AI ASSISTED CODING

# Assignment-10.4 :

**Lab 10 – Code Review and Quality: Using AI to Improve Code Quality and Readability**

**2403A52107**

**Pavani Voddepalli**

**Batch : 24BTCAIAI05**

**Task 1: Syntax and Error Detection**

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

# buggy\_code\_task1.py

def add\_numbers(a, b)

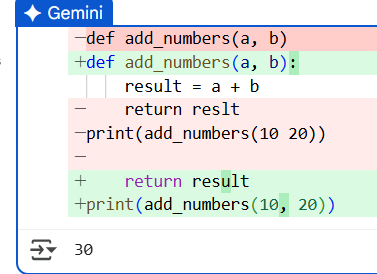
result = a + b

return reslt

print(add\_numbers(10 20))

**PROMPT :** **Generate a corrected code by identifying the errors in the given code**

**CODE , OUTPUT , EXPLANATION :**



* This code defines a simple Python function called add\_numbers that takes two arguments, a and b. It calculates their sum and returns the result. The code then calls this function with the values 10 and 20 and prints the returned value, which is 30.

**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])

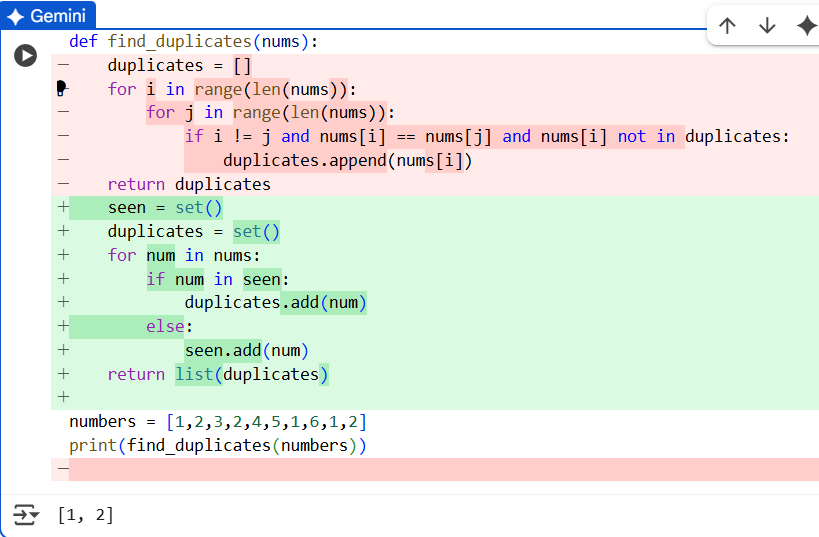
return duplicates

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

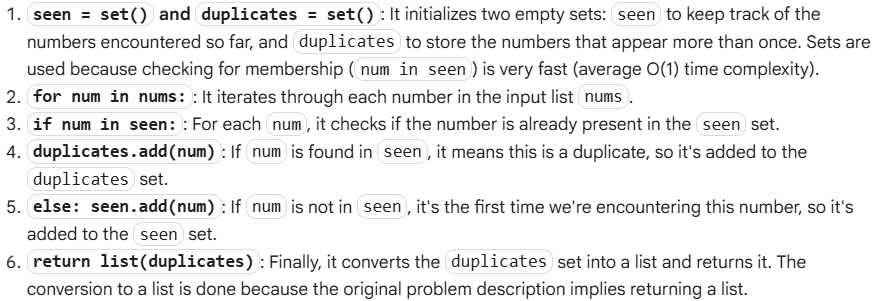
print(find\_duplicates(numbers))

**PROMPT :** **Generate an efficient logic for the given code to optimize using sets for duplicate detection**

**CODE , OUTPUT :**



**EXPLANATION :**

****

**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):

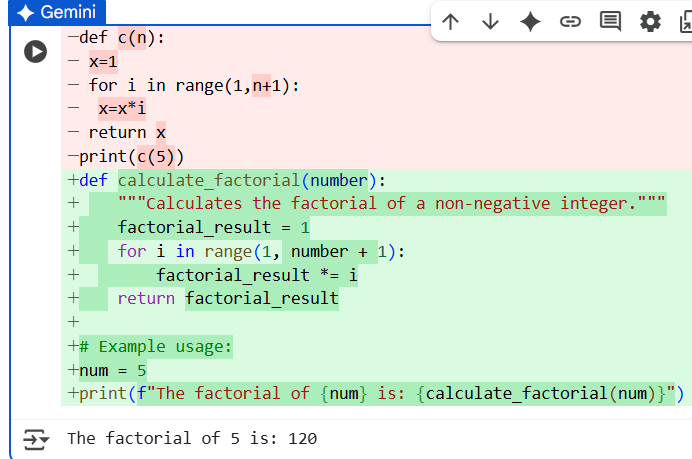
x=x\*i

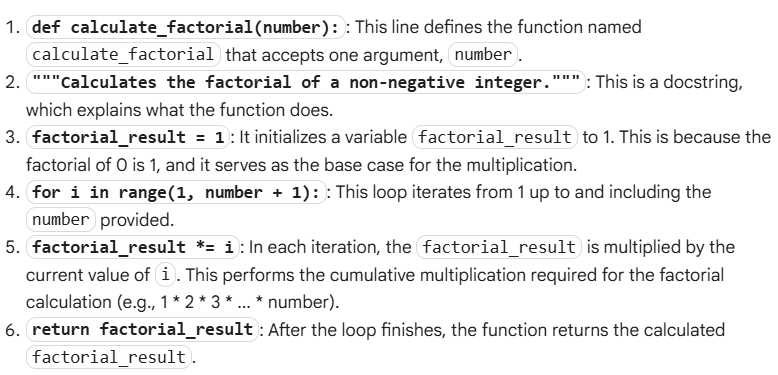
return x

print(c(5))

