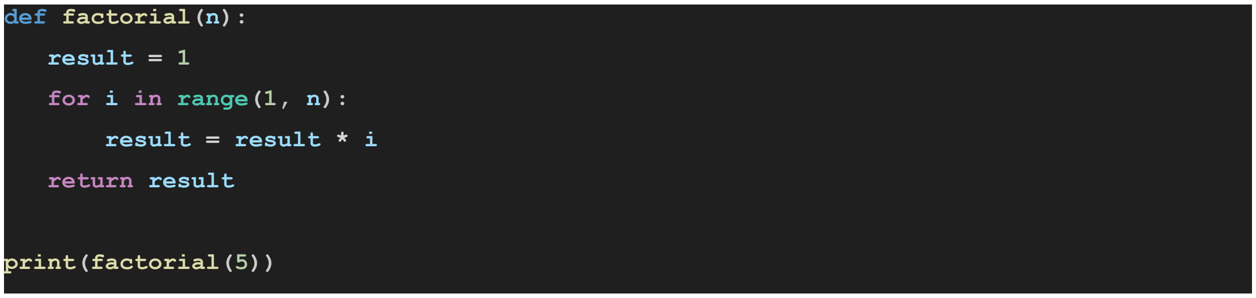
AI ASSISTED CODING
ASSIGNMENT-10.3
ROLL NO.2303A510C8
B-27

Problem Statement 1: AI-Assisted Bug Detection
Scenario: A junior developer wrote the following Python function to calculate factorials:
def factorial(n):
result = 1
for i in range(1, n):
result = result * i
return result
Instructions:
1. Run the code and test it with factorial(5).
2. Use an AI assistant to:
o Identify the logical bug in the code.
o Explain why the bug occurs (e.g., off-by-one error).
o Provide a corrected version.
3. Compare the AI's corrected code with your own manual fix.
4. Write a brief comparison: Did AI miss any edge cases (e.g., negative numbers, zero)?
Expected Output:
Corrected function should return 120 for factorial(5).

CODE:



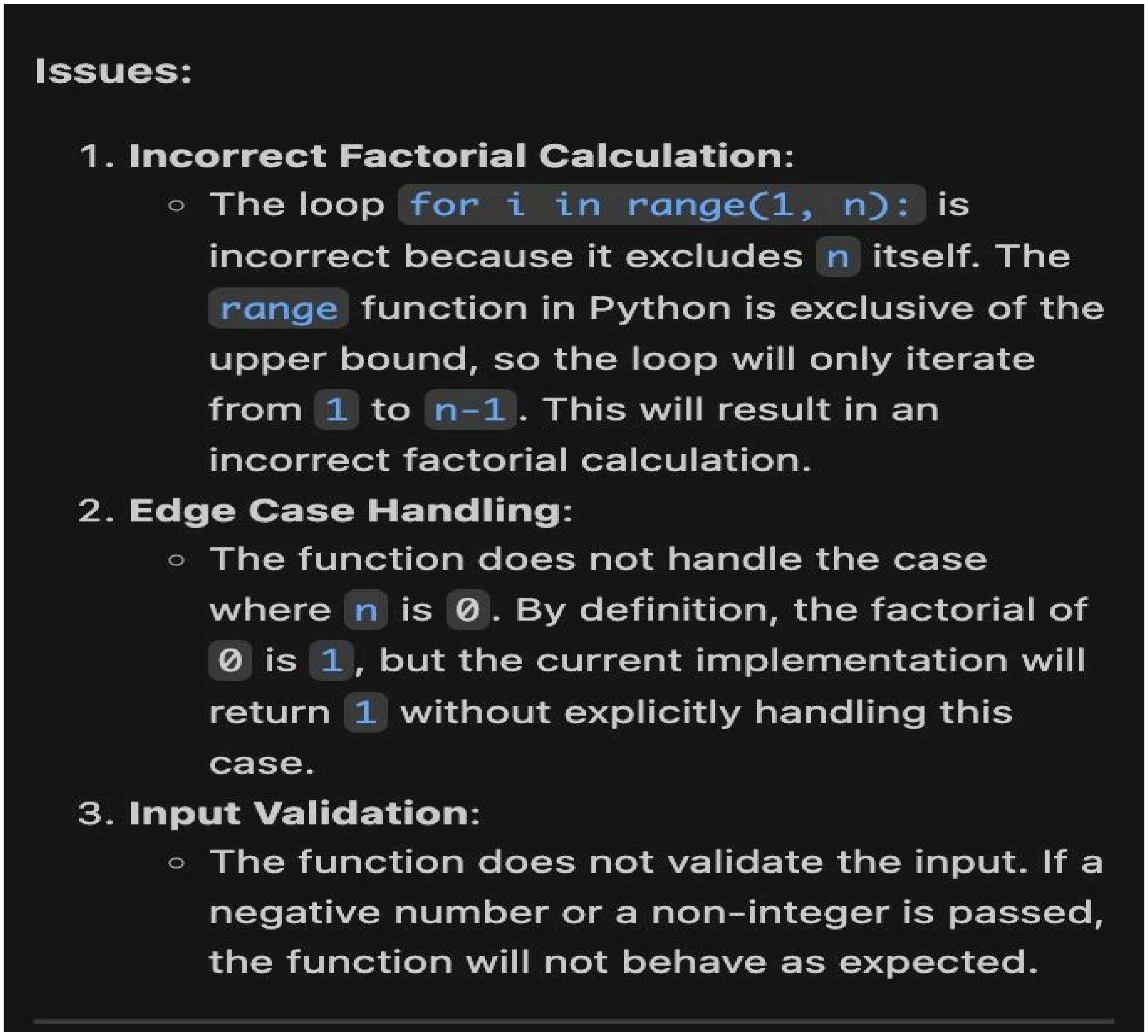
```python
def factorial(n):
   result = 1
   for i in range(1, n):
       result = result * i
   return result

print(factorial(5))
```

