ASSIGNMENT-10.3

TASK-1: 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 the errors in this code.

Code with Output:

```
def add_numbers(a, b):
    result = a + b
    return result

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

```
30
```

Explanation:

```
def add_numbers(a, b):
    result = a + b
    return result
```

- `def add_numbers(a, b):` This line defines the function named `add_numbers` that accepts two parameters, `a` and `b`. The colon at the end is essential for defining the function block.
- `result = a + b` : This line calculates the sum of `a` and `b` and stores it in a variable called `result`.
- `return result` : This line returns the value of `result` from the function.

The next part of the code calls the `add_numbers` function and prints the returned value.

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

- `add_numbers(10, 20)` : This calls the `add_numbers` function with the arguments `10` and `20`.
- `print(...)` : This prints the value returned by the `add_numbers` function, which is the sum of 10 and 20 (i.e., 30).

TASK-2: : 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: Identify the errors in this code.

Code with Output:

```
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))
```

```
[1, 2]
```

EXPLANATION:

```
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
```

1. `def find_duplicates(nums):` : This line defines a function named `find_duplicates` that takes one argument, `nums`, which is expected to be a list of numbers.
2. `duplicates = []` : This line initializes an empty list called `duplicates`. This list will store the unique duplicate numbers found in the `nums` list.
3. `for i in range(len(nums)):` : This is the outer loop. It iterates through the `nums` list using an index `i` from 0 up to (but not including) the length of the list.
4. `for j in range(len(nums)):` : This is the inner loop. For each element at index `i` in the outer loop, this inner loop iterates through the `nums` list again using an index `j` from 0 up to (but not including) the length of the list.
5. `if i != j and nums[i] == nums[j] and nums[i] not in duplicates:` : This is the core of the logic. It checks three conditions:
    - `i != j` : Ensures that the comparison is not between the same element at the same index.
    - `nums[i] == nums[j]` : Checks if the elements at the current indices `i` and `j` are equal.
    - `nums[i] not in duplicates` : Checks if the current duplicate number `nums[i]` has not already been added to the `duplicates` list. This is to ensure that each duplicate number is added only once to the `duplicates` list.
6. `duplicates.append(nums[i])` : If all three conditions in the `if` statement are true, it means a unique duplicate number has been found, and it is added to the `duplicates` list.
7. `return duplicates` : After the loops have finished iterating through all possible pairs of elements, the function returns the `duplicates` list containing the unique duplicate numbers found.

The code then defines a list `numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2]` and calls the `find_duplicates` function with this list, printing the returned list of duplicates.

In this specific example, the output will be `[1, 2]` because the numbers 1 and 2 are the only duplicates in the `numbers` list.

TASK-3: 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: Identify the errors.

Code with Output:

```
def c(n):
 x=1
 for i in range(1,n+1):
  x=x*i
 return x
print(c(5))
```

```
120
```

Explanation:

```
def c(n):
 x=1
 for i in range(1,n+1):
  x=x*i
 return x
print(c(5))
```

1. `def c(n):` : This line defines a function named `c` that takes one argument, `n`. This function is designed to calculate the factorial of `n`.
2. `x = 1` : Inside the function, a variable `x` is initialized to `1`. This variable will store the calculated factorial. We start with 1 because the factorial of 0 is 1, and it's the multiplicative identity.
3. `for i in range(1, n + 1):` : This is a `for` loop that iterates through a sequence of numbers starting from 1 up to and including the value of `n`.
   - `range(1, n + 1)` generates numbers from 1 up to `n`. For example, if `n` is 5, `range(1, 6)` will generate 1, 2, 3, 4, and 5.
4. `x = x * i` : Inside the loop, in each iteration, the current value of `x` is multiplied by the current value of `i`, and the result is assigned back to `x`. This is how the factorial is calculated: `1 * 2 * 3 * ... * n`.
5. `return x` : After the loop finishes, the function returns the final value of `x`, which is the factorial of `n`.

The line `print(c(5))` calls the function `c` with the argument `5`. The function calculates the factorial of 5 (5! = 5 * 4 * 3 * 2 * 1 = 120) and returns the value. The `print` function then displays this returned value to the console.

TASK-4: 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: Identify the errors in this code.

Code with output:

```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')")
```

```python
    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: 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: Identify the errors in this code.

Code with output:

```
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":
        if y == 0:
            return "Error: Division by zero"
        return x / y
    else:
        return "Error: Unknown operation"

# Test cases
print(calc(10, 5, "add"))   # 15
print(calc(10, 0, "div"))   # Error: Division by zero
print(calc(10, 5, "pow"))   # Error: Unknown operation
```

```
15
Error: Division by zero
Error: Unknown operation
```

Explanation:

1. `def calc(x, y, z):` : This line defines a function named `calc` that takes three arguments:
    - `x` : The first number.
    - `y` : The second number.
    - `z` : A string representing the operation to perform ("add", "sub", "mul", or "div").
2. `if z == "add":` : This is the first conditional check. If the value of `z` is the string "add", the code inside this `if` block is executed.
    - `return x + y` : If `z` is "add", the function returns the sum of `x` and `y`.
3. `elif z == "sub":` : This is an "else if" condition. If the previous `if` condition was false and `z` is the string "sub", the code inside this `elif` block is executed.
    - `return x - y` : If `z` is "sub", the function returns the difference between `x` and `y`.
4. `elif z == "mul":` : Another "else if" condition. If the previous conditions were false and `z` is the string "mul", the code inside this `elif` block is executed.
    - `return x * y` : If `z` is "mul", the function returns the product of `x` and `y`.

5. `elif z == "div":` : Another "else if" condition. If the previous conditions were false and `z` is the string "div", the code inside this `elif` block is executed.
    - `if y == 0:` : Inside the "div" block, there's a nested `if` condition to check if the divisor `y` is zero.
        - `return "Error: Division by zero"` : If `y` is zero, the function returns the string "Error: Division by zero" to indicate that division by zero is not possible.
    - `return x / y` : If `y` is not zero, the function returns the result of dividing `x` by `y`.
6. `else:` : This is the final "else" block. If none of the previous `if` or `elif` conditions were true (meaning `z` is not "add", "sub", "mul", or "div"), the code inside this `else` block is executed.
    - `return "Error: Unknown operation"` : The function returns the string "Error: Unknown operation" to indicate that the provided operation is not supported.

The code then includes several `print` statements to test the `calc` function with different inputs and operations:

- `print(calc(10, 5, "add"))` : Calls `calc` with `x=10`, `y=5`, and `z="add"`. It will print `15`.
- `print(calc(10, 0, "div"))` : Calls `calc` with `x=10`, `y=0`, and `z="div"`. Since `y` is 0, it will print `"Error: Division by zero"`.
- `print(calc(10, 5, "pow"))` : Calls `calc` with `x=10`, `y=5`, and `z="pow"`. Since "pow" is not a recognized operation, it will print `"Error: Unknown operation"`.

This function provides a simple way to perform basic arithmetic operations based on a textual command.