

# Lab Assignment-7.5

## Error Debugging with AI: Systematic approaches to finding and fixing bugs

Mohammed Awez Ali

2303A51767

Batch-12

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

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
1 # Bug: Mutable default argument
2 def add_item(item, items=[]):
3     items.append(item)
4     return items
5 print(add_item(1))
6 print(add_item(2))
7 print(add_item(3))
8 print(add_item(4))
9 print(add_item(5))
10 print(add_item(6))
11 print(add_item(7))
12 print(add_item(8))
13 print(add_item(9))
14 print(add_item(10))
```

**Output:**

```
Problems  Output  Debug Console  Terminal  Ports
PS C:\Users\Awez_\OneDrive\Desktop\Ai coding> & C:/Users/Awez_/AppData/Local/OneDrive/Desktop/Ai coding/lab1/lab1/lab 7.5.py"
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### Explanation:

Default items=[] is created once and shared by all calls, so the list keeps growing. Use items=None and set items = [] if items is None else items inside the function.

## Task 2 (Floating-Point Precision Error)

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
# Task 2(Floating-Point Precision Error)
# Bug: Floating point precision issue
def check_sum(a, b):
    return a + b == 0.3
print(check_sum(0.1, 0.2))
print(check_sum(0.2, 0.1))
print(check_sum(0.3, 0.4))
print(check_sum(0.4, 0.3))
print(check_sum(0.5, 0.5))
print(check_sum(0.6, 0.4))
print(check_sum(0.7, 0.3))
print(check_sum(0.8, 0.2))
print(check_sum(0.9, 0.1))
print(check_sum(1.0, 0.0))
```

**Output:**

```
PS C:\Users\Awez_\OneDrive\Desktop\Ai coding> python -u "c:\Users\Awez_\O
False
False
False
False
False
False
False
False
False
False
False
```

**Explanation:**

Floats like 0.1 and 0.2 can't be stored exactly, so  $0.1 + 0.2 == 0.3$  is often False. Use `math.isclose()` or a small tolerance instead of `==`.

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

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#task 3(Recursion Error – Missing Base Case)
# Bug: No base case
def countdown(n):
    if n == 0:
        return
    print(n)
    countdown(n-1)
countdown(5)
```

## Output:

```
PS C:\Users\Awez_\OneDrive\Desktop\Ai coding> python -u "c:  
5  
4  
3  
2  
1
```

## Explanation:

Recursion must stop at some point. Without a base case (e.g. if  $n == 0$ : return), the function keeps calling itself and can hit a stack overflow.

## Task 4 (Dictionary Key Error)

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#Task 4 (Dictionary Key Error)  
# Bug: Accessing non-existing key  
def get_value():  
    data = {"a": 1, "b": 2}  
    return data.get("c", "Key not found")  
print(get_value())
```

## Output:

```
PS C:\Users\Awez_\OneDrive\Desktop\Ai coding> python -u "c:  
Key not found
```

## Explanation:

Using `data["c"]` when "c" isn't in the dict raises `KeyError`. Use `data.get("c", "Key not found")` to return a default instead.

## Task 5 (Infinite Loop – Wrong Condition)

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#Task 5 (Infinite Loop – Wrong Condition)
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1
loop_example()
```

**Output:**

```
PS C:\Users\Awez_\OneDrive\Desk
0
1
2
3
4
```

**Explanation:**

if the loop condition never becomes false or the loop variable never changes, the loop never ends. Make sure the variable updates (e.g. `i += 1`) and the condition can eventually fail.

## Task 6 (Unpacking Error – Wrong Variables)

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#Task 6 (Unpacking Error – Wrong Variables)
# Bug: Wrong unpacking
a, b, c = (1, 2, 3)
print(a)
print(b)
```

### Output:

```
PS C:\Users\A
1
2
```

### Explanation:

You must have the same number of variables as values. `a, b = (1, 2, 3)` fails because there are 3 values. Use `a, b, c = (1, 2, 3)` or `a, b, *_ = (1, 2, 3)`.

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

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#Task 7 (Mixed Indentation – Tabs vs Spaces)
# Bug: Mixed indentation
def func():
    x = 5
    y = 10
    return x+y
print(func())
```

### Output:

```
PS C:\Users\A
15
```

### Explanation:

Don't mix tabs and spaces in the same file; Python will raise `TabError`. Use only spaces (e.g. 4 spaces) for indentation.

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

**Prompt:** Analyse the given code and fix it

**Code without errors:**

```
#Task 8 (Import Error – Wrong Module Usage)
# Bug: Wrong import
import math
print(math.sqrt(16))
```

**Output:**

```
PS C:\Users\A
4.0
```

**Explanation:**

Use the correct module and name. `import math` and then `math.sqrt(16)` is correct; calling a name that doesn't exist in the module (e.g. `math.squareroot`) causes an error.