

ASSIGNMENT-10.3

M.PARAMESH

HNO : 2303A53021

BATCH – 46

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:
 - ~Identify the logical bug in the code.
 - ~Explain why the bug occurs (e.g., off-by-one error).
 - ~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).

PROMPT USED :

Identify the logical error in this Python factorial function and provide a corrected version. Explain why the error occurs.

CODE :

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    elif n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

output:

The screenshot shows a code editor interface with a terminal window below it. The code editor has a tab bar with 'Run', 'Terminal', and 'Help'. The main area contains Python code for a factorial function, with some explanatory comments and a note about handling negative numbers. The terminal window below shows the command 'python f.py' being run, followed by the output '120', which is the factorial of 5.

```
# generate Analyze the following Python f.py ...
#> C:\Users\param> OneDrive > Documents > #generate Analyze the following Python factorial function. Identify any logical errors in the code, explain
1 #> #generate Analyze the following Python factorial function. Identify any logical errors in the code, explain
2 def factorial(n):
3     if n == 0:
4         return 1
5     else:
6         return n * factorial(n - 1)
7 # This provided factorial function has a logical error in that it does not handle negative numbers. The facto
8 # Here is the corrected version of the function that handles negative numbers by raising a ValueError:
9 def factorial(n):
10     if n < 0:
11         raise ValueError("Factorial is not defined for negative numbers")
12     elif n == 0:
13         return 1
14     else:
15         return n * factorial(n - 1)
16 print(factorial(5))
17 |
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\param\Downloads\AI Coding & C:/Users/param/AppData/Local/Python/pythoncore-3.14-64/python.exe "C:/Users/param/OneDrive/Documents/#generate Analyze the following Python f.py"
120
PS C:\Users\param\Downloads\AI Coding>
```

Explanation :

1) Bug Identified by AI

- The loop uses range(1, n)
- range(1, n) **excludes n**
- Factorial requires multiplying all numbers from 1 to n (inclusive)

This is an **off-by-one error**.

2) Why the Bug Occurs

For factorial(5):

- Loop runs as: $1 \times 2 \times 3 \times 4$

- Missing multiplication by 5
- Result becomes 24 instead of 120
- 3) AI **did not initially handle negative input**, which was later improved
- Corrected function now returns **120 for factorial(5)** .

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 byzero, non-string operation).

Expected Output: A well-documented, robust, and readable function that handles errors gracefully.

Prompt used:

Critique the readability and documentation of the following Python function. Rewrite it using descriptive names, a proper docstring, input validation, and exception handling for division by zero. Provide examples and explain improvements.

Code :

```
def div(a, b):
    return a / b

#!/usr/bin/env python3

def divide_numbers(numerator, denominator):
```

```
    """
```

Divides the numerator by the denominator.

Parameters:

numerator (float): The number to be divided.

denominator (float): The number by which to divide.

Returns:

float: The result of the division.

Raises:

ValueError: If the denominator is zero.

TypeError: If the inputs are not numbers.

Examples:

```
>>> divide_numbers(10, 2)
```

```
5.0
```

```
>>> divide_numbers(7, 0)
```

Traceback (most recent call last):

```
...
```

ValueError: Denominator cannot be zero.

```
>>> divide_numbers('10', 2)
```

Traceback (most recent call last):

...

TypeError: Both numerator and denominator must be numbers.

Input validation

```
if not isinstance(numerator, (int, float)) or not isinstance(denominator, (int, float)):  
    raise TypeError("Both numerator and denominator must be numbers.")
```

Check for division by zero

```
if denominator == 0:
```

```
    raise ValueError("Denominator cannot be zero.")
```

```
return numerator / denominator
```

```
print(divide_numbers(10, 5))
```

output :

