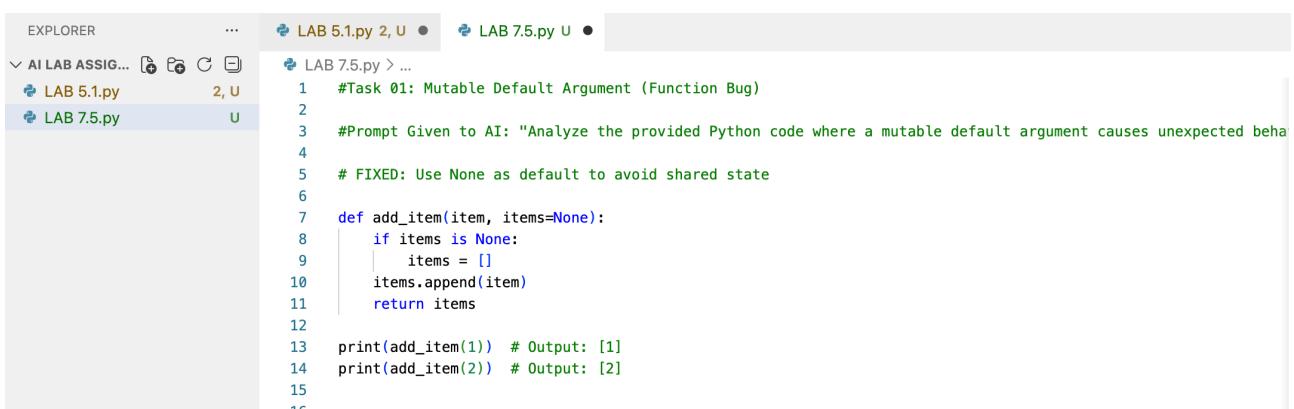


## Student Details :

- **Name:** Telu Sai Suraj
- **Roll Number:** 2403A54L01
- **Branch & Year:** B.Tech, 3rd Year
- **College:** SR University
- **Assignment Number:** Lab Assignment 7.5

### Task 01: Mutable Default Argument (Function Bug)

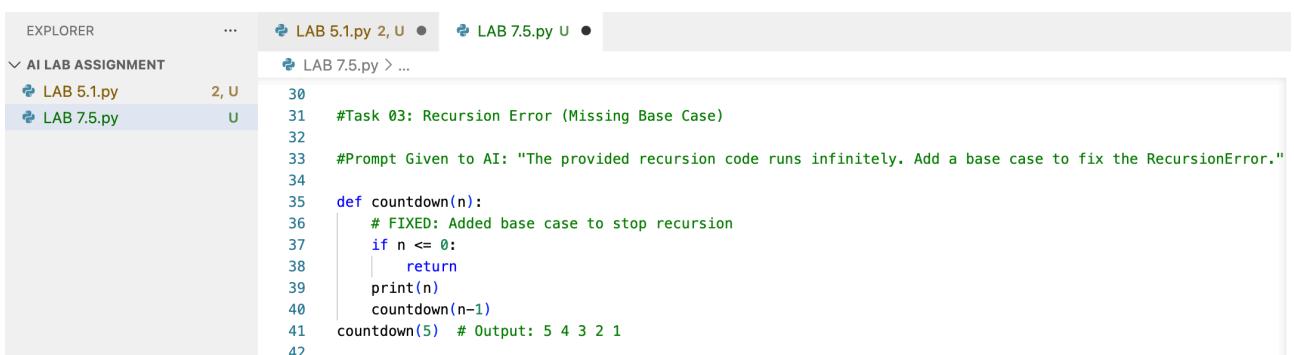


The screenshot shows a code editor interface with two files open: LAB 5.1.py and LAB 7.5.py. The LAB 7.5.py file is the active tab and contains the following code:

```
#Task 01: Mutable Default Argument (Function Bug)
#Prompt Given to AI: "Analyze the provided Python code where a mutable default argument causes unexpected behavior"
# FIXED: Use None as default to avoid shared state
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1)) # Output: [1]
print(add_item(2)) # Output: [2]
```

**Explanation of Code:** The original code used `items=[ ]` as a default argument. In Python, default arguments are created only once when the function is defined, not every time it is called. This meant the list was shared across all function calls, leading to data persisting between calls. The fix sets the default to `None` and creates a new empty list inside the function, ensuring each call gets its own fresh list.

### Task 02: Floating-Point Precision Error



The screenshot shows a code editor interface with two files open: LAB 5.1.py and LAB 7.5.py. The LAB 7.5.py file is the active tab and contains the following code:

```
#Task 03: Recursion Error (Missing Base Case)
#Prompt Given to AI: "The provided recursion code runs infinitely. Add a base case to fix the RecursionError."
def countdown(n):
    # FIXED: Added base case to stop recursion
    if n <= 0:
        return
    print(n)
    countdown(n-1)
countdown(5) # Output: 5 4 3 2 1
```

**Explanation of Code:** Computers store floating-point numbers as binary approximations, so `0.1 + 0.2` actually results in `0.3000000000000004`, which is not exactly equal to `0.3`. The corrected code uses `math.isclose()`, which compares the values with a small tolerance to handle these minor precision differences correctly.

