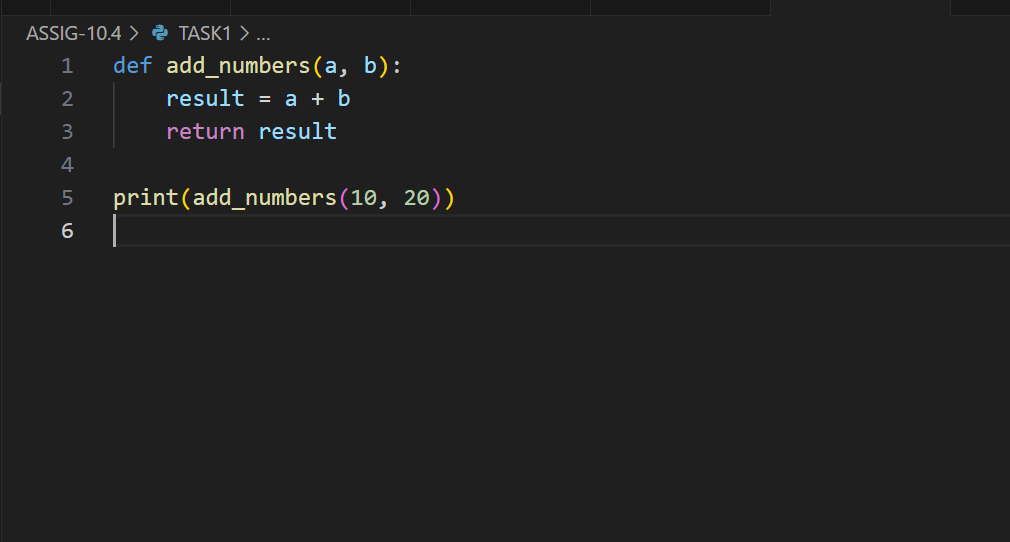
**AI ASSISTED CODING**

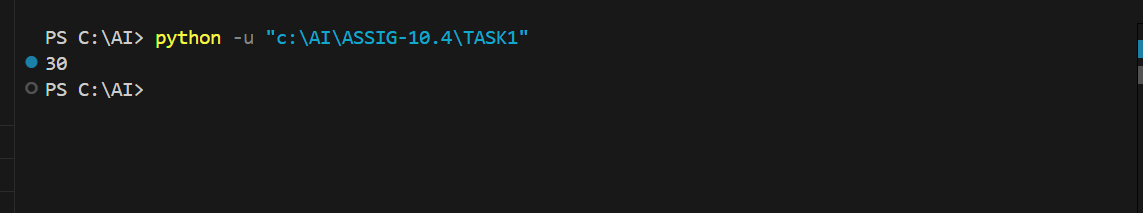
ASSIGNMENT-10.4

**TEST – 1**  
Task: 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))

Expected Output:  
• Corrected code with proper syntax (: after function, fixed variable  
name, corrected function call).  
• AI should explain what was fixed

