LAB ASSIGNMENT-10

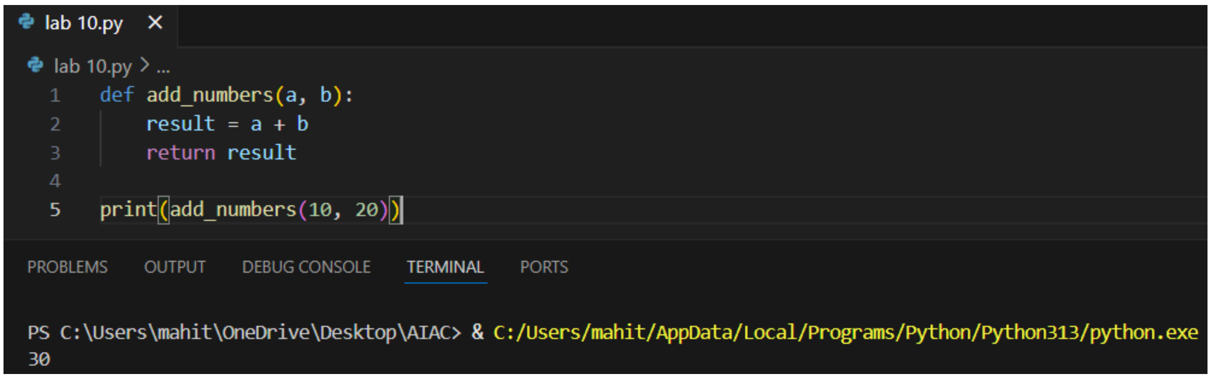
TASK-1:

Prompt:

Identify and correct the code

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

Code and Output:



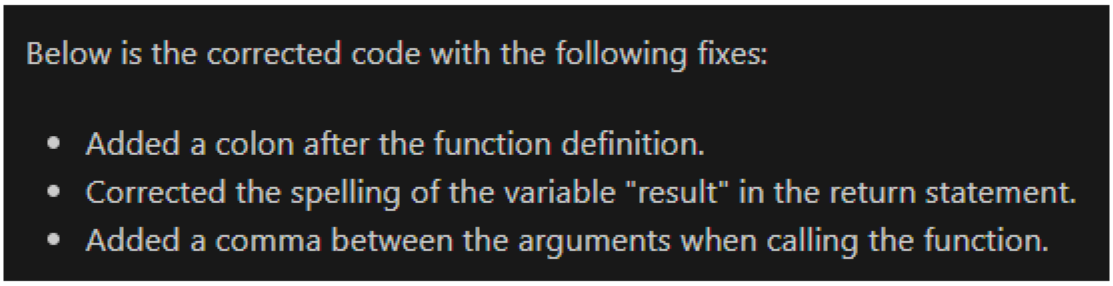
lab 10.py

```
def add_numbers(a, b):
    result = a + b
    return result

print(add_numbers(10, 20))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\mahit\OneDrive\Desktop\AIAC> & C:/Users/mahit/AppData/Local/Programs/Python/Python313/python.exe
30
```

Code Explanation:



Below is the corrected code with the following fixes:

- Added a colon after the function definition.
- Corrected the spelling of the variable "result" in the return statement.
- Added a comma between the arguments when calling the function.

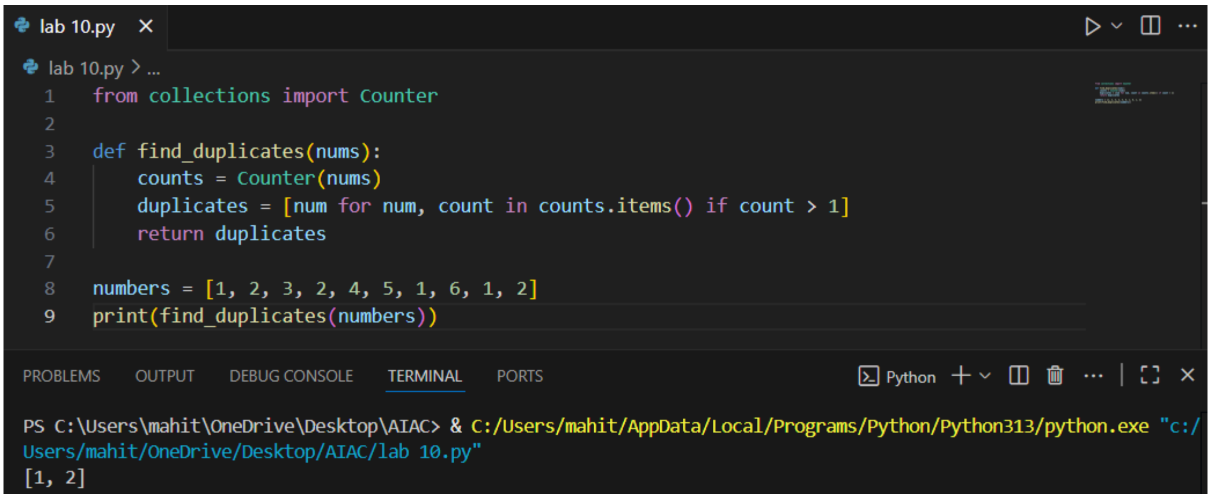
TASK-2:

Prompt:

Optimize inefficient logic while keeping the result correct.

```
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))
```

Code and Output:



```python
from collections import Counter

def find_duplicates(nums):
    counts = Counter(nums)
    duplicates = [num for num, count in counts.items() if count > 1]
    return duplicates

numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]
print(find_duplicates(numbers))
```

```
PS C:\Users\mahit\OneDrive\Desktop\AIAC> & C:/Users/mahit/AppData/Local/Programs/Python/Python313/python.exe "c:/
Users/mahit/OneDrive/Desktop/AIAC/lab 10.py"
[1, 2]
```

Code Explanation:

The original code uses two nested loops to compare every element with every other element, resulting in O(n^2) comparisons. In the optimized version, we use Python's `collections.Counter` to count the occurrence of each number in a single pass over the list (O(n) time complexity) and then extract those numbers that appear more than once. This reduces the complexity and improves efficiency significantly, especially for large lists.

TASK-3:

Prompt:

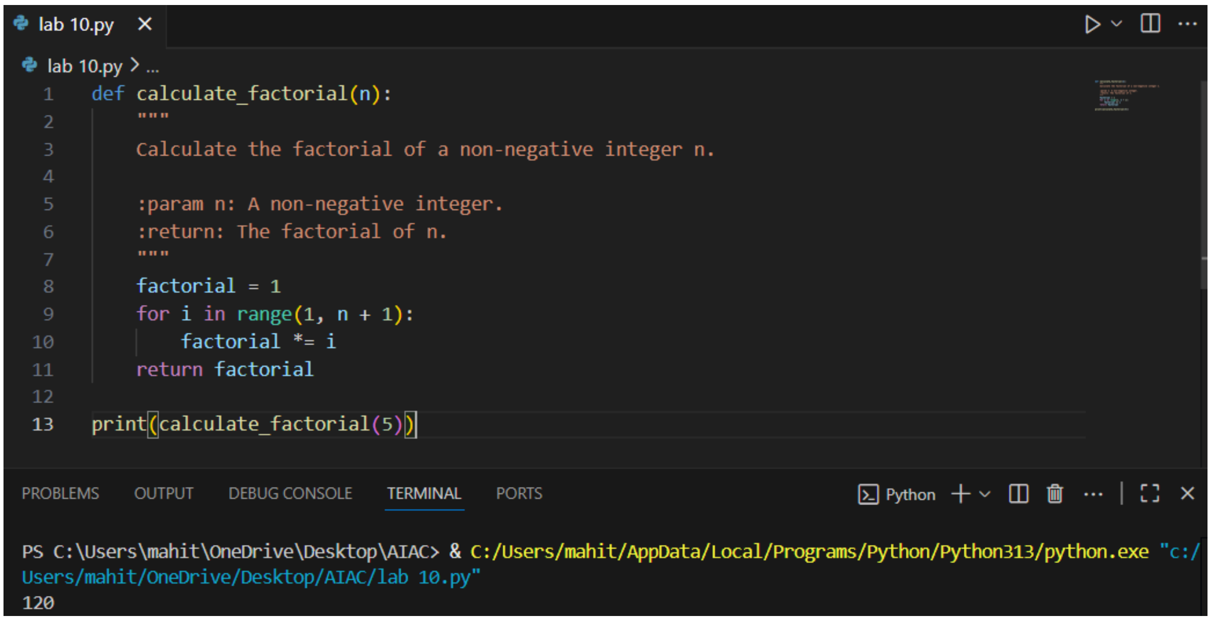
Generate a Function renamed to calculate_factorial.

