

AI ASSISTED CODING

T . ANANYA

2303A51128

BATCH – 03

06 – 02 – 2026

ASSIGNMENT – 7.5

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

TASK - 01 : Mutable Default Argument – Function Bug

ERROR AND FIXED CODE:

The screenshot shows a Google Colab notebook titled "AIAC_1127_LAB_7.5.ipynb". The notebook is open to a cell titled "Task 1 : Mutable Default Argument – Function Bug". The cell contains two code blocks. The first block, labeled "Bug: Mutable default: argument", shows a function definition:

```
def add_item(item, items=[]):  
    items.append(item)  
    return items  
print(add_item(1))  
print(add_item(2))
```

 Below the code, the output is shown:

```
[1]  
[1, 2]
```

 The second block, labeled "Correct Fix", shows the corrected function definition:

```
def add_item(item, items=None):  
    if items is None:  
        items = []  
    items.append(item)  
    return items  
print(add_item(1)) # [1]  
print(add_item(2)) # [2]
```

 Below the code, the output is shown:

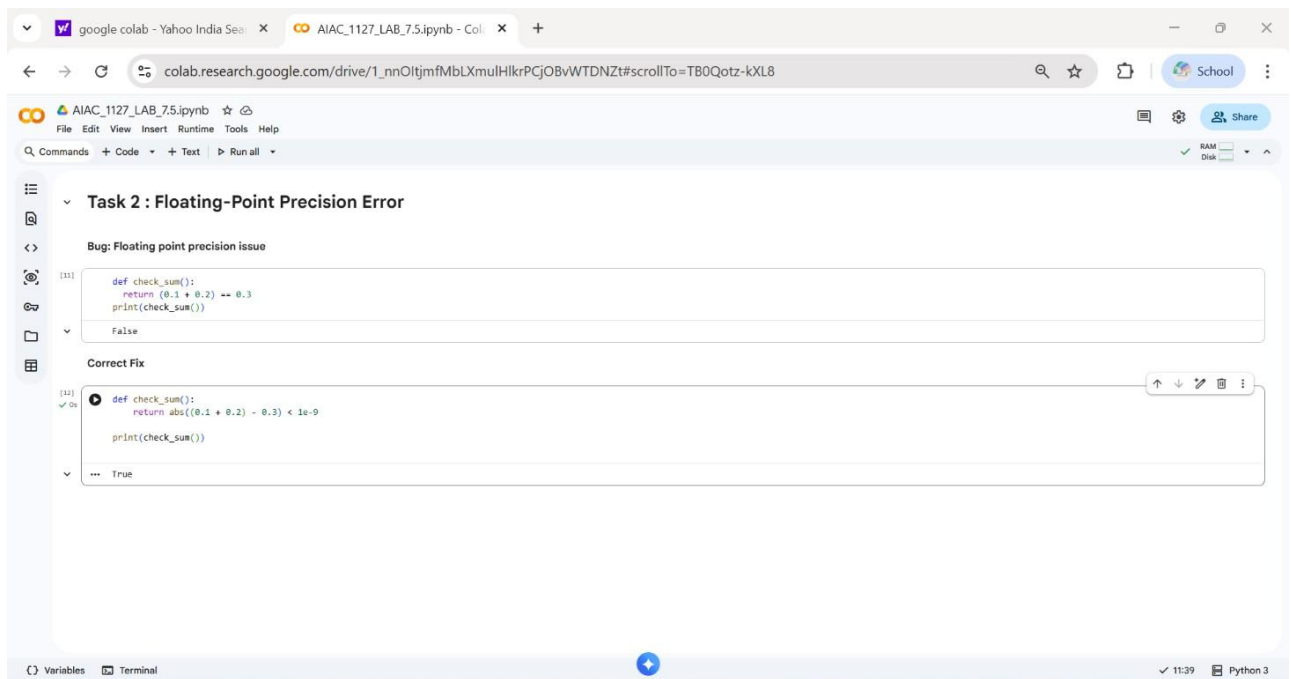
```
[1]  
[2]
```

 The notebook interface includes a menu bar with "File", "Edit", "View", "Insert", "Runtime", and "Tools". The status bar at the bottom shows "Variables", "Terminal", and "Python 3".