****

**OUTPUT:**

****

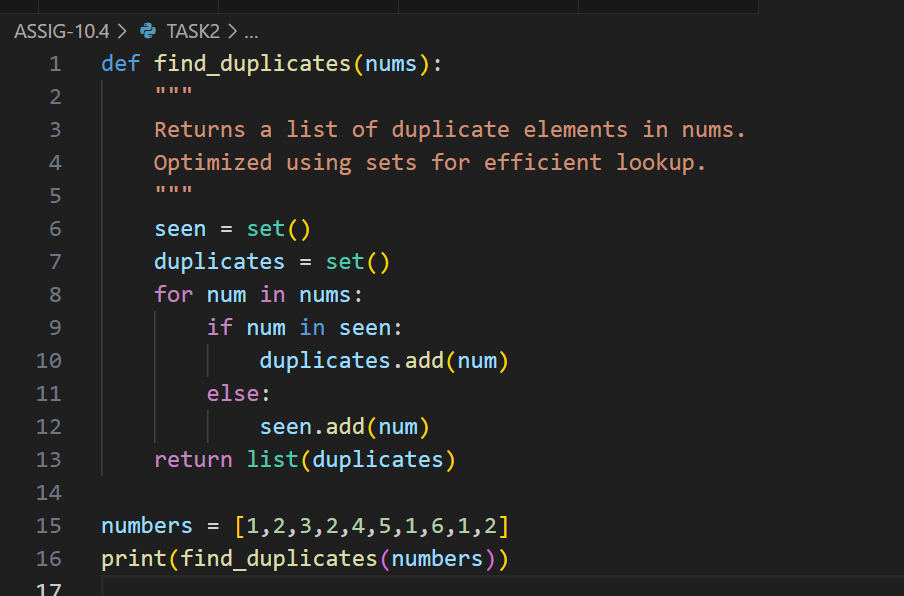
**Explanation of fixes:**

* Added missing colon (:) after the function definition line.
* Indented the function body correctly (2 spaces or 4 spaces).
* Corrected the variable name from reslt to result.
* Added missing comma between arguments in the function call add\_numbers(10, 20).

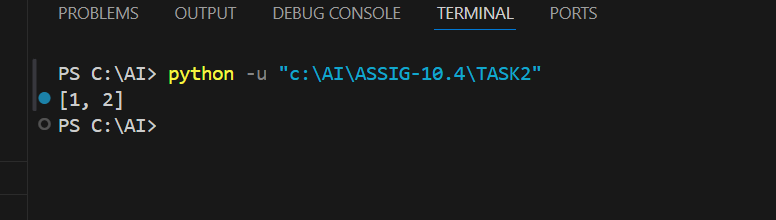
**TEST – 2**

Task: 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))

Expected Output:  
• More efficient duplicate detection (e.g., using sets).  
• AI should explain the optimization.



**OUTPUT:**



**Explanation :**

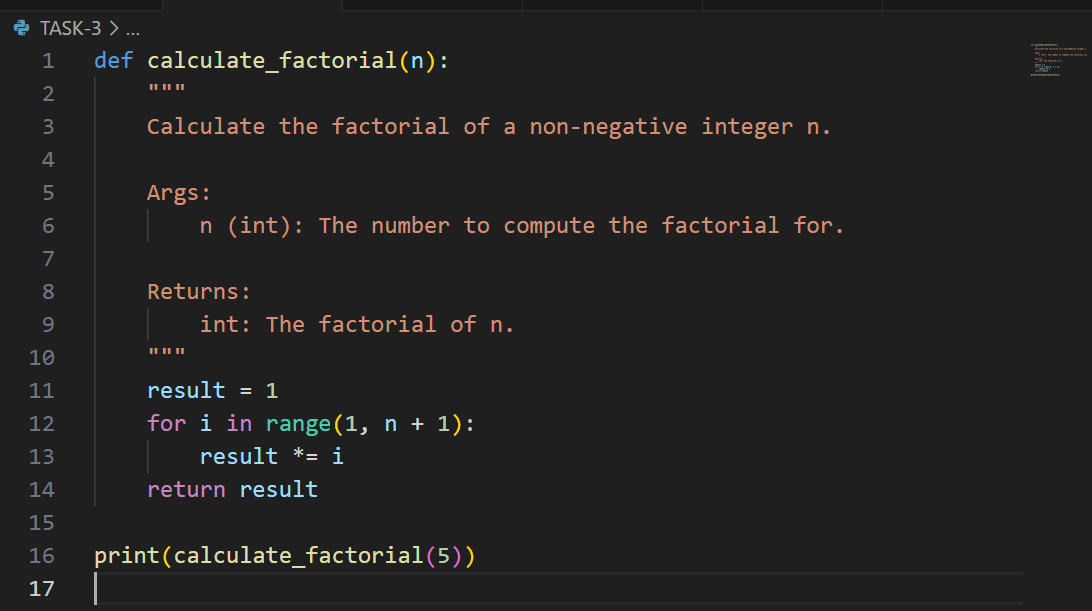
* Originalapproach: Uses nested loops (O(n^2)) comparing every element with every other element, which is inefficient for large lists.
* Newapproach: Uses two sets:
  + seen to keep track of elements already encountered.
  + duplicates to store elements that appear more than once.
* This reduces the complexity to **O(n)** because each element is processed only once.
* Sets provide **O(1)** average-time complexity for membership tests, making the duplicate detection much faster.
* Finally, the duplicates set is converted back to a list for output.