Proper indentation, variable naming, docstrings, and formatting.

Provide a more readable version.

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

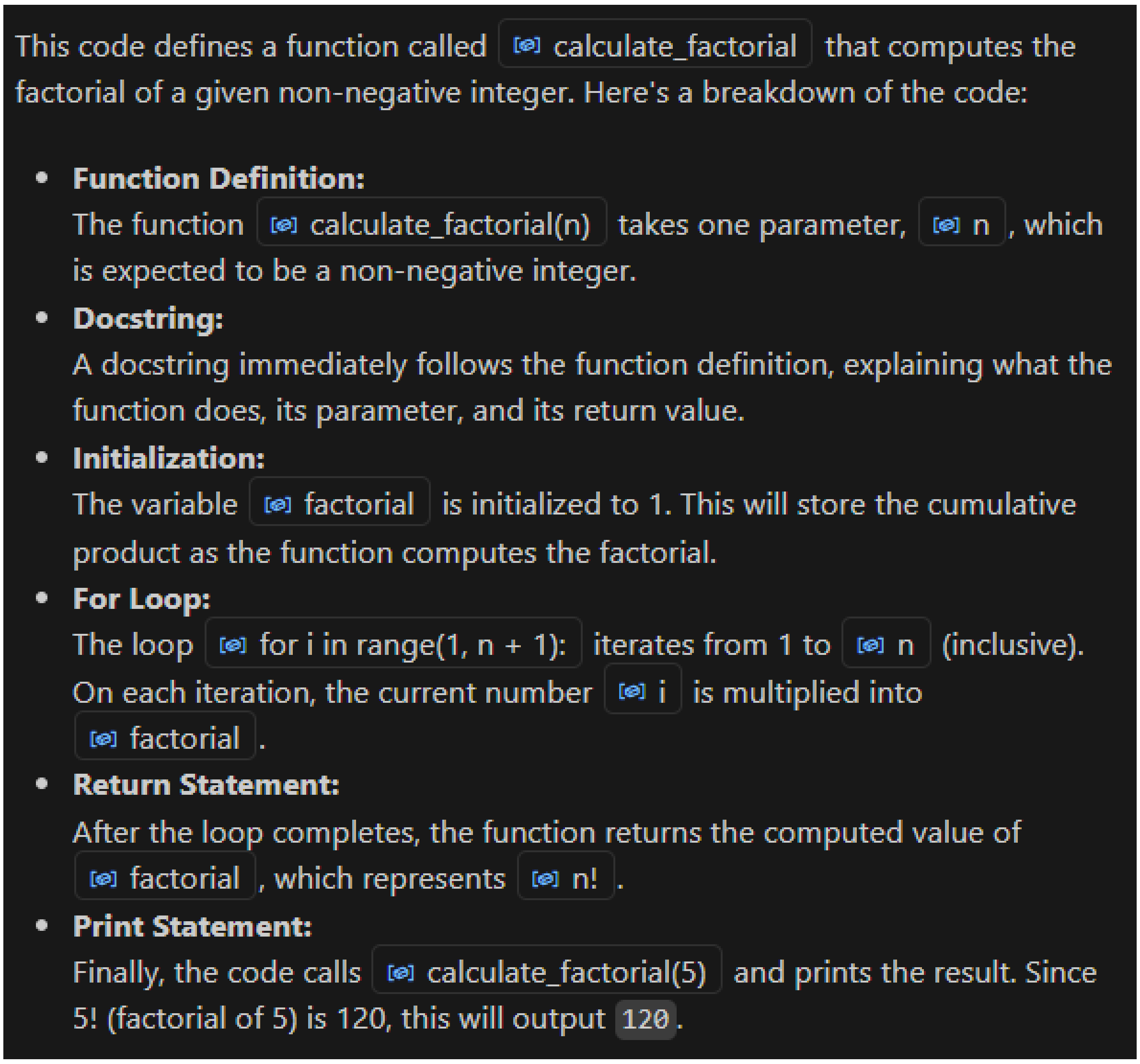
Code and Output:



```python
def calculate_factorial(n):
    """
    Calculate the factorial of a non-negative integer n.

    :param n: A non-negative integer.
    :return: The factorial of n.
    """
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial

print(calculate_factorial(5))
```

```
PS C:\Users\mahit\OneDrive\Desktop\AIAC> & C:/Users/mahit/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/mahit/OneDrive/Desktop/AIAC/lab 10.py"
120
```

