

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 :

```

1  #generate a python program to fix the syntax errors that add two numbers with functions with a missing colon and give
2  def add_numbers(num1, num2):
3      return num1 + num2
4  # Get user input for the numbers
5  number1 = float(input("Enter the first number: "))
6  number2 = float(input("Enter the second number: "))
7  # Call the function to add the numbers and print the result
8  result = add_numbers(number1, number2)
9  print("The sum of the two numbers is:", result)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER

PS C:\Users\hp\OneDrive\Desktop\AI> & C:/Users/hp/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/hp/OneDrive/Desktop/AI/lab-7.3.py
Enter the first number: 3
Enter the second number: 6
The sum of the two numbers is: 9.0
PS C:\Users\hp\OneDrive\Desktop\AI>

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

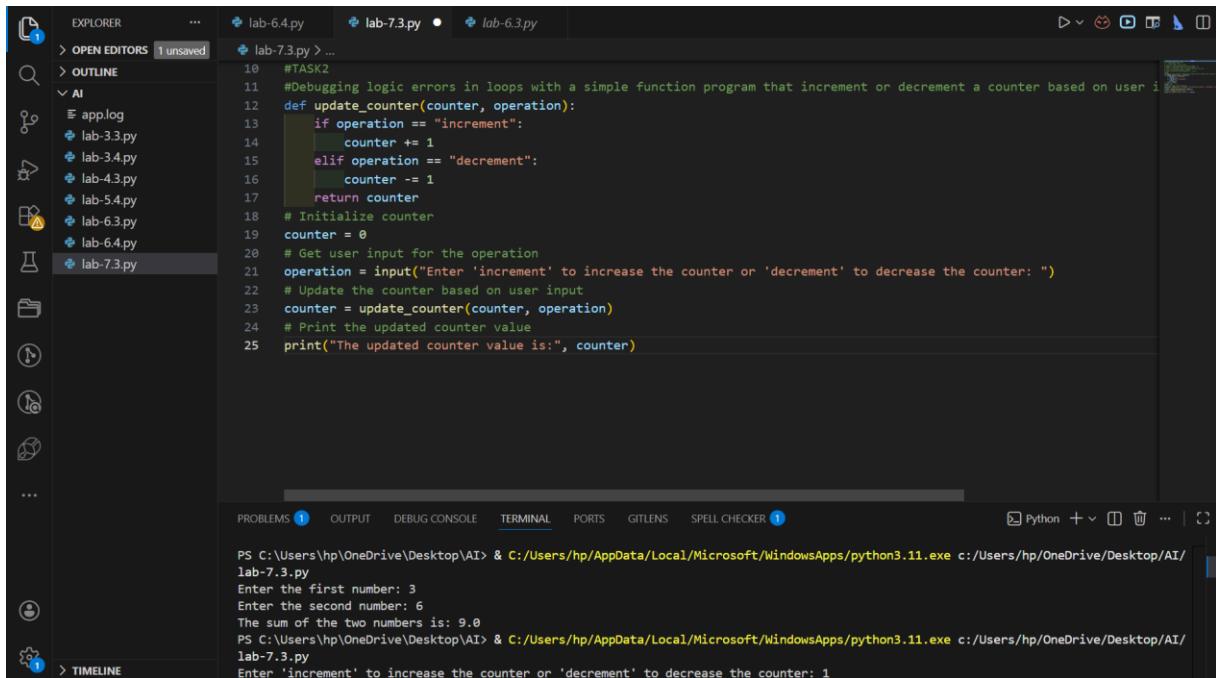
# Taking user input for action
action = input("Enter
action (increment/decrement): ")

# Updating counter based on user input and printing
the result
counter = update_counter(counter, action)

print(f"Counter value after {action}: {counter}")

```

Output :



The screenshot shows a code editor interface with the following details:

