ASSIGNMENT – 10.4

NAME: NALLALA SIRI

HT.NO: 2403A52037

BATCH: 03

TASK 1:

Identify and fix syntax, indentation, and variable errors in the given script.

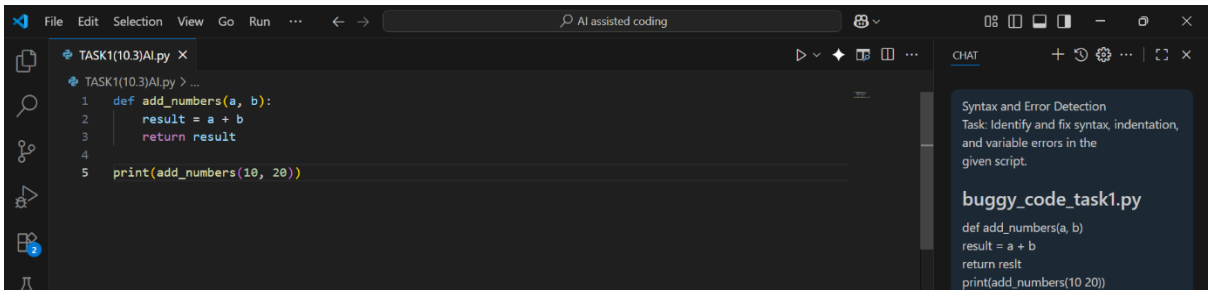
```
# buggy_code_task1.py
def add_numbers(a, b)
result = a + b
return reslt
print(add_numbers(10 20))
```

PROMPT:

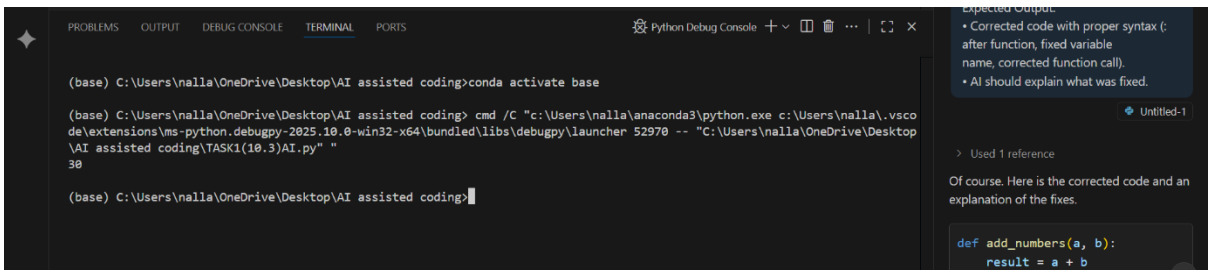
Detect and correct the syntax,identation and variable errors from the given code:

```
def add_numbers(a, b)
result = a + b
return reslt
print(add_numbers(10 20))
```

CODE:



OUTPUT:



OBSERVATION:

The colon is missed after the function definition and didn't indent the code block. It also had a typo in the return statement, using reslt instead of result. Finally, the comma between the arguments at the add_numbers function.

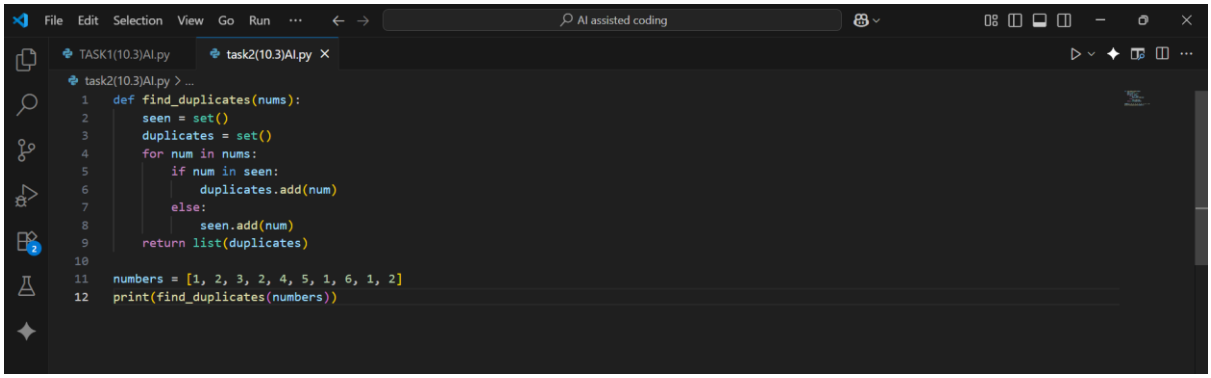
TASK 2:

