NAME : P.CHARAN KUMAR

BATCH:41

ROLL NO :-2303A52138

Task 1: Fixing Syntax Errors

SCENARIOS:

You are reviewing a Python program where a basic function definition contains a syntax error.

Requirements

• Provide a Python function add(a, b) with a missing colon

• Use an AI tool to detect the syntax error

• Allow AI to correct the function definition

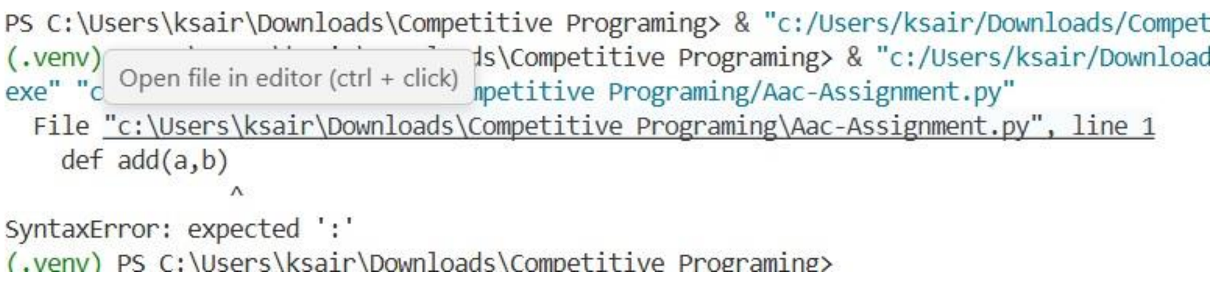
• Observe how AI explains the syntax issue

Prompt:

This script keeps the faulty code as a string (missing colon), shows the

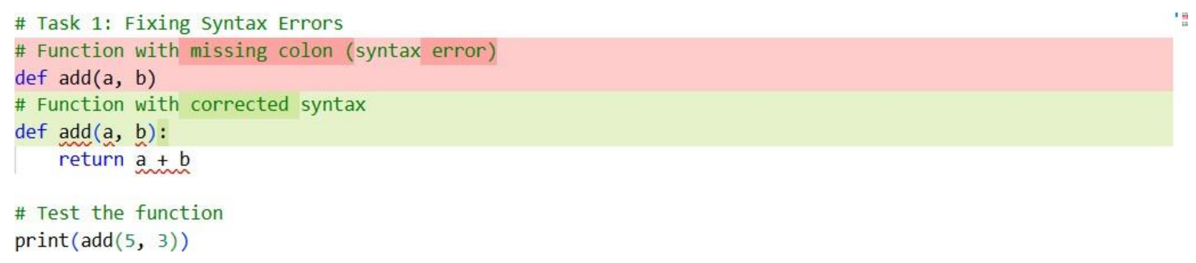
detected Syntax Error, then runs the corrected function.

Code:



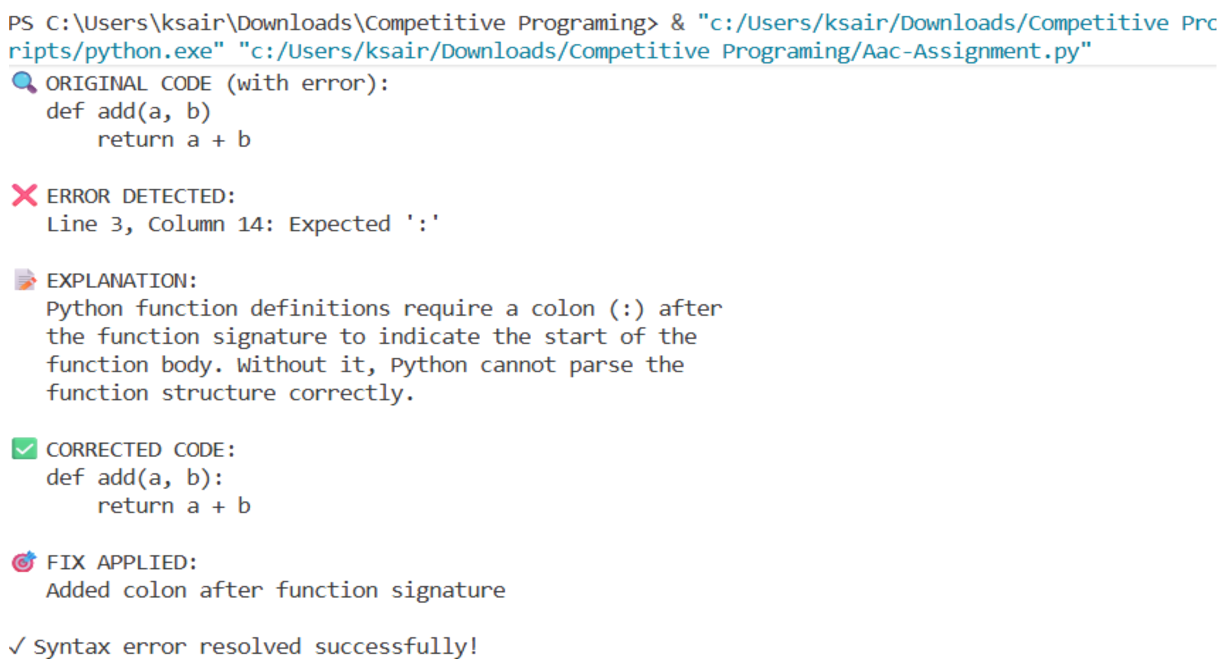
```
PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Compet
(.venv) Open file in editor (ctrl + click) ds\Competitive Programing> & "c:/Users/ksair/Download
exe" "c                                   mpetitive Programing/Aac-Assignment.py"
  File "c:\Users\ksair\Downloads\Competitive Programing\Aac-Assignment.py", line 1
    def add(a,b)
               ^
SyntaxError: expected ':'
(.venv) PS C:\Users\ksair\Downloads\Competitive Programing>
```

Corrected code:



```
# Task 1: Fixing Syntax Errors
# Function with missing colon (syntax error)
def add(a, b)
# Function with corrected syntax
def add(a, b):
    return a + b

# Test the function
print(add(5, 3))
```

Correct output:



```
PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Competitive Prc
ripts/python.exe" "c:/Users/ksair/Downloads/Competitive Programing/Aac-Assignment.py"
🔍 ORIGINAL CODE (with error):
   def add(a, b)
       return a + b

❌ ERROR DETECTED:
   Line 3, Column 14: Expected ':'

📝 EXPLANATION:
   Python function definitions require a colon (:) after
   the function signature to indicate the start of the
   function body. Without it, Python cannot parse the
   function structure correctly.

✅ CORRECTED CODE:
   def add(a, b):
       return a + b

🎯 FIX APPLIED:
   Added colon after function signature

✓ Syntax error resolved successfully!
```

Justification:

In Python, function definitions follow the pattern def function_name(parameters): where the colon is mandatory. It separates the function signature from the body and tells Python's parser that an indented code block follows. Without it, Python cannot parse the function structure correctly, resulting in a syntax error**.**

Task-2: Debugging Logic Errors in Loops

Scenario:

You are debugging a loop that runs infinitely due to a logical mistake.

