Assignment-7.5

Name: Jillela Akshaya

Ht.no: 2303A51629

Batch. No: 22

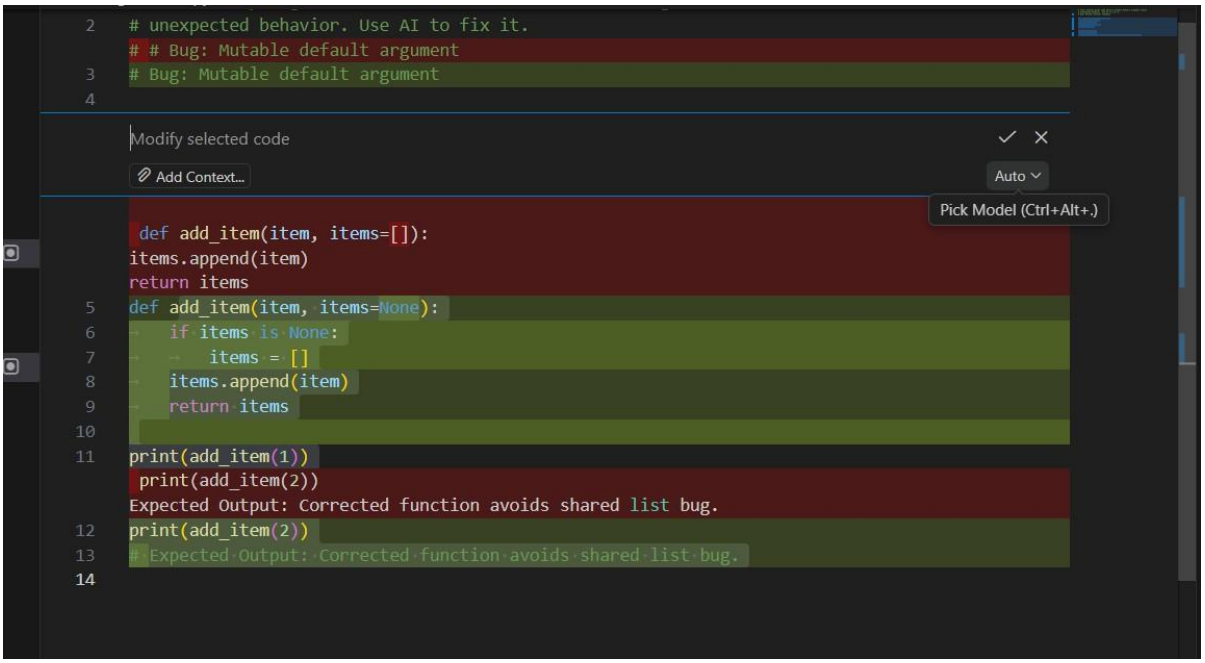
Task 1 (Mutable Default Argument – Function Bug)

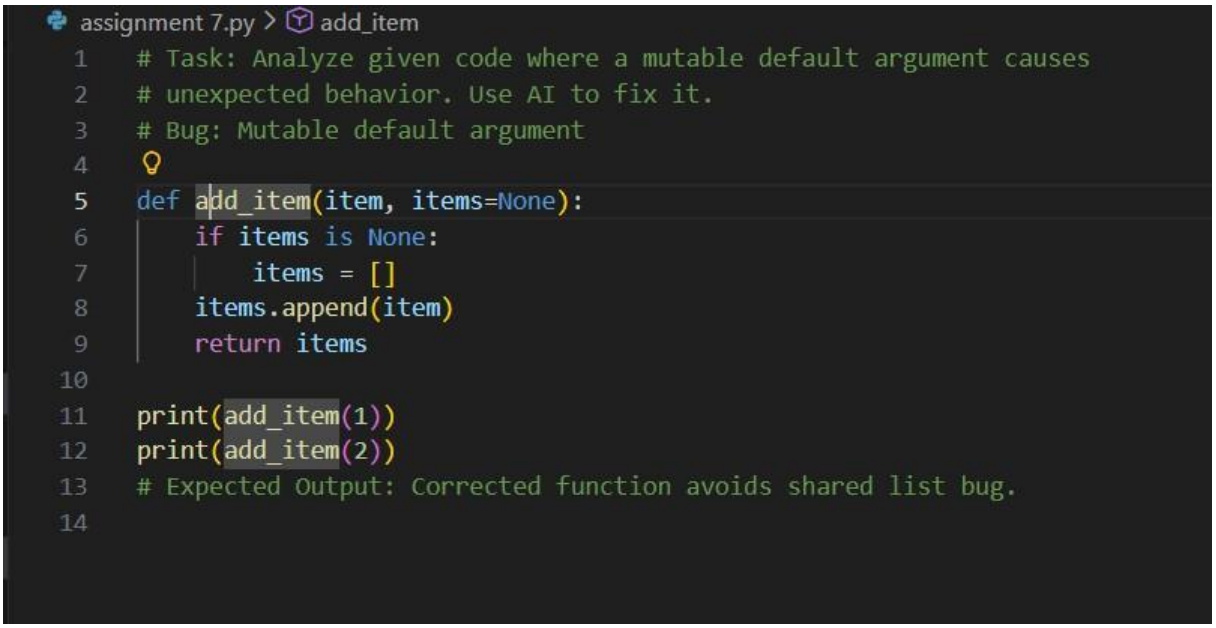
Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

```
# Bug: Mutable default argument
def add_item(item, items=[]):
items.append(item)
return items
print(add_item(1))
print(add_item(2))
```

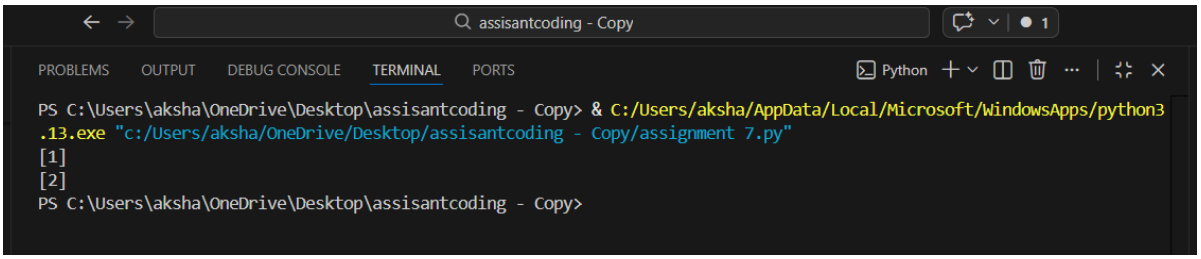
Expected Output: Corrected function avoids shared list bug.