Explanation : Using None instead of a mutable default argument creates a new list on every function call and avoids shared data issues.

Task 2: Floating-Point Precision Error

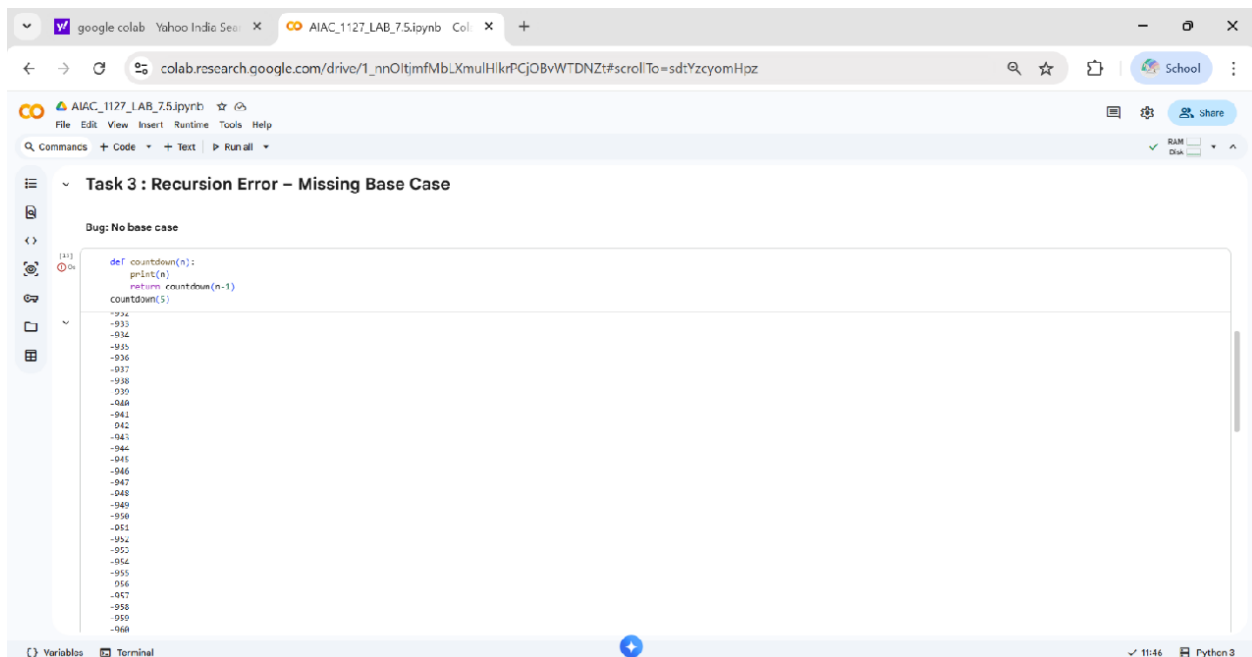
ERROR AND FIXED CODE:

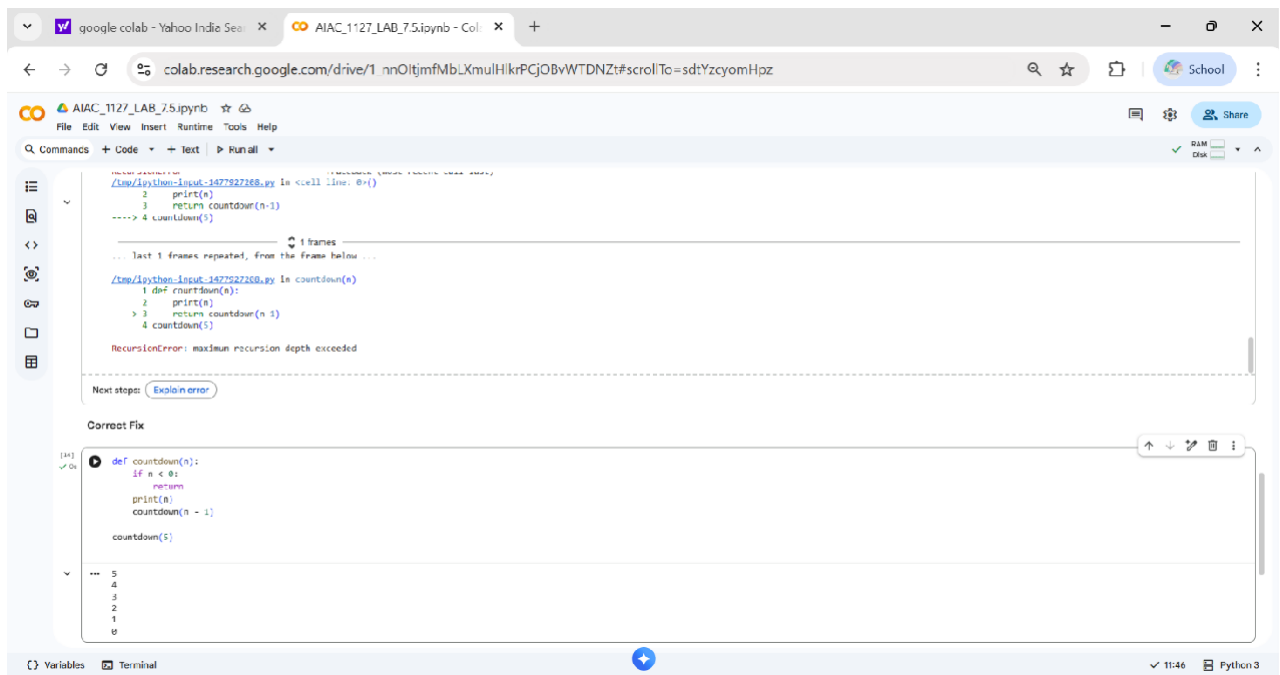


Explanation: Floating-point values are compared using a tolerance (or `math.isclose`) instead of direct equality to handle precision errors.

Task 3: Recursion Error – Missing Base Case

ERROR AND FIXED CODE :

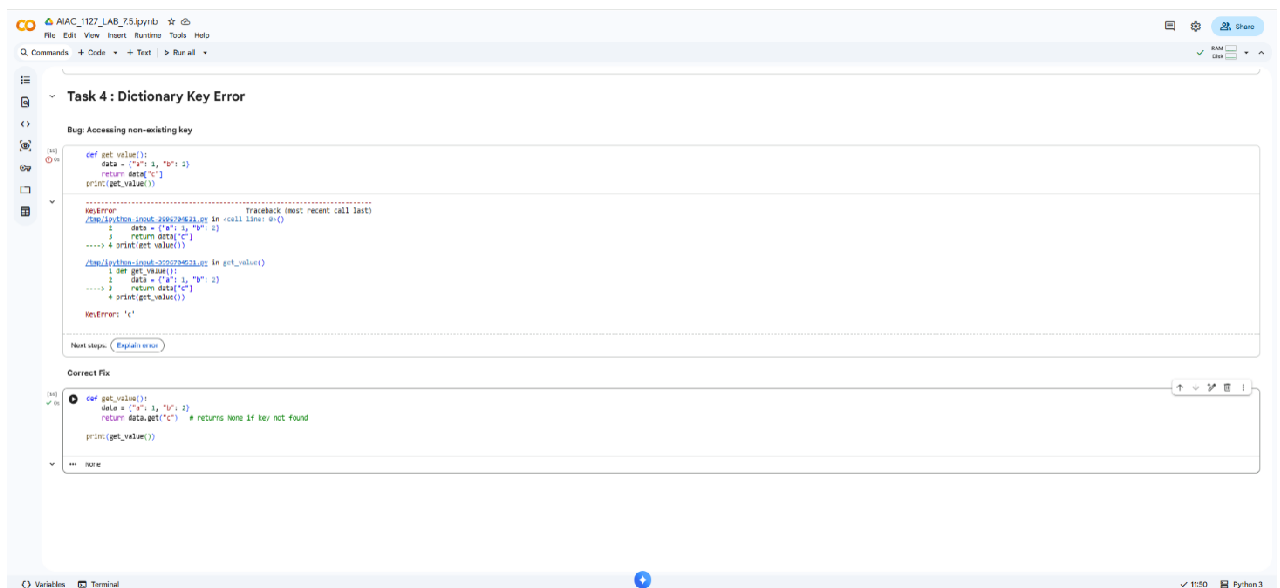




Explanation: A base case is added to stop recursive calls and prevent infinite recursion.