**TEST – 3**

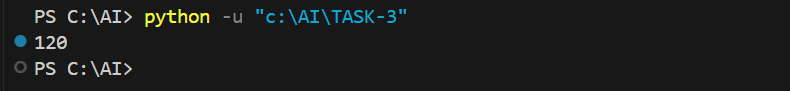
Task: Refactor messy code into clean, PEP 8–compliant, well-  
structured code.  
# buggy\_code\_task3.py

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

Expected Output:  
Function renamed to calculate\_factorial.  
Proper indentation, variable naming, docstrings, and formatting.  
AI should provide a more readable version.

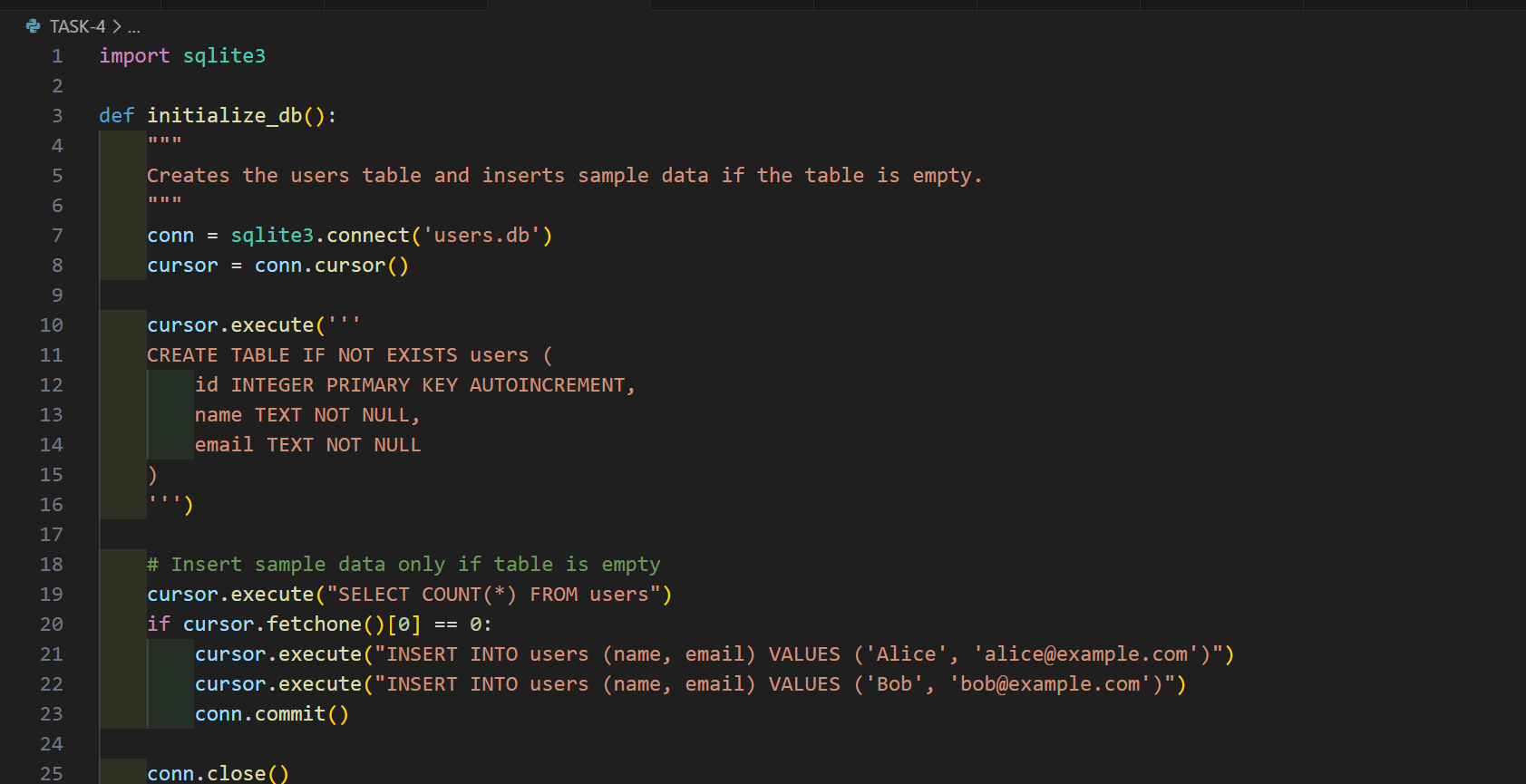
****

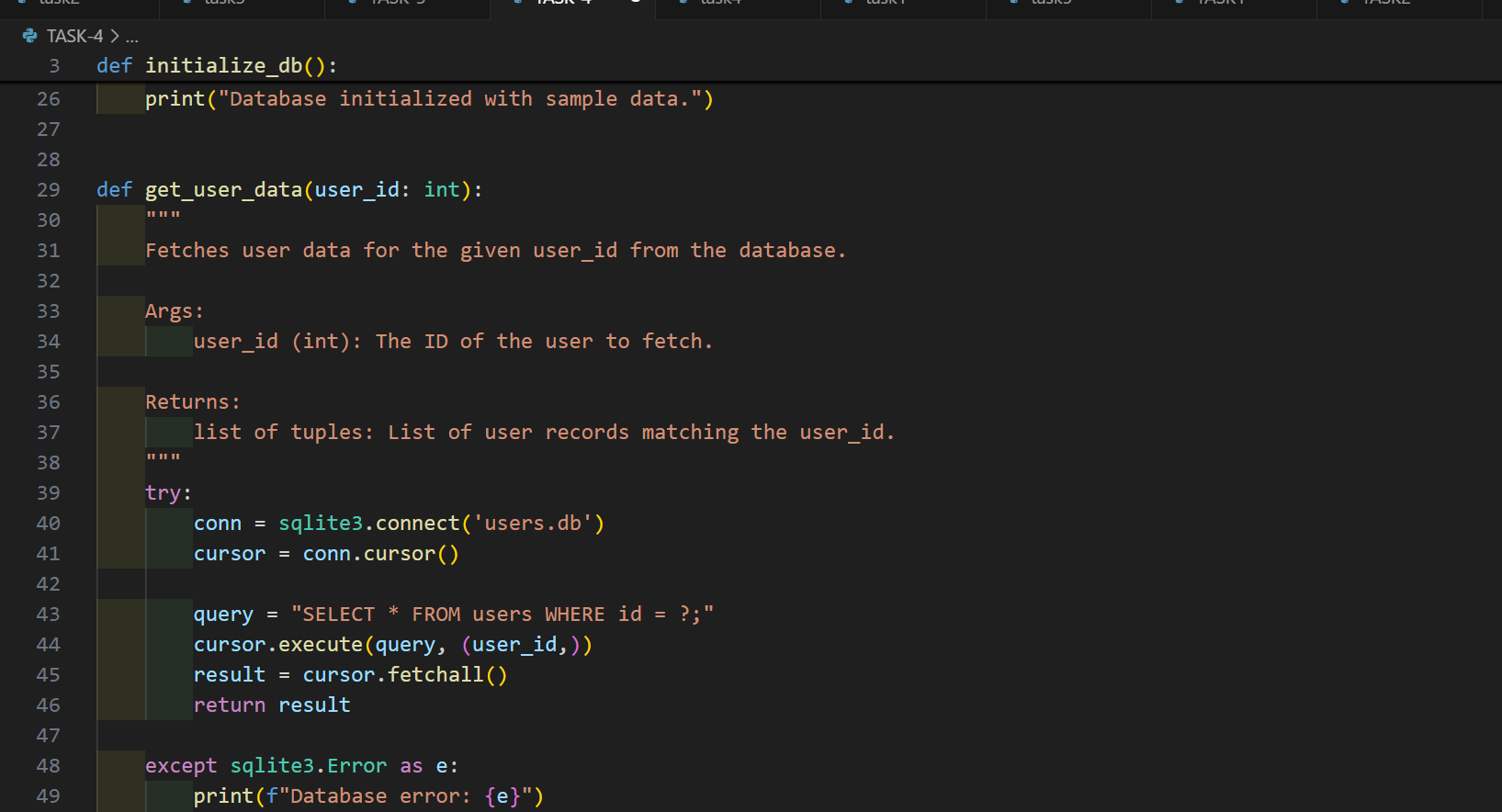
**OUTPUT:**

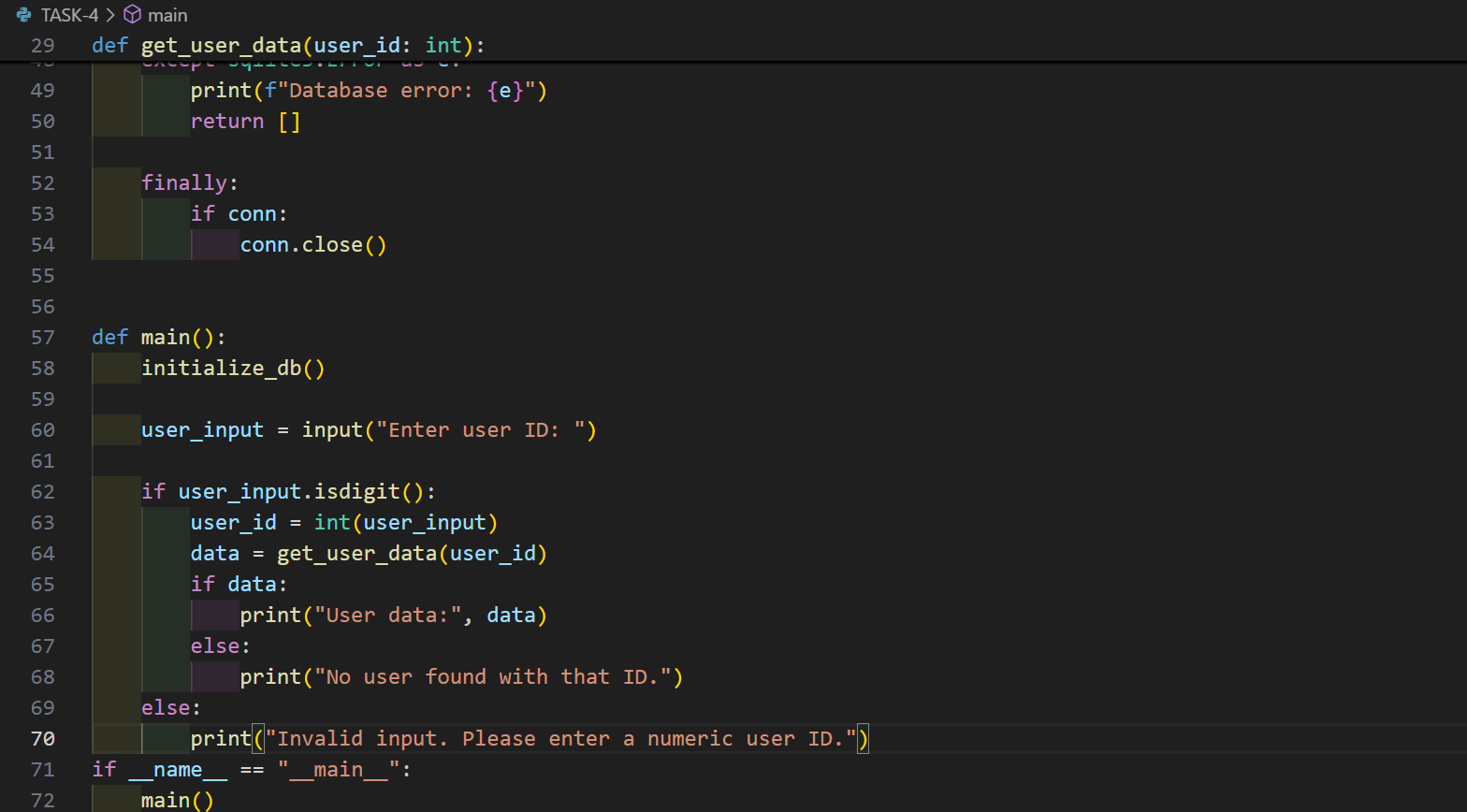