Code:





Output:



Observation:

The default list items=[] is created only once, so all function calls share the same list.

Because the list is mutable, values added in earlier calls remain in later calls, causing unexpected accumulation.

Using items=None and creating a new list inside the function ensures each call works with a fresh list and fixes the bug.

Task 2 (Floating-Point Precision Error)

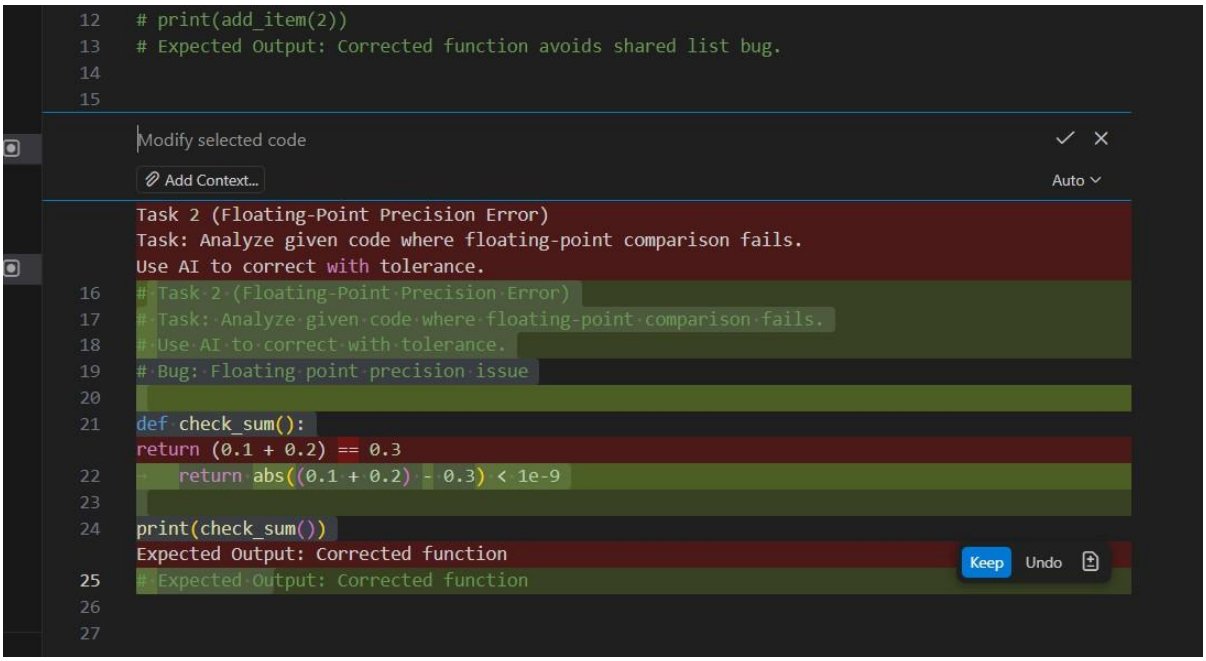
Task: Analyze given code where floating-point comparison fails.

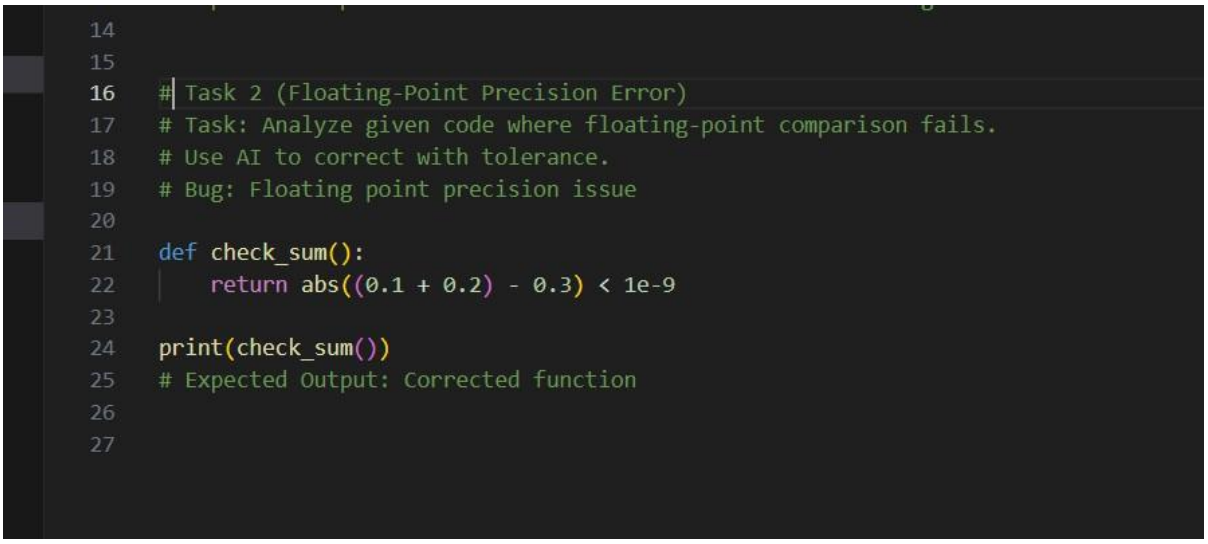
Use AI to correct with tolerance.

```
# Bug: Floating point precision issue
def check_sum():
return (0.1 + 0.2) == 0.3
print(check_sum())
```

