LAB ASSIGNMENT – 10.4

NAME : M.SHIVA

HALL.NO : 2403A52377

BATCH.NO : AI 14

ASSIGNMENT.NO : 10.4

PROMPT 01

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: AND explain what was fixed in point wise

CODE :

```python
# buggy_code_task1.py
def add_numbers(a, b):
    result = a + b
    return result

print(add_numbers(10, 20))
```

OUTPUT :

```
30
```

CODE EXPLANATION :

- Syntax Error: Added a colon (:) at the end of the function definition line def add_numbers(a, b).
- Indentation Error: Indented the lines result = a + b and return result to be inside the function block.
- Variable Error: Corrected the variable name reslt to result in the return statement to match the variable where the sum is stored.
- Syntax Error: Added a comma (,) between the arguments 10 and 20 in the function call print(add_numbers(10 20)).

PROMPT 02 :

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)) EXPLAIN THE OPTIMISATION IN POINT WISE

CODE :

```python
# buggy_code_task2.py
def find_duplicates(nums):
    seen = set()
    duplicates = []
    for num in nums:
        if num in seen and num not in duplicates:
            duplicates.append(num)
        seen.add(num)
    return duplicates

numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

OUTPUT :

```
[2, 1]
```

CODE EXPLANATION :

- Using a set for efficient lookups: The original code used nested loops, resulting in a time complexity of O(n^2). The optimized code uses a set called seen to keep track of numbers encountered so far. Checking if a number is in a set is an O(1) operation on average.
- Single pass: The optimized code iterates through the list only once, checking for duplicates as it goes. This reduces the time complexity to O(n).
- Avoiding redundant checks: The condition num not in duplicates prevents adding the same duplicate multiple times to the duplicates list.

PROMPT 03 :

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)) .provide a more readable version GIVE EXPLANATION IN POINT WISE

CODE :

```python
# buggy_code_task3.py

def calculate_factorial(number):
    """Calculates the factorial of a non-negative integer."""
    factorial = 1
    for i in range(1, number + 1):
        factorial *= i
    return factorial

# Example usage:
num = 5
print(f"The factorial of {num} is: {calculate_factorial(num)}")
```

OUTPUT :

```
The factorial of 5 is: 120
```

CODE EXPLANATION :

- Descriptive Function Name: Changed the function name from c to calculate_factorial to clearly indicate its purpose.
- Descriptive Variable Names: Changed the variable names from n to number and x to factorial for better understanding of what they represent.
- Docstring: Added a docstring to the function to explain what it does, its arguments, and what it returns. This is a standard practice in Python for code documentation.
- Improved Spacing and Indentation: Ensured consistent spacing around operators and proper indentation to enhance readability, adhering to PEP 8 guidelines.
- Clearer Calculation: Used the augmented assignment operator *= for a more concise way to update the factorial variable.
- Example Usage: Included an example of how to use the function with a clear variable name (num) and an f-string for a more readable output message.

PROMPT 04 :

Add security practices and exception handling to the code.# buggy_code_task4.py import sqlite3def 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)) output shoulb be Safe query using parameterized SQL (? placeholders). Try-except block for database errors. Input validation before query execution. GIVE EXLPLANATION POINT WISE

CODE :

```
import sqlite3

def get_user_data(username):
    """
    Retrieves user data from the database using a safe, parameterized
query.

    Args:
        username: The username to search for.

    Returns:
        The user data if found, otherwise None.
    """
    if not isinstance(username, str) or not username:
        print("Error: Invalid username input.")
        return None

    conn = None
    try:
        conn = sqlite3.connect('my_database.db')
        cursor = conn.cursor()

        # Parameterized SQL query to prevent SQL injection
        query = "SELECT id, email FROM users WHERE username = ?"
        cursor.execute(query, (username,))

        user_data = cursor.fetchone()
        return user_data

    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None
```

```
    finally:
        if conn:
            conn.close()

# Example usage:
user_to_find = "testuser"
user_info = get_user_data(user_to_find)

if user_info:
    print(f"Found user: ID - {user_info[0]}, Email - {user_info[1]}")
else:
    print(f"User '{user_to_find}' not found or an error occurred.")

