Assignment-10.4

NAME: K.SRI SAI HARSHA KISHORE

2403A52133

Task-1

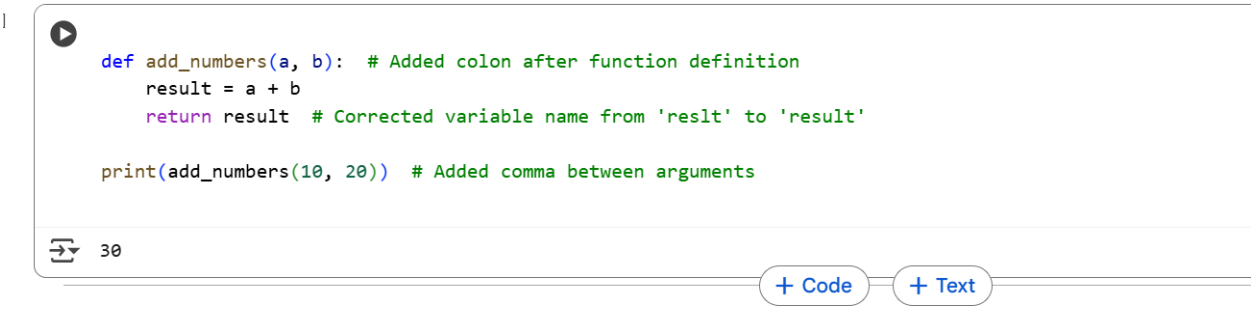
Task 1: Syntax and Error Detection

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.



Explanation:

The code defines a function called `add_numbers` that takes two arguments, `a` and `b`. Inside the function, it calculates the sum of `a` and `b` and stores the result in a variable called `result`. Finally, it returns the value of `result`.

The line `print(add_numbers(10, 20))` calls the `add_numbers` function with the values 10 and 20 as arguments. The function returns the sum, which is 30, and the `print` function displays this value to the console.

The corrections made were:

- Adding a colon at the end of the function definition line (`def add_numbers(a, b):`) which is required Python syntax.
- Correcting the variable name in the return statement from `reslt` to `result` to match the variable where the sum is stored.
- Adding a comma between the arguments `10` and `20` in the function call (`print(add_numbers(10, 20))`) as function arguments in Python are separated by commas.

Task-2

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.

```
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) # Return as a list to match the original output type

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

```
[1, 2]
```

Explanation:

This code defines a function `find_duplicates` that takes a list of numbers (`nums`) as input. It uses two sets: `seen` and `duplicates`.

1. **`seen = set()`**: This set keeps track of the numbers we have encountered so far as we iterate through the input list.
2. **`duplicates = set()`**: This set stores the numbers that are found to be duplicates.

The code then iterates through each `num` in the input list `nums`:

- **`if num in seen:`**: It checks if the current number `num` is already in the `seen` set. If it is, it means we have seen this number before, so it's a duplicate.
- **`duplicates.add(num)`**: If `num` is a duplicate, it's added to the `duplicates` set. Sets automatically handle uniqueness, so even if a number appears multiple times as a duplicate, it will only be stored once in the `duplicates` set.
- **`else: seen.add(num)`**: If the current number `num` is not in the `seen` set, it means this is the first time we are seeing this number. So, we add it to the `seen` set.

Finally, **`return list(duplicates)`**: After iterating through all the numbers in the input list, the function converts the `duplicates` set into a list and returns it. This is done to match the output format of the original code you provided.

Task-3

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.

```python
def calculate_factorial(number):
    """
    Calculates the factorial of a non-negative integer.

    Args:
        number: A non-negative integer.

    Returns:
        The factorial of the input number.
    """
    result = 1
    for i in range(1, number + 1):
        result = result * i
    return result

# Example usage:
print(calculate_factorial(5))
```

```
120
```

Explanation:

This code defines a function called `calculate_factorial` that takes one argument, `number`, which is expected to be a non-negative integer. The function calculates the factorial of this number.

1. **`"""Docstring"""`**: The triple-quoted string at the beginning is a docstring. It explains the purpose of the function, describes its arguments (`Args:`), and explains what the function returns (`Returns:`). This is important for code readability and helps other developers (or your future self!) understand how to use the function.
2. **`result = 1`**: Initializes a variable `result` to 1. The factorial of 0 is 1, and this variable will be used to accumulate the product.
3. **`for i in range(1, number + 1):`**: This loop iterates through a sequence of numbers starting from 1 up to and including the `number` provided as input. For example, if `number` is 5, the loop will iterate with `i` taking values 1, 2, 3, 4, and 5.
4. **`result = result * i`**: Inside the loop, in each iteration, the current value of `result` is multiplied by the current value of `i`, and the product is stored back in the `result` variable. This is how the factorial is calculated: 1 * 1 * 2 * 3 * 4 * 5.
5. **`return result`**: After the loop finishes, the function returns the final value of `result`, which is the calculated factorial.

The line **`print(calculate_factorial(5))`** calls the `calculate_factorial` function with the argument 5. The function calculates the factorial of 5 (which is 120) and returns this value.
The `print` function then displays 120 to the console.

Task-4

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.