Requirements

• Provide a loop with an increment or decrement error

• Use AI to identify the cause of infinite iteration

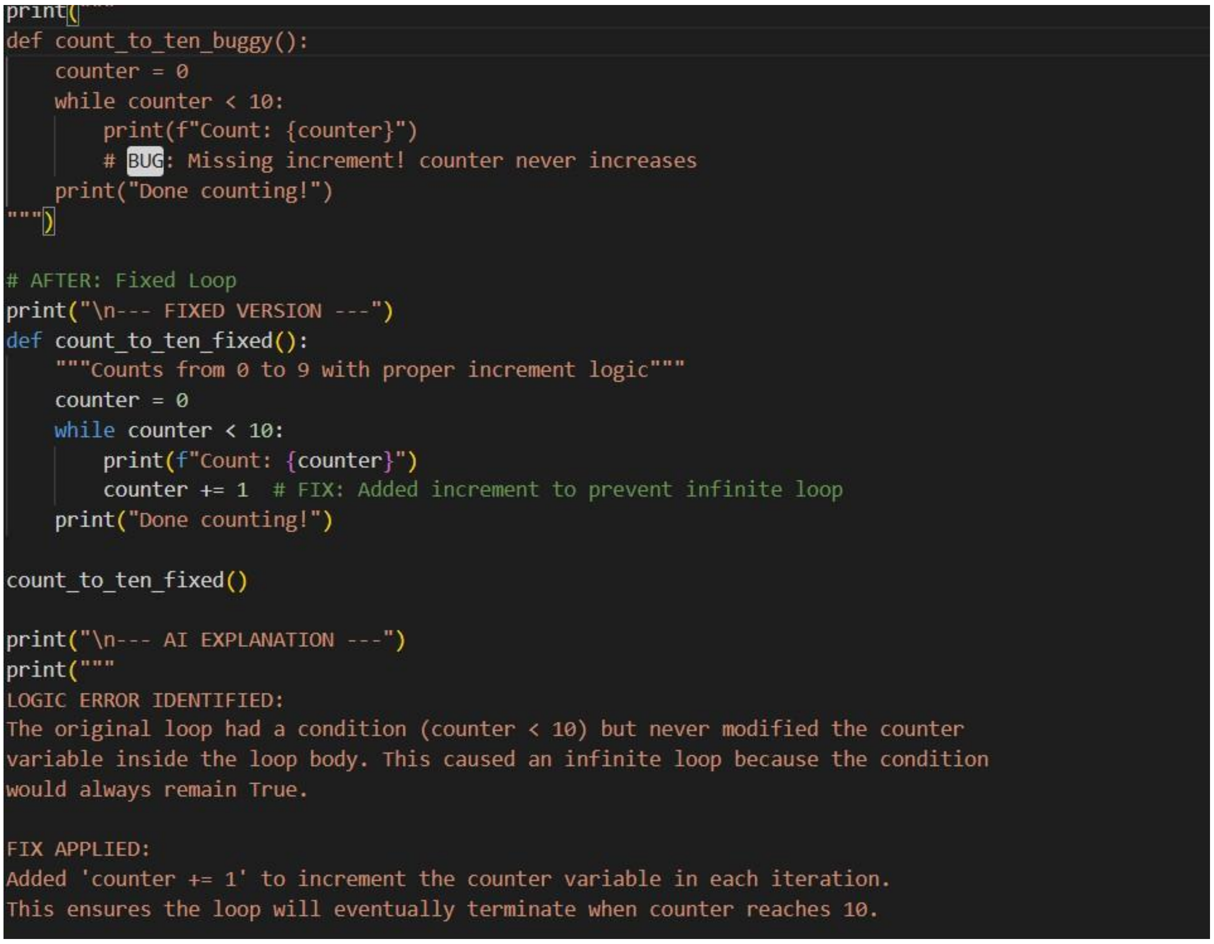
• Let AI fix the loop logic

• Analyze the corrected loop behavior

Prompt:

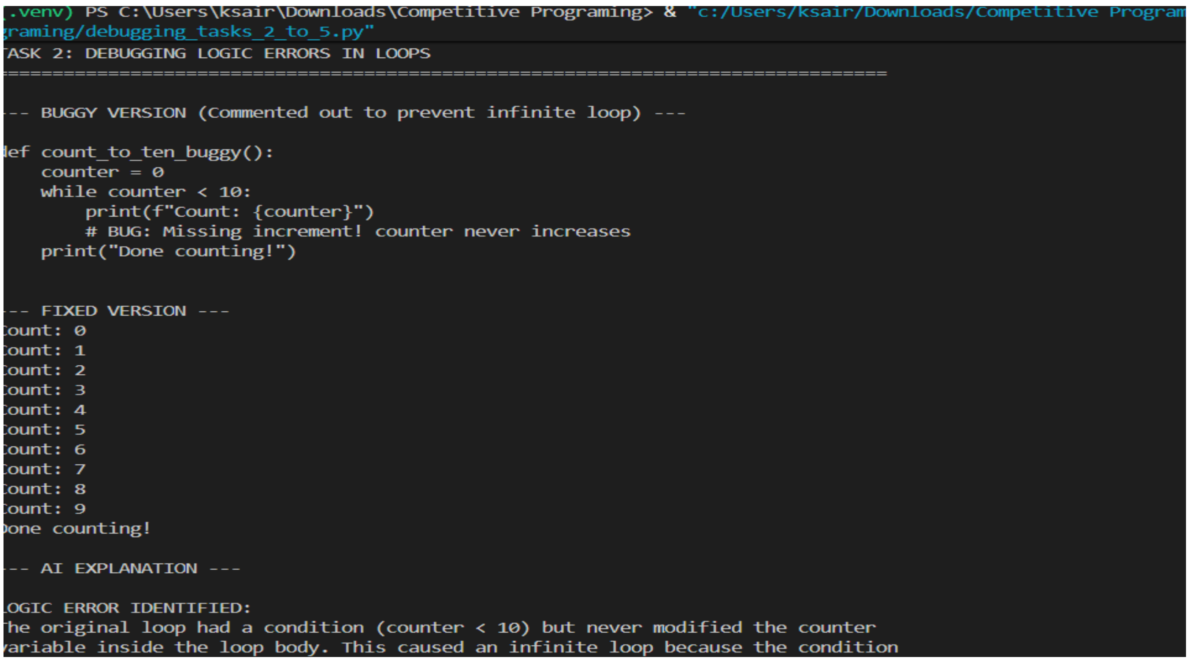
This loop is running infinitely. Can you identify the logic error:

Code and Fixed Code:



```
print("""
def count_to_ten_buggy():
    counter = 0
    while counter < 10:
        print(f"Count: {counter}")
        # BUG: Missing increment! counter never increases
    print("Done counting!")
""")

# AFTER: Fixed Loop
print("\n--- FIXED VERSION ---")
def count_to_ten_fixed():
    """Counts from 0 to 9 with proper increment logic"""
    counter = 0
    while counter < 10:
        print(f"Count: {counter}")
        counter += 1  # FIX: Added increment to prevent infinite loop
    print("Done counting!")

count_to_ten_fixed()

print("\n--- AI EXPLANATION ---")
print("""
LOGIC ERROR IDENTIFIED:
The original loop had a condition (counter < 10) but never modified the counter
variable inside the loop body. This caused an infinite loop because the condition
would always remain True.

FIX APPLIED:
Added 'counter += 1' to increment the counter variable in each iteration.
This ensures the loop will eventually terminate when counter reaches 10.
```

Output and Explaination:



```
.venv) PS C:\Users\ksair\Downloads\Competitive Programing> & "c:/Users/ksair/Downloads/Competitive Program
graming/debugging_tasks_2_to_5.py"
TASK 2: DEBUGGING LOGIC ERRORS IN LOOPS
============================================================================

-- BUGGY VERSION (Commented out to prevent infinite loop) ---

lef count_to_ten_buggy():
   counter = 0
   while counter < 10:
       print(f"Count: {counter}")
       # BUG: Missing increment! counter never increases
   print("Done counting!")


-- FIXED VERSION ---
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Count: 6
Count: 7
Count: 8
Count: 9
Done counting!

-- AI EXPLANATION ---

LOGIC ERROR IDENTIFIED:
The original loop had a condition (counter < 10) but never modified the counter
variable inside the loop body. This caused an infinite loop because the condition
```

Justification:

Infinite loops are critical bugs that freeze applications and consume 100% CPU resources, making systems unresponsive. They commonly occur in web servers, game loops, and data processing scripts where loop conditions aren't properly updated. Learning to identify missing increment/decrement logic is essential because these bugs are difficult to debug in production environments and can cause financial losses in cloud computing due to continuous resource consumption.

