AI ASSISTED CODING

ASSIGNMENT-7

B.Sri Laxmi Gayathri

2303a52033

Batch:38

**Lab 7: Error Debugging with AI: Systematic approaches to finding and fixing bugs**

**Lab Objectives:**

• To identify and correct syntax, logic, and runtime errors in Python programs using AI tools

• To understand common programming bugs and AI-assisted debugging suggestions

• To evaluate how AI explains, detects, and fixes different types of coding errors

• To build confidence in using AI for structured debugging practices

**Task 1: Fixing Syntax Errors**

**Scenario:**

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

**Python:**

```
def add(a,b)
    return a+b
```

**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

**Expected Output:**

• Corrected function with proper syntax

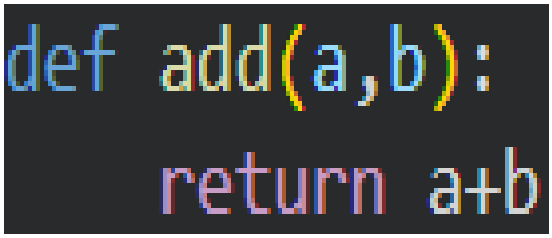
• Syntax error resolved successfully

• AI-generated explanation of the fix

**Prompt:**

Check the Python function below for errors, resolve the syntax mistake, and explain the correction made.

**Code & Output:**




```
def add(a, b):
    return a + b
# Example usage:
result = add(5, 3)
print(result)
```

```
8
```

**Explanation:**

The code above has a SyntaxError. In Python, a function definition (def) must be followed by parentheses containing parameters, and then a colon (:) before the indented block of code that forms the function body. Without the colon, Python doesn't recognize the start of the function's code block, leading to an error.

**Task 2: Debugging Logic Errors in Loops**

**Scenario:**

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

**Python:**

```
def count_down(n):
    while n>=0:
```

print(n)

n+=1  #should be n-=1

**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

**Expected Output:**

• Infinite loop issue resolved

• Correct increment/decrement logic applied

• AI explanation of the logic error

**Prompt:**

Analyze the following Python loop, identify the cause of the infinite loop, correct the logic error, and explain how the fix changes the loop behavior.

**Code & Output:**

```
def count_down(n):
    while n >= 0:
        print(n)
        n += 1 # This should be n -= 1

# Example of the infinite loop (do not run this if you want to avoid an infinite loop)
# count_down(5)
```

```
def count_down_corrected(n):
    while n >= 0:
        print(n)
        n -= 1 # Corrected: n is now decremented

# Example usage:
print("Counting down from 5:")
count_down_corrected(5)
print("Counting down from 0:")
count_down_corrected(0)
print("Counting down from -2 (loop will not run):")
count_down_corrected(-2)
```

```
Counting down from 5:
5
4
3
2
1
0
Counting down from 0:
0
Counting down from -2 (loop will not run):
```

**Explanation:**

The count_down function above contains a logic error that results in an infinite loop. The while condition n >= 0 checks if n is greater than or equal to zero. Inside the loop, n is *incremented* (n += 1) instead of being decremented.
If n starts as a non-negative number (e.g., 5), n will keep increasing (5, 6, 7, ...), and therefore n >= 0 will always remain true, causing the loop to run forever. To count down, n should be decremented.

**Task 3: Handling Runtime Errors (Division by Zero)**

**Scenario:**

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

#Debug the following code

```
def divide (a,b):
    return a/b
print(divide(10,0))
```

**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

**Expected Output:**

• Function executes safely without crashing

• Division by zero handled using try-except

• Clear AI-generated explanation of runtime error handling

**Prompt:**

Identify the runtime error in the following Python code, modify the function using try-except blocks to handle the error safely, and explain how the error handling works.