Task 4: Dictionary Key Error

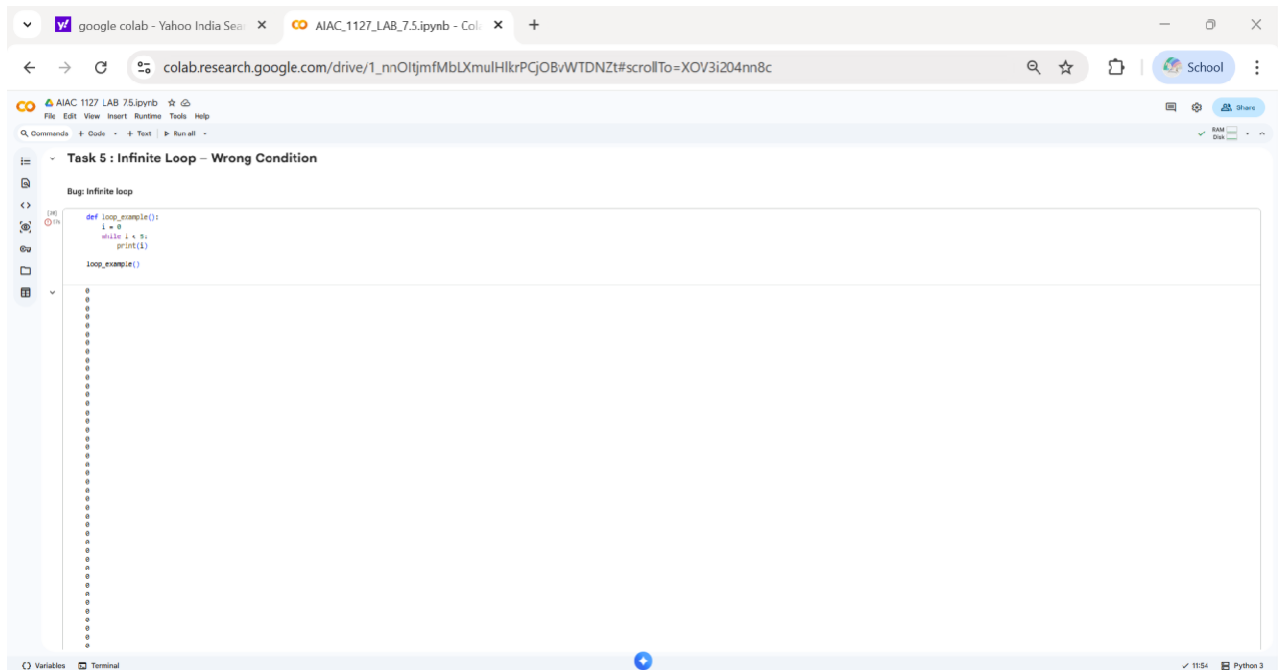
ERROR AND FIXED CODE:



Explanation: Using dict.get() or exception handling prevents KeyError when accessing missing dictionary keys.