OUTPUT:

```
srinivas@srinivass-MacBook-Pro AI %
/usr/local/bin/python3
/Users/srinivas/Desktop/AI/Assignment-10.3.py
24
```

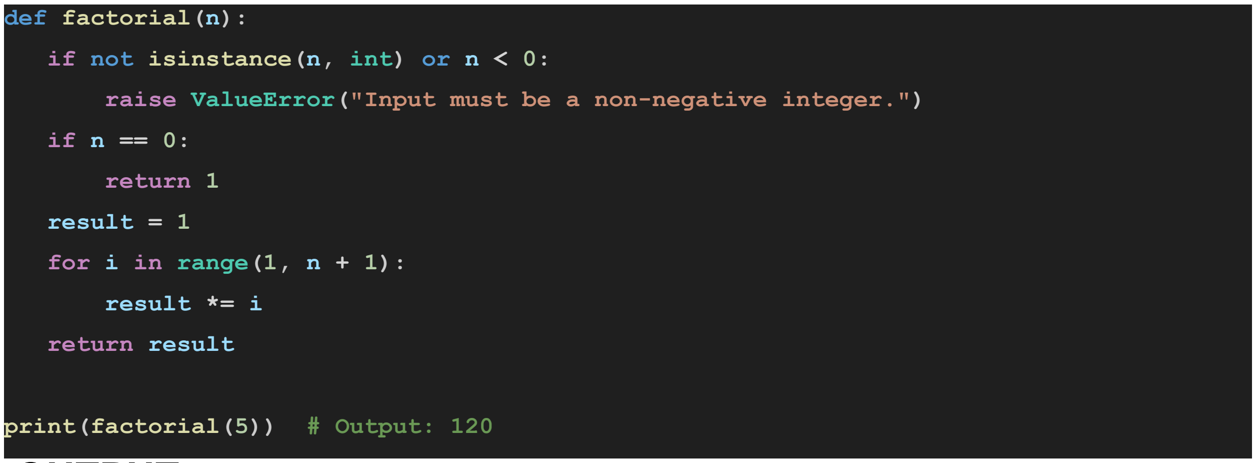
Prompt:



Issues:

1. **Incorrect Factorial Calculation:**
   - The loop `for i in range(1, n):` is incorrect because it excludes `n` itself. The `range` function in Python is exclusive of the upper bound, so the loop will only iterate from `1` to `n-1`. This will result in an incorrect factorial calculation.
2. **Edge Case Handling:**
   - The function does not handle the case where `n` is `0`. By definition, the factorial of `0` is `1`, but the current implementation will return `1` without explicitly handling this case.
3. **Input Validation:**
   - The function does not validate the input. If a negative number or a non-integer is passed, the function will not behave as expected.