Optimize inefficient logic while keeping the result correct.
# buggy_code_task2.py
def find_duplicates(nums):
duplicates = []
for i in range(len(nums)):
for j in range(len(nums)):
if i != j and nums[i] == nums[j] and nums[i] not in duplicates:
duplicates.append(nums[i])
return duplicates
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))

PROMPT:

Write a Python function to find all duplicate elements in a list. The function should take a list of numbers as input and return a list containing only the numbers that appear more than once. Your solution should be efficient and avoid nested loops. For example, for the input [1, 2, 3, 2, 4, 5, 1, 6, 1, 2], the function should return [1, 2].

CODE:



OUTPUT:

OBSERVATION:

The Counter class from this module is specifically designed to count hashable objects, making it perfect for this task. By creating a Counter object from the input list, you get a dictionary-like mapping of each number to its frequency. A simple list comprehension can then iterate through this counter's items, returning a new list that includes only those numbers with a count greater than one. This approach is not only more compact, fitting into a single line, but is also highly readable and idiomatic in Python for frequency-related problems, achieving the same efficient O(n) time complexity as the set-based solution.

TASK 3:

Refactor messy code into clean, PEP 8–compliant, well-structured code.
buggy_code_task3.py

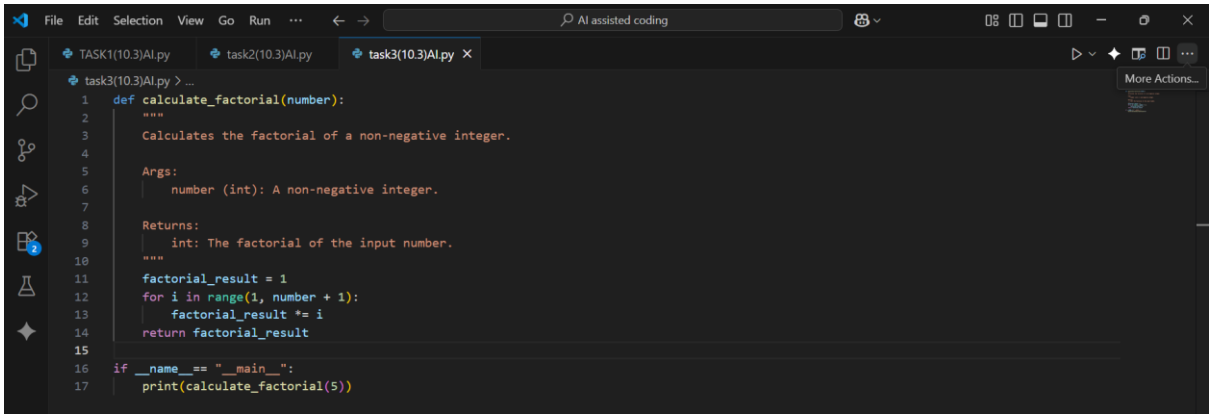
```
def c(n):
x=1
for i in range(1,n+1):
x=x*i
return x
print(c(5))
```

PROMPT:

Refactor the following Python code to improve readability and adhere to PEP 8 standards. Rename the function and variables to be more descriptive, add a comprehensive docstring, and wrap the main execution block appropriately.

```
def c(n):
x=1
for i in range(1,n+1):
x=x*i
return x
print(c(5))
```

CODE:



OUTPUT:

OBSERVATION:

This Python script defines a function calculate_factorial that computes the factorial of a given non-negative integer by initializing a result to 1 and then iteratively multiplying it by each integer from 1 up to the input number. The script then uses a standard if __name__ == "__main__" block, which ensures the code within it only runs when the file is executed directly. Inside this block, it calls the calculate_factorial function with the number 5 and prints the resulting value, 120, to the console.

