

# Assignment 7.5

G SAIANIRUDH

2303A51891

Batch 04

## 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 to perform structured debugging practices.

### Lab Outcomes (LOs):

After completing this lab, students will be able to:

- Use AI tools to detect and correct syntax, logic, and runtime errors.
- Interpret AI-suggested bug fixes and explanations.
- Apply systematic debugging strategies supported by AI-generated insights.

Refactor buggy code using responsible and reliable programming patterns.

### Task 1 :

#### Task 1 (Mutable Default Argument – Function Bug)

Task: Analyze given code where a mutable default argument causes unexpected behavior.  
Use AI to fix it.

```
# Bug: Mutable default argument
```

```
def add_item(item, items=[]):
```

```
    items.append(item)
```

```
    return items
```

```
print(add_item(1))
```

```
print(add_item(2))
```

Expected Output: Corrected function avoids shared list bug

## Code

The screenshot shows a code editor with multiple tabs at the top: lab01.py, lab02.py, assignments01.py, lab7.5.py (which is the active tab), and lab03.py. The code in lab7.5.py is as follows:

```
1  # You are an AI debugging assistant.
2
3  # Analyze the following Python code where a mutable default argument causes unexpected behavior.
4  # 1. Identify the bug type.
5  # 2. Explain why the issue occurs.
6  # 3. Fix the code using best Python practices.
7  # 4. Show the corrected output.
8
9  # Code:
10 # def add_item(item, items=[]):
11 #     items.append(item)
12 #     return items
13
14 # print(add_item(1))
15 # print(add_item(2))
16 # Bug Type: Mutable Default Argument
17 # Explanation: In Python, default arguments are evaluated only once when the function is defined, not each time the function is called.
18 # This means that if a mutable object (like a list or dictionary) is used as a default argument, it will retain its state between function calls.
19 # As a result, when we call add_item multiple times without providing the second argument, the same list is used and modified, leading to unexpect
20 def add_item(item, items=None):
21     if items is None:
22         items = []
23     items.append(item)
24     return items
25 print(add_item(1)) # Output: [1]
26 print(add_item(2)) # Output: [2]
27 # Corrected Output:
28 # [1]
29 # [2]
30
```

Below the code editor, there is a terminal window showing the command PS C:\Users\aniru\OneDrive\AI\_Assistant\_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI\_Assistant\_Coding/lab7.5.py followed by the output [1] [2]. To the right of the terminal, there is a sidebar with icons for powershell and Python.

## Explanation / Justification:

In Python, default arguments are evaluated only once when the function is defined. When a mutable object like a list is used as a default argument, the same list is shared across multiple function calls. This causes previously added elements to persist unexpectedly. To avoid this, `None` is used as the default value and a new list is created inside the function for each call.

## Task 2 (Floating-Point Precision Error)

Task: Analyze given code where floating-point comparison fails.

Use AI to correct with tolerance.

```
# Bug: Floating point precision issue
```

```
def check_sum():
```

```
    return (0.1 + 0.2) == 0.3
```

```
print(check_sum())
```

Expected Output: Corrected function

## Code:

```
47
48 #TASK 2
49 # You are an AI debugging assistant.
50
51 # Analyze the Python code where floating-point comparison fails.
52 # 1. Identify the logical issue.
53 # 2. Explain floating-point precision error.
54 # 3. Fix the code using tolerance or math methods.
55 # 4. Show the corrected output.
56
57 # Code:
58 # def check_sum():
59 #     return (0.1 + 0.2) == 0.3
60
61 # print(check_sum())
62 import math
63 def check_sum():
64     return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9)
65 print(check_sum()) # Output: True
66 # Corrected Output:
67 # True
68
```

## Explanation / Justification:

Floating-point numbers are stored in binary format, which cannot represent some decimal values exactly. As a result, operations like  $0.1 + 0.2$  do not produce an exact  $0.3$ , leading to incorrect comparisons. This issue is resolved by comparing numbers using a tolerance value or by using functions like `math.isclose()` to handle precision errors safely.

## Task 3 (Recursion Error – Missing Base Case)

Task: Analyze given code where recursion runs infinitely due to

missing base case. Use AI to fix.

```
# Bug: No base case
```

```
def countdown(n):
```

```
    print(n)
```

```
    return countdown(n-1)
```

```
countdown(5)
```

Expected Output : Correct recursion with stopping condition.