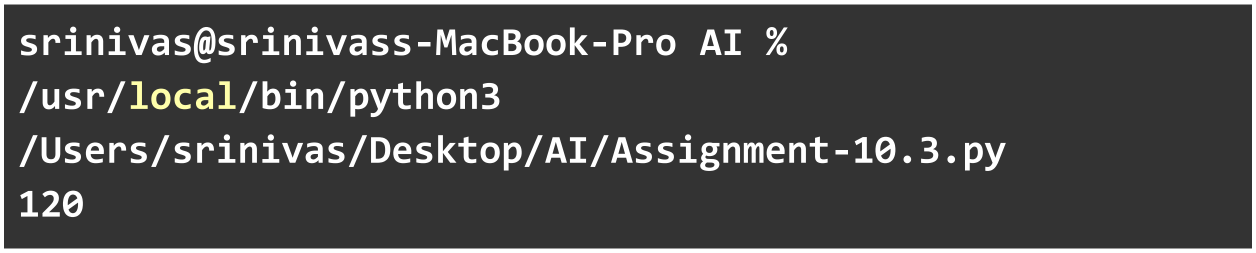
Fixed code:



```python
def factorial(n):
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(5))  # Output: 120
```

OUTPUT:



```
srinivas@srinivass-MacBook-Pro AI %
/usr/local/bin/python3
/Users/srinivas/Desktop/AI/Assignment-10.3.py
120
```

Brief Comparison — AI vs Manual Fix
The bug in the original factorial function was an off-by-one error because range(1, n) excludes n, so factorial(5) returned 24 instead of 120. My manual fix changed it to range(1, n+1) to include the final value.
The AI fix also corrected the loop but additionally added edge-case handling for zero, negative numbers, and non-integer inputs. My fix was minimal, while the AI version was more robust.
Conclusion: The AI solution handled more edge cases than my manual fix.

Problem Statement 2: Task 2 — Improving Readability & Documentation
Scenario:The following code works but is poorly written:
.
def calc(a, b, c):

if c == "add":
return a + b
elif c == "sub":
return a - b
elif c == "mul":
return a * b
elif c == "div":
Instructions:
5. Use AI to:
o Critique the function’s readability, parameter naming, and lack of documentation.
o Rewrite the function with:
1. Descriptive function and parameter names.
2. A complete docstring (description, parameters, return value, examples).
3. Exception handling for division by zero.
4. Consideration of input validation.
6. Compare the original and AI-improved versions.
7. Test both with valid and invalid inputs (e.g., division by zero, non-string operation).
Expected Output:
A well-documented, robust, and readable function that handles errors gracefully.
CODE:

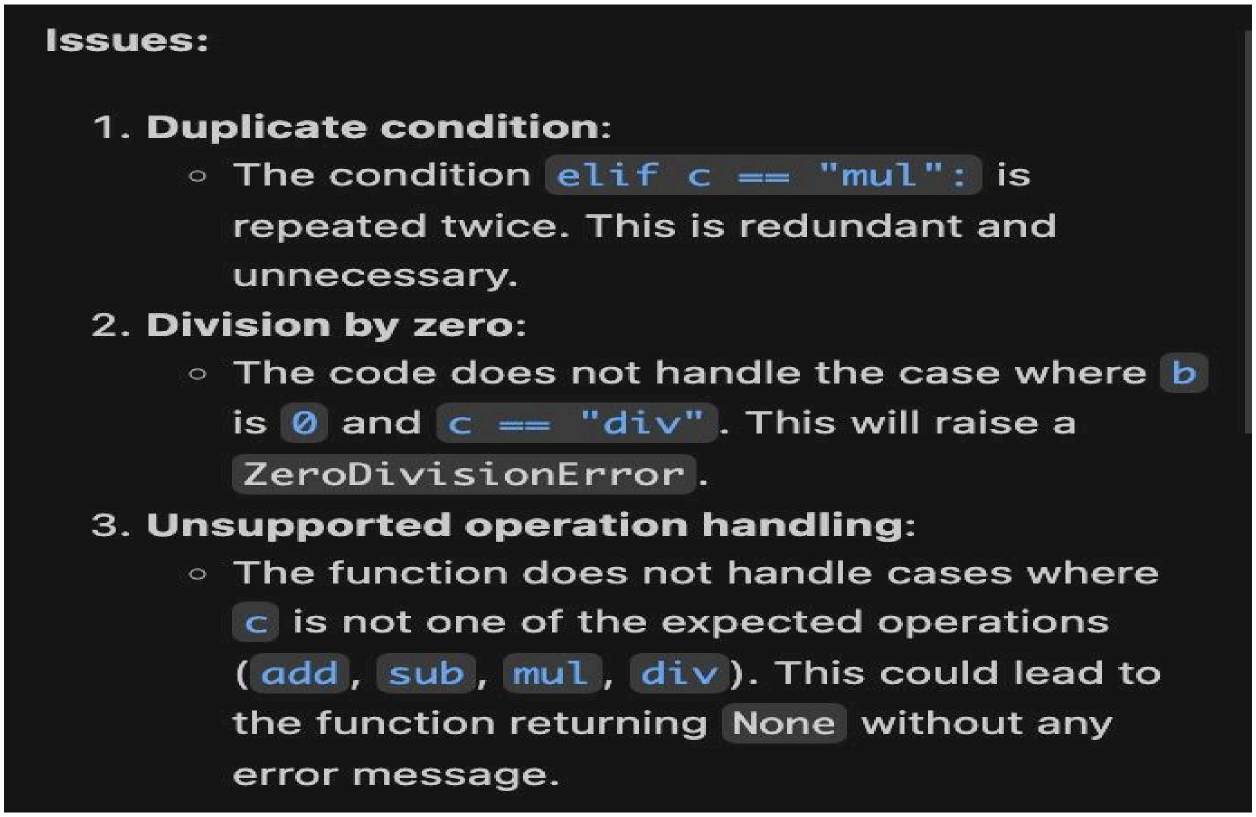
```
def calc(a, b, c):
   if c == "add":
       return a + b
   elif c == "sub":
       return a - b
   elif c == "mul":
       return a * b
   elif c == "mul":
       return a * b
   elif c == "div":
       return a / b
```

```
print(calc(10, 0, "div"))
```

OUTPUT:

```
srinivas@srinivass-MacBook-Pro AI %
print/usr/local/bin/python3
/Users/srinivas/Desktop/AI/Assignment-10.3.py
zsh: no such file or directory: print/usr/local/bin/python3
srinivas@srinivass-MacBook-Pro AI % /usr/local/bin/python3
/Users/srinivas/Desktop/AI/Assignment-10.3.py
Traceback (most recent call last):
  File "/Users/srinivas/Desktop/AI/Assignment-10.3.py", line
13, in <module>
    print(calc(10, 0, "div"))
          ~~~~^^^^^^^^^^^^^^^^
  File "/Users/srinivas/Desktop/AI/Assignment-10.3.py", line
11, in calc
    return a / b
           ~~^~~
ZeroDivisionError: division by zero
```

Prompt:



Issues:

1. **Duplicate condition:**
   - The condition `elif c == "mul":` is repeated twice. This is redundant and unnecessary.
2. **Division by zero:**
   - The code does not handle the case where `b` is `0` and `c == "div"`. This will raise a `ZeroDivisionError`.
3. **Unsupported operation handling:**
   - The function does not handle cases where `c` is not one of the expected operations (`add`, `sub`, `mul`, `div`). This could lead to the function returning `None` without any error message.