The screenshot shows a Python code editor interface with a dark theme. The code in the editor is as follows:

```
➊ #!/usr/bin/env python3  
➋ def divide(a, b):  
⌁     """  
⌂     Divides the numerator by the denominator.  
⌃     Parameters:  
⌄     numerator (float): The number to be divided.  
⌅     denominator (float): The number by which to divide.  
⌆     Returns:  
⌇     float: The result of the division.  
⌈     Raises:  
⌉     ValueError: If the denominator is zero.  
⌊     TypeError: If the inputs are not numbers.  
⌋ Examples:  
⌌ >>> divide(10, 2)  
⌍ 5.0  
⌎ >>> divide(7, 0)  
⌏ Traceback (most recent call last):  
⌐     ...  
⌑     ValueError: Denominator cannot be zero.  
⌒ >>> divide('10', 2)  
⌓ Traceback (most recent call last):  
⌔     ...  
⌕     TypeError: Both numerator and denominator must be numbers.  
⌖ # Input validation  
⌗ if not isinstance(numerator, (int, float)) or not isinstance(denominator, (int, float)):  
⌘     raise TypeError("Both numerator and denominator must be numbers.")  
⌙ # Check for division by zero  
⌚ if denominator == 0:  
⌛     raise ValueError("Denominator cannot be zero.")  
⌜ return numerator / denominator  
⌝ print(divide(10, 5))
```

Below the code editor, there is a terminal window showing the command PS C:\Users\paran> and the output of the code execution. The terminal also shows the path C:\Users\paran>param>OneDrive>Documents>Untitled-1.py and the command & C:/Users/paran/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/paran/OneDrive/Documents/Untitled-1.py. The terminal window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. At the bottom of the screen, there are status bars for RUN DEV, DEBUG STAGING, DEPLOY PRODUCTION, and other system information like line count and character count.

Explanation :

1. Descriptive names improve code understanding.
2. Docstrings help future developers and users.
3. Input validation prevents unexpected crashes.
4. Exception handling makes the function robust.
5. AI significantly improves maintainability and safety.
 - Handles invalid inputs gracefully
 - Meets all expected output requirements

Problem Statement 3: Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer 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
```

Problem Statement 4: AI as a Code Reviewer in Real Projects submits:

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

Instructions:

Verify the function works correctly for sample inputs.

- Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:

- List all PEP8 violations.
 - Refactor the code (function name, spacing, indentation, naming).
 - Apply the AI-suggested changes and verify functionality is preserved.
- 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
```

prompt used : #genaerate Analyze the following Python function for PEP8 violations. List all style issues, refactor the code to be PEP8-compliant (naming, spacing, indentation), and ensure the functionality remains the same

code :

```
ef calculate_area(radius):  
    pi=3.14159  
    area = pi * radius **2  
    return area
```

```
print(calculate_area(5))  
def Checkprime(num):  
    if num <= 1:  
        return False  
    for i in range(2, int(num**0.5) + 1):  
        if num % i == 0:
```

```

    return False

return True

print(Checkprime(7))

print(Checkprime(10))

```

output :

The screenshot shows a dark-themed interface for AI Coding. At the top, there's a menu bar with View, Go, Run, Terminal, Help, and a search bar labeled "AI Coding". Below the menu is a code editor window titled "Untitled-1.py" containing the provided Python code. To the right of the code editor is a panel with tabs for "CHAT" and "RECENT". The main workspace below the code editor shows the output of running the script. The output window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The OUTPUT tab shows the results of the script execution:

```

C:\> Users > param > OneDrive > Documents > Untitled-1.py ...
1 #generate Analyze the following Python function for PEP8 violations. List all style issues, refactor the code to be PEP8-compliant (naming, spacing, etc)
2 def calculate_area(radius):
3     pi=3.14159
4     area = pi * radius **2
5     return area
6
7
8 print(calculate_area(5))
9 def Checkprime(num):
10     if num <= 1:
11         return False
12     for i in range(2, int(num**0.5) + 1):
13         if num % i == 0:
14             return False
15     return True
16 print(Checkprime(7))
17 print(Checkprime(10))
18

PS C:\> C:\Users\param\Downloads\AI Coding & C:/Users/param/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/param/OneDrive/Documents/Untitled-1.py
78.53975
True
False
PS C:\>

```

At the bottom of the interface, there are buttons for RUN DEV, DEPLOY STAGING, and DEPLOY PRODUCTION, along with other status indicators like "In 18, Col 1", "Spaces 4", "UTF-8", and "CR LF".