Task 3: Handling Runtime Errors (Division by Zero)

Scenario

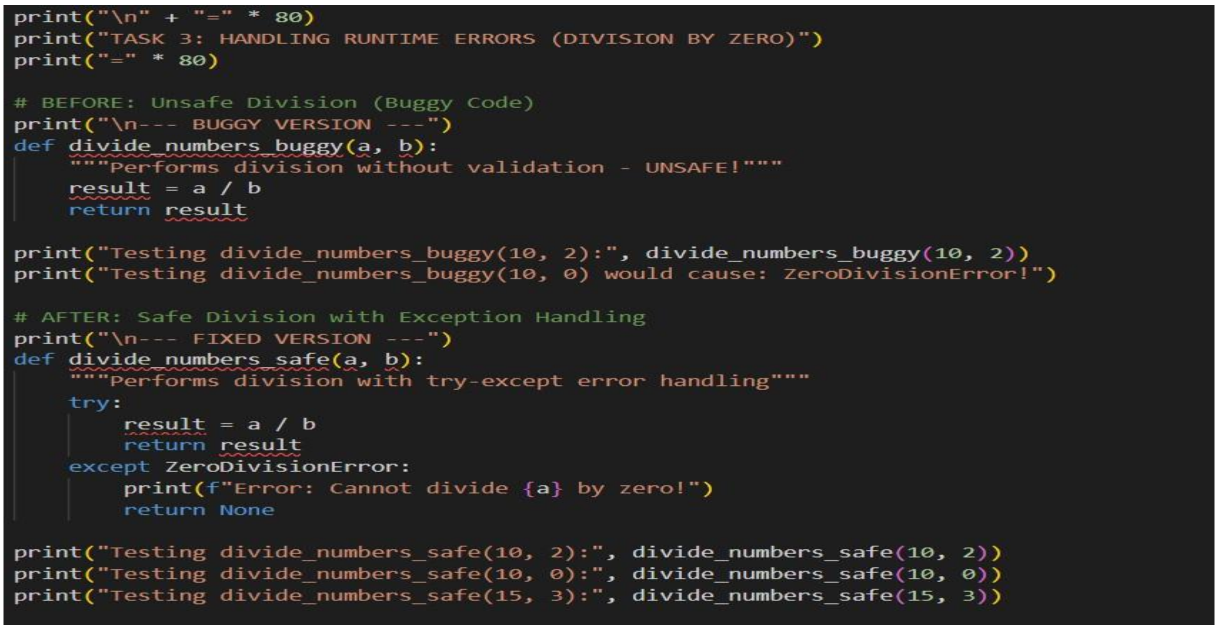
A Python function crashes during execution due to a division by zero error. Requirements

- Provide a function that performs division without validation
- Use AI to identify the runtime error
- Let AI add try-except blocks for safe execution
- Review AI's error-handling approach

Prompt:

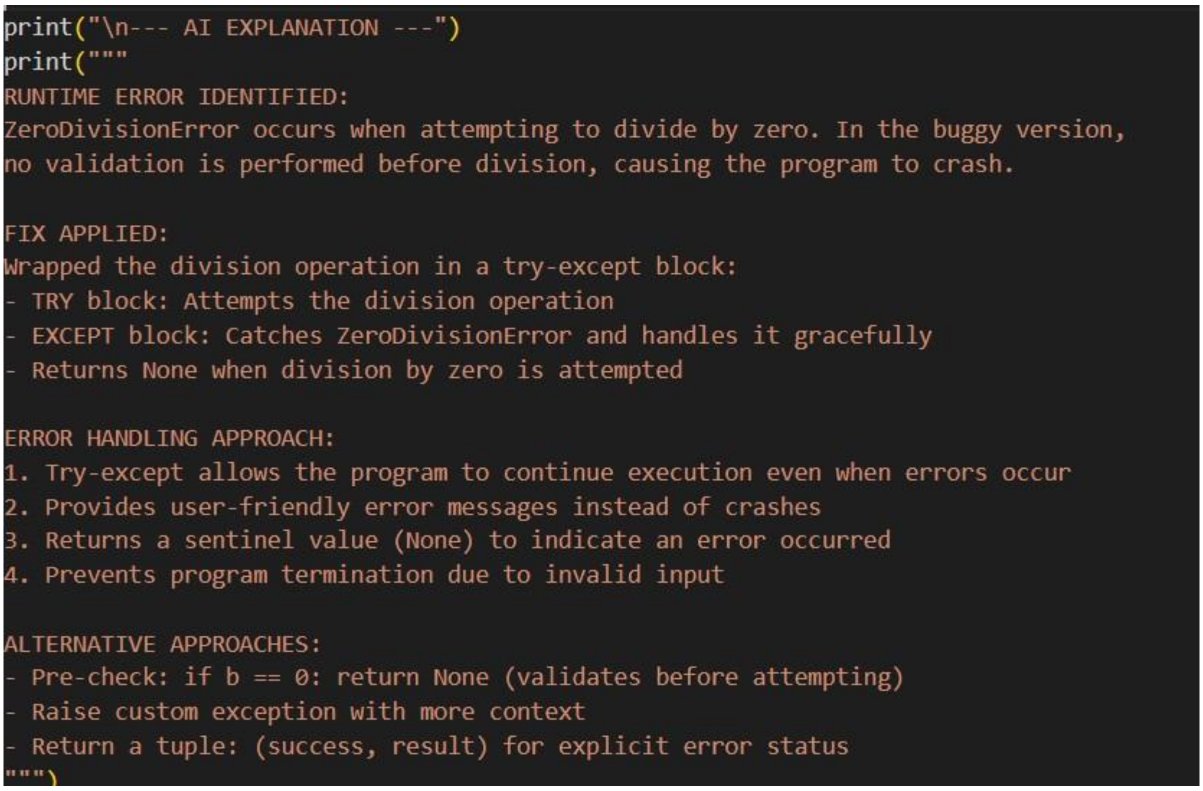
This function crashes with division by zero. Can you help fix it:

Code:



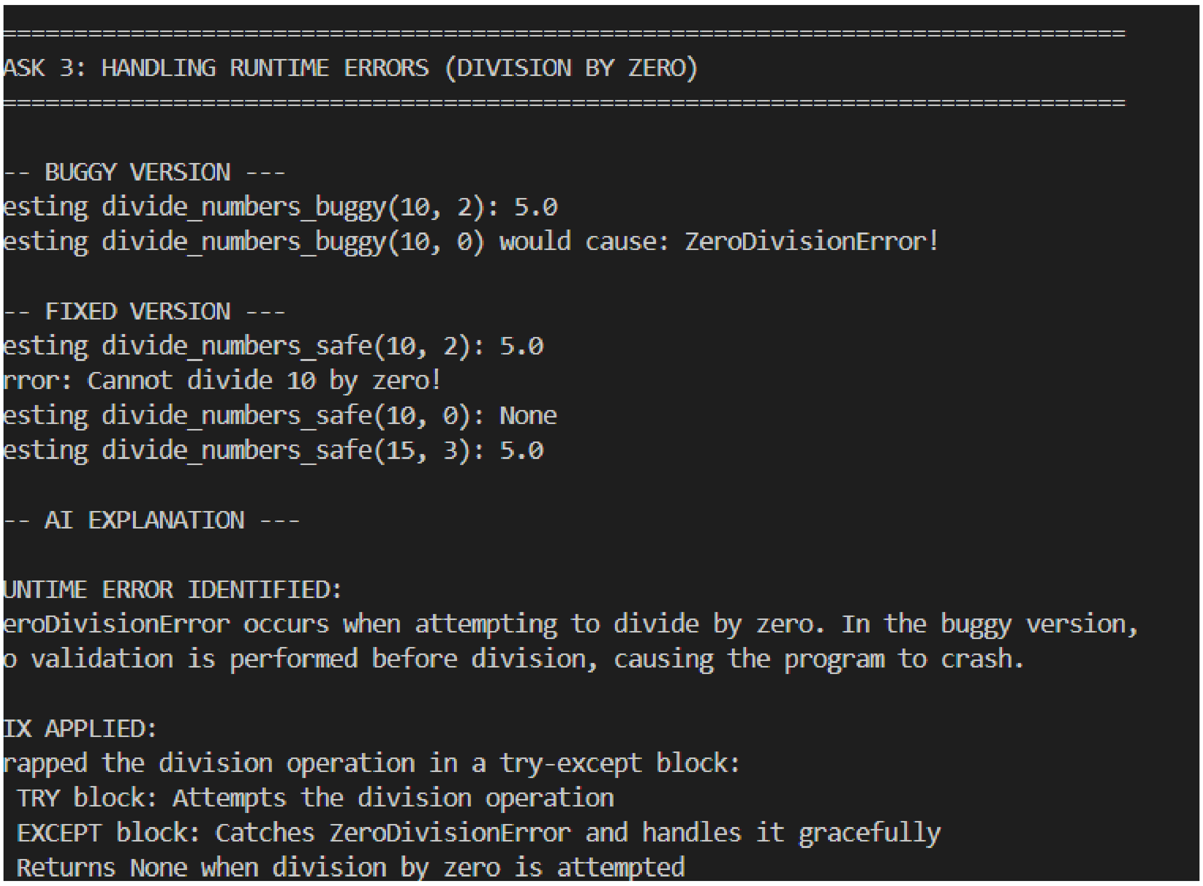
```python
print("\n" + "=" * 80)
print("TASK 3: HANDLING RUNTIME ERRORS (DIVISION BY ZERO)")
print("=" * 80)

# BEFORE: Unsafe Division (Buggy Code)
print("\n--- BUGGY VERSION ---")
def divide_numbers_buggy(a, b):
    """Performs division without validation - UNSAFE!"""
    result = a / b
    return result

print("Testing divide_numbers_buggy(10, 2):", divide_numbers_buggy(10, 2))
print("Testing divide_numbers_buggy(10, 0) would cause: ZeroDivisionError!")

# AFTER: Safe Division with Exception Handling
print("\n--- FIXED VERSION ---")
def divide_numbers_safe(a, b):
    """Performs division with try-except error handling"""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print(f"Error: Cannot divide {a} by zero!")
        return None

print("Testing divide_numbers_safe(10, 2):", divide_numbers_safe(10, 2))
print("Testing divide_numbers_safe(10, 0):", divide_numbers_safe(10, 0))
print("Testing divide_numbers_safe(15, 3):", divide_numbers_safe(15, 3))
```

Fixed Code and Explaination:



```
print("\n--- AI EXPLANATION ---")
print("""
RUNTIME ERROR IDENTIFIED:
ZeroDivisionError occurs when attempting to divide by zero. In the buggy version,
no validation is performed before division, causing the program to crash.

FIX APPLIED:
Wrapped the division operation in a try-except block:
- TRY block: Attempts the division operation
- EXCEPT block: Catches ZeroDivisionError and handles it gracefully
- Returns None when division by zero is attempted

ERROR HANDLING APPROACH:
1. Try-except allows the program to continue execution even when errors occur
2. Provides user-friendly error messages instead of crashes
3. Returns a sentinel value (None) to indicate an error occurred
4. Prevents program termination due to invalid input

ALTERNATIVE APPROACHES:
- Pre-check: if b == 0: return None (validates before attempting)
- Raise custom exception with more context
- Return a tuple: (success, result) for explicit error status
""")
```

Output:



```
===================================================================================
ASK 3: HANDLING RUNTIME ERRORS (DIVISION BY ZERO)
===================================================================================

-- BUGGY VERSION ---
esting divide_numbers_buggy(10, 2): 5.0
esting divide_numbers_buggy(10, 0) would cause: ZeroDivisionError!

-- FIXED VERSION ---
esting divide_numbers_safe(10, 2): 5.0
rror: Cannot divide 10 by zero!
esting divide_numbers_safe(10, 0): None
esting divide_numbers_safe(15, 3): 5.0

-- AI EXPLANATION ---

UNTIME ERROR IDENTIFIED:
eroDivisionError occurs when attempting to divide by zero. In the buggy version,
o validation is performed before division, causing the program to crash.

IX APPLIED:
rapped the division operation in a try-except block:
 TRY block: Attempts the division operation
 EXCEPT block: Catches ZeroDivisionError and handles it gracefully
 Returns None when division by zero is attempted
```

Justification:

Division by zero errors cause immediate program crashes and data loss, creating poor user experience and potential security vulnerabilities. This occurs frequently in financial calculations, analytics dashboards, and scientific applications where denominators come from user input or dynamic data. Proper try-except error handling is fundamental to defensive programming, ensuring applications remain stable and handle edge cases gracefully without terminating unexpectedly.

Task 4: Debugging Class Definition Errors

Scenario

You are given a faulty Python class where the constructor is incorrectly defined.

Requirements

• Provide a class definition with missing self-parameter

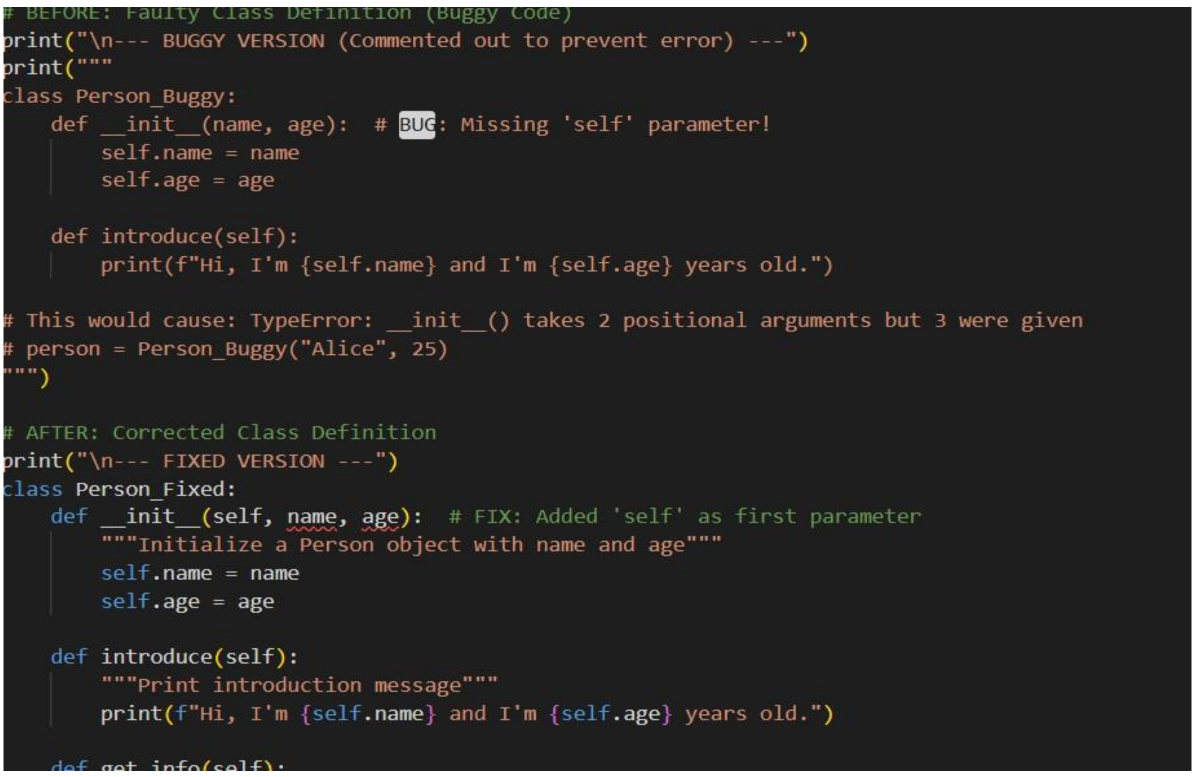
• Use AI to identify the issue in the __init__() method

