

# ASSIGNMENT – 10

NAME: G. Bala Varshitha

HT.NO: 2403A52050

BATCH: AIB03

## Task-1:

Syntax and Error Detection

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

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

Expected Output:

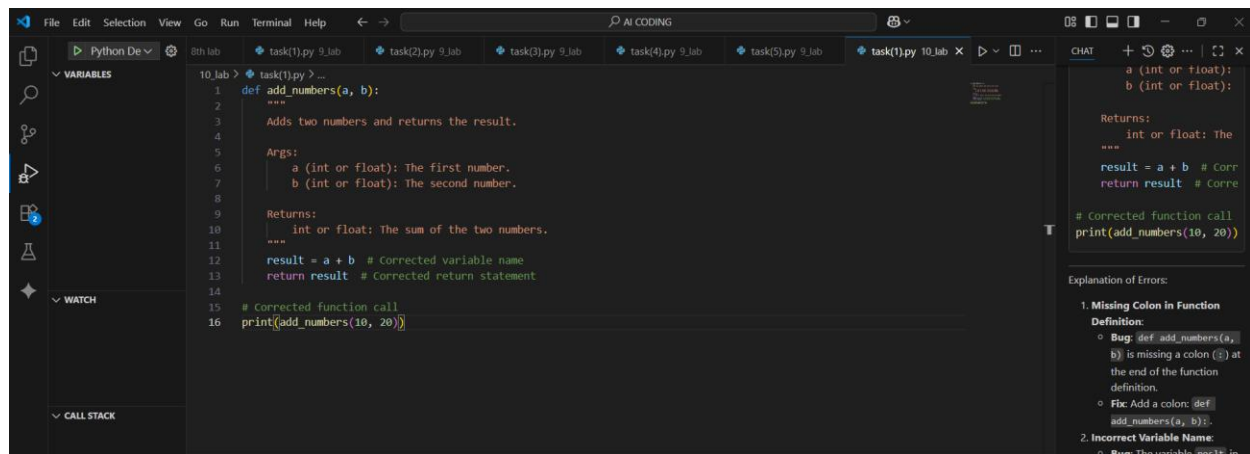
- Corrected code with proper syntax (: after function, fixed variable name, corrected function call).
- AI should explain what was fixed

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

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

even explain the errors.

Code:

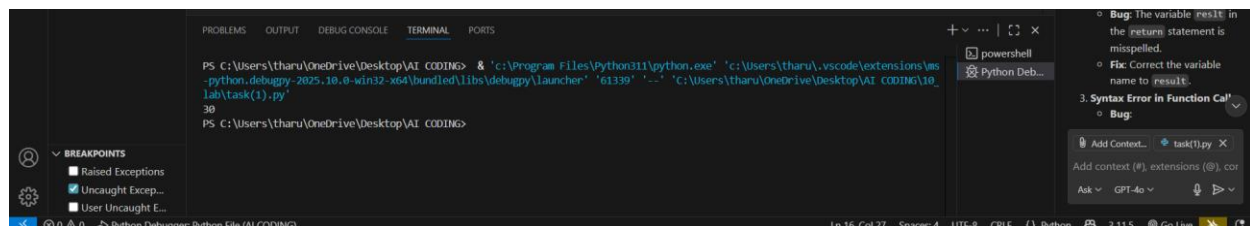


```
1 def add_numbers(a, b):
2     """
3     Adds two numbers and returns the result.
4
5     Args:
6         a (int or float): The first number.
7         b (int or float): The second number.
8
9     Returns:
10        int or float: The sum of the two numbers.
11    """
12    result = a + b # Corrected variable name
13    return result # Corrected return statement
14
15 # Corrected function call
16 print(add_numbers(10, 20))
```

Explanation of Errors:

- Missing Colon in Function Definition:**
  - Bug:** def add\_numbers(a, b) is missing a colon (:) at the end of the function definition.
  - Fix:** Add a colon: def add\_numbers(a, b):.
- Incorrect Variable Name:**
  - Bug:** The variable reslt in the return statement is misspelled.
  - Fix:** Correct the variable name to result.
- Syntax Error in Function Call:**
  - Bug:** print(add\_numbers(10 20)) is missing a comma between 10 and 20.
  - Fix:** Add a comma: print(add\_numbers(10, 20)).
- Extra Closing Parenthesis:**
  - Bug:** The print statement has an extra closing parenthesis: print(add\_numbers(10, 20)).
  - Fix:** Remove the extra parenthesis.