Explanation :

1. PEP8 Naming Rule

- Function names should use lowercase with underscores.
- Checkprime → check_prime

2. Proper Indentation

- Python uses indentation to define blocks.
- Correct indentation improves readability and avoids errors.

3. Consistent Spacing

- Spaces after keywords and operators make code easier to read.

4. AI as a Code Reviewer

- AI quickly detects style and formatting issues.

- Ensures uniform coding standards across teams.

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:

- Readability and naming.
- Reusability and modularity.
- Edge cases (non-list input, empty list, non-integer elements).

2. Use AI to generate a code review covering:

- Better naming and function purpose clarity.
- Input validation and type hints.
- 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]
```

Prompt used : Review the following Python function as if you were a GitHub code reviewer. Suggest better naming, add type hints, input validation, and propose improvements to make the function more reusable and robust

Code : def sum_of_squares(numbers):

```
total = 0

for num in numbers:
    total += num ** 2

return total

# Example usage

nums = [1, 2, 3, 4]

print(f"Sum of squares: {sum_of_squares(nums)}") # Output: Sum of squares: 30
```

output :

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows two files: "Untitled-1.py" and "A-5 (1).py".
- Code Editor:** Displays the Python code for "A-5 (1).py". The code defines a function `sum_of_squares` that calculates the sum of squares of a list of numbers. It includes a comment for example usage and prints the result.
- Terminal:** Shows the command line output:

```
PS C:\Users\param\Downloads\AI Coding & C:/Users/param/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/param/Downloads/AI Coding/A-5 (1).py"
Sum of squares: 30
PS C:\Users\param\Downloads\AI Coding
```
- Python Interpreter:** A sidebar on the right shows two Python environments: "Python" and "Python".

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)
```

Prompt used :

Analyze the performance of the following Python function that calculates the sum of squares of a list of integers. Determine its time complexity, suggest optimizations to improve performance (such as using built-in functions or generator expressions), and provide an optimized version. Also compare execution time and discuss trade-offs between readability and performance

code :

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

```

return total

# Example usage

nums = [1, 2, 3, 4]

print(f"Sum of squares: {sum_of_squares(nums)}") # Output: Sum of squares

def sum_of_squares_optimized(numbers):

    return sum(x * x for x in numbers)

# Example usage

nums = [1, 2, 3, 4]

print(f"Optimized Sum of squares: {sum_of_squares_optimized(nums)}") # Output:
Optimized Sum of squares: 30: 30

```

output:

```

View Go Run Terminal Help ← → Q AI Coding
A-1.py A-5 (1).py ...
A-5 (1).py > ...
124
125
126
127 def sum_of_squares(numbers):
128     total = 0
129     for num in numbers:
130         total += num ** 2
131     return total
132 # Example usage
133 nums = [1, 2, 3, 4]
134 print(f"Sum of squares: {sum_of_squares(nums)}") # Output: Sum of squares
135 def sum_of_squares_optimized(numbers):
136     return sum(x * x for x in numbers)
137 # Example usage
138 nums = [1, 2, 3, 4]
139 print(f"Optimized Sum of squares: {sum_of_squares_optimized(nums)}") # Output: Optimized Sum of squares: 30: 30
140
141
142
143
144
145
146
147
148
149

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\param\Downloads\AI Coding> & C:/Users/param/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/param/Downloads/AI Coding/A-5 (1).py"

Sum of squares: 30
Optimized Sum of squares: 30
PS C:\Users\param\Downloads\AI Coding>

Explanation:

- 1) The original function uses a loop, which becomes slow for very large lists.
- 2) Its time complexity is $O(n)$ because each number is processed once.
- 3) AI suggested using Python's built-in `sum()` with a generator expression.
- 4) The optimized version is faster because built-in functions are more efficient.

5) Performance improves slightly, while readability remains good for large datasets.