**Code & Output:**

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

# This will cause a ZeroDivisionError
# print(divide(10, 0))
```

```
def divide_safe(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
        return None # Or raise a custom exception, or return a specific value

# Example usage:
print("10 divided by 2:", divide_safe(10, 2))
print("10 divided by 0:", divide_safe(10, 0))
print("5 divided by 1:", divide_safe(5, 1))
```

```
10 divided by 2: 5.0
Error: Cannot divide by zero!
10 divided by 0: None
5 divided by 1: 5.0
```

**Explanation:**

The program crashes because dividing a number by zero raises a ZeroDivisionError at runtime. The AI identifies this issue and adds a try-except block to handle the error safely. The try block contains the normal division operation, while the except block catches the division by zero error. Instead of crashing, the program displays a meaningful error message. This ensures safe and stable program execution.

**Task 4: Debugging Class Definition Errors**

**Scenario:**

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

**Python**

class Rectangle:

    def__init__(length,width):

        self.length=length

        self.width=width

**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

**Expected Output:**

• Corrected __init__() method

• Proper use of self in class definition

• AI explanation of object-oriented error

**Prompt:**

Identify the error in the following Python class constructor, explain why the error occurs, and correct the __init__() method using proper object-oriented syntax.

**Code & Output:**

```
class Rectangle:
    def __init__(length, width):
        length = length
        width = width
# Example of attempting to create an instance (this will cause an error)
# rect = Rectangle(10, 5)
# print(rect.length)
```

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
    def perimeter(self): # Added a perimeter method for demonstration
        return 2 * (self.length + self.width)
# Example usage:
rect1 = Rectangle(10, 5)
print(f"Rectangle 1 - Length: {rect1.length}, Width: {rect1.width}")
print(f"Rectangle 1 Area: {rect1.area()}")
print(f"Rectangle 1 Perimeter: {rect1.perimeter()}")
rect2 = Rectangle(7, 3)
print(f"Rectangle 2 - Length: {rect2.length}, Width: {rect2.width}")
print(f"Rectangle 2 Area: {rect2.area()}")
```

```
Rectangle 1 - Length: 10, Width: 5
Rectangle 1 Area: 50
Rectangle 1 Perimeter: 30
Rectangle 2 - Length: 7, Width: 3
Rectangle 2 Area: 21
```

**Explanation:**

The constructor method is incorrect because it does not include the self parameter and also misses a space in def __init__. In Python, self represents the current object and is required to access instance variables. Without self, Python cannot associate variables with the object being created. The AI detects this object-oriented error and corrects the constructor definition. After fixing it, the class works properly and can store object-specific data.

**Task 5: Resolving Index Errors in Lists**

**Scenario:**

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

**Python**

numbers=[1,2,3]

print(numbers[5])

**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

**Expected Output:**

- Index error resolved
- Safe list access logic implemented
- AI suggestion using length checks or exception handling

**Prompt:**

Analyze the following Python code, identify the IndexError caused by invalid list access, and suggest a safe way to handle the error using bounds checking or exception handling.

**Code & Output:**

```
numbers = [1, 2, 3]

# This will cause an IndexError
# print(numbers[5])
```

```
def get_list_element_safe(a_list, index):
    try:
        return a_list[index]
    except IndexError:
        print(f"Error: Index {index} is out of bounds for list of length {len(a_list)}.")
        return None
def get_list_element_with_check(a_list, index):
    if 0 <= index < len(a_list):
        return a_list[index]
    else:
        print(f"Error: Index {index} is out of bounds for list of length {len(a_list)}.")
        return None
numbers = [10, 20, 30]
print("--- Using try-except ---")
print(f"Element at index 1: {get_list_element_safe(numbers, 1)}")
print(f"Element at index 5: {get_list_element_safe(numbers, 5)}")
print(f"Element at index -1: {get_list_element_safe(numbers, -1)}") # Python allows negative indexing
print("\n--- Using bounds checking ---")
print(f"Element at index 1: {get_list_element_with_check(numbers, 1)}")
print(f"Element at index 5: {get_list_element_with_check(numbers, 5)}")
print(f"Element at index -1: {get_list_element_with_check(numbers, -1)}") # Will report error if negative indexing is not explicitly handled
```

```
--- Using try-except ---
Element at index 1: 20
Error: Index 5 is out of bounds for list of length 3.
Element at index 5: None
Element at index -1: 30
```

```
--- Using bounds checking ---
Element at index 1: 20
Error: Index 5 is out of bounds for list of length 3.
Element at index 5: None
Error: Index -1 is out of bounds for list of length 3.
Element at index -1: None
```

**Explanation:**

The program crashes because it tries to access an index that does not exist in the list. The AI identifies this as an IndexError, which occurs when the index value is outside the list's valid range. To prevent the crash, the AI suggests checking the index against the list length before accessing it. Alternatively, a try-except block can be used to catch the error gracefully. Both approaches ensure safe list access without program termination.