****

**Task – 4**Task: 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))

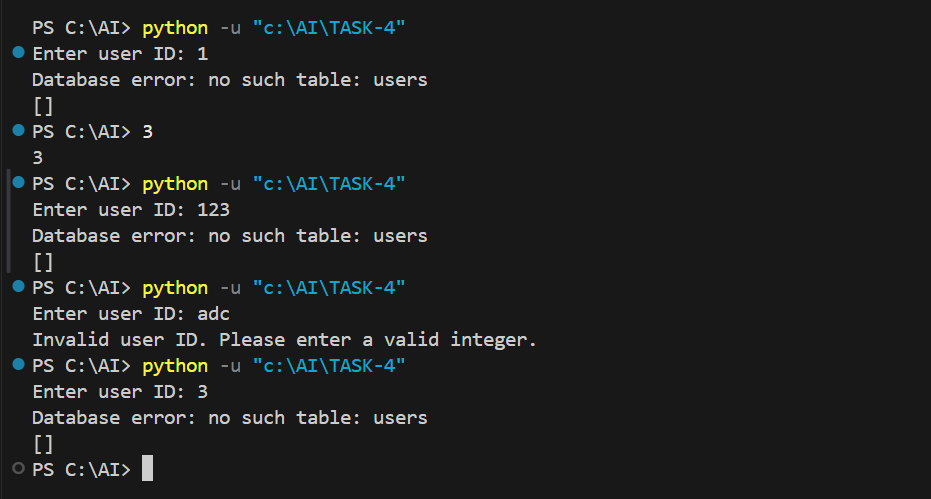
Expected Output:  
Safe query using parameterized SQL (? placeholders).  
Try-except block for database errors.  
Input validation before query execution.

