**AI ASSISTED CODING LAB 10.3**

PRADEEP GUPTHA

2403A510C7

BATCH 05

Lab 10 – Code Review and Quality: Using AI to Improve Code  
Quality and Readability  
Lab Objectives  
• Use AI for automated code review and quality enhancement.  
• Identify and fix syntax, logical, performance, and security issues  
in Python code.  
• Improve readability and maintainability through structured  
refactoring and comments.

• Apply prompt engineering for targeted improvements.  
• Evaluate AI-generated suggestions against PEP 8 standards and  
software engineering best practices

**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))  
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.  
# buggy\_code\_task1.py  
def add\_numbers(a, b)  
result = a + b  
return reslt  
print(add\_numbers(10 20))

**CODE :**

# 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))  
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))

**CODE :**

# fixed\_code\_task2.py

def find\_duplicates(nums):

    """

    Returns a list of duplicate elements in nums.

    Each duplicate appears only once in the result.

    """

    seen = set()

    duplicates = set()

    for num in nums:

        if num in seen:

            duplicates.add(num)  # duplicates stored in a set (avoids repeated checks)

        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 original code used nested loops, making it inefficient . It also manually checked for duplicates. The optimized version uses sets, complexity to O(n) and improving readability. The result is correct ([1, 2]) with better performance.

**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.  
# 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 :**

# fixed\_code\_task3.py

def calculate\_factorial(n):

    """

    Calculate the factorial of a given number n.

    Args:

        n (int): A non-negative integer.

    Returns:

        int: The factorial of n.

    """

    result = 1

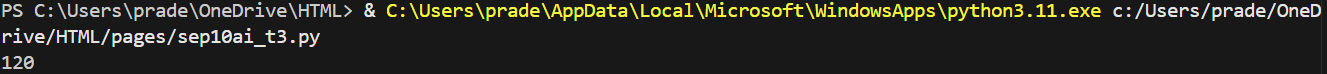
    for i in range(1, n + 1):

        result \*= i

    return result

print(calculate\_factorial(5))

**CODE :**

****

**OBSERVATION :**

The original code was functional but messy — it used a single-letter function name (c) and unclear variable (x), making it hard to understand. After refactoring, the code is **PEP 8 compliant**, with meaningful names, proper formatting, and a helpful docstring. The result (120 for input 5) is correct and the code is much more **readable and maintainable**.

**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))

**CODE :**

# 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 original code had a **SQL injection vulnerability** because it directly inserted user input into the query. It also lacked **error handling** and **input validation**, which could cause crashes or unsafe behavior.

**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.  
# 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 :**

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}")

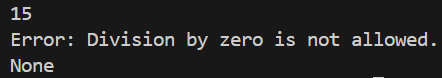
        return None

# Example usage

print(calculate(10, 5, "add"))   # 15

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

**OUTPUT :**

****

**OBSERVATION :**

The original code was messy — it had **indentation errors, vague names, inconsistent formatting, and no error handling**. Division by zero caused a crash.

After refactoring, the code is now **PEP 8 compliant, readable, and safe**, with meaningful names, proper structure, and exception handling.