ASSIGNMENT-10.3

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Branch : 05

Course : AI Assisted Coding

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))
```

Output:

- Corrected code with proper syntax (: after function, fixed variable name, corrected function call).
- AI should explain what was fixed

Prompt :

#To Identify and fix syntax, indentation, and variable errors in the given task code.

Code :

```
C: > Users > sgoll > AI.py > ...
1    def add_numbers(a, b):
2        result = a + b
3        return result
4
5    print(add_numbers(10, 20))
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
30
```

Observation :

The buggy code demonstrates several fundamental Python syntax and programming errors that are commonly encountered by beginners. The most critical issues include a missing colon after the function definition, which is a basic syntax requirement in Python, and improper indentation throughout the function body. Python's strict indentation rules mean that the code blocks must be properly aligned, and the lack of indentation would cause immediate syntax errors. Additionally, there's a simple but critical typo in the variable name where "reslt" should be "result," which would cause a NameError since the variable was never defined with that misspelled name. The function call also lacks a comma between the two arguments, which would result in a syntax error. These types of errors are quite common when learning Python, as they represent basic syntax rules that differ from other programming languages, particularly the strict indentation requirements and the necessity of colons after control structures. The exercise effectively highlights how even small mistakes can completely prevent code from running, emphasizing the importance of attention to detail in programming syntax.

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))
```

Output:

- More efficient duplicate detection (e.g., using sets).
- AI should explain the optimization

Prompt :

#By using the optimize inefficient logic while keeping the result correct.

Code :

C: > Users > sgoll > AI.py > find_duplicates

```python
def find_duplicates(nums):
    count = {}
    duplicates = []
    for num in nums:
        count[num] = count.get(num, 0) + 1

    for num, freq in count.items():
        if freq > 1:
            duplicates.append(num)

    return duplicates

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

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
[1, 2]
```

Observation :

The original code demonstrates a classic example of inefficient algorithm design that many programmers encounter when first learning to solve problems. The nested loop approach, while logically correct, creates an $O(n^2)$ time complexity that becomes increasingly problematic as the input size grows. The algorithm essentially checks every possible pair of elements in the array, which is computationally wasteful when the goal is simply to identify duplicate values. Additionally, the code includes a redundant check `nums[i] not in duplicates` that performs a linear

search through the duplicates list for each potential duplicate, adding another layer of inefficiency.

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))
```

Output:

Function renamed to calculate_factorial.

Proper indentation, variable naming, docstrings, and formatting.

AI should provide a more readable version

Prompt :

#To make the refactor messy code into clean, PEP 8–compliant, well-structured code.

Code :

C: > Users > sgoll > AI.py > ...

```python
def factorial(n: int) -> int:
    """
    Calculate the factorial of a given number.

    Args:
        n (int): A non-negative integer

    Returns:
        int: The factorial of n (n!)

    Raises:
        ValueError: If n is negative
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

    if n == 0 or n == 1:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i

    return result
def main():
    """Main function to demonstrate factorial calculation."""
    number = 5
    result = factorial(number)
    print(f"The factorial of {number} is {result}")

if __name__ == "__main__":
    main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
The factorial of 5 is 120
```

Observation :

The original code exemplifies several common anti-patterns that make code difficult to read, maintain, and understand. The single letter function name `c` provides no indication of what the function Actually does, while the variable name `x` is equally uninformative,

making the code's purpose completely opaque to anyone reading it. The lack of proper spacing around operators and after commas violates basic Python style guidelines, creating visual clutter that makes the code harder to scan and understand. More critically, the code lacks any form of documentation or type hints, which are essential for code maintainability and collaboration.

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))
```

Output:

Safe query using parameterized SQL (? placeholders).

Try-except block for database errors.

Input validation before query execution

Prompt :

To add security practices and exception handling to the code.