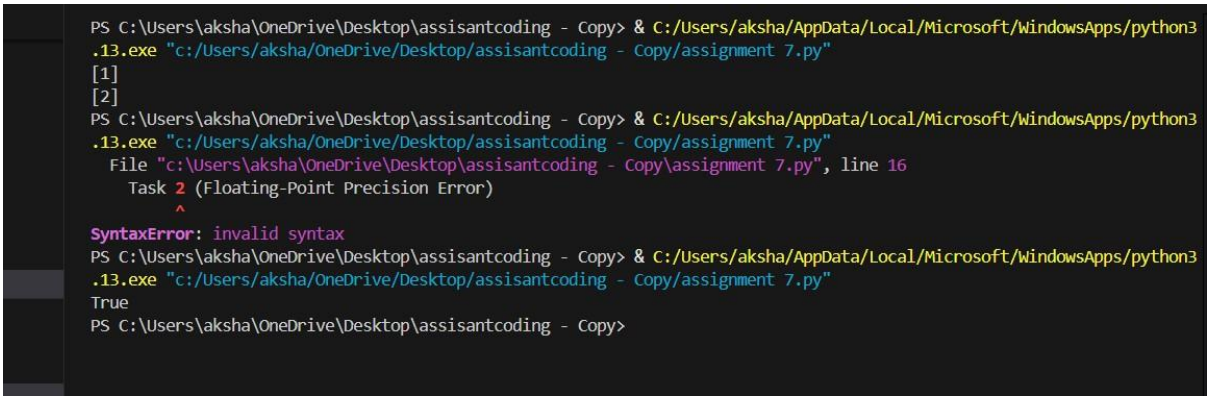
Expected Output: Corrected function

Code:





Output:



Observation:

Floating-point numbers cannot always represent decimal values exactly, so (0.1 + 0.2) is not exactly equal to 0.3.
Direct comparison using == may therefore return False even when the values should logically be equal.
Using a tolerance (epsilon) and checking whether the absolute difference is smaller than that tolerance correctly handles floating-point precision errors.

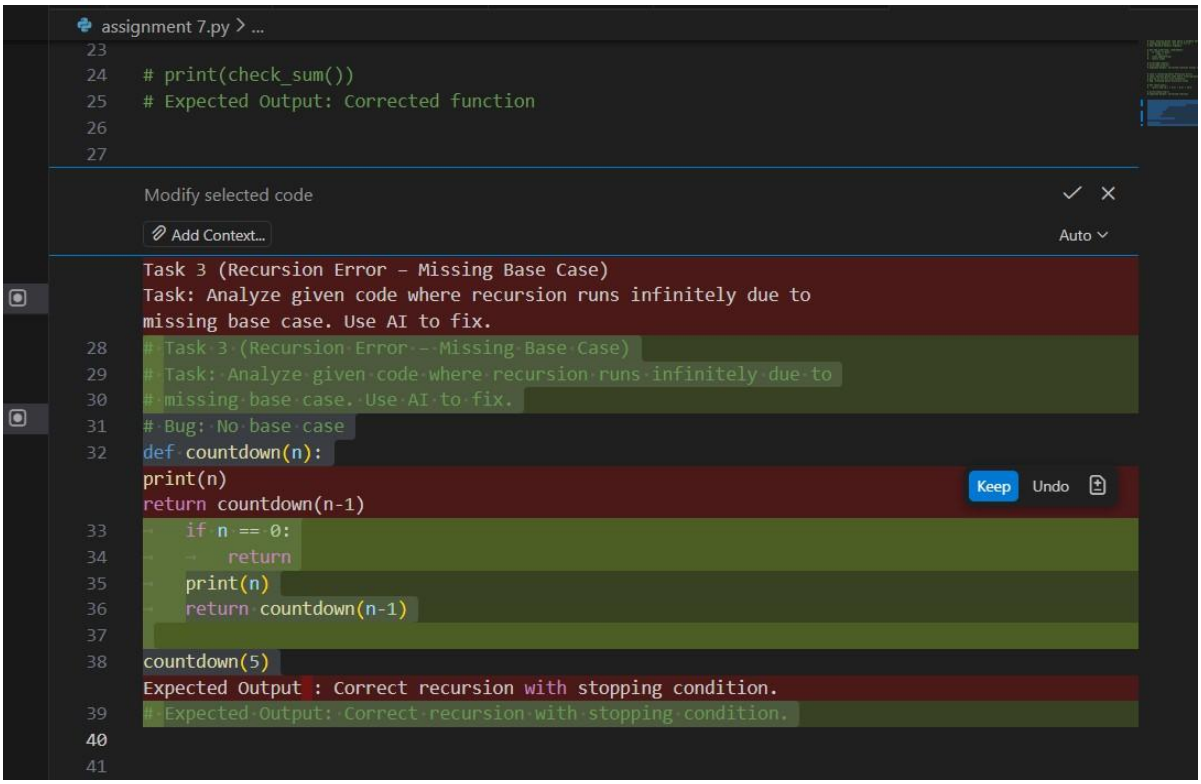
Task 3 (Recursion Error – Missing Base Case)