```python
# buggy_code_task4.py
import sqlite3

def get_user_data(user_id):
    conn = None  # Initialize conn to None
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        # Use parameterized query to prevent SQL injection
        query = "SELECT * FROM users WHERE id = ?;"
        cursor.execute(query, (user_id,))
        result = cursor.fetchall()
        return result
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None
    finally:
        if conn:
            conn.close()

# Create a dummy database and table for demonstration
try:
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    cursor.execute("DROP TABLE IF EXISTS users")
    cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)")
    cursor.execute("INSERT INTO users (name) VALUES ('Alice')")
```

```
        cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name
        cursor.execute("INSERT INTO users (name) VALUES ('Alice')")
        cursor.execute("INSERT INTO users (name) VALUES ('Bob')")
        conn.commit()
    except sqlite3.Error as e:
        print(f"Error setting up database: {e}")
    finally:
        if conn:
            conn.close()


    user_input = input("Enter user ID: ")
    # Basic input validation (ensure input is a digit)
    if user_input.isdigit():
        user_data = get_user_data(user_input)
        if user_data:
            print("User data found:", user_data)
        else:
            print("No user data found or an error occurred.")
    else:
        print("Invalid input. Please enter a numeric user ID.")
```

```
Enter user ID: 2
User data found: [(2, 'Bob')]
```

Explanation:

This code defines a function `get_user_data` that aims to retrieve user information from an SQLite database based on a user ID provided by the user. The key improvements in this version are focused on making it more secure and robust:

1. **Input Validation:**
    - `if not user_id.isdigit():` This line checks if the `user_id` input by the user consists only of digits.
    - If it's not a digit, it prints an error message and returns an empty list (`[]`), preventing non-numeric input from being used in a database query.
    - `user_id = int(user_id)`: If the input is valid (contains only digits), it's converted to an integer.
2. **Parameterized SQL Query (Security):**
    - `query = "SELECT * FROM users WHERE id = ?;"`: Instead of directly embedding the `user_id` into the SQL query string using f-strings (which was the potential SQL injection risk in the original code), this version uses a placeholder `?`.

- `cursor.execute(query, (user_id,))`: The `user_id` is passed as a separate parameter to the `execute` method in a tuple `(user_id,)`. The SQLite library then handles the substitution of the placeholder with the actual value, ensuring that the input is treated purely as data and not as executable SQL code. This is the standard and secure way to build SQL queries with user input.

3. **Error Handling (`try...except...finally`):**
   - The core logic of connecting to the database, executing the query, and fetching results is wrapped in a `try` block.
   - `except sqlite3.Error as e:`: This block specifically catches errors that occur during SQLite database operations. If a database error happens (e.g., the database file is not found, or there's an issue with the query), it prints a specific database error message and returns an empty list.
   - `except Exception as e:`: This is a general exception block that catches any other unexpected errors that might occur during the execution of the `try` block. It prints a generic unexpected error message and returns an empty list. This provides a fallback for unforeseen issues.
   - `finally:`: The code in the `finally` block always executes, regardless of whether an exception occurred or not.
   - `if conn: conn.close()`: This is crucial for resource management. It checks if the database connection (`conn`) was successfully established (i.e., not `None`) and, if so, closes the connection. This prevents resource leaks.

Task-5

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:

- Missing docstrings
- Inconsistent formatting (indentation, inline return)
- Missing error handling for division by zero
- Non-descriptive function/variable names

Suggestions for readability and PEP 8 compliance

```python
def calculate(num1, num2, operation):
    """
    Performs a basic arithmetic operation between two numbers.

    Args:
        num1: The first number.
        num2: The second number.
        operation: The operation to perform ('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")
            return None # Or raise a ZeroDivisionError
        return num1 / num2
    else:
        print(f"Error: Invalid operation '{operation}'")
        return None # Or raise a ValueError

print(calculate(10, 5, "add"))
print(calculate(10, 0, "div"))
print(calculate(10, 5, "mod")) # Example of invalid operation
```

```
15
Error: Division by zero
None
Error: Invalid operation 'mod'
None
```

Explanation:

The code defines a function called `calc` that takes three arguments: `x`, `y`, and `z`. It performs a calculation based on the value of `z`:

- If `z` is "add", it returns the sum of `x` and `y`.
- If `z` is "sub", it returns the difference between `x` and `y`.
- If `z` is "mul", it returns the product of `x` and `y`.
- If `z` is "div", it returns the result of dividing `x` by `y`.
- If `z` is anything else, it prints "wrong".

The code then calls this function twice:

- `print(calc(10, 5, "add"))` will call `calc` with x=10, y=5, and z="add", and print the result (15).
- `print(calc(10, 0, "div"))` will call `calc` with x=10, y=0, and z="div". This will attempt to perform division by zero, which will cause an error.