## Code:

```
70    """
71 #Task 3
72 You are an AI debugging assistant.
73
74 Analyze the recursive Python function that runs infinitely.
75 1. Identify the missing condition.
76 2. Explain why infinite recursion occurs.
77 3. Fix the function by adding a base case.
78 4. Show the corrected output.
79
80 Code:
81 def countdown(n):
82     print(n)
83     return countdown(n-1)
84
85 countdown(5)"""
86 def countdown(n):
87     if n <= 0: # Base case to stop recursion
88         print("Countdown finished!")
89         return
90     print(n)
91     return countdown(n-1)
92 countdown(5) # Output: 5, 4, 3, 2, 1, Countdown finished!
93
94
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\aniru\OneDrive\AI\_Assistant\_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI\_Assistant\_Coding/lab7.5.py

Countdown finished!

● PS C:\Users\aniru\OneDrive\AI\_Assistant\_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI\_Assistant\_Coding/lab7.5.py

5

4

3

2

1

Countdown finished!

## Explanation / Justification:

Recursion requires a base case to stop further function calls. In the given code, the function keeps calling itself without any stopping condition, leading to infinite recursion and a runtime error. Adding a proper base case ensures that the recursion terminates when a specific condition is met.

## Task 4 (Dictionary Key Error) :

Task: Analyze given code where a missing dictionary key causes

error. Use AI to fix it.

```
# Bug: Accessing non-existing key
```

```
def get_value():
```

```
data = {"a": 1, "b": 2}
```

```
return data["c"]
```

```
print(get_value())
```

Expected Output: Corrected with .get() or error handling

### Code :

```
93  '''You are an AI debugging assistant.
94
95
96 Analyze the Python code where accessing a dictionary key causes an error.
97 1. Identify the runtime error.
98 2. Explain why the error occurs.
99 3. Fix the code using safe dictionary access or exception handling.
100 4. Show the corrected output.
101
102 Code:
103 def get_value():
104     data = {"a": 1, "b": 2}
105     return data["c"]
106
107 print(get_value())
108 def get_value():
109     data = {"a": 1, "b": 2}
110     return data.get("c", "Key not found") # Safe access with default message
111 print(get_value()) # Output: Key not found
112 # Corrected Output:
113 # Key not found
114
115
```

The screenshot shows a code editor interface with a dark theme. At the top, there are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. Below the tabs, the terminal window displays the command: PS C:\Users\aniru\OneDrive\AI\_Assitant\_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI\_Assistant\_Coding/la. It also shows two items in the problem list: a red circle next to 'Key not found' and a blue circle next to 'PS C:\Users\aniru\OneDrive\AI\_Assitant\_Coding>'.

### Explanation / Justification:

Accessing a key that does not exist in a dictionary raises a `KeyError`. In the given code, the key "c" is not present in the dictionary. This issue is fixed by using the `.get()` method or exception handling, which allows safe access to dictionary values without causing runtime errors.

### Task 5 (Infinite Loop – Wrong Condition)

Task: Analyze given code where loop never ends. Use AI to detect and fix it.

```
# Bug: Infinite loop
```

```
def loop_example():
```

```
i = 0
```

```
while i < 5:
```

```
    print(i)
```

Expected Output: Corrected loop increments i.

```
14
15      #Task 5
16      '''You are an AI debugging assistant.
17
18      Analyze the Python code that results in an infinite loop.
19      1. Identify why the loop never ends.
20      2. Explain the logic error.
21      3. Fix the loop condition.
22      4. Show the corrected output.
23
24  Code:
25  def loop_example():
26      i = 0
27      while i < 5:
28          print(i)'''
29  def loop_example():
30      i = 0
31      while i < 5:
32          print(i)
33          i += 1 # Increment i to avoid infinite loop
34  loop_example() # Output: 0, 1, 2, 3, 4
35  # Corrected Output:
36
37
38
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

```
>S C:\Users\aniru\OneDrive\AI_Assistant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_A
0
1
2
3
4
>S C:\Users\aniru\OneDrive\AI_Assistant_Coding>
```

### Explanation / Justification:

An infinite loop occurs when the loop condition never becomes false. In this case, the loop variable `i` is not updated inside the loop, so the condition `i < 5` remains true forever.