OP:



```
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> & 'c:\Program Files\Python311\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundle\libs\debugpy\launcher' '61339' '-' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\10 lab\task(1).py'
30
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
```

Explanation of Errors:

- Missing Colon in Function Definition:**
  - Bug:** def add\_numbers(a, b) is missing a colon (:) at the end of the function definition.
  - Fix:** Add a colon: def add\_numbers(a, b):.
- Incorrect Variable Name:**
  - Bug:** The variable reslt in the return statement is misspelled.
  - Fix:** Correct the variable name to result.
- Syntax Error in Function Call:**
  - Bug:** print(add\_numbers(10 20)) is missing a comma between 10 and 20.
  - Fix:** Add a comma: print(add\_numbers(10, 20)).
- Extra Closing Parenthesis:**
  - Bug:** The print statement has an extra closing parenthesis: print(add\_numbers(10, 20)).
  - Fix:** Remove the extra parenthesis.

Observation: It has corrected the errors and even explained it. **Bug:** def add\_numbers(a, b) is missing a colon (:) at the end of the function definition. **Fix:** Add a colon: def add\_numbers(a, b):. **Bug:** The variable reslt in the return statement is misspelled. **Fix:** Correct the variable name to result. **Bug:** print(add\_numbers(10 20)) is missing a comma between 10 and 20. **Fix:** Add a comma: print(add\_numbers(10, 20)). **Bug:** The print statement has an extra closing parenthesis: print(add\_numbers(10, 20)). **Fix:** Remove the extra parenthesis.

Task-2:

## Logical and Performance Issue Review

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.

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:
```

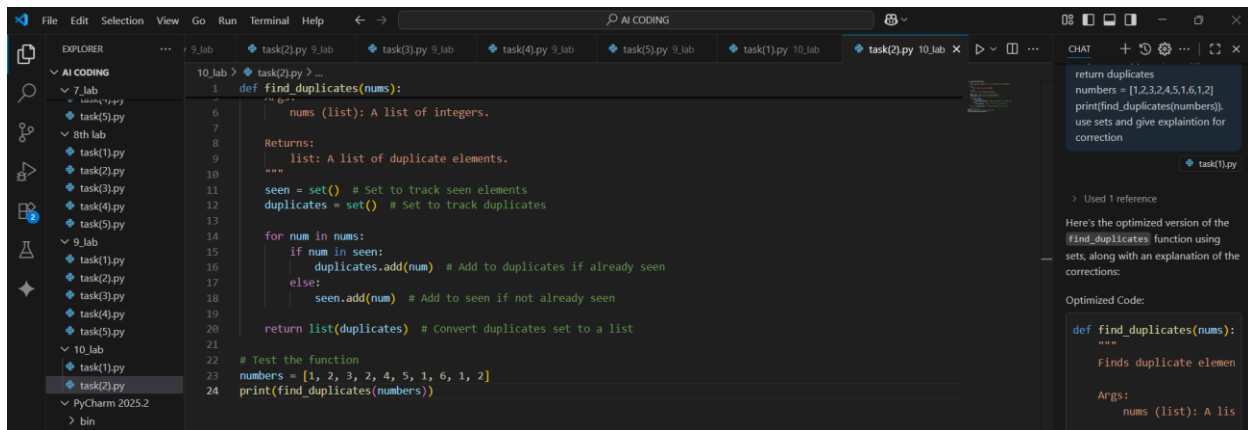
```
                duplicates.append(nums[i])
```

```
    return duplicates
```

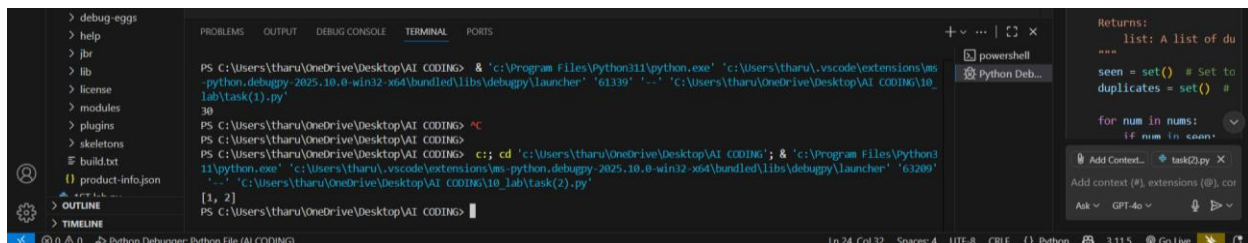
```
numbers = [1,2,3,2,4,5,1,6,1,2]
```