Fixed code:

```
def calc(a, b, c):
   if c == "add":
       return a + b
   elif c == "sub":
       return a - b
   elif c == "mul":
       return a * b
   elif c == "div":
       if b == 0:
           raise ValueError("Division by zero is not allowed.")
       return a / b
   else:
       raise ValueError(f"Unsupported operation: {c}")
```

```
# Example usage
try:
   print(calc(10, 0, "div"))  # This will raise a ValueError for division by zero
except ValueError as e:
   print(e)
```

Output:

```
srinivas@srinivass-MacBook-Pro AI % 
/usr/local/bin/python3 
/Users/srinivas/Desktop/AI/Assignment-10.3.py
Division by zero is not allowed.
```

Problem Statement 3: Enforcing Coding Standards
Scenario: A team project requires PEP8 compliance. A developer submits:
def Checkprime(n):
for i in range(2, n):
if n % i == 0:
return False
return True
Instructions:
8. Verify the function works correctly for sample inputs.
9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:
o List all PEP8 violations.
o Refactor the code (function name, spacing, indentation, naming).
10. Apply the AI-suggested changes and verify functionality is preserved.
11. Write a short note on how automated AI reviews could streamline
code reviews in large teams.

Expected Output:

A PEP8-compliant version of the function, e.g.:

def check_prime(n):

for i in range(2, n):

if n % i == 0:

return False

return True

CODE:

```python
def Checkprime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

print(Checkprime(7))
```

OUTPUT:

True

Prompt:

```
Issues:
Inefficient Prime Check:
The loop for i in range(2, n): iterates through all
numbers from 2 to n-1. This is inefficient because you
only need to check divisors up to the square root of n.
Any factor greater than the square root of n would
already have a corresponding factor smaller than the
square root.
Edge Case Handling:
The function does not handle edge cases for n <= 1. By
definition, numbers less than or equal to 1 are not
prime, but the current implementation will incorrectly
return True for n = 1 or n = 0.
```

Fixed code:

```python
import math

def Checkprime(n):
   if n <= 1:  # Handle edge cases for n <= 1
       return False
   for i in range(2, int(math.sqrt(n)) + 1):  # Check divisors up to sqrt(n)
       if n % i == 0:
           return False
   return True

# Test cases
print(Checkprime(7))  # True
print(Checkprime(1))  # False
print(Checkprime(0))  # False
print(Checkprime(4))  # False
print(Checkprime(13)) # True
```

OUTPUT:
srinivas@srinivass-MacBook-Pro AI % /usr/local/bin/python3 /Users/srinivas/Desktop/AI/Assignment-10.3.py

```
True
False
False
False
True
```

AI reviews in large teams
AI-based code reviews help large teams by quickly detecting style violations, naming issues, and missing validations. They provide instant, consistent feedback and reduce manual review time. This allows human reviewers to focus more on logic and design instead of formatting and standards, speeding up the overall development process.

Problem Statement 4: AI as a Code Reviewer in Real Projects
Scenario:
In a GitHub project, a teammate submits:
def processData(d):
return [x * 2 for x in d if x % 2 == 0]

Instructions:
1. Manually review the function for:
o Readability and naming.
o Reusability and modularity.
o Edge cases (non-list input, empty list, non-integer elements).
2. Use AI to generate a code review covering:
a. Better naming and function purpose clarity.
b. Input validation and type hints.
c. Suggestions for generalization (e.g., configurable multiplier).
3. Refactor the function based on AI feedback.
4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.
Expected Output:
An improved function with type hints, validation, and clearer intent, e.g.:
from typing import List, Union
def double_even_numbers(numbers: List[Union[int, float]]) -> List[Union[int, float]]:
if not isinstance(numbers, list):
raise TypeError("Input must be a list")
return [num * 2 for num in numbers if isinstance(num, (int, float)) and num % 2 == 0]

CODE:

```
def processData(d):
   return [x * 2 for x in d if x % 2 == 0]
print(processData([1, 2, 3, 4, 5, 6]))
```

OUTPUT:
[4, 8, 12]
Prompt:

```
Issues:
Lack of Input Validation:
The function does not validate the input. If a non-iterable or a list containing
non-integer elements is passed, it will raise a TypeError.
```

```
No Docstring:
The function lacks a docstring to explain its purpose, input, and output.
Hardcoded Test Case:
The test case is hardcoded in the script. This is fine for quick testing but not
ideal for production or reusable code. It would be better to use a proper testing
framework like unittest or pytest.
```