• Allow AI to correct the class definition

• Understand why self is required

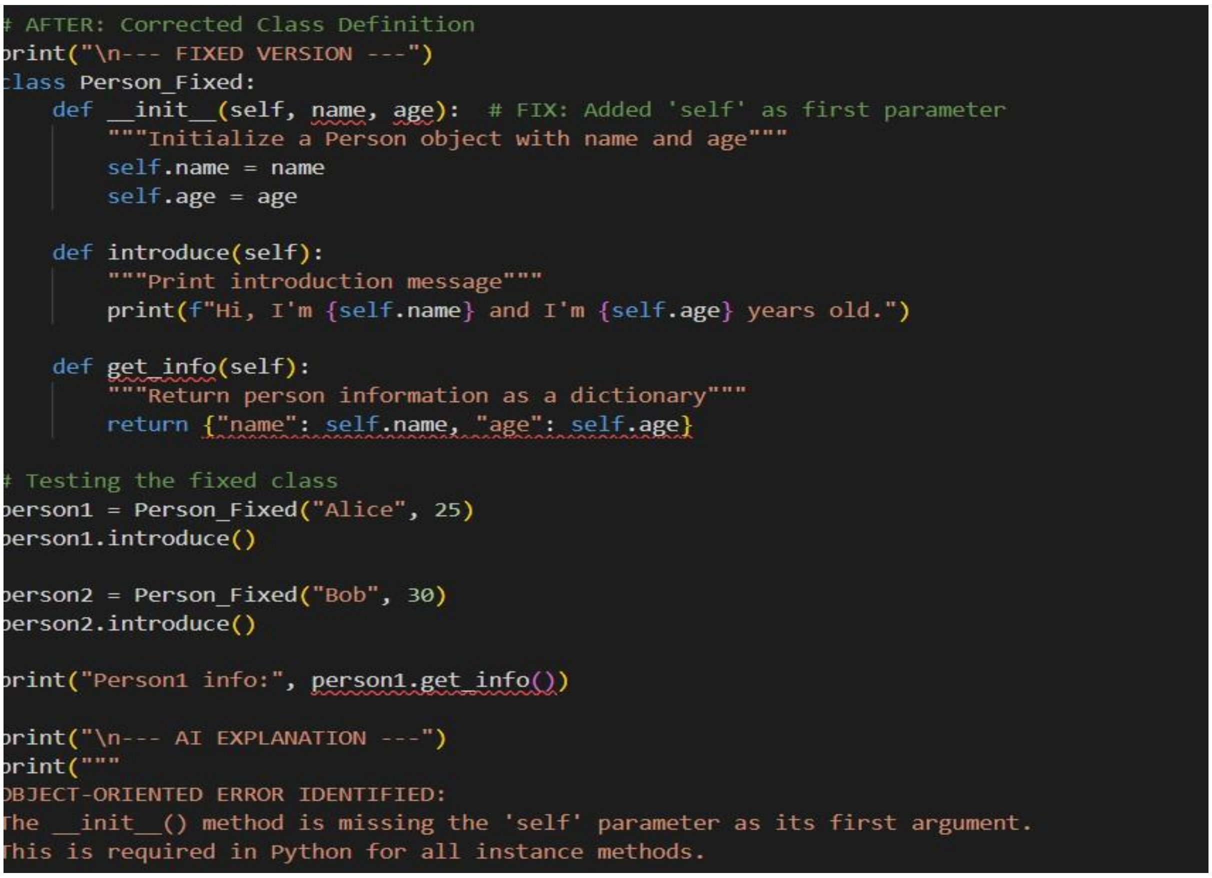
Prompt: "My Python class won't work. What's wrong with this code:

Code:



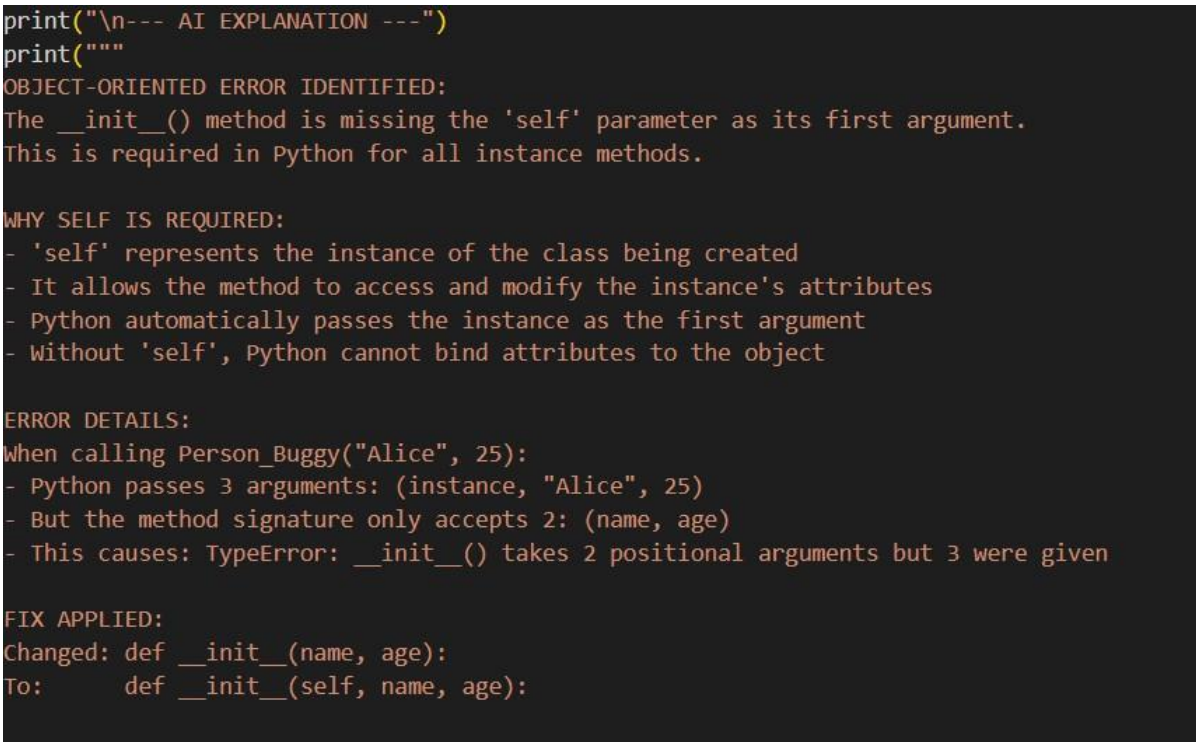
```python
# BEFORE: Faulty Class Definition (Buggy Code)
print("\n--- BUGGY VERSION (Commented out to prevent error) ---")
print("""
class Person_Buggy:
    def __init__(name, age):  # BUG: Missing 'self' parameter!
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")

# This would cause: TypeError: __init__() takes 2 positional arguments but 3 were given
# person = Person_Buggy("Alice", 25)
""")

# AFTER: Corrected Class Definition
print("\n--- FIXED VERSION ---")
class Person_Fixed:
    def __init__(self, name, age):  # FIX: Added 'self' as first parameter
        """Initialize a Person object with name and age"""
        self.name = name
        self.age = age

    def introduce(self):
        """Print introduction message"""
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")

    def get_info(self):
```

Fixed code:



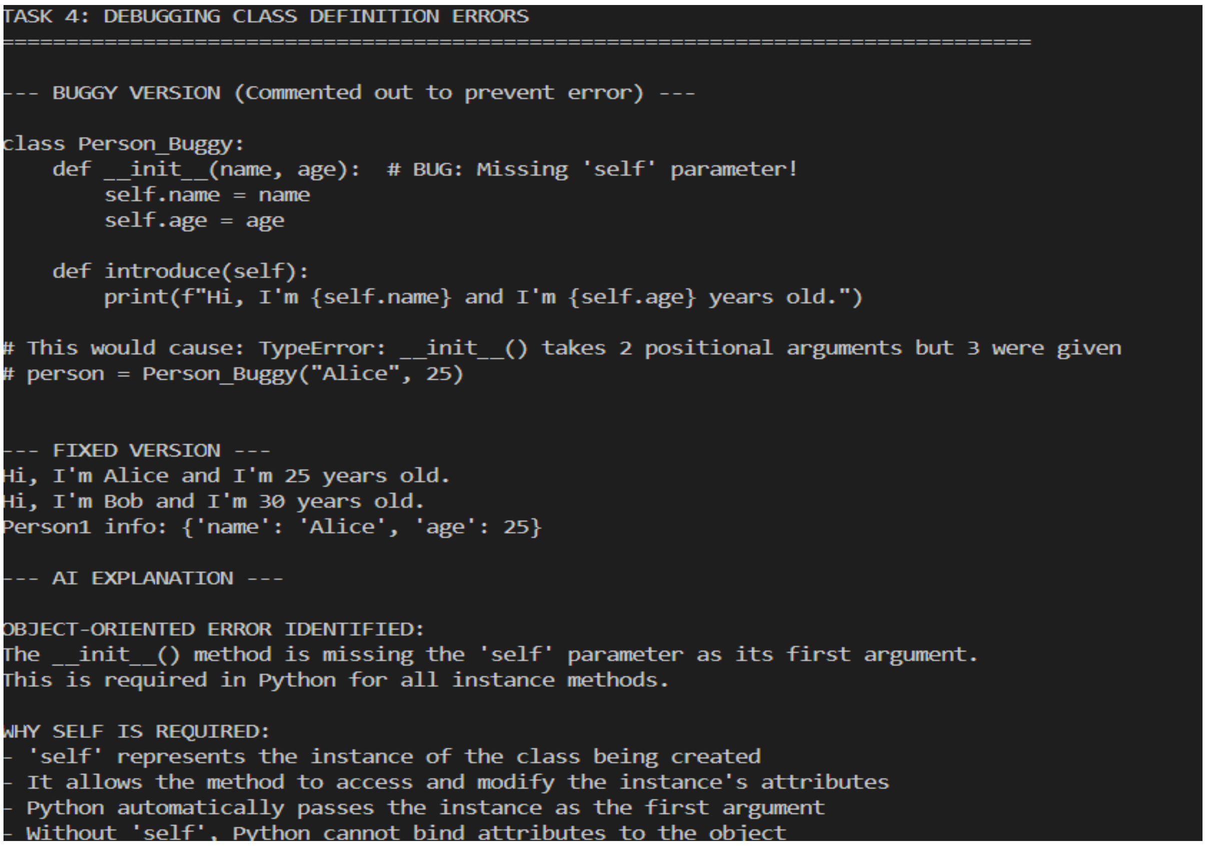
```
# AFTER: Corrected Class Definition
rint("\n--- FIXED VERSION ---")
lass Person_Fixed:
    def __init__(self, name, age):  # FIX: Added 'self' as first parameter
        """Initialize a Person object with name and age"""
        self.name = name
        self.age = age

    def introduce(self):
        """Print introduction message"""
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")

    def get_info(self):
        """Return person information as a dictionary"""
        return {"name": self.name, "age": self.age}

# Testing the fixed class
erson1 = Person_Fixed("Alice", 25)
erson1.introduce()

erson2 = Person_Fixed("Bob", 30)
erson2.introduce()

rint("Person1 info:", person1.get_info())

rint("\n--- AI EXPLANATION ---")
rint("""
BJECT-ORIENTED ERROR IDENTIFIED:
he __init__() method is missing the 'self' parameter as its first argument.
his is required in Python for all instance methods.
```

Ai-explaination:



```
print("\n--- AI EXPLANATION ---")
print("""
OBJECT-ORIENTED ERROR IDENTIFIED:
The __init__() method is missing the 'self' parameter as its first argument.
This is required in Python for all instance methods.

WHY SELF IS REQUIRED:
- 'self' represents the instance of the class being created
- It allows the method to access and modify the instance's attributes
- Python automatically passes the instance as the first argument
- Without 'self', Python cannot bind attributes to the object

ERROR DETAILS:
When calling Person_Buggy("Alice", 25):
- Python passes 3 arguments: (instance, "Alice", 25)
- But the method signature only accepts 2: (name, age)
- This causes: TypeError: __init__() takes 2 positional arguments but 3 were given

FIX APPLIED:
Changed: def __init__(name, age):
To:      def __init__(self, name, age):
```

Output:



```
TASK 4: DEBUGGING CLASS DEFINITION ERRORS
================================================================================

--- BUGGY VERSION (Commented out to prevent error) ---

class Person_Buggy:
    def __init__(name, age):  # BUG: Missing 'self' parameter!
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, I'm {self.name} and I'm {self.age} years old.")

# This would cause: TypeError: __init__() takes 2 positional arguments but 3 were given
# person = Person_Buggy("Alice", 25)


--- FIXED VERSION ---
Hi, I'm Alice and I'm 25 years old.
Hi, I'm Bob and I'm 30 years old.
Person1 info: {'name': 'Alice', 'age': 25}

--- AI EXPLANATION ---

OBJECT-ORIENTED ERROR IDENTIFIED:
The __init__() method is missing the 'self' parameter as its first argument.
This is required in Python for all instance methods.

WHY SELF IS REQUIRED:
- 'self' represents the instance of the class being created
- It allows the method to access and modify the instance's attributes
- Python automatically passes the instance as the first argument
- Without 'self', Python cannot bind attributes to the object
```

Justification:

Missing the 'self' parameter in class methods breaks object-oriented programming fundamentals and prevents object instantiation entirely. This error is extremely common for beginners learning OOP and appears in real-world frameworks like Django, Flask, and SQL Alchemy that rely heavily on class-based structures. Understanding 'self' is crucial because it represents the instance being created and allows methods to access object attributes, making it essential for building maintainable, production-grade applications.

Task 5: Resolving Index Errors in Lists

Scenario

A program crashes when accessing an invalid index in a list.