TASK 4:

Add security practices and exception handling to the code.
# buggy_code_task4.py
import sqlite3
def get_user_data(user_id):
conn = sqlite3.connect("users.db")
cursor = conn.cursor()
query = f"SELECT * FROM users WHERE id = {user_id};" #
Potential SQL injection risk
cursor.execute(query)
result = cursor.fetchall()
conn.close()
return result

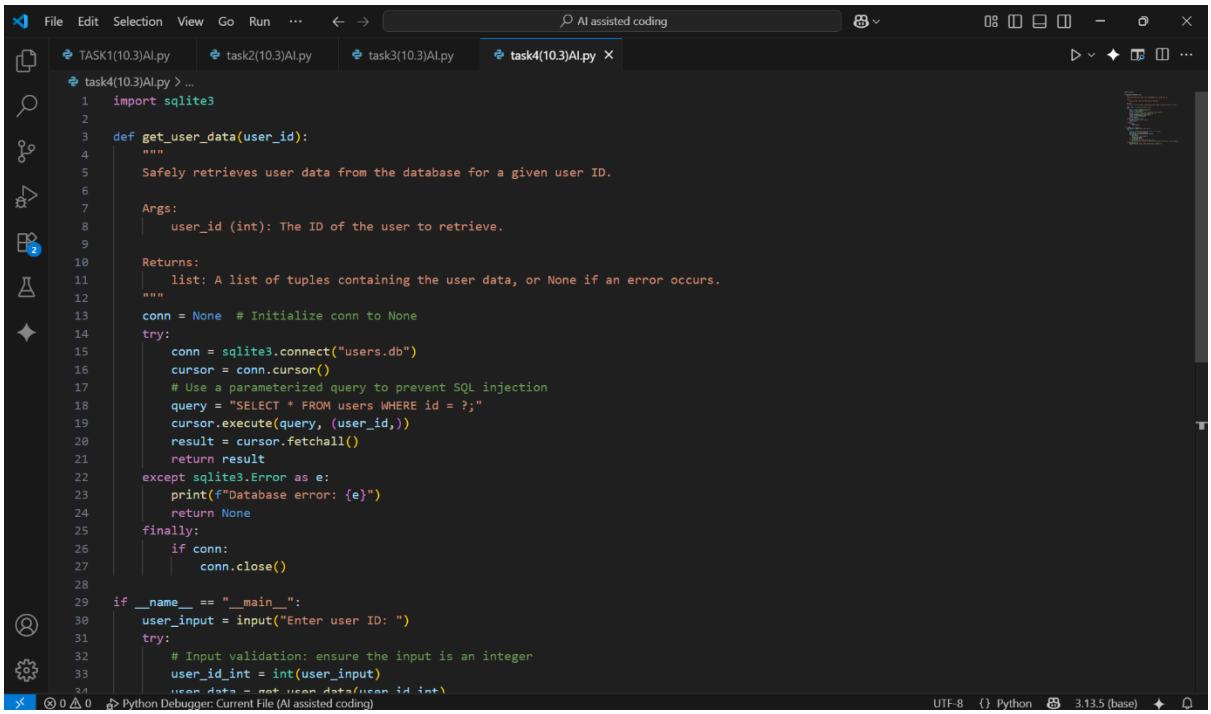
user_input = input("Enter user ID: ")
print(get_user_data(user_input))

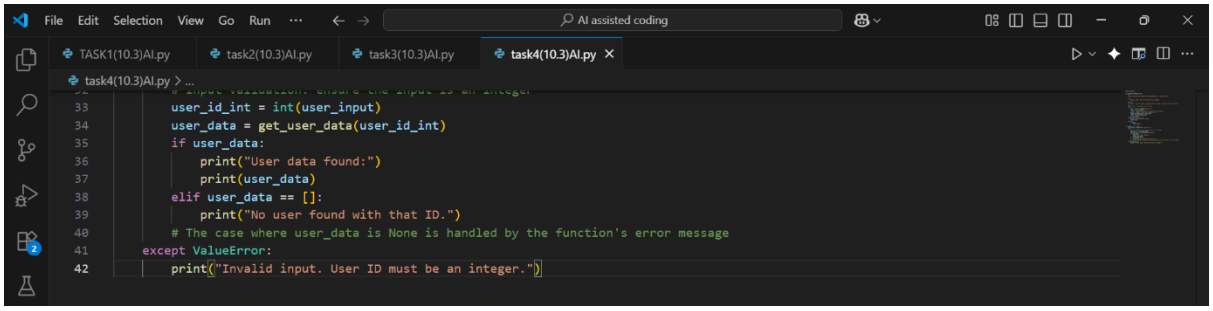
PROMPT:

Refactor the following Python script to fix a critical security vulnerability and add robust error handling. The current code is susceptible to SQL injection and lacks proper exception management.

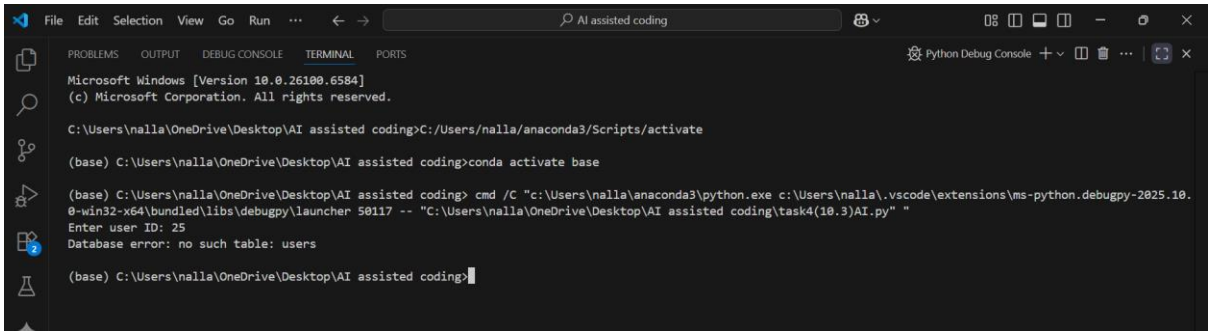
import sqlite3
def get_user_data(user_id):
conn = sqlite3.connect("users.db")
cursor = conn.cursor()
query = f"SELECT * FROM users WHERE id = {user_id};" #
Potential SQL injection risk
cursor.execute(query)
result = cursor.fetchall()
conn.close()
return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input))

CODE:





OUTPUT:



OBSERVATION:

This Python script queries a SQLite database (users.db) to retrieve user data by ID. The get_user_data function uses a parameterized query (id = ?) to prevent SQL injection and wraps operations in try...except...finally to handle errors and ensure the connection closes. The main block (if __name__ == "__main__") prompts for an ID, validates numeric input, and gives clear feedback—showing results if found, noting when no user exists, or reporting invalid input.

TASK 5:

Generate a review report for this messy code.
buggy_code_task5.py

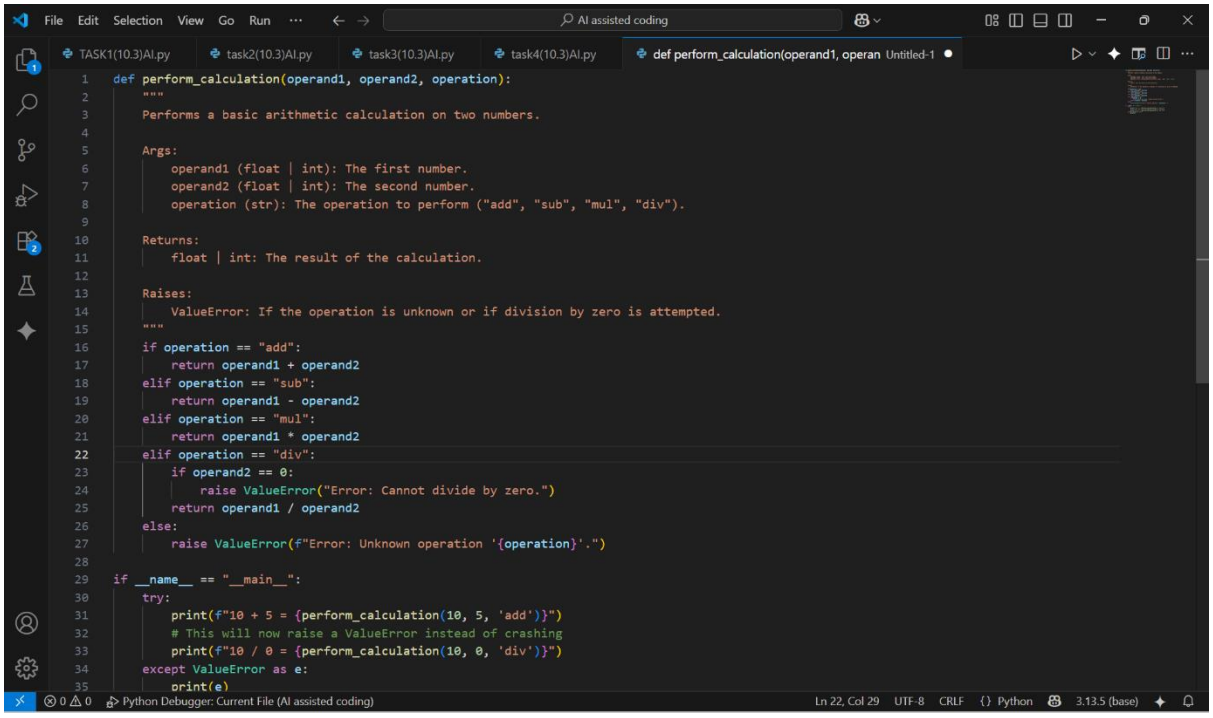
```
def calc(x,y,z):
if z=="add":
return x+y
elif z=="sub": return x-y
elif z=="mul":
return x*y
elif z=="div":
return x/y
else: print("wrong")
print(calc(10,5,"add"))
print(calc(10,0,"div"))
```

PROMPT:

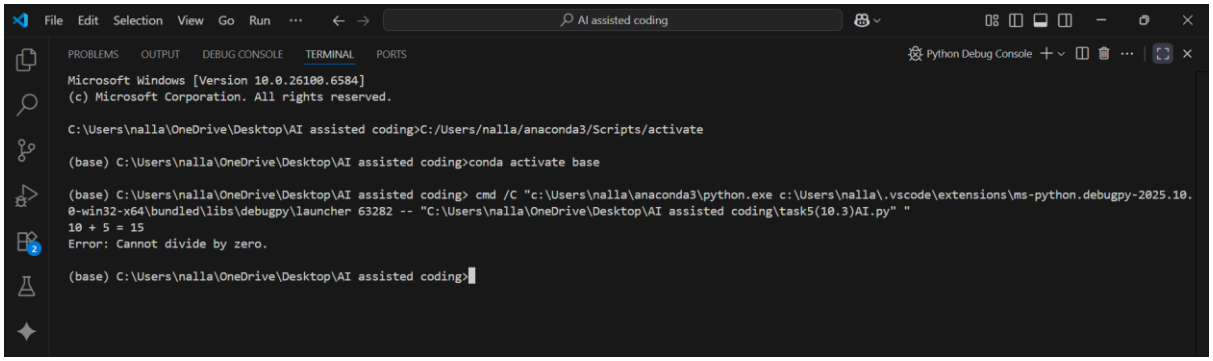
Act as an automated code review tool. Analyze the following Python script and generate a formal review report.

```
def calc(x,y,z):
if z=="add":
return x+y
elif z=="sub": return x-y
elif z=="mul":
return x*y
elif z=="div":
return x/y
else: print("wrong")
print(calc(10,5,"add"))
print(calc(10,0,"div"))
```

CODE:



OUTPUT:



OBSERVATION:

This Python script defines a function calc that performs four basic arithmetic operations: addition, subtraction, multiplication, and division. It checks if the operation requested is valid and raises an error if it is not. Division is handled carefully by raising an error when the second number is zero to prevent a crash. In the main block, the script demonstrates how the function works by performing a valid calculation (10 + 5) and an invalid one (10 / 0). The results are displayed on the screen, and any errors are caught and printed in a user-friendly way.