# Example with invalid input:
invalid_user = 123
get_user_data(invalid_user)
```

OUTPUT :

```
Database error: no such table: users
User 'testuser' not found or an error occurred.
Error: Invalid username input.
```

CODE EXPLANATION :

- Parameterized SQL Query: The line query = "SELECT id, email FROM users WHERE username = ?" uses a placeholder ? for the username. The actual username value is passed as a separate tuple (username,) in the cursor.execute() method. This is the standard and safest way to pass data to SQL queries in Python's sqlite3 module (and most database connectors), as it prevents SQL injection vulnerabilities by ensuring the input is treated as data, not executable code.

- Try-Except Block for Database Errors: The code is wrapped in a try...except sqlite3.Error as e: block. This allows the program to gracefully handle any errors that might occur during database operations (like connection issues, syntax errors in the query, etc.). If an error occurs, it's caught, an error message is printed, and the function returns None instead of crashing the program. The finally block ensures that the database connection is closed even if an error occurs.
- Input Validation: Before executing the query, the code checks if the username input is a string and not empty (if not isinstance(username, str) or not username:). This basic input validation helps prevent unexpected behavior or errors by ensuring that the input is in the expected format. You can add more comprehensive validation based on your specific requirements.

PROMPT 05 :

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")) IT SHOULD BE WITH 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 AND GIVE POINT WISE WXPLANATION

CODE :

```
# buggy_code_task5.py

def perform_operation(num1, num2, operation):
    """
    Performs a basic arithmetic operation based on the provided string.

    Args:
        num1: The first number.
        num2: The second number.
        operation: A string representing the operation ("add", "sub",
"mul", "div").

    Returns:
        The result of the operation, or None if the operation is
invalid or division by zero occurs.
    """
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
    elif operation == "div":
        if num2 == 0:
            print("Error: Division by zero is not allowed.")
            return None
        return num1 / num2
    else:
        print("Error: Invalid operation.")
        return None

# Example usage:
print(perform_operation(10, 5, "add"))
print(perform_operation(10, 0, "div"))
print(perform_operation(10, 5, "sub"))
print(perform_operation(10, 5, "mul"))
print(perform_operation(10, 5, "mod")) # Example of invalid operation
```

OUTPUT :

```
15
Error: Division by zero is not allowed.
None
5
50
Error: Invalid operation.
None
```

CODE EXPLANATION :

- Parameterized SQL Query: The line query = "SELECT id, email FROM users WHERE username = ?" uses a placeholder ? for the username. The actual username value is passed as a separate tuple (username,) in the cursor.execute() method. This is the standard and safest way to pass data to SQL queries in Python's sqlite3 module (and most database connectors), as it prevents SQL injection vulnerabilities by ensuring the input is treated as data, not executable code.
- Try-Except Block for Database Errors: The code is wrapped in a try...except sqlite3.Error as e: block. This allows the program to gracefully handle any errors that might occur during database operations (like connection issues, syntax errors in the query, etc.). If an error occurs, it's caught, an error message is printed, and the function returns None instead of crashing the program. The finally block ensures that the database connection is closed even if an error occurs.
- Input Validation: Before executing the query, the code checks if the username input is a string and not empty (if not isinstance(username, str) or not username:). This basic input validation helps prevent unexpected behavior or errors by ensuring that the input is in the expected format. You can add more comprehensive validation based on your specific requirements.

AI-Generated Code Review Report :

Here is a review of the provided Python code, highlighting areas for improvement:

- Missing Docstrings: The function calc lacks a docstring. A docstring should be included to explain what the function does, its parameters (x, y, z), and what it returns. This improves code documentation and makes it easier for others (or yourself in the future) to understand how to use the function.
- Inconsistent Formatting:
  - Indentation: The indentation is inconsistent, particularly in the else block where the print("wrong") statement is not properly indented under the else.
  - Inline Return: The return x-y statement in the elif z=="sub": block is on the same line as the elif. While Python allows this, it's generally more readable to have the return statement on a new line with proper indentation, consistent with the other elif blocks.
- Missing Error Handling for Division by Zero: The code does not handle the case where z is "div" and y is 0. This will result in a ZeroDivisionError at runtime. Robust code should include error handling (e.g., a try-except block) to gracefully manage this situation and provide a more informative message to the user.
- Non-Descriptive Function/Variable Names:
  - Function Name: The function name calc is very generic. A more descriptive name, such as perform_operation or simple_calculator, would better convey the function's purpose.
  - Variable Names: The variable names x, y, and z are not descriptive. Using names like num1, num2, and operation would make the code's intent clearer.
- Suggestions for Readability and PEP 8 Compliance:

- Follow PEP 8: Adhering to PEP 8 guidelines for code style (like consistent indentation, spacing around operators, etc.) significantly improves readability and maintainability.
- Use a Dictionary for Operations: Instead of a long chain of elif statements, you could use a dictionary to map the operation strings ("add", "sub", etc.) to the corresponding functions or lambda expressions. This can make the code more concise and easier to extend with new operations.
- Implement Error Handling: Add a try-except ZeroDivisionError block within the division case to handle division by zero gracefully.
- Return a Meaningful Value on Error: Instead of just printing "wrong" in the else block, consider returning a specific value (like None) or raising a custom exception to indicate an invalid operation.