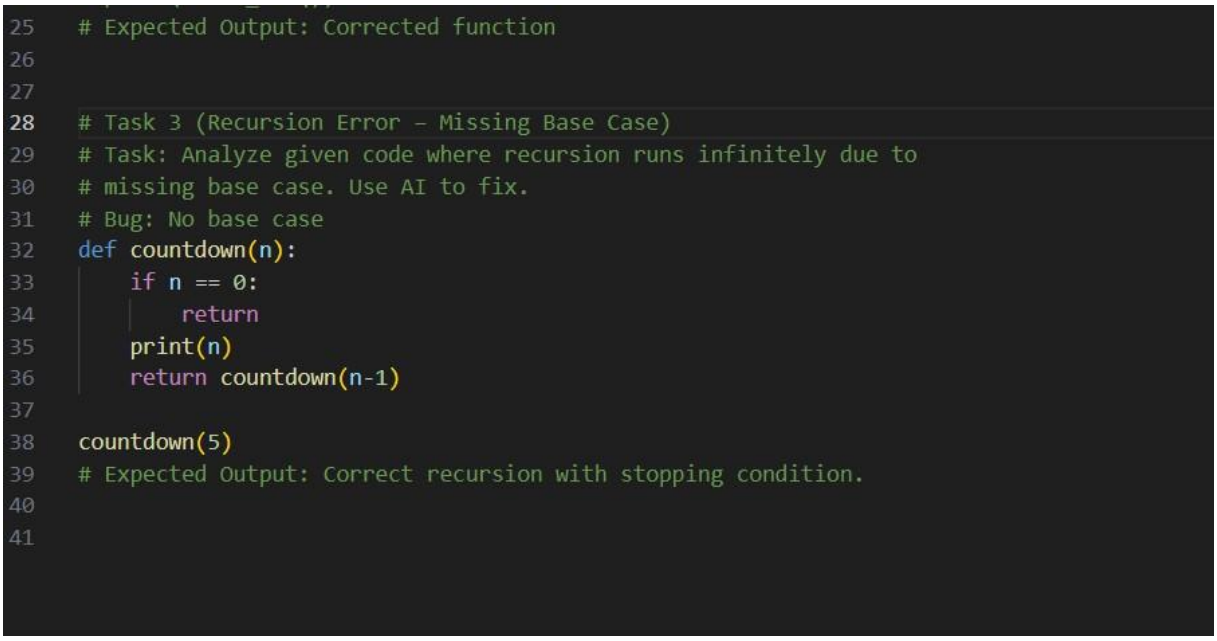
Task: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.

```
# Bug: No base case
def countdown(n):
print(n)
return countdown(n-1)
countdown(5)
```

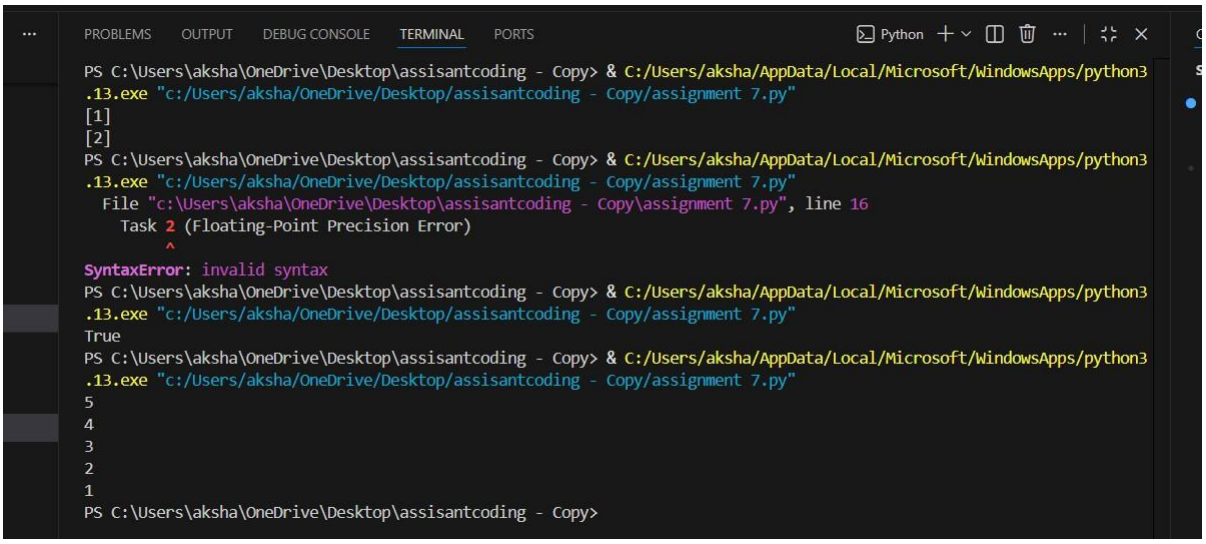
Expected Output : Correct recursion with stopping condition.

Code:


assignment 7.py > ...
print(check_sum())
Expected Output: Corrected function
Modify selected code
Add Context...
Auto
Task 3 (Recursion Error – Missing Base Case)
Task: Analyze given code where recursion runs infinitely due to
missing base case. Use AI to fix.
# 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)
Keep
Undo
if n == 0:
return
print(n)
return countdown(n-1)
countdown(5)
Expected Output : Correct recursion with stopping condition.
Expected Output: Correct recursion with stopping condition.


25 # Expected Output: Corrected function
26
27
28 # Task 3 (Recursion Error – Missing Base Case)
29 # Task: Analyze given code where recursion runs infinitely due to
30 # missing base case. Use AI to fix.
31 # Bug: No base case
32 def countdown(n):
33 if n == 0:
34 return
35 print(n)
36 return countdown(n-1)
37
38 countdown(5)
39 # Expected Output: Correct recursion with stopping condition.
40
41

Output:



Observation:

The original recursive function lacks a base case, so the function keeps calling itself indefinitely, leading to infinite recursion or a recursion depth error.

Adding a stopping condition such as if n == 0: return ensures that the recursion terminates properly.

After fixing, the function prints numbers from 5 down to 1 and then stops successfully.