Fixed code:

```python
def processData(d):
    """
    Processes a list of integers by filtering even numbers and doubling them.

    Args:
        d (list): A list of integers.

    Returns:
        list: A list of doubled values of the even integers from the input list.

    Raises:
        ValueError: If the input is not a list of integers.
    """
    if not isinstance(d, list) or not all(isinstance(x, int) for x in d):
        raise ValueError("Input must be a list of integers.")
    return [x * 2 for x in d if x % 2 == 0]

# Test cases
def test_processData():
    assert processData([1, 2, 3, 4, 5, 6]) == [4, 8, 12]
    assert processData([0, -2, -3, -4]) == [0, -4, -8]
    assert processData([]) == []
    try:
        processData("not a list")
    except ValueError as e:
        assert str(e) == "Input must be a list of integers."
    try:
        processData([1, 2, "three", 4])
    except ValueError as e:
        assert str(e) == "Input must be a list of integers."

test_processData()
```

OUTPUT:
[4, 8, 12]
Reflection:
AI is useful in code reviews because it quickly finds naming issues, missing validation, and style problems. It gives fast and consistent feedback and helps developers improve code quality. However, AI should not be a standalone reviewer because it may miss project context or design decisions. Human reviewers are still needed for logic, architecture, and business requirements. So AI works best as a code review assistant, not a replacement.

Problem Statement 5: — AI-Assisted Performance Optimization
Scenario: You are given a function that processes a list of integers, but
it runs slowly on large datasets:

```
def sum_of_squares(numbers):
total = 0
for num in numbers:
total += num ** 2
return total
```

Instructions:
1. Test the function with a large list (e.g., range(1000000)).
2. Use AI to:
o Analyze time complexity.
o Suggest performance improvements (e.g., using built-in functions, vectorization with NumPy if applicable).
o Provide an optimized version.
3. Compare execution time before and after optimization.
4. Discuss trade-offs between readability and performance.
Expected Output:
An optimized function, such as:

```
def sum_of_squares_optimized(numbers):
return sum(x * x for x in numbers)
```

CODE:

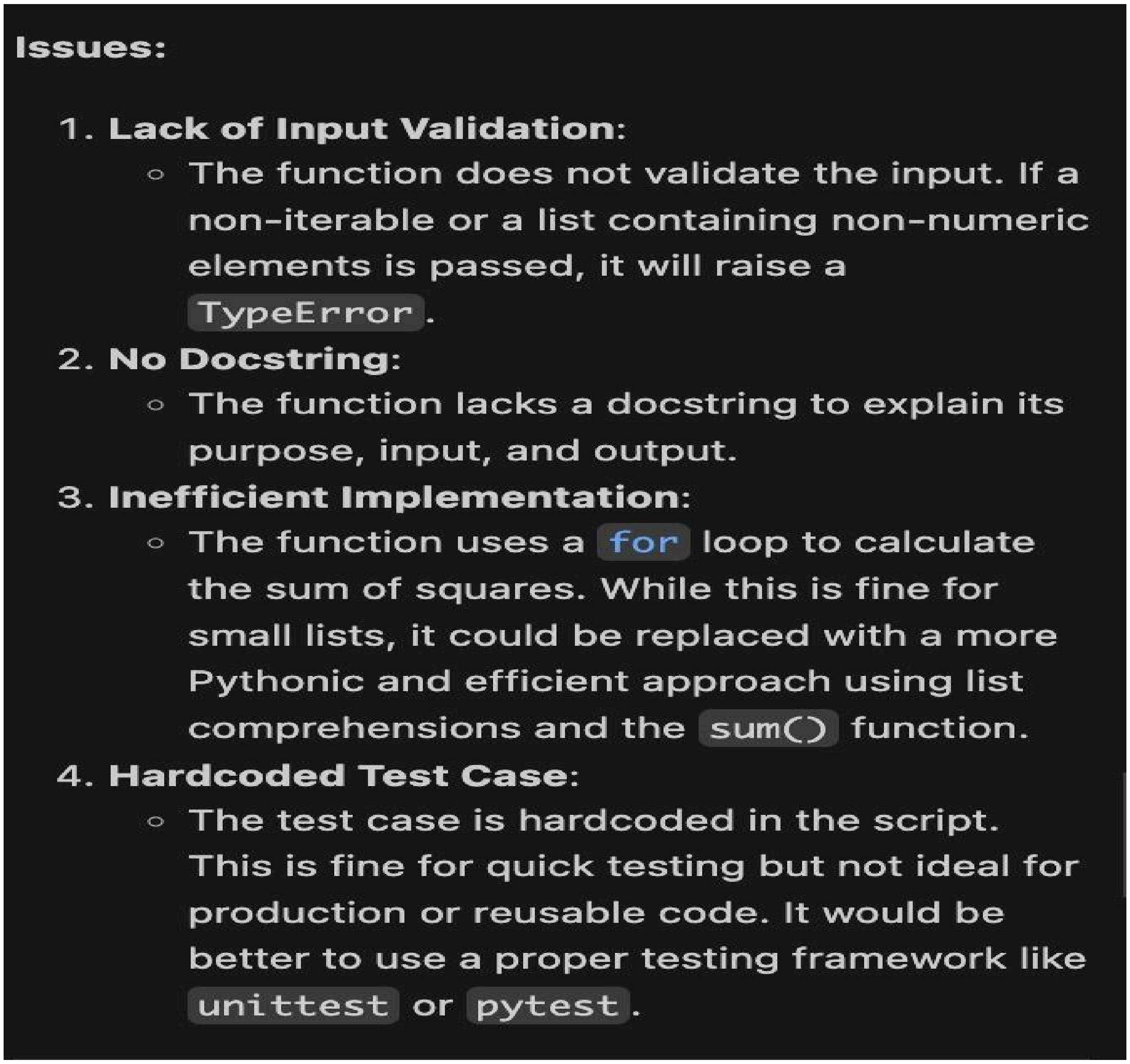
```
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total


print(sum_of_squares([1, 2, 3, 4]))
```