Incrementing the loop variable ensures that the condition eventually becomes false and the loop terminates correctly.

## Task 6 (Unpacking Error – Wrong Variables)

Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

# Bug: Wrong unpacking

```
a, b = (1, 2, 3)
```

Expected Output: Correct unpacking or using \_ for extra values.

### Code

```
130
137 #Task 6
138 '''You are an AI debugging assistant.
139
140 Analyze the Python code where tuple unpacking fails.
141 1. Identify the error type.
142 2. Explain why unpacking fails.
143 3. Fix the code using correct unpacking.
144 4. Show the corrected output.
145
146 Code:
147 a, b = (1, 2, 3)
148 '''
149 a, b, c = (1, 2, 3) # Correct unpacking with three variables
150 print(a, b, c) # Output: 1, 2, 3
151 # Corrected Output:
152 # 1 2 3
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

```
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users
```

```
1 2 3
```

```
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

### Explanation / Justification:

Tuple unpacking requires the number of variables on the left-hand side to match the number of elements in the tuple. In the given code, three values are assigned to only two variables, causing a ValueError. This can be fixed by using the correct number of variables or by ignoring extra values using \_ or the \* operator.

## Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.

```
# Bug: Mixed indentation
```

```
def func():
```

```
    x = 5
```

```
    y = 10
```

```
    return x+y
```

Expected Output : Consistent indentation applied.

**Code**

```
155 #Task 07
156 '''You are an AI debugging assistant.
157
158 Analyze the Python code with mixed indentation.
159 1. Identify the indentation issue.
160 2. Explain why Python throws an error.
161 3. Fix the code using consistent indentation.
162 4. Show the corrected output.
163
164 Code:
165 def func():
166     x = 5
167     |   y = 10
168     return x + y
169     ''
170 def func():
171     x = 5
172     y = 10  # Fixed indentation to be consistent with the rest of the function
173     return x + y
174 print(func())  # Output: 15
175 # Corrected output:
176 # 15
177'''
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

```
PS C:\Users\aniru\OneDrive\AI_Assistant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneD
15
PS C:\Users\aniru\OneDrive\AI_Assistant_Coding>
```

### Explanation / Justification:

Python uses indentation to define code blocks. Mixing tabs and spaces or incorrect indentation leads to syntax or indentation errors. In the given code, inconsistent indentation breaks execution. Using a consistent indentation style (preferably 4 spaces) ensures proper block structure and avoids indentation-related errors.

### Task 8 (Import Error – Wrong Module Usage)

Task: Analyze given code with incorrect import. Use AI to fix.

```
# Bug: Wrong import
```

```
import maths  
print(maths.sqrt(16))
```

Expected Output: Corrected to import math

## Code

The screenshot shows a code editor interface with a dark theme. At the top, there is a toolbar with a 'Breakpoint' button and other icons. Below the toolbar is a code editor area containing the following Python code:

```
Click to add a breakpoint  
177 # Task 08  
178 # You are an AI debugging assistant.  
179  
180 # Analyze the Python code with an incorrect module import.  
181 # 1. Identify the import error.  
182 # 2. Explain why the module is not found.  
183 # 3. Fix the import statement.  
184 # 4. Show the corrected output.  
185  
186 # Code:  
187 # import maths  
188 # print(maths.sqrt(16))  
189 import math # Corrected module name  
190 print(math.sqrt(16)) # Output: 4.0  
191 # Corrected Output:  
192 # 4.0  
193
```

Below the code editor, there is a navigation bar with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. The TERMINAL tab shows the following terminal output:

```
PS C:\Users\aniru\OneDrive\AI_Assistant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assistant_Coding.py  
4.0  
PS C:\Users\aniru\OneDrive\AI_Assistant_Coding>
```

## Explanation / Justification:

Python raises an `ImportError` when a module that does not exist or is incorrectly named is imported. In the given code, `maths` is not a valid standard Python module. Correcting the module name to `math` resolves the issue and allows access to mathematical functions like `sqrt()`.

-- THANK YOU --