Code Explanation:


This code defines a function called `calculate_factorial` that computes the factorial of a given non-negative integer. Here's a breakdown of the code:

- **Function Definition:**
  The function `calculate_factorial(n)` takes one parameter, `n`, which is expected to be a non-negative integer.
- **Docstring:**
  A docstring immediately follows the function definition, explaining what the function does, its parameter, and its return value.
- **Initialization:**
  The variable `factorial` is initialized to 1. This will store the cumulative product as the function computes the factorial.
- **For Loop:**
  The loop `for i in range(1, n + 1):` iterates from 1 to `n` (inclusive). On each iteration, the current number `i` is multiplied into `factorial`.
- **Return Statement:**
  After the loop completes, the function returns the computed value of `factorial`, which represents `n!`.
- **Print Statement:**
  Finally, the code calls `calculate_factorial(5)` and prints the result. Since 5! (factorial of 5) is 120, this will output `120`.

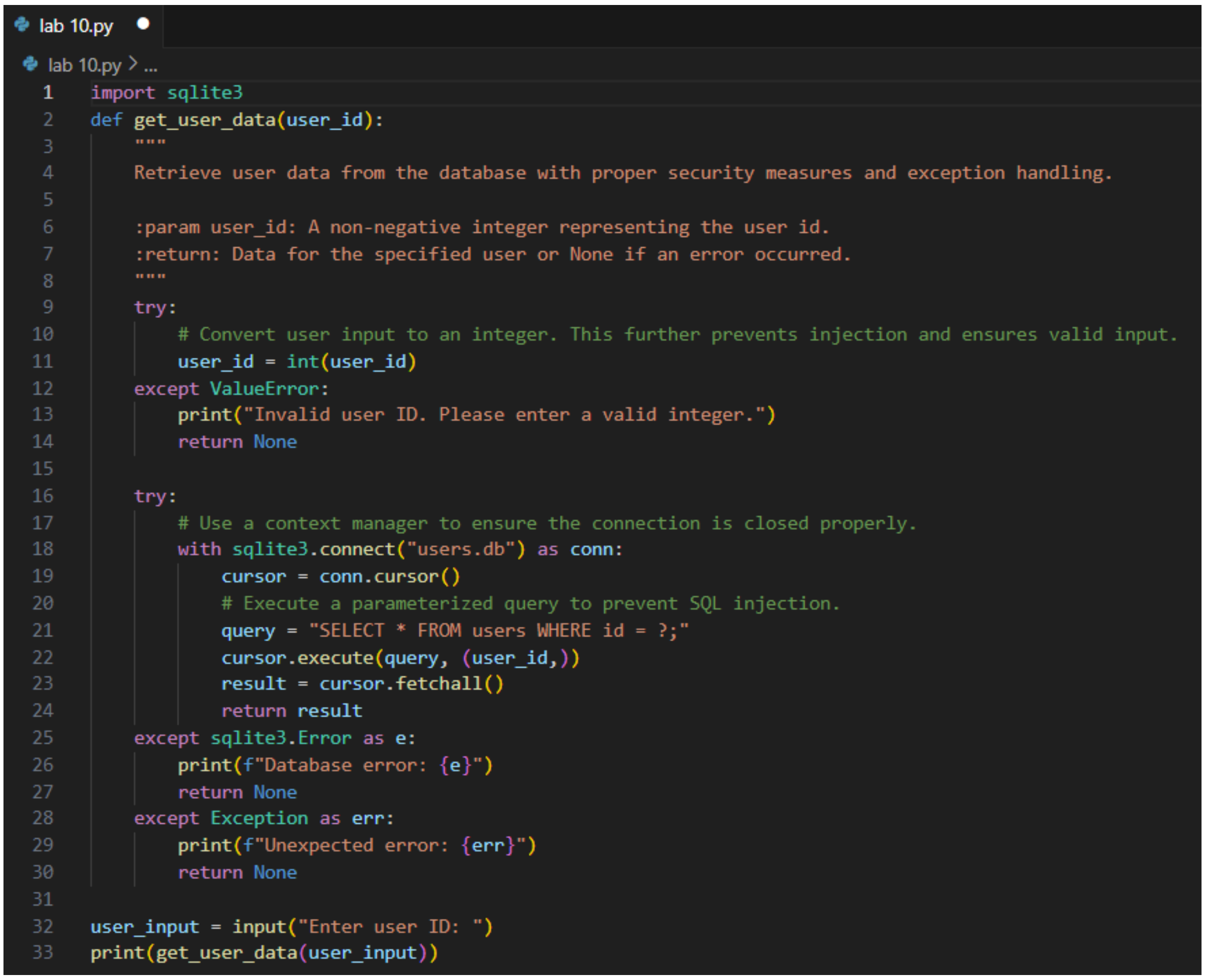
TASK-4:

