**AI ASSISTED CODING**

**LAB ASSIGNMENT-10.3**

**2403A510E4-BATCH 05**

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.  
Week5 -  
Thursday

• 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))

**GENERATED CODE:**

# buggy\_code\_task1.py

def add\_numbers(a, b):  # Added colon at the end of function definition

    result = a + b      # Fixed indentation

    return result       # Fixed variable name from 'reslt' to 'result'

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

**OUTPUT:**

**30**

**OBSERATION:**

The code in buggy\_code\_task1.py was corrected by making several important fixes. A colon was added at the end of the function definition def add\_numbers(a, b): to ensure proper syntax, and the indentation of the function body was fixed so that the statements are correctly nested inside the function. The variable name typo reslt was corrected to result, which prevents a NameError during execution. In the print statement, a missing comma between the arguments 10 and 20 was added, allowing the function to receive two inputs properly. With these corrections, the program now runs successfully and produces the correct output 30. Overall, the errors related to syntax, indentation, and variable naming were resolved, making the code functional and clean.

**Task 2: Logical and Performance Issue Review**

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))

**GENERATED CODE:**

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:**

[1, 2]

**OBSERVATION:**

The function find\_duplicates(nums) is designed to identify duplicate elements in a list. It uses two sets: seen to track elements that have already appeared, and duplicates to store values that occur more than once. During iteration, if a number is already in the seen set, it is added to duplicates; otherwise, it gets stored in seen. Finally, the function returns the duplicates as a list. In the given input list [1, 2, 3, 2, 4, 5, 1, 6, 1, 2], the numbers 1 and 2 appear multiple times, so the output will be [1, 2] (order may vary because sets are unordered). The code correctly implements duplicate detection using efficient set operations, making it both clean and optimal in terms of performance.

**Task 3: Code Refactoring for Readability**

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))

**GENERATED CODE:**

def calculate\_factorial(n):

    """

    Calculate the factorial of a given positive integer n.

    Args:

        n (int): The number to calculate the factorial for.

    Returns:

        int: The factorial of n.

    """

    result = 1

    for i in range(1, n + 1):

        result \*= i

    return result

print(calculate\_factorial(5))

**OUTPUT:**

120

**OBSERVATION:**

The function calculate\_factorial(n) correctly computes the factorial of a given positive integer using an iterative approach. It initializes result to 1 and then multiplies it by every number from 1 up to n using a for loop. This ensures that the product accumulates step by step, resulting in the factorial value. The docstring is well-written, clearly explaining the purpose of the function, its argument, and its return type. For the input 5, the loop executes as 1 × 2 × 3 × 4 × 5, giving the final result 120, which is correctly returned and printed. Overall, the code is clean, efficient, and produces the expected factorial output.

**Task 4: Security and Error Handling Enhancement**

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))

**GENERATED CODE**:

import sqlite3

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

    """

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

    Uses parameterized queries to prevent SQL injection.

    Handles database errors gracefully.

    """

    try:

        conn = sqlite3.connect("users.db")

        cursor = conn.cursor()

        query = "SELECT \* FROM users WHERE id = ?;"

        cursor.execute(query, (user\_id,))

        result = cursor.fetchall()

    except sqlite3.DatabaseError as e:

        print(f"Database error: {e}")

        result = []

    finally:

        if 'conn' in locals():

            conn.close()

    return result

def get\_valid\_user\_id():

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

    if not user\_input.isdigit():

        print("Invalid input. Please enter a numeric user ID.")

        return None

    return int(user\_input)

user\_id = get\_valid\_user\_id()

if user\_id is not None:

    print(get\_user\_data(user\_id))

**OBSERVATION:**

 **Security:** The get\_user\_data function uses a **parameterized query** (SELECT \* FROM users WHERE id = ?;). This is a crucial security practice that prevents **SQL injection attacks** by separating the SQL command from the user-provided data.

 **Input Validation:** The get\_valid\_user\_id function validates user input to ensure it is a numeric digit. This prevents non-numeric input from causing errors when passed to the database query and improves the overall reliability of the script.

 **Error Handling:** The try...except...finally block in get\_user\_data gracefully handles potential sqlite3.DatabaseError exceptions. This ensures that the program doesn't crash if there's a problem connecting to or querying the database.

 **Resource Management:** The finally block guarantees that the database connection (conn) is always closed, regardless of whether an error occurred. This is a good practice to free up system resources and prevent connection leaks.

**Task 5: Automated Code Review Report Generation**

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"))

**GENERATED CODE:**

def calculate(x, y, operation):

"""

Perform a basic arithmetic operation.

Args:

x (int/float): First number

y (int/float): Second number

operation (str): Operation type ("add", "sub", "mul", "div")

Returns:

int/float | str: Result of operation or error message

"""

try:

if operation == "add":

return x + y

elif operation == "sub":

return x - y

elif operation == "mul":

return x \* y

elif operation == "div":

return x / y

else:

return "Invalid operation"

except ZeroDivisionError:

return "Error: Division by zero"

print(calculate(10, 5, "add")) # 15

print(calculate(10, 0, "div")) # Error: Division by zero

**OBSERATION:**

The function calculate(x, y, operation) is designed to perform basic arithmetic operations in a clean and structured way. It accepts two numeric inputs (x and y) and an operation type as a string ("add", "sub", "mul", or "div"). Based on the operation, it performs the corresponding calculation and returns the result. If the operation provided does not match any of the valid options, it returns an "Invalid operation" message, ensuring graceful handling of incorrect inputs.

The function also includes exception handling for division by zero using a try-except block. If the user attempts to divide a number by zero, it returns a clear error message "Error: Division by zero" instead of crashing the program. For the given test cases, calculate(10, 5, "add") correctly returns 15, while calculate(10, 0, "div") safely returns the error message without causing a runtime exception.

Overall, the code is **robust, user-friendly, and well-documented**. It demonstrates good programming practices such as error handling, clear docstrings, and structured conditional logic.