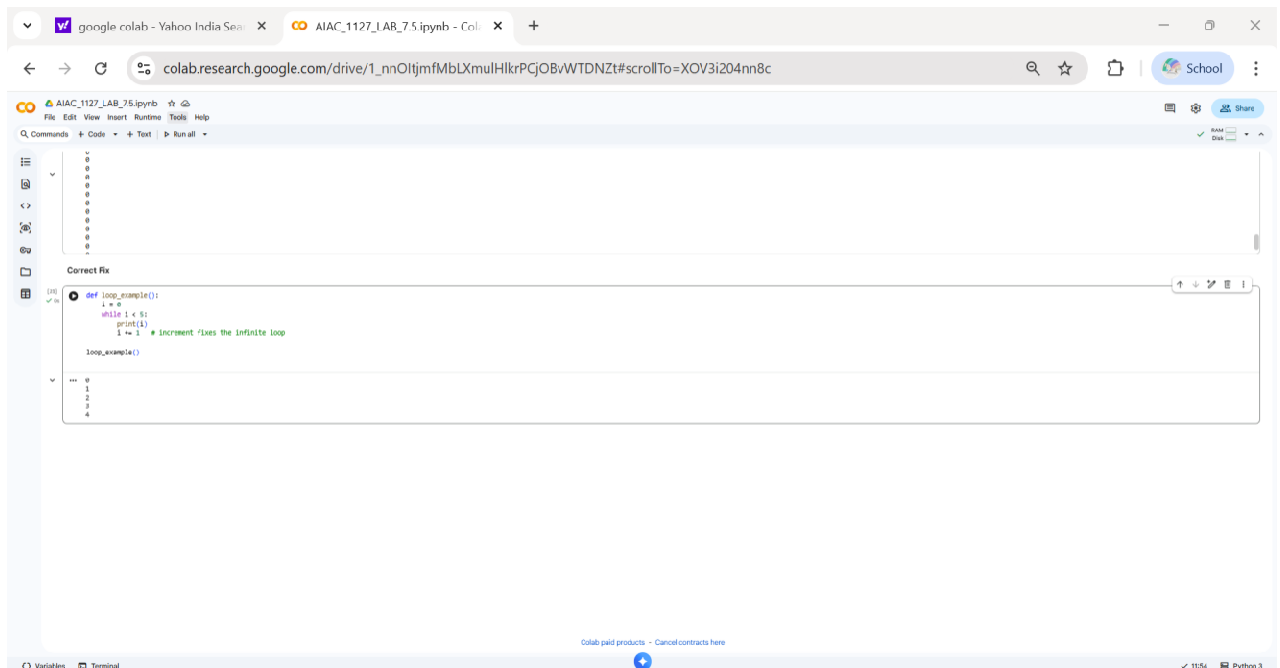
Task 5: Infinite Loop – Wrong Condition

ERROR AND FIXED CODE:



The screenshot shows a Google Colab notebook titled "Task 5: Infinite Loop – Wrong Condition". The code defines a function `loop_example()` with a `while` loop that has a constant condition `i < 5`. The loop prints the value of `i` and increments it by 5. This results in an infinite loop. The output shows a long list of zeros, indicating that the loop is not terminating. The error message "Bug: Infinite loop" is displayed at the top of the code cell.

```
def loop_example():  
    i = 0  
    while i < 5:  
        print(i)  
        i += 5  
    loop_example()  
  
loop_example()
```