Prompt:

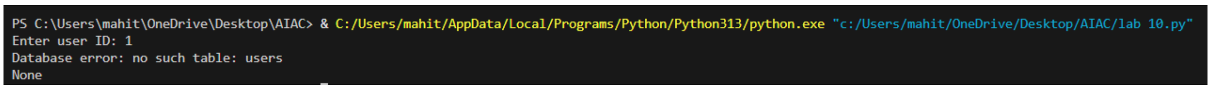
Add security practices and exception handling to the code.

```
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 and Output:

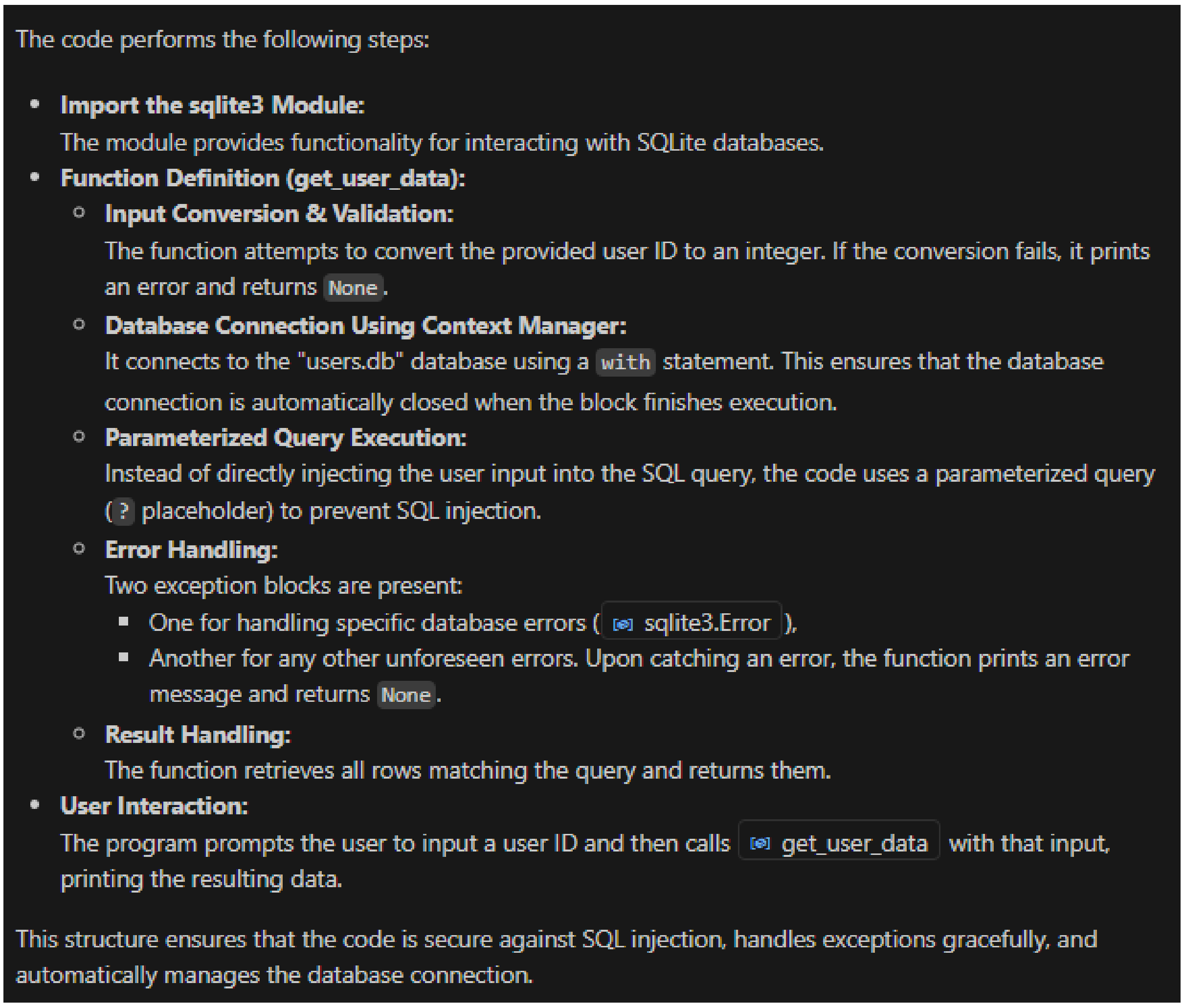


