input with functions

Code : def

```
divide_numbers(num1, num2):
    try:
        result = num1 / num2        return result
except ZeroDivisionError:        return "Error:
Division by zero is not allowed."
# Get user input for the numbers
number1 = float(input("Enter the numerator: "))
number2 = float(input("Enter the denominator:
")) # Call the function and display the result result
= divide_numbers(number1, number2)
print("The result of dividing", number1, "by", number2, "is:", result)
```

Output:

Code Analysis :



- The function attempts division inside a try block.
- If the denominator is zero, ZeroDivisionError is caught.
- The program does not crash due to exception handling.
- A user-friendly error message is returned instead.
- try-except ensures runtime stability.

Task-4

Prompt: #generate a code to debug the class definition errors for a rectangle .provide a class definition with missing self-parameter and correct it using __init_ method and explain why self is used in class definitions .take user input

Code :

class Rectangle: def

__init__(self, width, height):

self.width = width

self.height = height def

area(self):

return self.width * self.height # Get user input for

width and height width = float(input("Enter the width of

the rectangle: ")) height = float(input("Enter the height

of the rectangle: ")) # Create an instance of the Rectangle

class rectangle = Rectangle(width, height) # Calculate

and display the area of the rectangle print("The area of

the rectangle is:", rectangle.area())

Explanation: The self parameter is used in class definitions

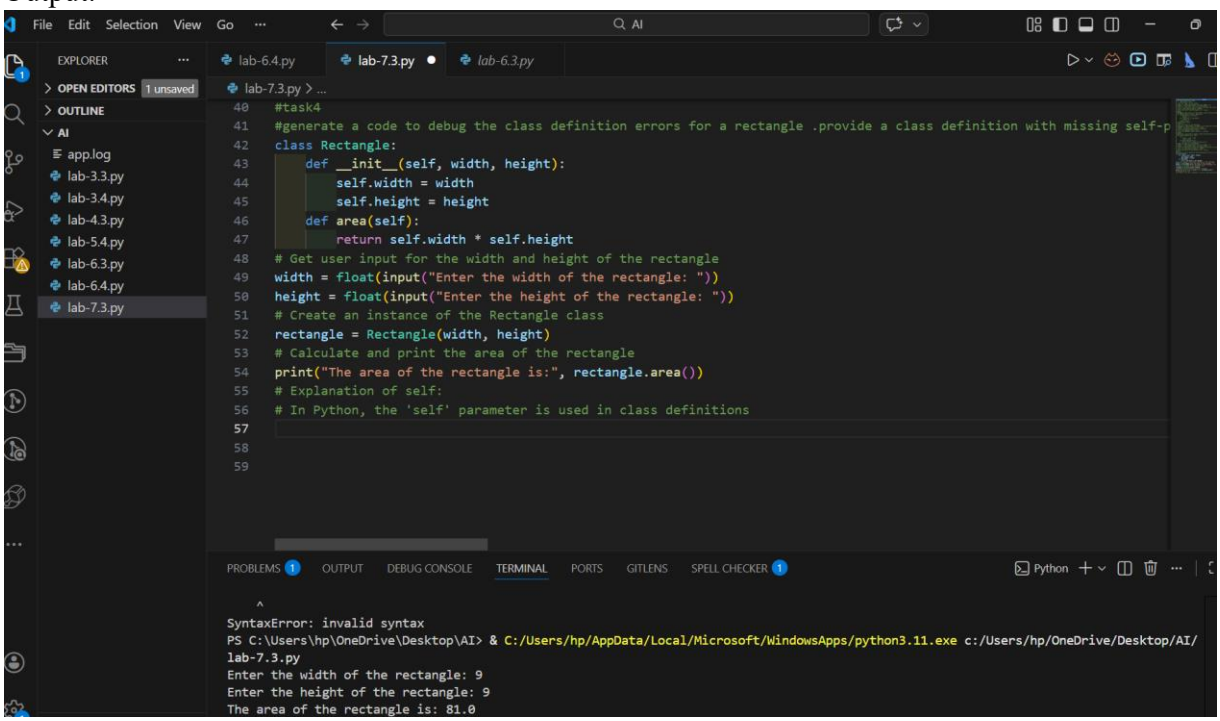
to refer to the instance of the class. It allows us to access and

modify the attributes of the instance.

In the __init__ method, we use self to assign the width and height

values to the instance variables.