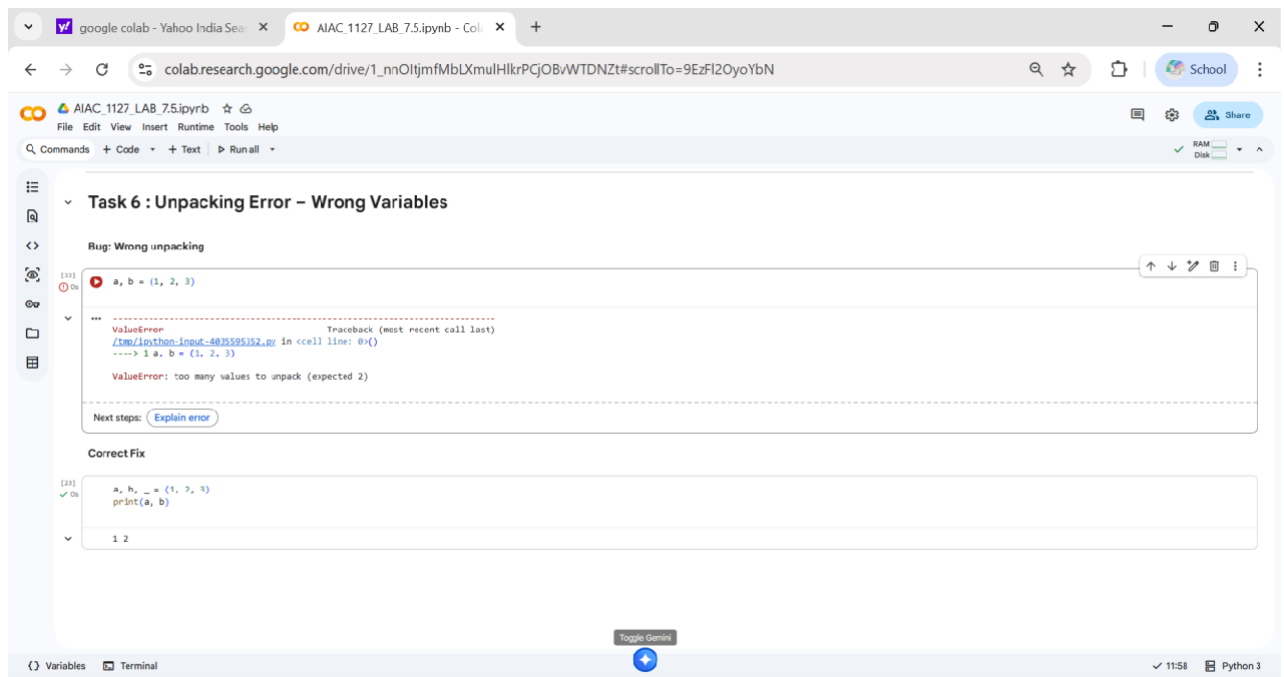
The screenshot shows the same Google Colab notebook, but with the code corrected. The `while` loop condition is now `i < 5`, and the increment is `i += 1`. The output shows the values 0, 1, 2, 3, and 4, indicating that the loop terminates correctly. The error message "Correct Fix" is displayed at the top of the code cell.

```
def loop_example():  
    i = 0  
    while i < 5:  
        print(i)  
        i += 1 # Increment fixes the infinite loop  
    loop_example()  
  
loop_example()
```

Explanation: Incrementing the loop variable ensures the loop condition eventually becomes false.

TASK 6: Unpacking Error – Wrong Variables

ERROR AND FIXED CODE:



The screenshot shows a Google Colab notebook titled "Task 6 : Unpacking Error – Wrong Variables". The notebook contains two code cells. The first cell, labeled "Bug: Wrong unpacking", contains the code `a, b = (1, 2, 3)`. This code has failed, resulting in a `ValueError: too many values to unpack (expected 2)`. The error message is displayed in a red box with a traceback. Below the error, there is a "Next steps: Explain error" button. The second cell, labeled "Correct Fix", contains the code `a, b, _ = (1, 2, 3)` followed by `print(a, b)`. This code has been executed successfully, and the output `1 2` is shown below the code.

Explanation: Correct unpacking is achieved by matching variable count or ignoring extra values using `_` or `*`.

Task 7: Mixed Indentation – Tabs vs Spaces

ERROR AND FIXED CODE:

```
def func():  
    x = 5  
    y = 10  
    return x + y
```

File: "/tmp/ipython-input-4064137757.py", line 2
x = 5
IndentationError: expected an indented block after function definition on line 1

Next steps: [Explain error](#)

Correct Fix

```
def func():  
    x = 5  
    y = 10  
    return x + y  
  
print(func())
```

15

EXPLANATION: Consistent indentation using spaces fixes IndentationError and allows proper code execution.

Task 8: Import Error – Wrong Module Usage

ERROR AND FIXED CODE: Correcting the module name to math resolves the import error.

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

ModuleNotFoundError: Traceback (most recent call last):
 File "/tmp/ipython-input-1532532258.py", in <cell line: 0>():
 1 import maths
 2 print(maths.sqrt(16))

ModuleNotFoundError: no module named 'maths'

NOTE: If your import is failing due to a missing package, you can manually install dependencies using either `!pip` or `!apt`.
To view examples of installing some common dependencies, click the "Open Examples" button below.

Next steps: [Explain error](#)

Correct Fix

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

4.0

Explanation: Correcting the module name to math resolves the import error.