```python
import sqlite3
def get_user_data(user_id):
    """
    Retrieve user data from the database with proper security measures and exception handling.

    :param user_id: A non-negative integer representing the user id.
    :return: Data for the specified user or None if an error occurred.
    """
    try:
        # Convert user input to an integer. This further prevents injection and ensures valid input.
        user_id = int(user_id)
    except ValueError:
        print("Invalid user ID. Please enter a valid integer.")
        return None

    try:
        # Use a context manager to ensure the connection is closed properly.
        with sqlite3.connect("users.db") as conn:
            cursor = conn.cursor()
            # Execute a parameterized query to prevent SQL injection.
            query = "SELECT * FROM users WHERE id = ?;"
            cursor.execute(query, (user_id,))
            result = cursor.fetchall()
            return result
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None
    except Exception as err:
        print(f"Unexpected error: {err}")
        return None

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



```
PS C:\Users\mahit\OneDrive\Desktop\AIAC> & C:/Users/mahit/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/mahit/OneDrive/Desktop/AIAC/lab 10.py"
Enter user ID: 1
Database error: no such table: users
None
```

Code Explanation:



The code performs the following steps:

- **Import the sqlite3 Module:**
  The module provides functionality for interacting with SQLite databases.
- **Function Definition (get_user_data):**
  - **Input Conversion & Validation:**
    The function attempts to convert the provided user ID to an integer. If the conversion fails, it prints an error and returns `None`.
  - **Database Connection Using Context Manager:**
    It connects to the "users.db" database using a `with` statement. This ensures that the database connection is automatically closed when the block finishes execution.
  - **Parameterized Query Execution:**
    Instead of directly injecting the user input into the SQL query, the code uses a parameterized query (`?` placeholder) to prevent SQL injection.
  - **Error Handling:**
    Two exception blocks are present:
    - One for handling specific database errors (sqlite3.Error),
    - Another for any other unforeseen errors. Upon catching an error, the function prints an error message and returns `None`.
  - **Result Handling:**
    The function retrieves all rows matching the query and returns them.
- **User Interaction:**
  The program prompts the user to input a user ID and then calls get_user_data with that input, printing the resulting data.

This structure ensures that the code is secure against SQL injection, handles exceptions gracefully, and automatically manages the database connection.