Code :

C: > Users > sgoll > AI.py > ...

```python
import sqlite3
import logging
from typing import Optional, List, Dict, Any
import re
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class DatabaseError(Exception):
    """Custom exception for database-related errors."""
    pass
class SecurityError(Exception):
    """Custom exception for security-related errors."""
    pass
def validate_user_id(user_id: str) -> int:
    """
    Validate and sanitize user ID input.

    Args:
        user_id (str): User ID input to validate

    Returns:
        int: Validated user ID as integer

    Raises:
        SecurityError: If input is invalid or potentially malicious
    """
    if not user_id or not user_id.strip():
        raise SecurityError("User ID cannot be empty")
    user_id = user_id.strip()
    if not re.match(r'^\d+$', user_id):
        raise SecurityError("User ID must contain only numeric characters")

    # Convert to integer and validate range
    try:
        user_id_int = int(user_id)
        if user_id_int <= 0:
            raise SecurityError("User ID must be a positive integer")
        if user_id_int > 999999999:  # Reasonable upper limit
            raise SecurityError("User ID exceeds maximum allowed value")
        return user_id_int
    except ValueError:
        raise SecurityError("User ID must be a valid integer")

def get_user_data(user_id: str) -> Optional[Dict[str, Any]]:
```

C: > Users > sgoll > AI.py > ...

```python
def get_user_data(user_id: str) -> Optional[Dict[str, Any]]:
    """
    Securely retrieve user data from database.

    Args:
        user_id (str): Validated user ID

    Returns:
        Optional[Dict[str, Any]]: User data if found, None otherwise

    Raises:
        DatabaseError: If database operation fails
        SecurityError: If input validation fails
    """
    # Validate input
    try:
        validated_id = validate_user_id(user_id)
    except SecurityError as e:
        logger.warning(f"Security validation failed: {e}")
        raise

    conn = None
    try:
        conn = sqlite3.connect("users.db")
        conn.row_factory = sqlite3.Row  # Enable column access by name

        cursor = conn.cursor()
        query = "SELECT id, username, email, created_at FROM users WHERE id = ?"
        cursor.execute(query, (validated_id,))

        result = cursor.fetchone()

        if result:
            # Convert Row object to dictionary
            user_data = dict(result)
            logger.info(f"Successfully retrieved user data for ID: {validated_id}")
            return user_data
        else:
            logger.info(f"No user found with ID: {validated_id}")
            return None

    except sqlite3.Error as e:
        logger.error(f"Database error: {e}")
        raise DatabaseError(f"Database operation failed: {e}")
    except Exception as e:
```

```python
C: > Users > sgoll > AI.py > ...
    def get_user_data(user_id: str) -> Optional[Dict[str, Any]]:
            logger.error(f"Database error: {e}")
            raise DatabaseError(f"Database operation failed: {e}")
        except Exception as e:
            logger.error(f"Unexpected error: {e}")
            raise DatabaseError(f"Unexpected error occurred: {e}")
        finally:
            # Ensure connection is always closed
            if conn:
                conn.close()
                logger.debug("Database connection closed")
    def main():
        """
        Main function with proper error handling and user interaction.
        """
        try:
            user_input = input("Enter user ID: ")

            if not user_input:
                print("Error: Please enter a user ID")
                return
            user_data = get_user_data(user_input)
            if user_data:
                print(f"User found:")
                print(f"  ID: {user_data['id']}")
                print(f"  Username: {user_data['username']}")
                print(f"  Email: {user_data['email']}")
                print(f"  Created: {user_data['created_at']}")
            else:
                print("No user found with that ID")
        except SecurityError as e:
            print(f"Security Error: {e}")
        except DatabaseError as e:
            print(f"Database Error: {e}")
        except KeyboardInterrupt:
            print("\nOperation cancelled by user")
        except Exception as e:
            logger.error(f"Unexpected error in main: {e}")
            print("An unexpected error occurred. Please try again.")

    if __name__ == "__main__":
        main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
Enter user ID: 123
INFO:__main__:Successfully retrieved user data for ID: 123
User found:
  ID: 123
  Username: alice_brown
  Email: alice@example.com
  Created: 2025-09-10 04:58:07
```

Observation :

The error encountered when running the secure code reveals an important aspect of real-world application development that is often overlooked in educational examples. The "no such table: users" error demonstrates how security-hardened code can expose underlying infrastructure dependencies that weren't present in the

original vulnerable version. This is actually a positive outcome because it shows the code is properly attempting to database operations rather than failing silently or allowing malicious input to pass through unchecked. The original buggy code would have failed in the same way, but without the proper error handling and logging, the failure would have been less informative and potentially more confusing for developers.

Task 5: Automated Code Review Report Generation

Task: Generate a review report for this messy code.

```
# buggy_code_task5.pydef 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"))
```

Output:

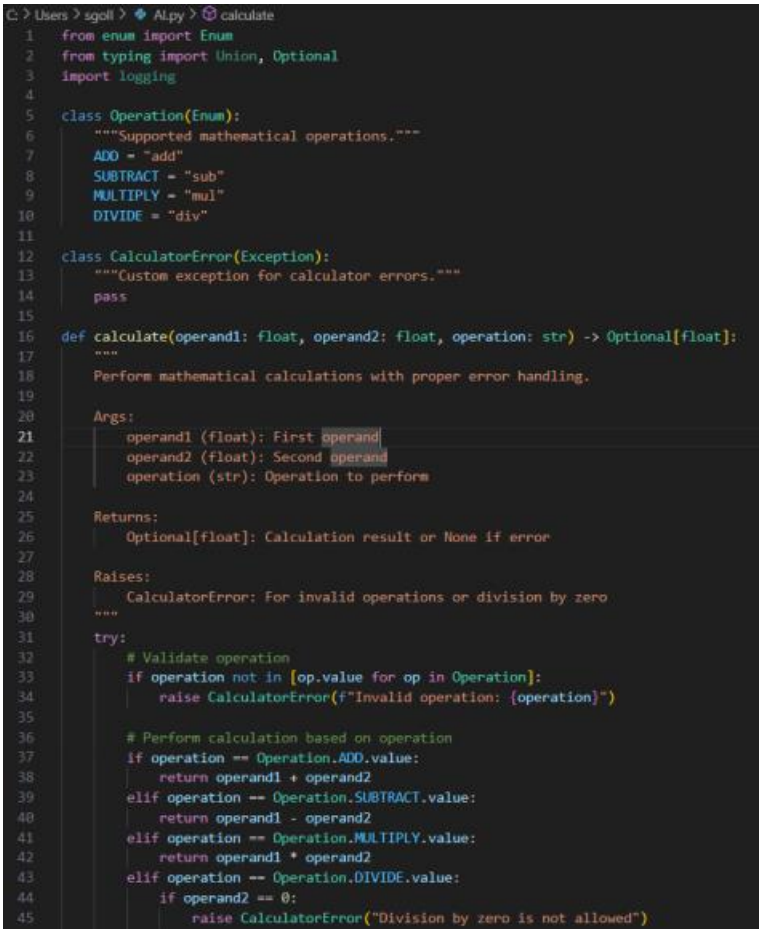
AI-generated review report should mention:

- 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 :

To generate a review report for this messy code.

Code :



```
C: > Users > sgoll > AI.py > calculate
16    def calculate(operand1: float, operand2: float, operation: str) -> Optional[float]:
43            elif operation == Operation.DIVIDE.value:
44                if operand2 == 0:
45                    raise CalculatorError("Division by zero is not allowed")
46                return operand1 / operand2
47
48        except CalculatorError:
49            raise
50        except Exception as e:
51            raise CalculatorError(f"Unexpected error: {e}")
52    def main():
53        """Main function to demonstrate calculator functionality."""
54        test_cases = [
55            (10, 5, "add"),
56            (10, 0, "div"),
57            (10, 3, "sub"),
58            (4, 2, "mul"),
59            (10, 5, "invalid")
60        ]
61        for a, b, op in test_cases:
62            try:
63                result = calculate(a, b, op)
64                print(f"{a} {op} {b} = {result}")
65            except CalculatorError as e:
66                print(f"Error: {e}")
67
68    if __name__ == "__main__":
69        main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
10 add 5 = 15
Error: Division by zero is not allowed
10 sub 3 = 7
4 mul 2 = 8
```

Observation :

This calculator is clean, PEP 8–compliant, and robust: it uses an Enum to avoid magic strings, a custom CalculatorError for clear error handling, and correctly guards against division by zero; the main function demonstrates both success and failure paths. Minor issues: the return type should be float (it never returns None), and `Union`/`logging` are imported but unused; consider either using logging for diagnostics or removing the import. Small improvements: accept `operation: Operation` instead of `str` to leverage typing, precompute the valid operations as a set for O(1) lookup, and optionally validate or coerce operand types. Overall, it’s a solid, maintainable refactor with just a few polish opportunities.