```
print(find_duplicates(numbers)). use sets and give explaintion for  
correction
```

Code:



OP:



Observation:

**Bug:** The original code uses nested loops (for i in range(len(nums)) and for j in range(len(nums))), resulting in an  $O(n^2)$  time complexity. **Fix:** Replaced the nested loops with a single loop and used sets to track seen elements and duplicates, reducing the time complexity to  $O(n)$ . **Bug:** The original code uses a list (duplicates) to store duplicates, which requires checking if an element is already in the list (nums[i] not in duplicates), making it inefficient. **Fix:** Used a set (duplicates) to store duplicates, as set lookups are faster (average  $O(1)$  time complexity). **Bug:** The original code checks i != j and compares every pair of elements, which is unnecessary. **Fix:** Simplified the logic by using a single loop. If an element is already in the seen set, it is added to the duplicates set. **Fix:** Converted the duplicates set to a list before returning, as the output format is expected to be a list.

### Task-3:

#### Code Refactoring for Readability

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

Prompt: Refactor messy code into clean, PEP 8–compliant, well-structured code.

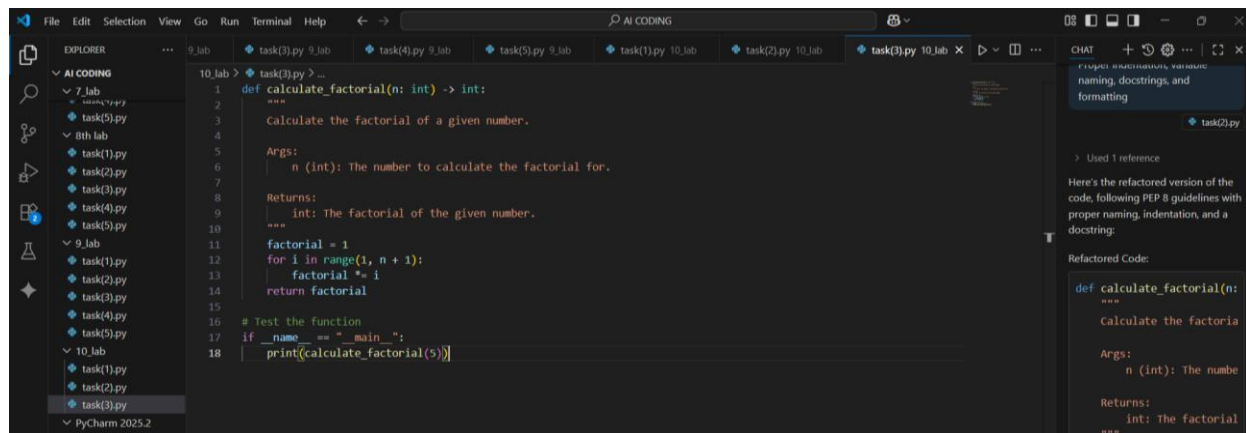
```
def c(n):  
    x=1  
  
    for i in range(1,n+1):  
        x=x*i
```

return x

print(c(5))

Function renamed to calculate\_factorial. Proper indentation, variable naming, docstrings, and formatting

Code:



```
10_lab > task(3).py > ...
1 def calculate_factorial(n: int) -> int:
2     """
3     Calculate the factorial of a given number.
4
5     Args:
6         n (int): The number to calculate the factorial for.
7
8     Returns:
9         int: The factorial of the given number.
10    """
11    factorial = 1
12    for i in range(1, n + 1):
13        factorial *= i
14    return factorial
15
16 # Test the function
17 if __name__ == "__main__":
18     print(calculate_factorial(5))
```

OP:



```
> modules [1, 2]
> plugins PS C:\Users\tharu\OneDrive\Desktop\AI CODING> ^C
> skeletons PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
> build.txt PS C:\Users\tharu\OneDrive\Desktop\AI CODING> cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Program Files\Python
11\python.exe' 'c:\Users\tharu\OneDrive\Desktop\AI CODING\venv\python\python.exe' -u -python.debuggy-2025.10.0-win32-x64\lib\debuggy\launcher' '50164'
'-' 'c:\Users\tharu\OneDrive\Desktop\AI CODING\10_lab\task(3).py'
120 PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
> TIMELINE
```

Observation: Renamed c to calculate\_factorial to make the function name descriptive and meaningful. Renamed x to factorial for clarity. Added a Google-style docstring to describe the function's purpose, arguments, and return value. Ensured proper indentation and spacing.

Added type hints for the function argument and return value. Wrapped the test call in an if `__name__ == "__main__":` block to ensure the function is only executed when the script is run directly.

Task-4:

## Security and Error Handling Enhancement

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.

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

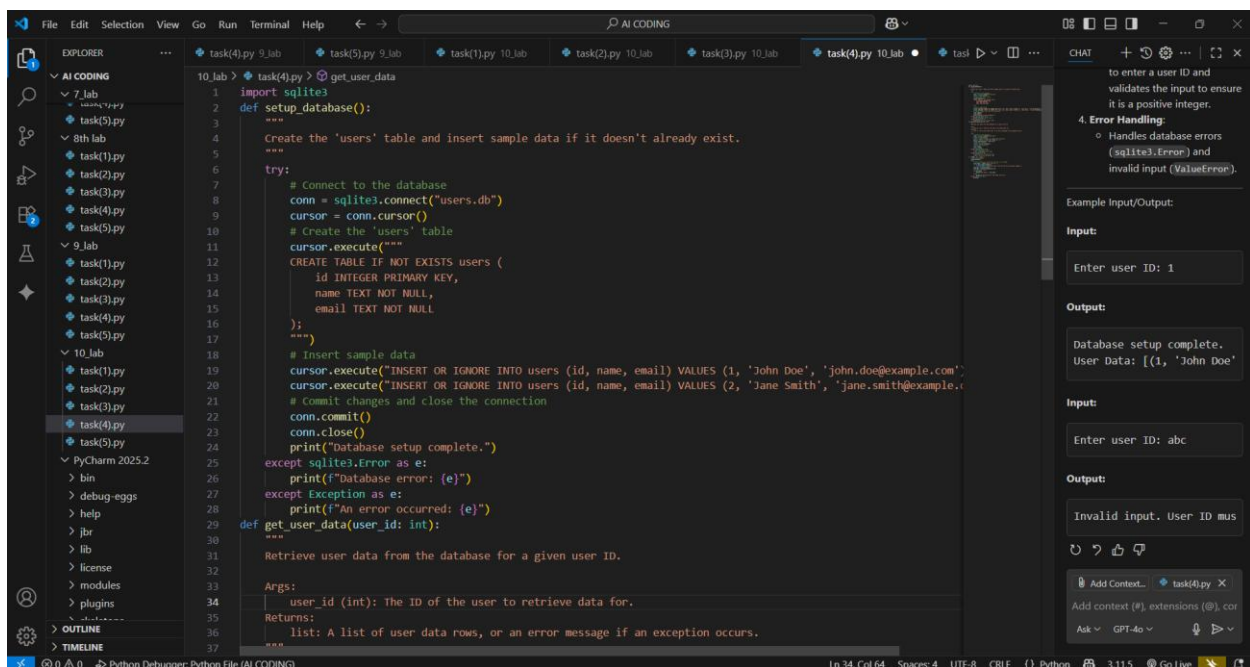
```
    cursor.execute(query)
```

