

# AI Assisted Coding

## Assignment 7.5

Name: MD SHAHID

Ht.no: 2303A51421

Batch: 21

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.

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

**Task:** Analyze given code where a mutable default argument cause 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:

```

AIAC Assignment-7.5.py
C:\Users\chaitanya> AIAC Assignment-7.5.py ...
def add_item(item, items=[]):
    items.append(item)
    return items
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
print(add_item(3, [4, 5]))
```

(1) Python mutable default argu...  
 def add\_item(item, items=[]):  
 items.append(item)  
 return items  
 ...

def add\_item(item, items=None):  
 items.append(item)  
 return items  
 ...

AIAC Assignment-7.5.py +7 -3  
 def add\_item(item, items=[]):  
 items.append(item)  
 return items  
 def add\_item(item, items=None):  
 if items is None:  
 ...

I've updated the file c:\Users\chaitanya\AIAC Assignment-7.5.py with the corrected add\_item function.  
 The updated code now looks like this:

```

def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1))
...
```

### Output:

```

[1]
[2]
[4, 5, 3]
PS C:\Program Files\cursor>
```

To avoid this, a common practice is to use None as the default value and then create a new list inside the function if items is No

## **Observation:**

The function safely handles mutable defaults by using None and creating a new list only when needed, ensuring each call uses an independent instance and preventing unintended shared state

## **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:**

The screenshot shows a code editor window with two tabs: 'AIAC Assignment-7.5.py' and 'Browser Tab'. The 'AIAC Assignment-7.5.py' tab contains the following Python code:`1 # Bug: Floating point precision issue
2 def check_sum():
3 return (0.1 + 0.2) == 0.3
4 print(check_sum())`The code is annotated with AI-generated comments:

- (1) Python mutable default argu...  
# Bug: Floating point precision issue  
def check\_sum():  
 return (0.1 + 0.2) == 0.3  
print(check\_sum())
- will likely print False.  
To correctly compare floating-point numbers, you should generally check if their absolute difference is within a small tolerance (epsilon). Here's an improved version:  

```
import math
# Corrected: Handling floating point
def check_sum_corrected():
    # Using math.isclose for robust c
    return math.isclose(0.1 + 0.2, 0,
print(check_sum_corrected())
```

This version uses `math.isclose()` which is designed for this exact purpose and will correctly print True. I've updated your file.

## Output:

```
False
PS C:\Program Files\cursor> []
```

To correctly compare floating-point numbers, you should generally check if their absolute difference is within a small tolerance (epsilon). Here's an improved version:

## Observation:

The expression `(0.1 + 0.2) == 0.3` typically evaluates to `False` because floating-point numbers cannot be represented exactly in binary. Values like `0.1` and `0.2` are stored as approximations, so their sum is slightly off from `0.3`, causing a direct equality check to fail. The correct approach is to compare floats using a small tolerance, such as with `math.isclose()`.

## 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:

```
C:\> Users > chaitanya > AIAC Assignment-7.5.py > ...
^ 3 of 3 | Undo File Keep File Ctrl+D
7 # Corrected: Added base case for recursion
8 def countdown_corrected(n):
9     if n <= 0: # Base case: stop when n is 0 or less
10        print("Done!")
11        return
12    print(n)
13    countdown_corrected(n-1)
14
15 countdown_corrected(5) ^ 3 of 3 | Undo Ctrl+N Keep Ctrl+Shift+Y
```

**(1) Python mutable default argu...**  
Now +15 -5 · gemini-2.5-flash

```
# Bug: No base case
def countdown(n):
    print(n)
    return countdown(n)
error.
```

I will now update your file c:\Users\chaitanya\AIAC Assignment-7.5.py with this corrected code.

**AIAC Assignment-7.5.py +15 -4**

```
# Bug: Floating point precision issue
def check_sum():
    return (0.1 + 0.2) == 0.3
print(check_sum())
# Bug: No base case
```

I've added the corrected countdown\_corrected function to c:\Users\chaitanya\AIAC Assignment-7.5.py, along with the original for context.

## Output:

```
5
4
3
2
1
Done!
PS C:\Program Files\cursor> █
```

## Observation:

The recursion error occurred because the function lacked a base case, causing infinite recursive calls and eventually a stack overflow. Adding a stopping condition that terminates when the value reaches zero fixed the issue and ensured the function executes correctly. This emphasizes the necessity of defining a base case in every recursive function.

## 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:

```
43
44  # Task 4 (Dictionary Key Error)
45  ...
46  Task: Analyze given code where a missing dictionary key causes
47  error. Use AI to fix it.
48  # Bug: Accessing non-existing key
● 49  Expected Output: Corrected with .get() or error handling.
50  ...
51  def get_value():
52      data = {"a": 1, "b": 2}
53 → |     return data["c"]
54      return data.get("c", "Key not found")
55
56  print(get_value())
```

## Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Key not found
```

## Observation:

The program attempted to access a non-existent dictionary key, resulting in a `KeyError`. Replacing direct key access with `.get()` or adding proper error handling prevented the crash and made the code more robust. This reinforces the need to validate keys before accessing them..

## 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.

### Code:

```
57 ...
58 Task 5 (Infinite Loop □ Wrong Condition)
59 Task: Analyze given code where loop never ends. Use AI to detect
60 and fix it.
61 # Bug: Infinite loop
62 Expected Output: Corrected loop increments i.
63 ...
64 def loop_example():
65     i = 0
66     while i < 5:
67 →| | | print(i) | | print(i)
    | | | i += 1
```

### Output:

```
C:\Users\acer\Desktop\Btech_3_2\AI Assistant coding>python
0
1
2
3
4
```

### Observation:

The infinite loop happened because the control variable i was never incremented inside the while loop, so the termination condition was never reached. Adding  $i += 1$  allowed the loop to progress and exit correctly, preventing non-terminating execution.

## 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:**

```
71  # Task 6 (Unpacking Error - Wrong Variables)
72  ...
73  Task: Analyze given code where tuple unpacking fails. Use AI to
74  fix it.
75  # Bug: Wrong unpacking
76  Expected Output: Correct unpacking or using _ for extra values.
77  ...
78 →| a, b = (1, 2, 3)
    a, b, _ = (1, 2, 3)
```

**Output:**

```
C:\Users\acer\Desktop\Btech_3_2\AI Assistant coding>python
1 2
```

## Observation:

The unpacking error occurred because the number of variables did not match the number of elements in the tuple. Matching the variable count or using placeholders like `_` for unused values resolved the issue and ensured correct unpacking.

## 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():

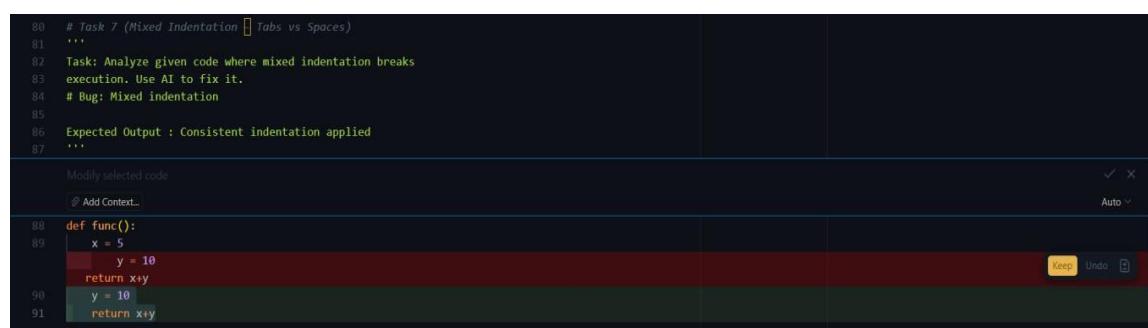
x = 5 y =

10

return x+y

Expected Output : Consistent indentation applied.

## Code:



```
80 # Task 7 (Mixed Indentation [ Tabs vs Spaces )
81 ...
82 Task: Analyze given code where mixed indentation breaks
83 execution. Use AI to fix it.
84 # Bug: Mixed indentation
85
86 Expected Output : Consistent indentation applied
87 ...

Modify selected code
 Add Context...
88 def func():
89     x = 5
90         y = 10
91     return x+y
92     y = 10
93     return x+y
```

## Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python  
15
```

## Observation:

The error was caused by mixing tabs and spaces, which led to an `IndentationError`. Reformattting the code with consistent indentation—preferably four spaces—resolved the issue and ensured proper execution.

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

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

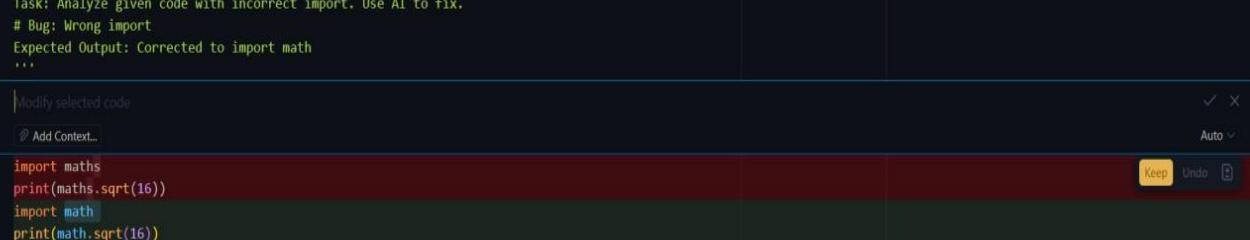
# Bug: Wrong import import

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

Expected Output: Corrected to import math

**Code:**

```
93 # Task 8 (Import Error - Wrong Module Usage)  
94 ...  
95 Task: Analyze given code with incorrect import. Use AI to fix.  
96 # Bug: Wrong import  
97 Expected Output: Corrected to import math  
98 ...
```



## Output:

```
C:\Users\acer\Desktop\Btech_3_2\AI Assistant coding>python  
4.0
```

## **Observation:**

The program tried to import a non-existent module maths, which raised a ModuleNotFoundError. Correcting the import to the built-in math module resolved the issue and allowed math.sqrt(16) to run successfully.

## **Observation:**

- This lab involved identifying and fixing various Python errors, including logical, runtime, syntax, recursion, and import issues. The root causes were analyzed, such as mutable default arguments, floating-point precision limitations, and missing base cases in recursion. Corrective practices included tolerant float comparisons, safe dictionary access with .get(), proper loop control, and accurate tuple unpacking..

- Overall, the lab improved our systematic debugging approach and increased confidence in using AI for structured and efficient error correction.