Requirements

• Provide code that accesses an out-of-range list index

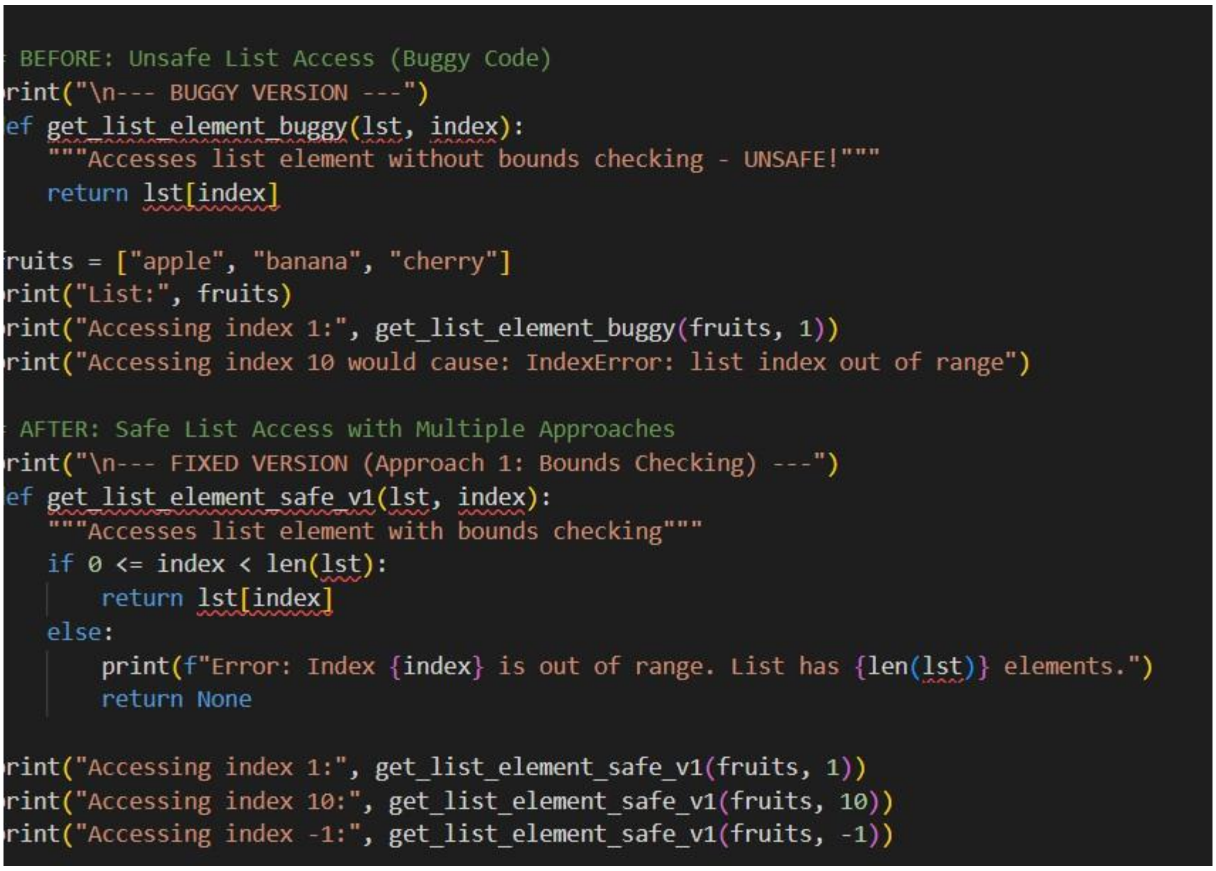
• Use AI to identify the Index Error

• Let AI suggest safe access methods

• Apply bounds checking or exception handling

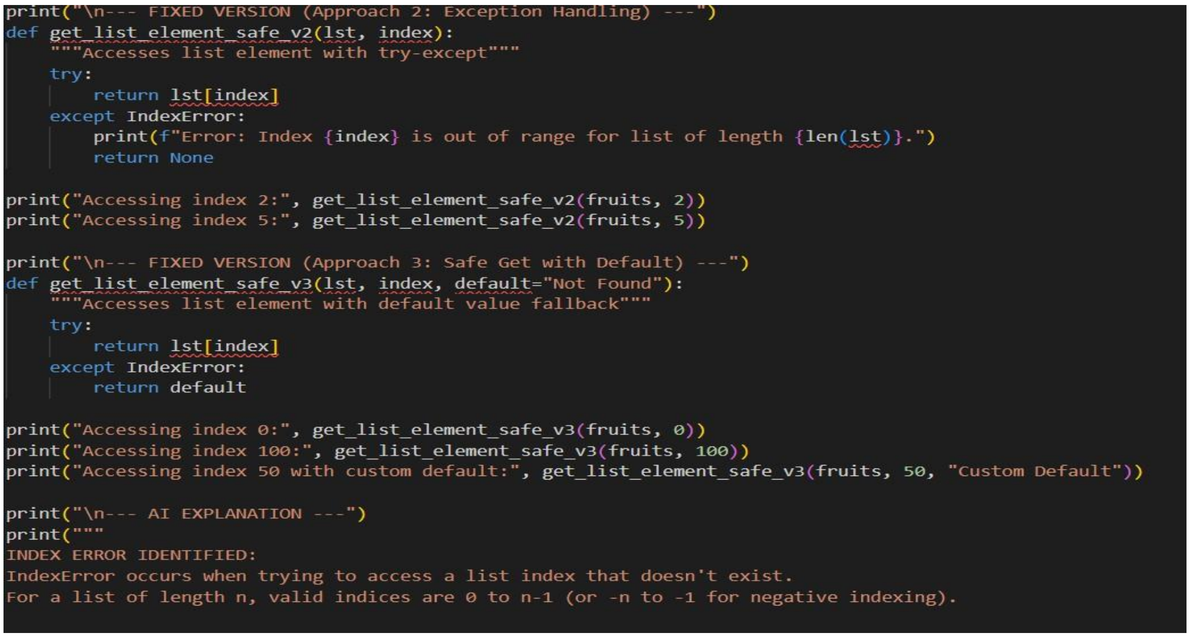
Prompt: "My program crashes with an IndexError. Can you fix this code:

Code:



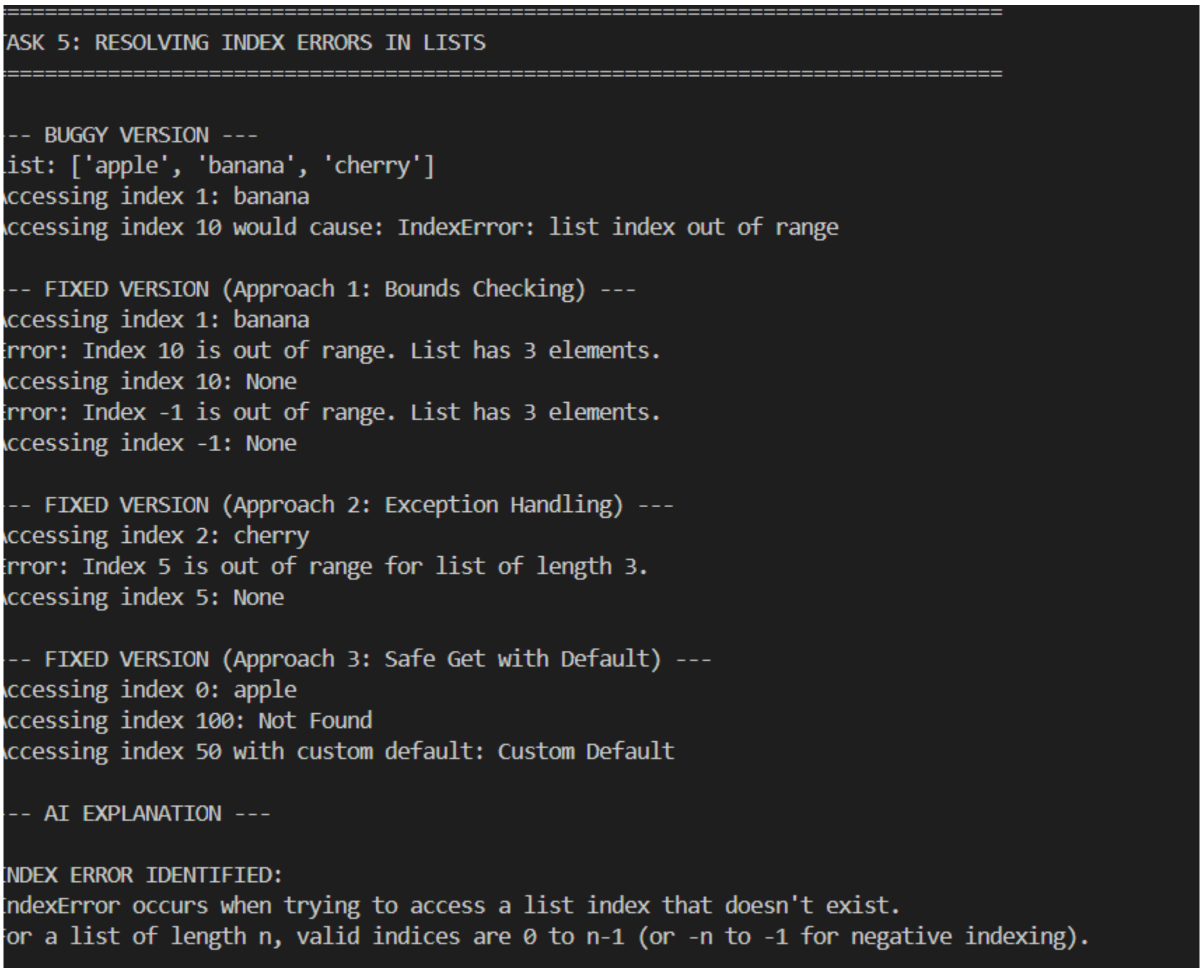
```
 BEFORE: Unsafe List Access (Buggy Code)
rint("\n--- BUGGY VERSION ---")
ef get_list_element_buggy(lst, index):
   """Accesses list element without bounds checking - UNSAFE!"""
   return lst[index]

ruits = ["apple", "banana", "cherry"]
rint("List:", fruits)
rint("Accessing index 1:", get_list_element_buggy(fruits, 1))
rint("Accessing index 10 would cause: IndexError: list index out of range")

 AFTER: Safe List Access with Multiple Approaches
rint("\n--- FIXED VERSION (Approach 1: Bounds Checking) ---")
ef get_list_element_safe_v1(lst, index):
   """Accesses list element with bounds checking"""
   if 0 <= index < len(lst):
       return lst[index]
   else:
       print(f"Error: Index {index} is out of range. List has {len(lst)} elements.")
       return None

rint("Accessing index 1:", get_list_element_safe_v1(fruits, 1))
rint("Accessing index 10:", get_list_element_safe_v1(fruits, 10))
rint("Accessing index -1:", get_list_element_safe_v1(fruits, -1))
```

