AI ASSISTED CODING

ASSIGNMENT-10.4

Akula Vivek

2403A51258

BATCH-11

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.

PROMPT:

TASK: Fix the syntax errors, indentation, and variable names in the code below.

The corrected function should return the sum of two numbers. Also correct the function call to print the result of adding 10 and 20.

CODE AND THE OUTPUT:

```
task1.py > ...
def add_numbers(a, b):
    return a + b

print(add_numbers(10, 20))

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\siris\New folder> & C:/Users/siris/AppData/Local/M
30
```

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.

PROMPT:

TASK: Optimize the find_duplicates function below.

Requirements:

- Return the same duplicates as before

- Use sets or other efficient logic to improve performance to O(n)

- Keep output as a list of duplicates.

Buggy code:

```python
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
# Example usage:
```

Corrected code:

```python
def find_duplicates(nums):
    """Return a list of duplicate elements in nums (optimized with sets)."""
    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))
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\siris\New folder> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/siris/New folder/task1.py"
[1, 2]
PS C:\Users\siris\New folder>
```

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

prompt:

TASK: Refactor the factorial function below for readability:

- Rename function to calculate_factorial
- Follow PEP 8 formatting and indentation
- Use descriptive variable names
- Add a Google-style docstring and type hints
- Include error handling for negative input.

Messy code:

task3.py > ...

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

Corrected code:

```python
def calculate_factorial(n: int) -> int:
    """Calculate the factorial of a non-negative integer.

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

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is negative.
    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(calculate_factorial(5))
```

```
PS C:\Users\siris\New folder> & C:/Users/siris/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/siris/New folder/task3.py"
120
```

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.

Prompt:

TASK: Enhance security and error handling in the function below:

- Use parameterized SQL instead of f-strings to prevent injection
- Add try/except/finally for database connection and errors
- Validate that user input is an integer before executing the query
- Use clear function and variable names, plus docstrings.

Messy 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.ex (variable) cursor: Cursor
    result = cursor.fetchall()
    conn.close()
    return result

user_input = input("Enter user ID: ")
print(get_user_data(user_input))
```

Problems with messy code:

- **SQL injection risk** — f-string directly inserting user_id.

- **No input validation** — user_id might not be an integer.
- **No exception handling** — DB errors could crash the program.

Updated code:

```python
task4.py > get_user_data_by_id
import sqlite3
def get_user_data_by_id(user_id: int):
    """Retrieve user data from the database by user ID.

    Args:
        user_id (int): The ID of the user to retrieve.

    Returns:
        list: List of tuples containing user data, or an empty list if not found or error occurs.
    """
    conn = None
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        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 []
    finally:
        if conn:
            conn.close()

def get_valid_user_id():
    """Prompt the user for a valid integer user ID."""
    user_input = input("Enter user ID: ")
    if not user_input.isdigit():
        print("Invalid input. Please enter a valid integer user ID.")
        return None
    return int(user_input)

user_id = get_valid_user_id()
if user_id is not None:
    print(get_user_data_by_id(user_id))
```

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:

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:

TASK: Perform an automated code review of the following function.

- Identify missing docstrings

- Identify inconsistent formatting and indentation

- Identify missing error handling (like division by zero)

- Identify non-descriptive names

- Suggest improvements for readability and PEP 8 compliance.

Messy code:

task5.py > ...

```python
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"))
```

Updated code:

```python
def calculate(a, b, operation):
    """Perform a basic arithmetic operation on two numbers.

    Args:
        a (float): The first operand.
        b (float): The second operand.
        operation (str): The operation to perform: 'add', 'sub', 'mul', or 'div'.

    Returns:
        float: The result of the operation.
        str: An error message if the operation is invalid or causes an error.
    """
    if operation == "add":
        return a + b
    elif operation == "sub":
        return a - b
    elif operation == "mul":
        return a * b
    elif operation == "div":
        if b == 0:
            return "Error: Division by zero."
        return a / b
    else:
        return "Error: Invalid operation."

print(calculate(10, 5, "add"))
print(calculate(10, 0, "div"))
```

Report:

1. **Missing Docstrings**
    - The function calc lacks a docstring explaining its purpose, parameters, and return value.
2. **Inconsistent Formatting and Indentation**
    - The function body is not properly indented.
    - The elif and else statements are not consistently formatted.
    - The print statements are not separated from the function definition.
3. **Missing Error Handling**

    - Division by zero is not handled, which will raise a ZeroDivisionError if y is zero and z is "div".

4. **Non-descriptive Names**
    - The function name calc and parameter names x, y, and z are not descriptive. More meaningful names like a, b, and operation would improve readability.

5. **Other Suggestions**
    - Use return instead of print for error messages to keep function output consistent.
    - Consider using a dictionary to map operations to functions for scalability.
    - Follow PEP 8: add spaces after commas, use lowercase with underscores for function names, and add two blank lines before function definitions.

Suggested Improvements

- Add a docstring.
- Use descriptive parameter names.
- Properly indent the function.
- Add error handling for division by zero.
- Use PEP 8 formatting.
- Return error messages instead of printing them inside the function.