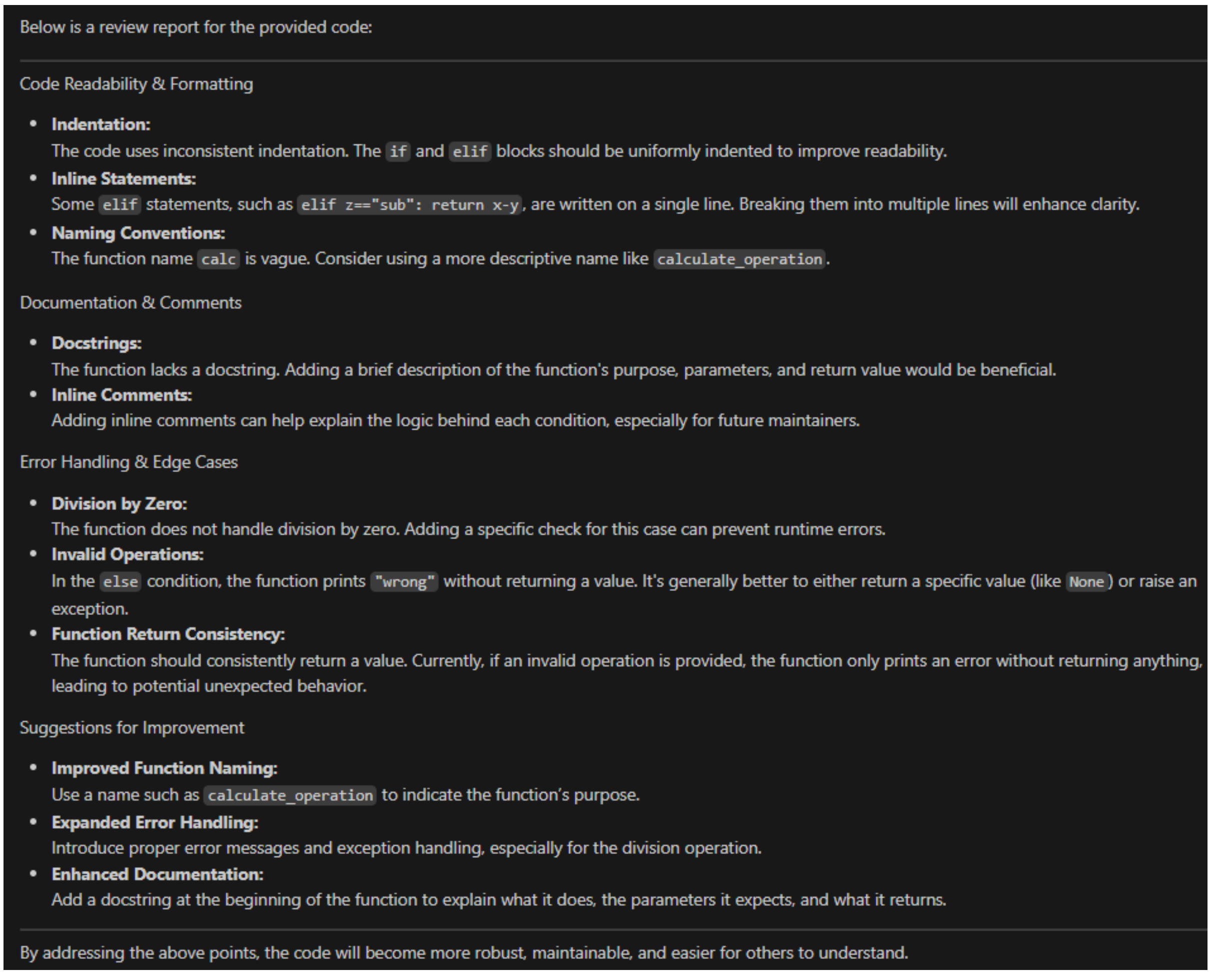
TASK-5:

Prompt:

Generate a review report for this messy code.

```
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"))
```

Output:



Below is a review report for the provided code:

Code Readability & Formatting

- **Indentation:**
  The code uses inconsistent indentation. The `if` and `elif` blocks should be uniformly indented to improve readability.
- **Inline Statements:**
  Some `elif` statements, such as `elif z=="sub": return x-y`, are written on a single line. Breaking them into multiple lines will enhance clarity.
- **Naming Conventions:**
  The function name `calc` is vague. Consider using a more descriptive name like `calculate_operation`.

Documentation & Comments

- **Docstrings:**
  The function lacks a docstring. Adding a brief description of the function's purpose, parameters, and return value would be beneficial.
- **Inline Comments:**
  Adding inline comments can help explain the logic behind each condition, especially for future maintainers.

Error Handling & Edge Cases

- **Division by Zero:**
  The function does not handle division by zero. Adding a specific check for this case can prevent runtime errors.
- **Invalid Operations:**
  In the `else` condition, the function prints `"wrong"` without returning a value. It's generally better to either return a specific value (like `None`) or raise an exception.
- **Function Return Consistency:**
  The function should consistently return a value. Currently, if an invalid operation is provided, the function only prints an error without returning anything, leading to potential unexpected behavior.

Suggestions for Improvement

- **Improved Function Naming:**
  Use a name such as `calculate_operation` to indicate the function's purpose.
- **Expanded Error Handling:**
  Introduce proper error messages and exception handling, especially for the division operation.
- **Enhanced Documentation:**
  Add a docstring at the beginning of the function to explain what it does, the parameters it expects, and what it returns.

By addressing the above points, the code will become more robust, maintainable, and easier for others to understand.