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():

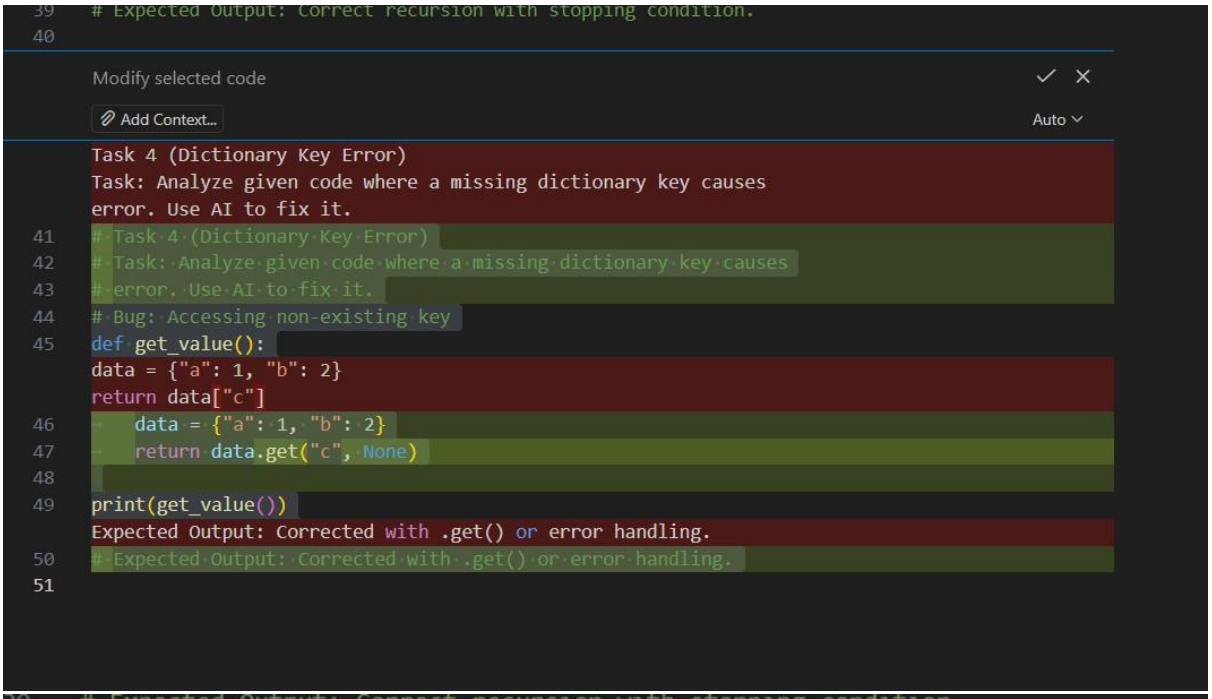
data = {"a": 1, "b": 2}

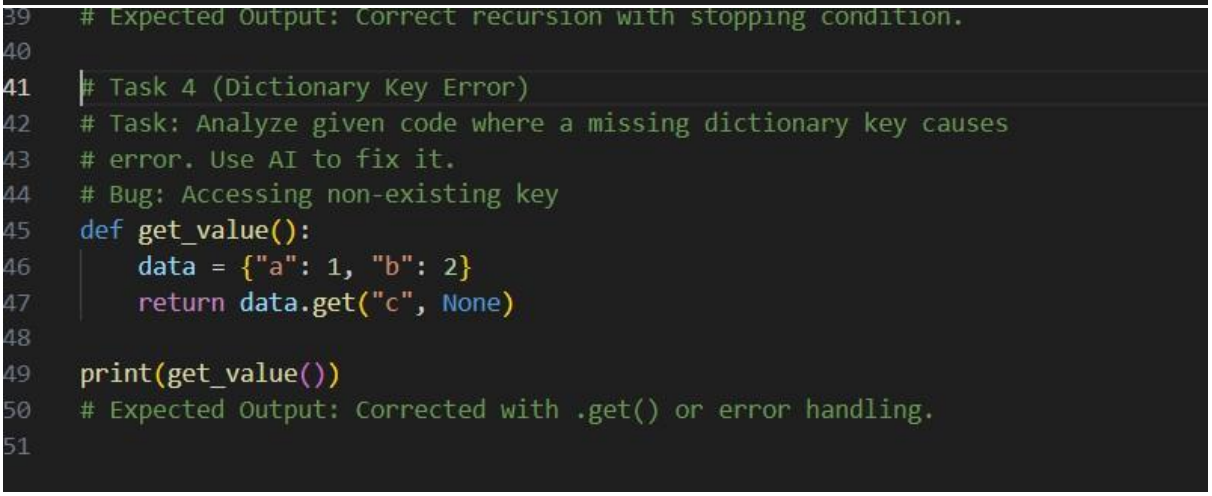
return data["c"]

print(get_value())

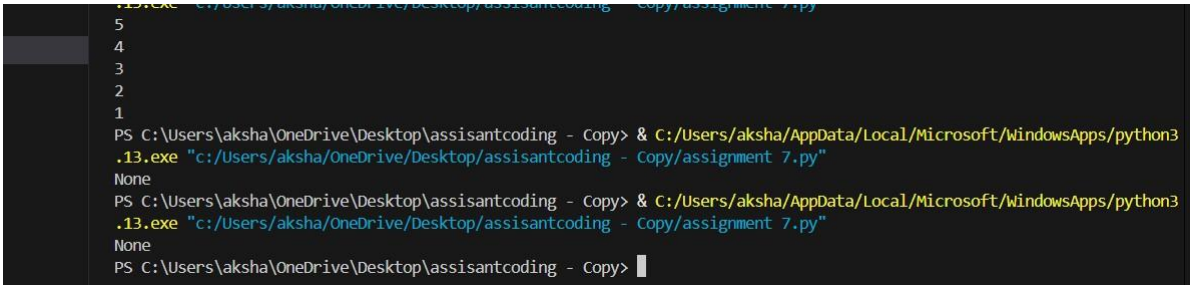
Expected Output: Corrected with .get() or error handling.

Code:





Output:



Observation:

Accessing a non-existing dictionary key using data["c"] raises a KeyError because the key is not present in the dictionary.

Using data.get("c", "Key not found") safely returns a default value instead of causing an error.

This fix ensures the program runs without crashing and handles missing keys gracefully.

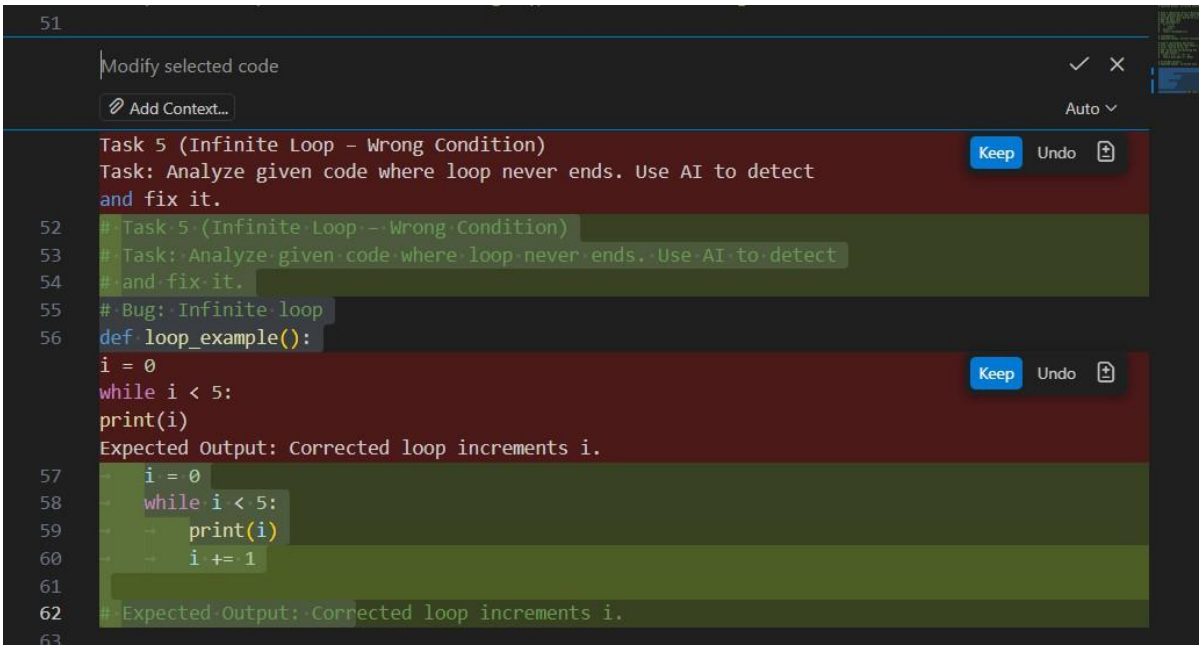
Task 5 (Infinite Loop – Wrong Condition)

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.

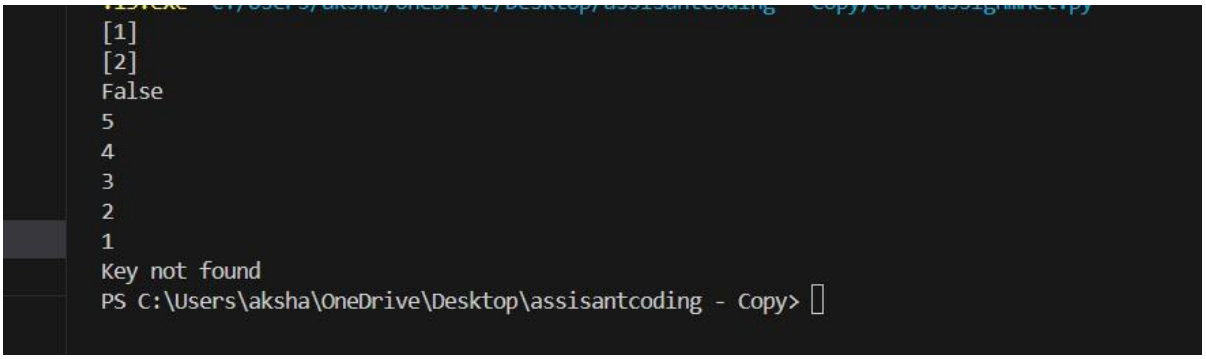
Code:




```python
# 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)
        i += 1

# Expected Output: Corrected loop increments i.
```

Output:



Observation:

The original loop does not update the value of i, so the condition i < 5 always remains true, causing an infinite loop.

Adding the increment statement i += 1 inside the loop ensures that i increases after each iteration.

With this fix, the loop prints values from 0 to 4 and then terminates correctly.

Task 6 (Unpacking Error – Wrong Variables)

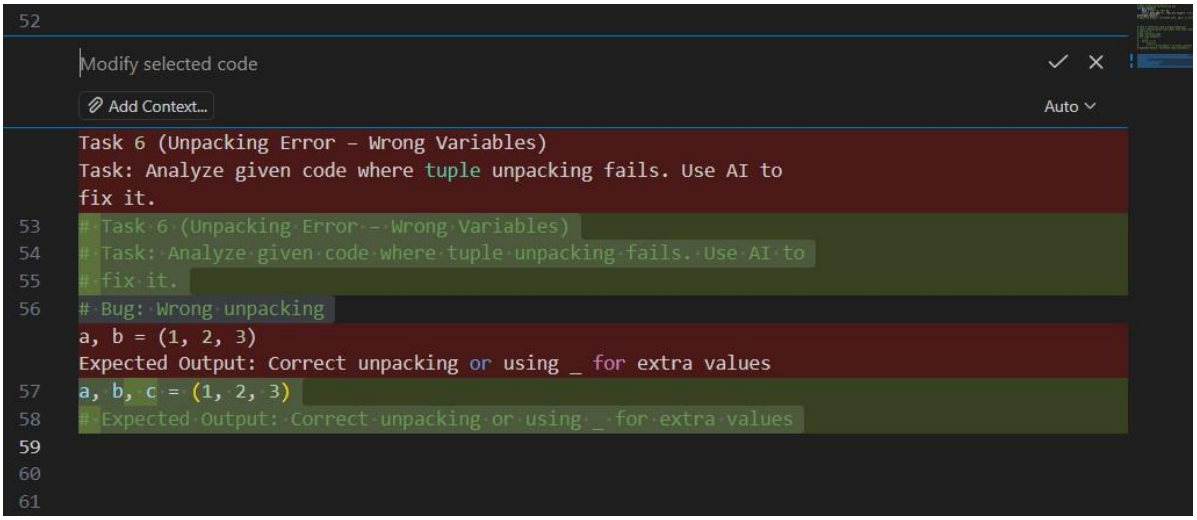
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

Bug: Wrong unpacking

a, b = (1, 2, 3)

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

Code:



```
51
52
53  # Task 6 (Unpacking Error – Wrong Variables)
54  # Task: Analyze given code where tuple unpacking fails. Use AI to
55  # fix it.
56  # Bug: Wrong unpacking
57  a, b, c= (1, 2, 3)
58  # Expected Output: Correct unpacking or using _ for extra values
59
60
61
62
```

Output:

```
    ^^^^^
ValueError: too many values to unpack (expected 2)
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy> & C:/Users/aksha/AppData/Local/Microsoft/WindowsApps/python3
.13.exe "c:/Users/aksha/OneDrive/Desktop/assisantcoding - Copy/errorassignmnet.py"
[1]
[2]
PS C:\Users\aksha\OneDrive\Desktop\assisantcoding - Copy>
```

Observation:

The tuple (1, 2, 3) contains three values, but only two variables (a, b) are provided, causing a “too many values to unpack” error.

Tuple unpacking requires the number of variables to match the number of values unless a placeholder like _ is used.

Fixing it as a, b, _ = (1, 2, 3) or using three variables correctly resolves the unpacking error.

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():

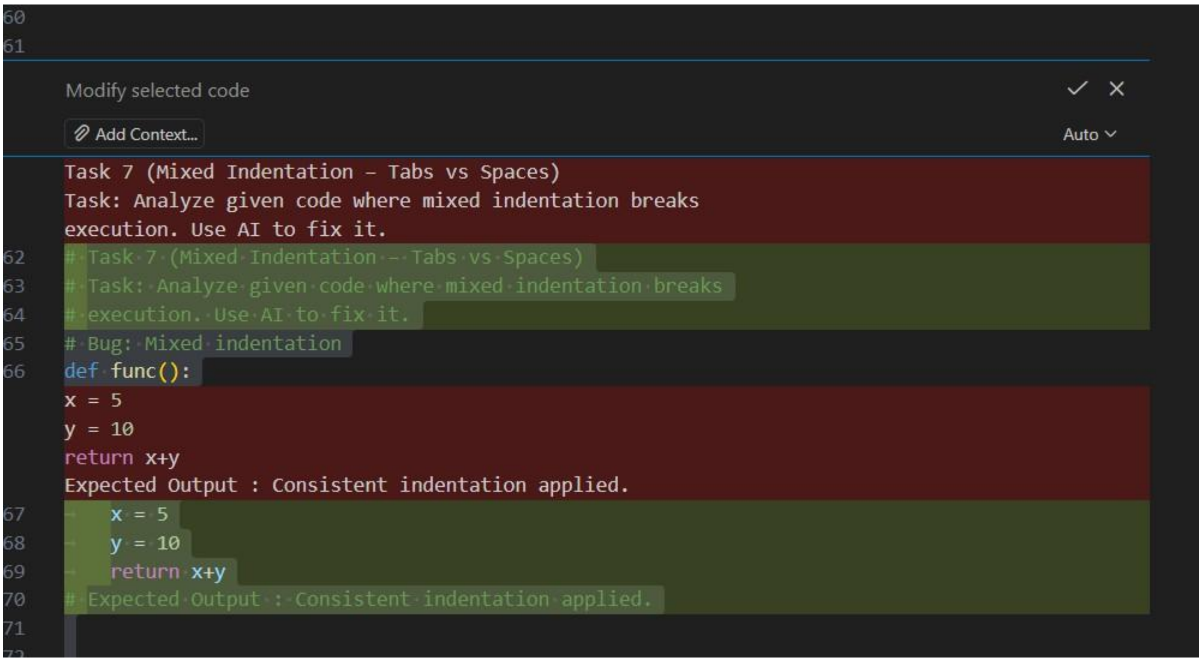
x = 5

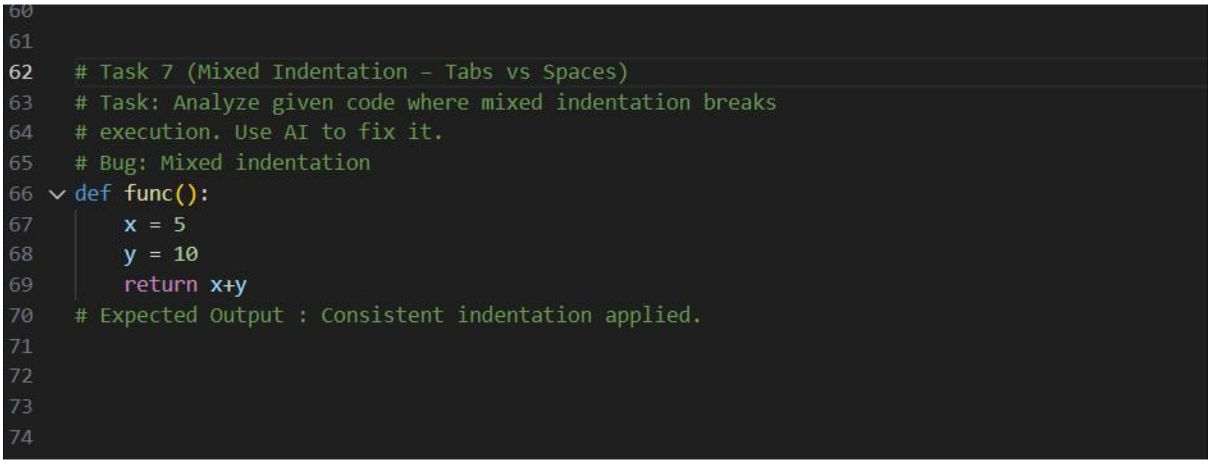
y = 10

return x+y

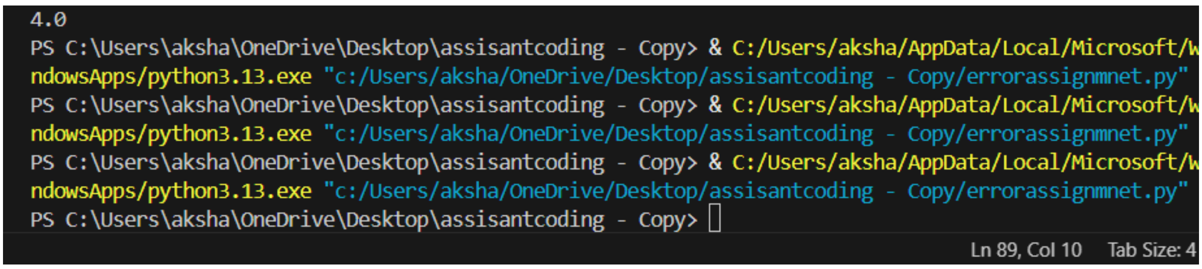
Expected Output : Consistent indentation applied.

Code:





Output:



Observation:

The original code uses mixed indentation (tabs and spaces), which causes an indentation error and prevents the program from executing correctly.

Python requires consistent indentation within a block to define the function body properly.

Using the same indentation style (for example, four spaces for each line) fixes the issue and produces the correct output 15.

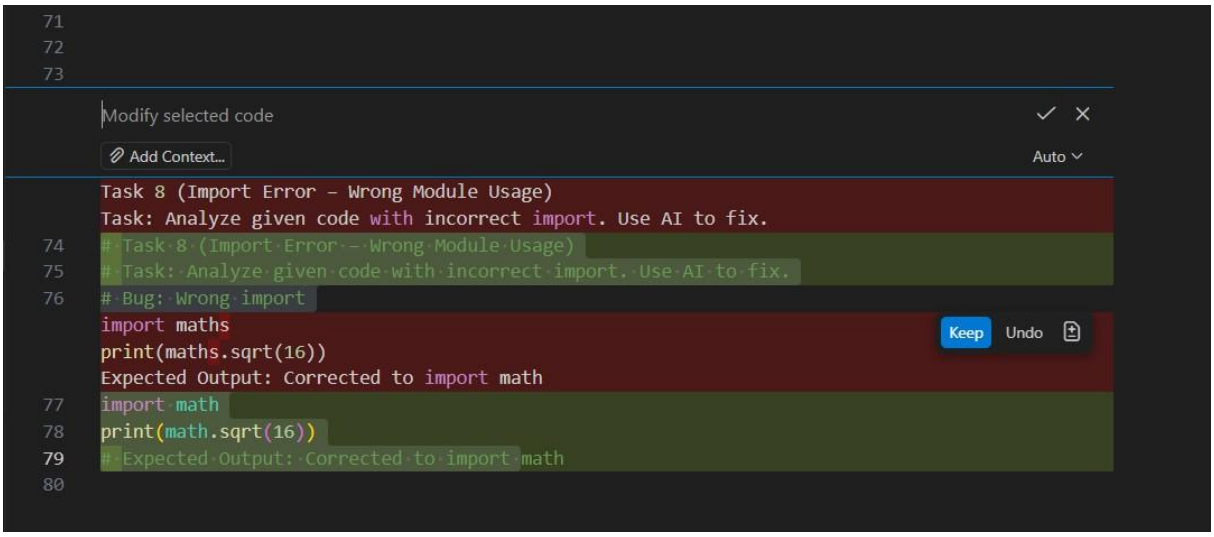
Task 8 (Import Error – Wrong Module Usage)

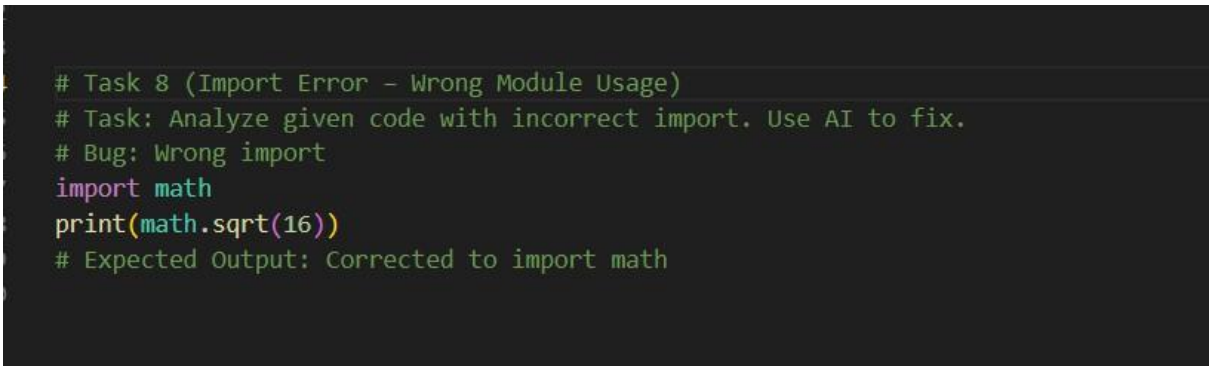
Task: Analyze given code with incorrect import. Use AI to fix.

```
# Bug: Wrong import
import maths
print(maths.sqrt(16))
```

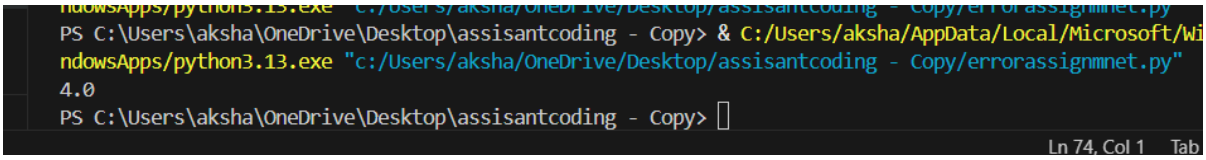
Expected Output: Corrected to import math

Code:





Output:



Observation:

The original code fails because it uses mixed indentation, which Python does not allow within the same block.

This causes an indentation error and stops the program from running correctly.

Applying consistent indentation (such as four spaces for all lines inside the function) fixes the issue and allows the function to execute properly.