### Task 03: Recursion Error (Missing Base Case)

```
EXPLORER ... LAB 5.1.py 2, U ● LAB 7.5.py U ●
AI LAB ASSIG... 📁 🗑 C ⚙️
LAB 5.1.py 2, U
LAB 7.5.py U

LAB 7.5.py > ...
17 #Task 02: Floating-Point Precision Error
18
19 #Prompt Given to AI: "Fix the floating-point comparison error in this function using a tolerance method."
20
21 import math
22
23 def check_sum():
24     # FIXED: Use math.isclose for safe floating-point comparison
25     # Direct comparison (0.1 + 0.2 == 0.3) fails due to precision issues
26     return math.isclose(0.1 + 0.2, 0.3)
27
28 print(check_sum()) # Output: True
29
30
```

**Explanation of Code:** The original function lacked a "base case," meaning it would call `countdown(n-1)` forever until the program crashed with a `RecursionError`. The fix adds a condition `if n <= 0: return`, which tells the function when to stop calling itself, ensuring the program terminates safely.

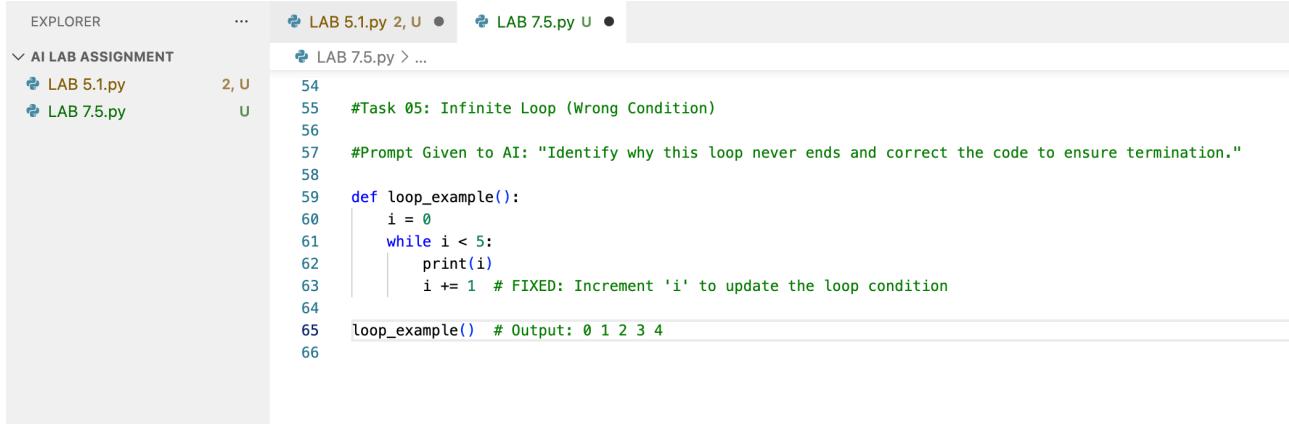
### Task 04: Dictionary Key Error

```
EXPLORER ... LAB 5.1.py 2, U ● LAB 7.5.py U ●
AI LAB ASSIGNMENT 📁 🗑
LAB 5.1.py 2, U
LAB 7.5.py U

LAB 7.5.py > ...
43 #Task 04: Dictionary Key Error
44
45 #Prompt Given to AI: "Fix the KeyError caused by accessing a non-existent dictionary key. Use a safe retrieval"
46
47 def get_value():
48     data = {"a": 1, "b": 2}
49     # FIXED: Use .get() method to handle missing keys gracefully
50     return data.get("c", "Key not found")
51
52 print(get_value())
53
```

**Explanation of Code:** Trying to access `data[ "c" ]` directly causes a crash because the key "c" does not exist in the dictionary. The fixed code uses the `.get()` method, which safely returns `None` (or a custom default message like "Key not found") instead of crashing the program.

## Task 05: Infinite Loop (Wrong Condition)



The screenshot shows a code editor interface with the following details:

- EXPLORER**: Shows two files: LAB 5.1.py (2, U) and LAB 7.5.py (U).
- LAB 7.5.py** (selected tab):
  - Line 54: 54
  - Line 55: 55 #Task 05: Infinite Loop (Wrong Condition)
  - Line 56: 56
  - Line 57: 57 #Prompt Given to AI: "Identify why this loop never ends and correct the code to ensure termination."
  - Line 58: 58
  - Line 59: 59 def loop\_example():
  - Line 60: | i = 0
  - Line 61: | while i < 5:
  - Line 62: | | print(i)
  - Line 63: | | i += 1 # FIXED: Increment 'i' to update the loop condition
  - Line 64: 64
  - Line 65: 65 loop\_example() # Output: 0 1 2 3 4
  - Line 66: 66

**Explanation of Code:** The original loop was infinite because the variable `i` was never updated; it stayed at 0, so the condition `i < 5` was always true. The fix adds `i += 1` inside the loop, ensuring that `i` eventually reaches 5 and the loop terminates as expected.

Thank You....