AI-ASSISTED CODING

ASSIGNMENT-10.3

Code Review and Quality: Using AI to Improve Code Quality and Readability

NAME: GATTU VARSHITH

HALL-TICKET: 2403A510E3

BATCH NO: 05

**TASK-1**: **Syntax and Error Detection**  
**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))

**Prompt:**

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))

**CODE GENERATED:**

# fixed\_code\_task1.py

def add\_numbers(a, b):   # Added missing colon

    result = a + b       # Fixed indentation

    return result        # Fixed typo: reslt -> result

print(add\_numbers(10, 20))  # Added missing comma between arguments

**OUTPUT:**

****

**OBSERVATION :**

The original code had syntax errors (missing colon, missing comma), indentation issues, and a variable name typo. After fixing these, the function works correctly and outputs 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: **:** 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))

CODE GENERATED:

# optimized\_code\_task2.py

def find\_duplicates(nums):

    seen = set()

    duplicates = set()

    for num in nums:

        if num in seen:

            duplicates.add(num)

        else:

            seen.add(num)

    return list(duplicates)

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

print(find\_duplicates(numbers))

OUTPUT:  


OBSERVATION:

The function checks each number once, remembers numbers it has already seen, and collects those that appear again into a duplicates list. This makes it faster and avoids unnecessary repeated comparisons.

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:

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))

CODE GENERATED:

# refactored\_code\_task3.py

def factorial(n: int) -> int:

    """Calculate the factorial of a given number.

    Args:

        n (int): A non-negative integer.

    Returns:

        int: The factorial of n.

    """

    result = 1

    for i in range(1, n + 1):

        result \*= i

    return result

if \_\_name\_\_ == "\_\_main\_\_":

    print(factorial(5))

OUTPUT:



OBSERVATION:

The refactored code is easier to read and understand because it uses meaningful names (factorial, result), follows Python style rules, and includes a docstring to explain what the function does.

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: **:** 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))

CODE GENERATED:

# fixed\_code\_task4\_with\_data.py

import sqlite3

def setup\_database():

    """

    Creates the users table and inserts sample data if not already present.

    """

    with sqlite3.connect("users.db") as conn:

        cursor = conn.cursor()

        # Create table if it doesn't exist

        cursor.execute("""

            CREATE TABLE IF NOT EXISTS users (

                id INTEGER PRIMARY KEY,

                name TEXT NOT NULL,

                email TEXT NOT NULL

            )

        """)

        # Insert sample data (only if table is empty)

        cursor.execute("SELECT COUNT(\*) FROM users")

        if cursor.fetchone()[0] == 0:

            sample\_data = [

                (1, "Alice", "alice@example.com"),

                (2, "Bob", "bob@example.com"),

                (3, "Charlie", "charlie@example.com"),

            ]

            cursor.executemany("INSERT INTO users VALUES (?, ?, ?)", sample\_data)

            conn.commit()

def get\_user\_data(user\_id):

    """

    Safely fetch user data from the database by user\_id.

    """

    try:

        with sqlite3.connect("users.db") as conn:

            cursor = conn.cursor()

            # ✅ Safe parameterized query

            cursor.execute("SELECT \* FROM users WHERE id = ?", (user\_id,))

            result = cursor.fetchall()

            return result

    except sqlite3.Error as e:

        print(f"Database error: {e}")

        return []

# --- Main Program ---

if \_\_name\_\_ == "\_\_main\_\_":

    setup\_database()

    try:

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

        if not user\_input.isdigit():

            raise ValueError("User ID must be a valid integer.")

        user\_id = int(user\_input)

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

    except ValueError as ve:

        print(f"Invalid input: {ve}")

OUTPUT:  


OBSERVATION:

The setup script creates a users table with some sample records, so when the secure query code runs, it can safely fetch real user data for valid IDs and handle invalid or malicious inputs properly.

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.  
# 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"))

CODE GENERATED:

def calculate(x, y, operation):

    """

    Perform basic arithmetic operations.

    Args:

        x (float): First number.

        y (float): Second number.

        operation (str): One of 'add', 'sub', 'mul', 'div'.

    Returns:

        float | None: Result of the operation, or None if invalid.

    """

    try:

        if operation == "add":

            return x + y

        elif operation == "sub":

            return x - y

        elif operation == "mul":

            return x \* y

        elif operation == "div":

            if y == 0:

                raise ZeroDivisionError("Division by zero is not allowed.")

            return x / y

        else:

            print("Invalid operation.")

            return None

    except Exception as e:

        print(f"Error: {e}")

        return None

# Example usage

print(calculate(10, 5, "add"))   # 15

print(calculate(10, 0, "div"))   # Error message

OUTPUT:  
A black background with white text

AI-generated content may be incorrect.

OBSERVATION:

The code works for basic math operations, but it is messy, hard to read, and breaks on division by zero or invalid inputs because it lacks proper error handling and consistent return values.