**PROMPT :** **Generate a structured code to refactor ,rename for readability format**

**CODE , OUTPUT , EXPLANATION :**





**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()

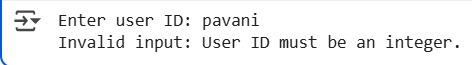
return result

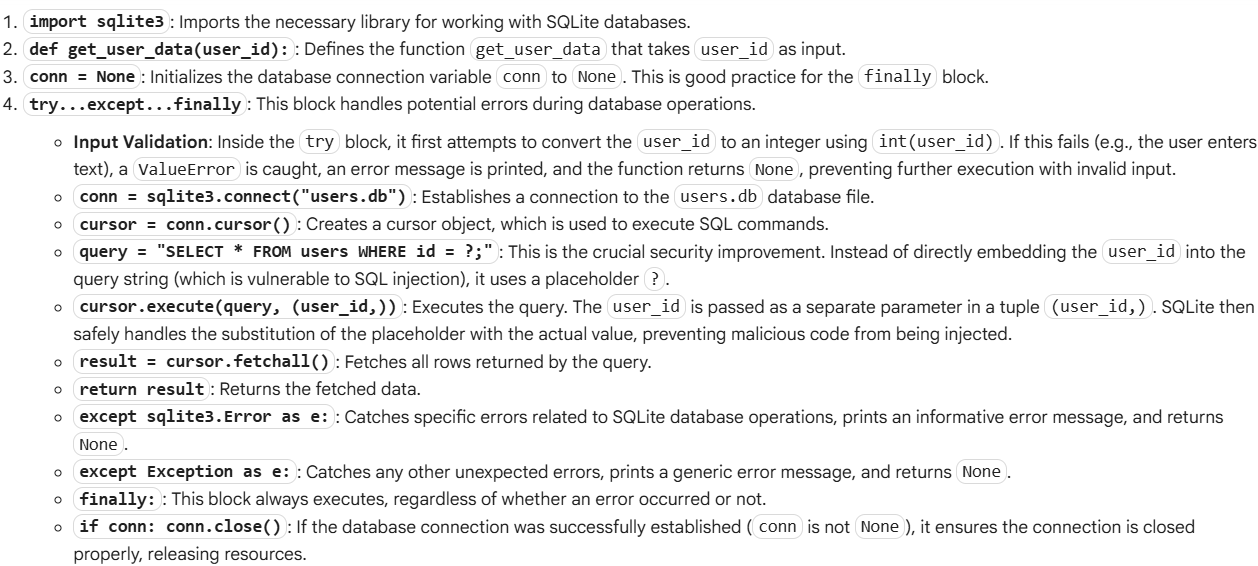
user\_input = input("Enter user ID: ")

print(get\_user\_data(user\_input))

**PROMPT :** **Generate the Security and Error Handling Enhancement Task: Add security practices and exception handling to the code for Safe query using parameterized SQL (? placeholders). Try-except block for database errors. Input validation before query execution.**

**CODE , OUTPUT :****:**



**EXPLANATION :**  **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"))

**PROMPT : Generate a review report for this messy code mentioning 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**

**CODE , OUTPUT , EXPLANATION :**

Code Review Report for the provided `calc` function.

Overall Assessment:

The code is functional but lacks readability, consistency, and robustness. It does not adhere to standard Python style guidelines (PEP 8) and has potential error handling issues.

Detailed Findings:

1. \*\*Missing Docstrings:\*\*

\* The function `calc` lacks a docstring explaining its purpose, arguments, and what it returns. This makes it difficult for other developers (or even the original author) to understand its functionality without reading the entire code.

2. \*\*Inconsistent Formatting:\*\*

\* \*\*Indentation:\*\* The indentation within the `if/elif/else` blocks is inconsistent. Some blocks are indented with 1 space, while others use 2 spaces. PEP 8 recommends 4 spaces for indentation.

\* \*\*Inline Return:\*\* The `elif z=="sub": return x-y` line has the `return` statement on the same line as the `elif` condition. While syntactically valid, it's generally preferred to have the `return` statement on a new, indented line for better readability, consistent with the other `return` statements.

3. \*\*Missing Error Handling for Division by Zero:\*\*

\* The code does not handle the case where the operation is "div" and the denominator `y` is zero. This will result in a `ZeroDivisionError` and crash the program if this scenario occurs, as shown in the example usage `print(calc(10,0,"div"))`.

4. \*\*Non-Descriptive Function/Variable Names:\*\*

\* \*\*Function Name:\*\* The function name `calc` is too short and does not clearly indicate what kind of calculation it performs. A name like `perform\_basic\_operation` or `calculator` would be more descriptive.

\* \*\*Variable Names:\*\* The variable names `x`, `y`, and `z` are not descriptive. It's unclear what `x` and `y` represent (operands?) and what `z` represents (the operation?). More meaningful names like `operand1`, `operand2`, and `operation` would significantly improve understanding.

5. \*\*Suggestions for Readability and PEP 8 Compliance:\*\*

\* Add a clear docstring to the `calc` function.

\* Use consistent indentation of 4 spaces.

\* Place the `return` statement on a new line within each conditional block.

\* Rename the function to a more descriptive name (e.g., `perform\_basic\_operation`).

\* Rename variables `x`, `y`, and `z` to more descriptive names (e.g., `operand1`, `operand2`, `operation`).

\* Implement a check for division by zero in the "div" case and handle it gracefully (e.g., return an error message or `None`, or raise a specific exception).

\* Consider using a dictionary to map operation strings to functions, which can make the code more extensible and potentially cleaner than a long `if/elif` chain for more operations.

15

Error: Division by zero is not allowed.