****

****

****

**OUTPUT:**

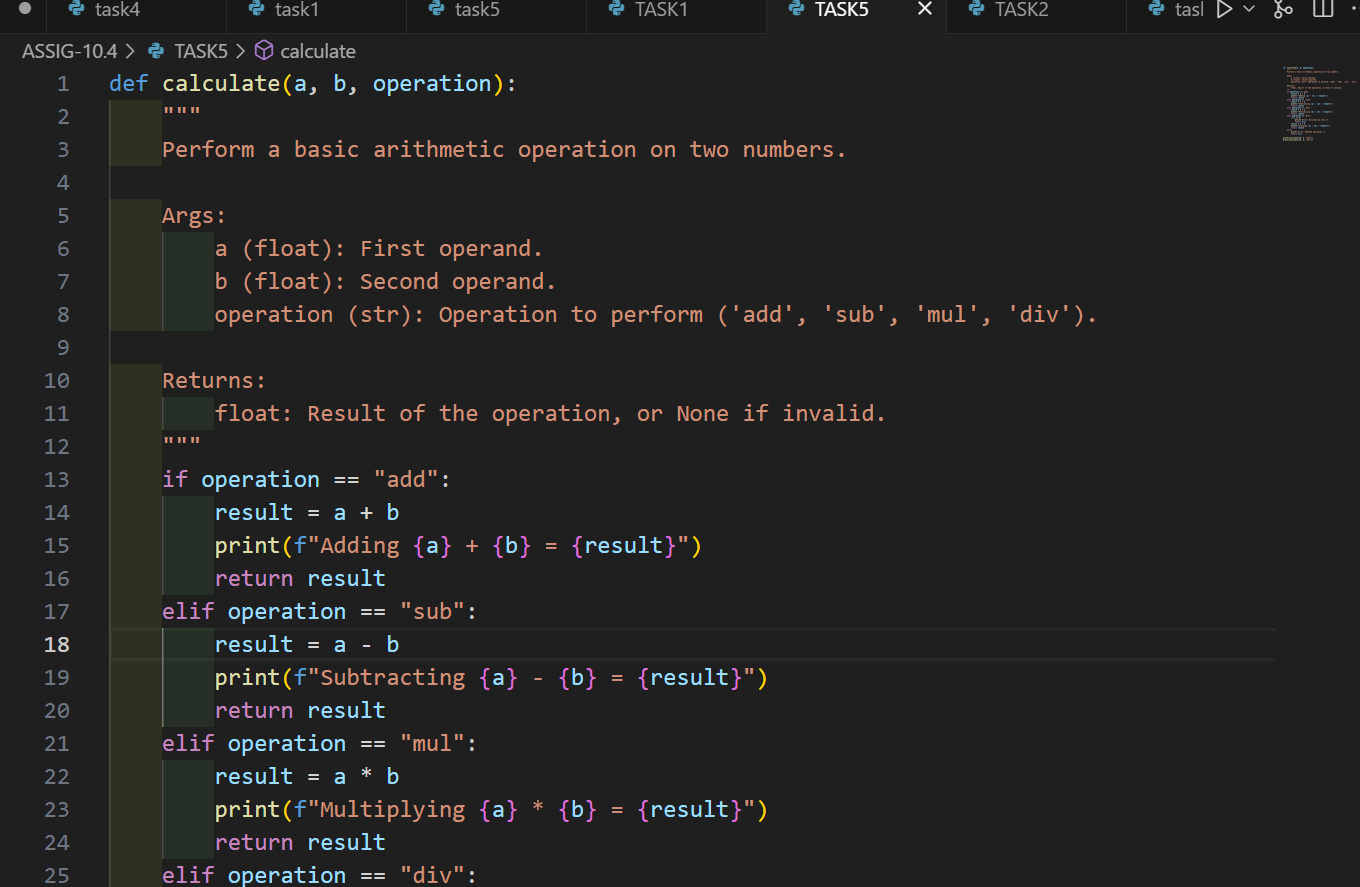
****

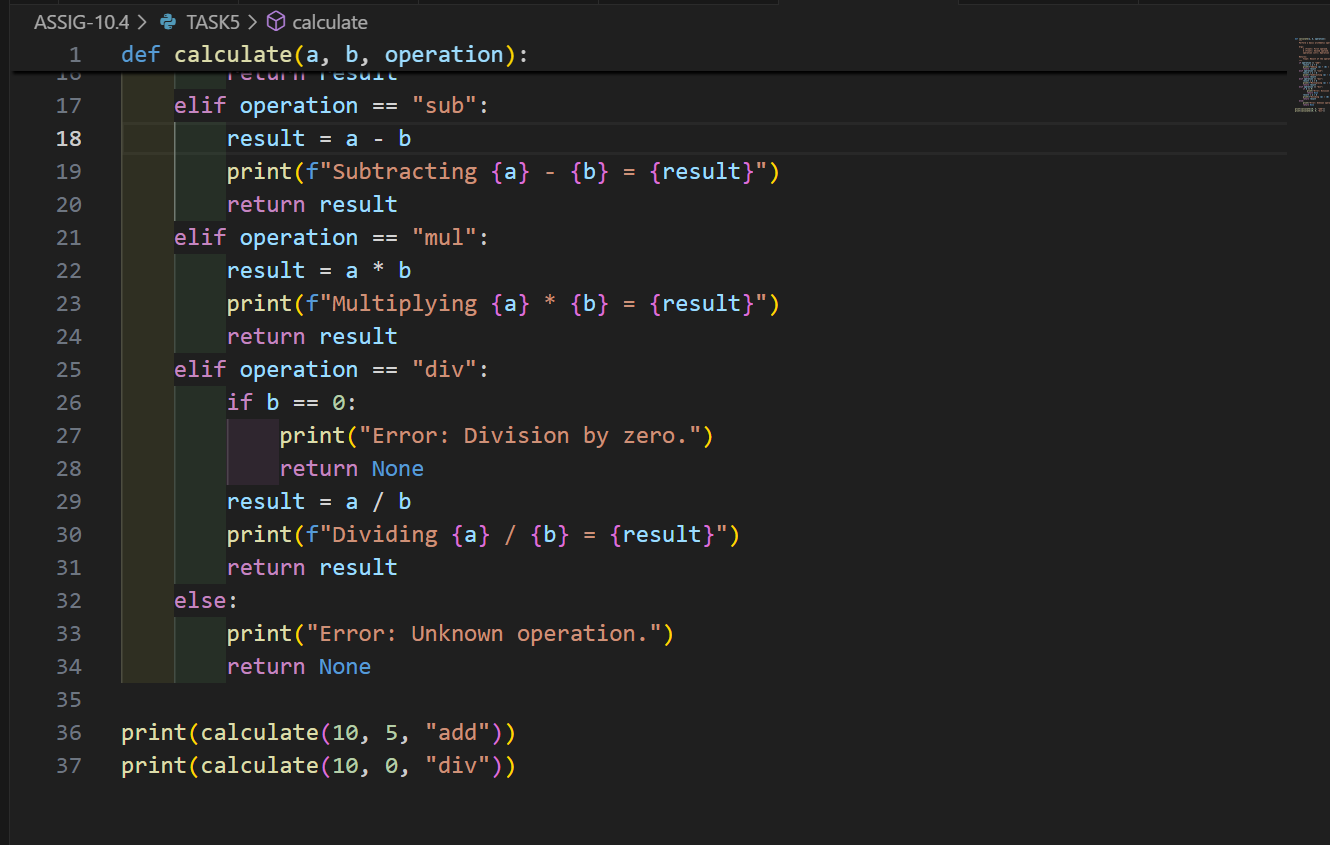
**TEST – 5**

Task: 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"))

Expected Output:  
AI-generated review report should mention:  
o Missing docstrings  
o Inconsistent formatting (indentation, inline return)  
o Missing error handling for division by zero  
o Non-descriptive function/variable names  
o Suggestions for readability and PEP 8 compliance





**OUTPUT:**