- EXPLORER:** Shows files in the workspace, including `app.log`, `lab-3.3.py`, `lab-3.4.py`, `lab-4.3.py`, `lab-5.4.py`, `lab-6.3.py`, `lab-6.4.py`, and `lab-7.3.py`.
- EDITOR:** Displays the Python script `lab-7.3.py` with the following code:

```

10 #TASK2
11 #Debugging logic errors in loops with a simple function program that increment or decrement a counter based on user i
12 def update_counter(counter, operation):
13     if operation == "increment":
14         counter += 1
15     elif operation == "decrement":
16         counter -= 1
17     return counter
18 # Initialize counter
19 counter = 0
20 # Get user input for the operation
21 operation = input("Enter 'increment' to increase the counter or 'decrement' to decrease the counter: ")
22 # Update the counter based on user input
23 counter = update_counter(counter, operation)
24 # Print the updated counter value
25 print("The updated counter value is:", counter)

```
- TERMINAL:** Shows the command-line output of running the script:

```

PS C:\Users\hp\OneDrive\Desktop\AI> & C:/Users/hp/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/hp/OneDrive/Desktop/AI/
lab-7.3.py
Enter the first number: 3
Enter the second number: 6
The sum of the two numbers is: 9.0
PS C:\Users\hp\OneDrive\Desktop\AI> & C:/Users/hp/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/hp/OneDrive/Desktop/AI/
lab-7.3.py
Enter 'increment' to increase the counter or 'decrement' to decrease the counter: 1

```

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 screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- Left Sidebar:** Shows the file tree with files like `app.log`, `lab-3.4.py`, `lab-4.3.py`, `lab-5.4.py`, `lab-6.3.py`, `lab-6.4.py`, and `lab-7.3.py`.
- Top Bar:** Shows tabs for `lab-6.4.py`, `lab-7.3.py` (the active tab), and `lab-6.3.py`. The status bar indicates there is 1 unsaved change.
- Code Editor:** Displays Python code for handling division by zero. The code defines a function `divide_numbers` that attempts to divide two numbers. If a `ZeroDivisionError` occurs, it returns a specific error message. The code also includes a type annotation for the parameters and a call to the function with sample values.
- Bottom Status Bar:** Shows the command palette icon, Python language icon, and other status indicators.
- Terminal:** Shows the output of running the script `lab-7.3.py`. It prompts the user to increment or decrement a counter, updates the counter value, and then prints the result of dividing 9 by 8, which is 1.125.

- 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:

The screenshot shows the Visual Studio Code interface. The left sidebar displays a file tree with several Python files: app.log, lab-3.3.py, lab-3.4.py, lab-4.3.py, lab-5.4.py, lab-6.3.py, lab-6.4.py, and lab-7.3.py. The main editor area contains the following Python code:

```

40     #generate a code to debug the class definition errors for a rectangle .provide a class definition with missing self-p
41     #task4
42     class Rectangle:
43         def __init__(self, width, height):
44             self.width = width
45             self.height = height
46         def area(self):
47             return self.width * self.height
48         # Get user input for the width and height of the rectangle
49         width = float(input("Enter the width of the rectangle: "))
50         height = float(input("Enter the height of the rectangle: "))
51         # Create an instance of the Rectangle class
52         rectangle = Rectangle(width, height)
53         # Calculate and print the area of the rectangle
54         print("The area of the rectangle is:", rectangle.area())
55         # Explanation of self:
56         # In Python, the 'self' parameter is used in class definitions
57
58
59

```

The terminal at the bottom shows the output of running the script:

```

^
SyntaxError: invalid syntax
PS C:\Users\hp\OneDrive\Desktop\AI> & C:/Users/hp/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/hp/OneDrive/Desktop/AI/
lab-7.3.py
Enter the width of the rectangle: 9
Enter the height of the rectangle: 9
The area of the rectangle is: 81.0

```

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 :

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files like lab-6.4.py, lab-7.3.py (the active file), and lab-6.3.py.
- Code Editor:** Displays Python code for handling index errors. The code prompts for a rectangle height, calculates its area, asks for a list of elements separated by commas, and then asks for an index to access. It handles `IndexError` and `ValueError`.
- Terminal:** Shows the command line output of the program's execution. It includes prompts for rectangle height, area calculation, list input, and index access, along with the resulting element printout.

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