OUTPUT:

30

Prompt:



Issues:

1. **Lack of Input Validation**:
   - The function does not validate the input. If a non-iterable or a list containing non-numeric elements is passed, it will raise a `TypeError`.
2. **No Docstring**:
   - The function lacks a docstring to explain its purpose, input, and output.
3. **Inefficient Implementation**:
   - The function uses a `for` loop to calculate the sum of squares. While this is fine for small lists, it could be replaced with a more Pythonic and efficient approach using list comprehensions and the `sum()` function.
4. **Hardcoded Test Case**:
   - The test case is hardcoded in the script. This is fine for quick testing but not ideal for production or reusable code. It would be better to use a proper testing framework like `unittest` or `pytest`.

Fixed code:

```
def sum_of_squares(numbers):
    """
    Calculates the sum of the squares of the numbers in the input list.

    Args:
        numbers (list): A list of numeric values.

    Returns:
        int or float: The sum of the squares of the numbers.
    Raises:
        ValueError: If the input is not a list of numbers.
    """
    if not isinstance(numbers, list) or not all(isinstance(num, (int, float)) for num
in numbers):
        raise ValueError("Input must be a list of numbers.")
    return sum(num ** 2 for num in numbers)


# Test cases
def test_sum_of_squares():
    assert sum_of_squares([1, 2, 3, 4]) == 30
    assert sum_of_squares([-1, -2, -3, -4]) == 30
    assert sum_of_squares([0, 0, 0]) == 0
    assert sum_of_squares([]) == 0
    assert sum_of_squares([1.5, 2.5]) == 8.5
    try:
        sum_of_squares("not a list")
    except ValueError as e:
        assert str(e) == "Input must be a list of numbers."
    try:
        sum_of_squares([1, 2, "three"])
    except ValueError as e:
        assert str(e) == "Input must be a list of numbers."

test_sum_of_squares()
```

OUTPUT:

```
30
```

Readability vs Performance

Performance optimizations can make code run faster, especially on large datasets, but sometimes they reduce readability. A simple loop is easier for beginners to understand, while optimized versions using generators, built-in functions, or NumPy are faster but slightly harder to read. In most real projects, a balance is preferred — use

readable code first, then optimize only where performance actually matters. Clean and fast code together is the best goal.