Fixed code and Explaination:

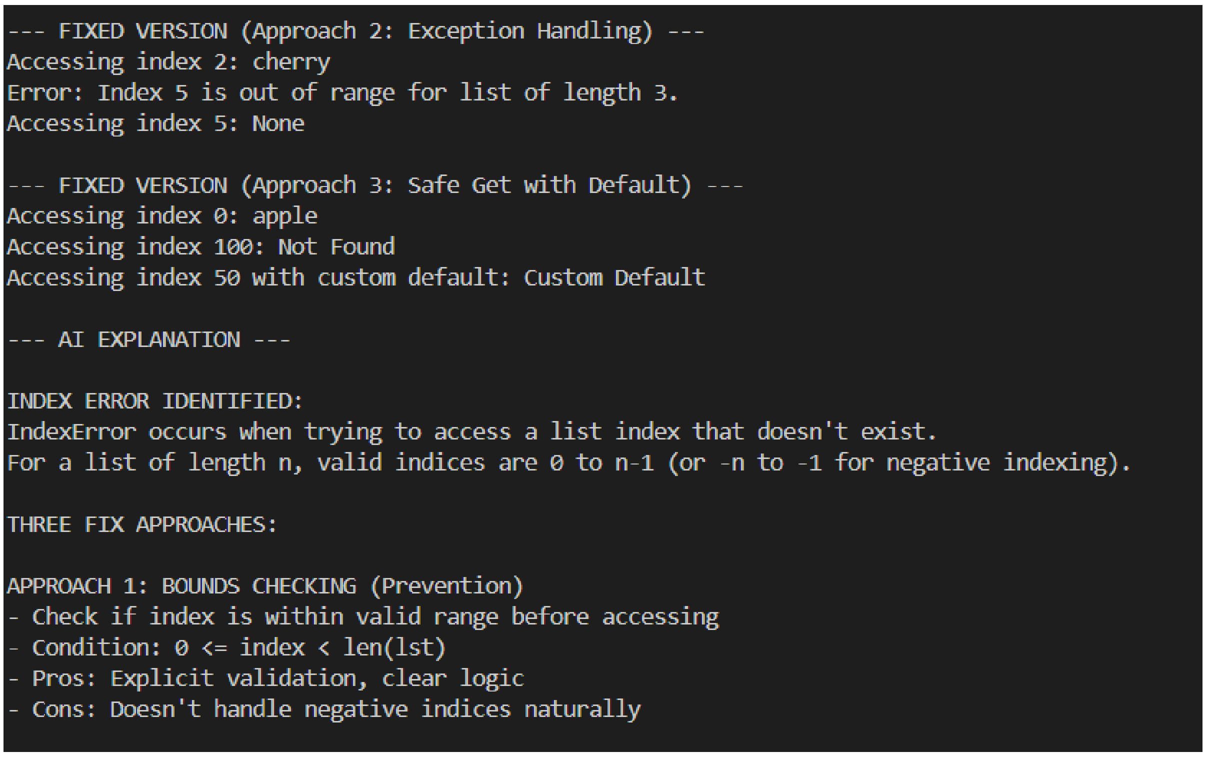


```
print("\n--- FIXED VERSION (Approach 2: Exception Handling) ---")
def get_list_element_safe_v2(lst, index):
    """Accesses list element with try-except"""
    try:
        return lst[index]
    except IndexError:
        print(f"Error: Index {index} is out of range for list of length {len(lst)}.")
        return None

print("Accessing index 2:", get_list_element_safe_v2(fruits, 2))
print("Accessing index 5:", get_list_element_safe_v2(fruits, 5))

print("\n--- FIXED VERSION (Approach 3: Safe Get with Default) ---")
def get_list_element_safe_v3(lst, index, default="Not Found"):
    """Accesses list element with default value fallback"""
    try:
        return lst[index]
    except IndexError:
        return default

print("Accessing index 0:", get_list_element_safe_v3(fruits, 0))
print("Accessing index 100:", get_list_element_safe_v3(fruits, 100))
print("Accessing index 50 with custom default:", get_list_element_safe_v3(fruits, 50, "Custom Default"))

print("\n--- AI EXPLANATION ---")
print("""
INDEX ERROR IDENTIFIED:
IndexError occurs when trying to access a list index that doesn't exist.
For a list of length n, valid indices are 0 to n-1 (or -n to -1 for negative indexing).
```

Output:



```
================================================================================
ASK 5: RESOLVING INDEX ERRORS IN LISTS
================================================================================

-- BUGGY VERSION ---
ist: ['apple', 'banana', 'cherry']
ccessing index 1: banana
ccessing index 10 would cause: IndexError: list index out of range

-- FIXED VERSION (Approach 1: Bounds Checking) ---
ccessing index 1: banana
rror: Index 10 is out of range. List has 3 elements.
ccessing index 10: None
rror: Index -1 is out of range. List has 3 elements.
ccessing index -1: None

-- FIXED VERSION (Approach 2: Exception Handling) ---
ccessing index 2: cherry
rror: Index 5 is out of range for list of length 3.
ccessing index 5: None

-- FIXED VERSION (Approach 3: Safe Get with Default) ---
ccessing index 0: apple
ccessing index 100: Not Found
ccessing index 50 with custom default: Custom Default

-- AI EXPLANATION ---

NDEX ERROR IDENTIFIED:
ndexError occurs when trying to access a list index that doesn't exist.
or a list of length n, valid indices are 0 to n-1 (or -n to -1 for negative indexing).
```



```
--- FIXED VERSION (Approach 2: Exception Handling) ---
Accessing index 2: cherry
Error: Index 5 is out of range for list of length 3.
Accessing index 5: None

--- FIXED VERSION (Approach 3: Safe Get with Default) ---
Accessing index 0: apple
Accessing index 100: Not Found
Accessing index 50 with custom default: Custom Default

--- AI EXPLANATION ---

INDEX ERROR IDENTIFIED:
IndexError occurs when trying to access a list index that doesn't exist.
For a list of length n, valid indices are 0 to n-1 (or -n to -1 for negative indexing).

THREE FIX APPROACHES:

APPROACH 1: BOUNDS CHECKING (Prevention)
- Check if index is within valid range before accessing
- Condition: 0 <= index < len(lst)
- Pros: Explicit validation, clear logic
- Cons: Doesn't handle negative indices naturally
```

Justification:

Index errors occur unpredictably when accessing collections with dynamic sizes, commonly happening in CSV processing, API responses, and user input parsing. These boundary errors crash programs when data structures change size or when invalid indices are requested. Learning multiple resolution approaches—bounds checking, exception handling, and default values—develops defensive programming skills essential for handling variable-length data from databases, APIs, and file operations in production systems.