Output:



Code Analysis :

- The constructor method must be __init__ (double underscores).
- self refers to the current object instance.
- Instance variables (self.width, self.height) store object data.
- The area() method accesses instance variables using self.
- Without self, Python cannot link data to the specific object.

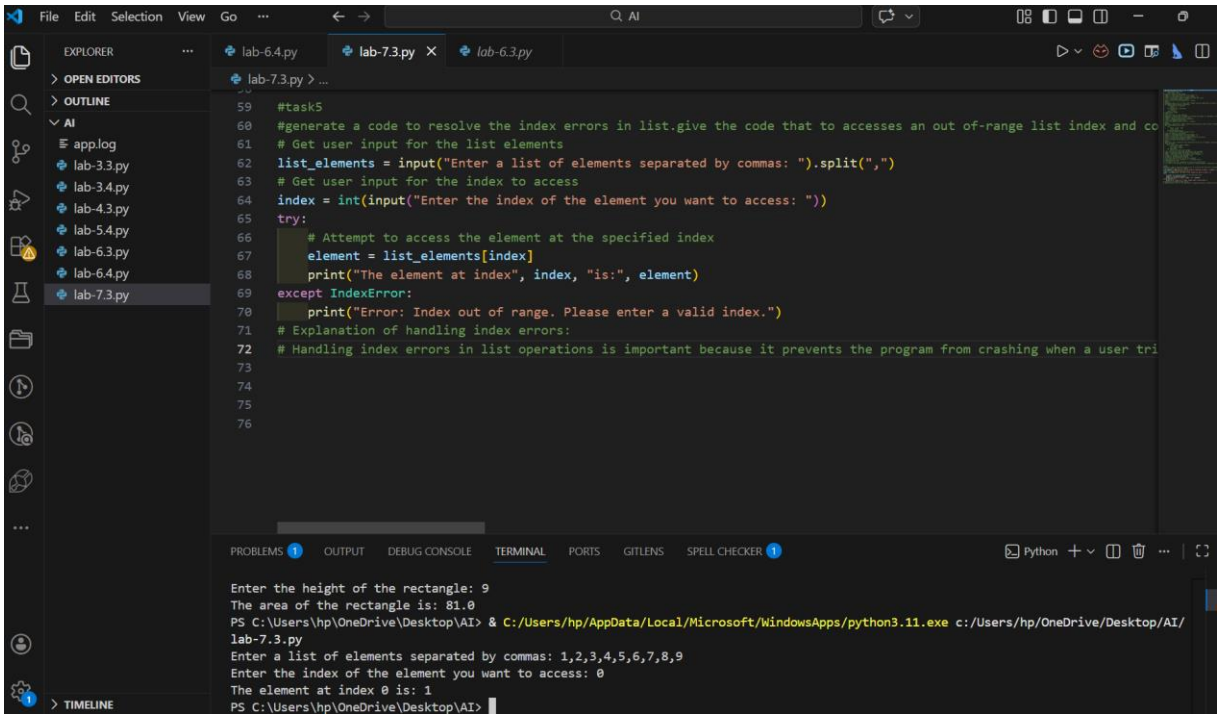
Task-5

Prompt: generate a code to resolve the index errors in list.give the code that to accesses an out of-range list index and correct it by using exception handling and explain the importance of handling index errors in list operations. take user input for list elements.

```
my_list = [1, 2, 3] try:
    # Attempting to access an out-of-range index
print(my_list[5]) except IndexError:
    print("Error: Index out of range. Please provide a valid index.")
# Get user input for list elements user_input = input("Enter a list of
numbers separated by commas: ")
# Convert the user input into a list of integers my_list
= [int(x.strip()) for x in user_input.split(",")] #
Attempt to access an index based on user input
try:
    index = int(input("Enter the index you want to access: "))
print("Element at index", index, "is:", my_list[index]) except
IndexError:
    print("Error: Index out of range. Please provide a valid index.")
# Explanation: Handling index errors in list operations is important because it prevents the
program from crashing when an invalid index is accessed. By using exception handling,
```

we can catch the error and provide a user-friendly message, allowing the program to continue running smoothly even when unexpected input is encountered.

Output :



Code Analysis :

- User input is converted into a list using split() and list comprehension.
- The program attempts to access a user-specified index.
- If index is invalid, IndexError is handled gracefully.
- ValueError ensures proper numeric input.
- Exception handling prevents program crashes and improves reliability