```
    result = cursor.fetchall()
```

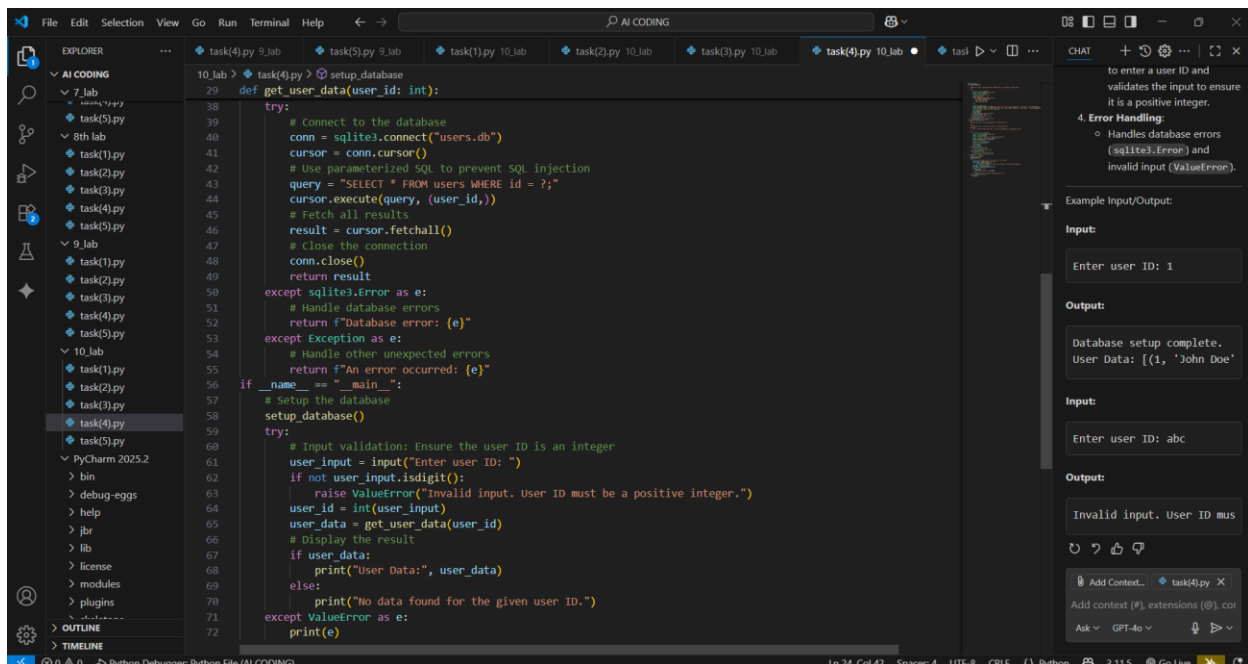
```
conn.close()
return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input)).
```

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

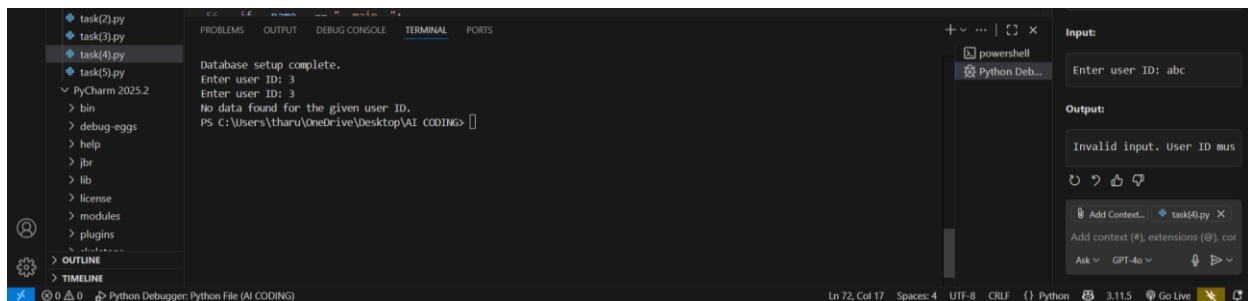
Code:







OP:



Observation:

This Python script is designed to manage a simple SQLite database for user data. It includes two main functions: `setup\_database` and `get\_user\_data`. The `setup\_database` function ensures that the database (users.db) is properly initialized. It creates a `users` table if it doesn't already exist and inserts sample data for testing purposes. This function uses the `CREATE TABLE IF NOT EXISTS` SQL statement to avoid duplicate table creation and the `INSERT OR IGNORE` statement to prevent duplicate entries. The `get\_user\_data` function retrieves user data for a given user ID using parameterized SQL queries (`?`)

placeholders) to prevent SQL injection attacks. It connects to the database, executes the query, fetches the results, and closes the connection. The script also includes robust error handling to manage database-related errors (``sqlite3.Error``) and unexpected exceptions. In the ``if __name__ == "__main__":`` block, the script first sets up the database and then prompts the user to enter a user ID. The input is validated to ensure it is a positive integer, and if valid, the ``get_user_data`` function is called to fetch and display the user data. If the user ID is invalid or no data is found, appropriate error messages are displayed. This script demonstrates best practices for database management, including input validation, parameterized queries, and proper resource handling.

#### Task-5:

Automated Code Review Report Generation

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

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"))
```

include: Missing docstrings. Inconsistent formatting (indentation, inline return). Missing error handling for division by zero. Non-descriptive function/variable names. Suggestions for readability and PEP 8 compliance

Code:

