# ASSIGNMENT- 7.5

**Name:** T.Sriharshitha

**HT. No:** 2303A51261

**Batch:** 19

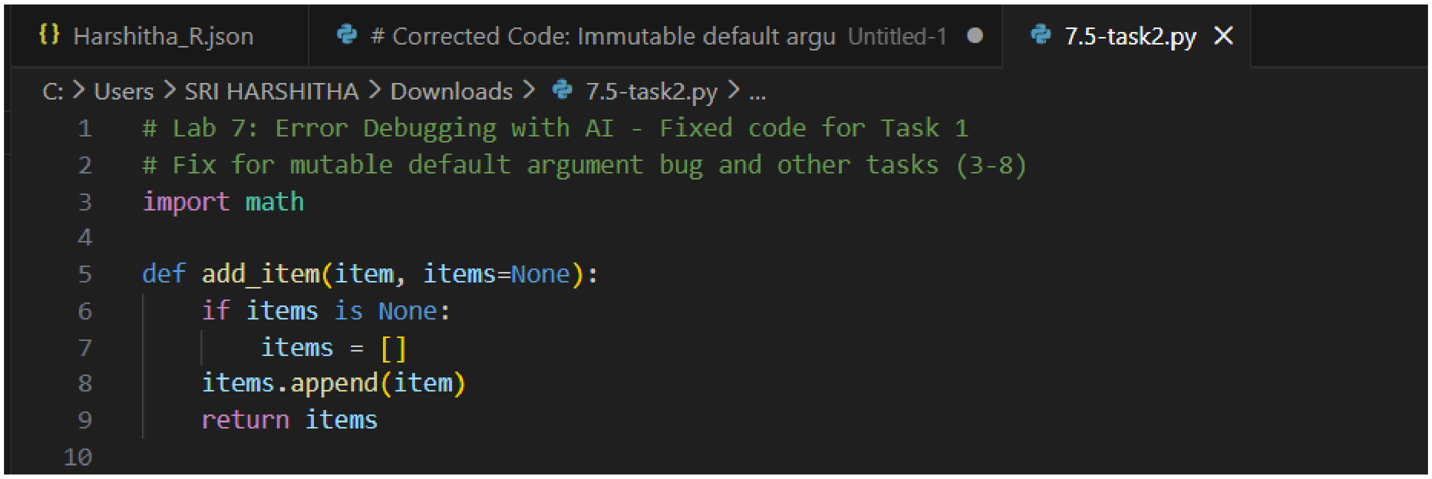
**Task 1:** Mutable Default Argument – Function Bug

The given function uses a mutable default argument, which causes data to persist across function calls and leads to unexpected behavior

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

.**Prompt: #Fix the Python function where a mutable default argument causes unexpected behavior.**

**Code:**



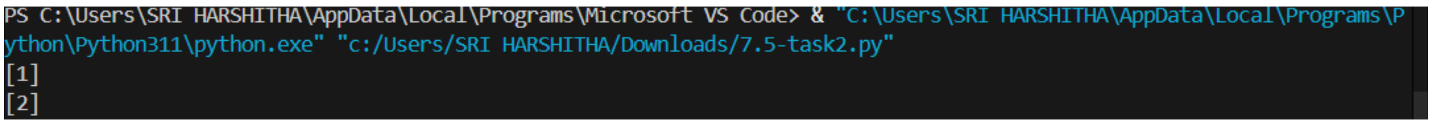
{} Harshitha_R.json
# Corrected Code: Immutable default argu Untitled-1
7.5-task2.py
C: > Users > SRI HARSHITHA > Downloads > 7.5-task2.py > ...

```python
# Lab 7: Error Debugging with AI - Fixed code for Task 1
# Fix for mutable default argument bug and other tasks (3-8)
import math

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

```

**Result:**



```
PS C:\Users\SRI HARSHITHA\AppData\Local\Programs\Microsoft VS Code> & "C:\Users\SRI HARSHITHA\AppData\Local\Programs\Python\Python311\python.exe" "c:/Users/SRI HARSHITHA/Downloads/7.5-task2.py"
[1]
[2]
```

**Observation:**
The AI correctly identified that mutable default arguments are shared across function calls. Replacing the default list with `None` and initializing it inside the function prevents unintended data sharing and ensures correct behavior.

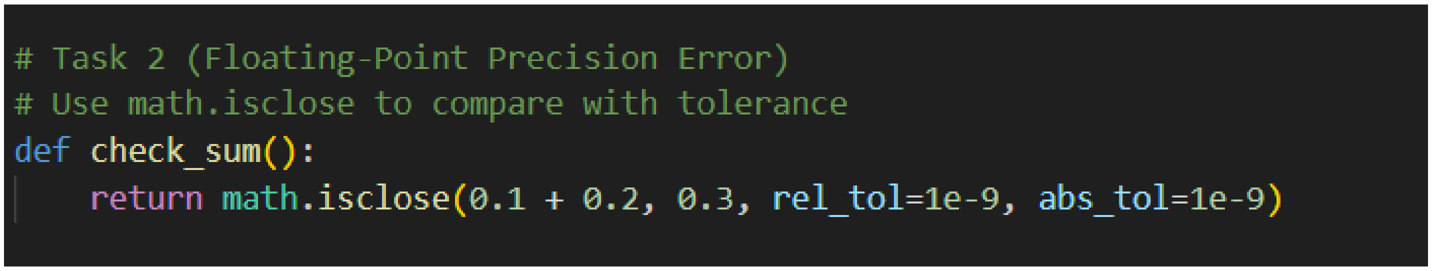
**Task 2:** Task 2: Floating-Point Precision Error

Direct comparison of floating-point numbers leads to incorrect results due to precision limitations.

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

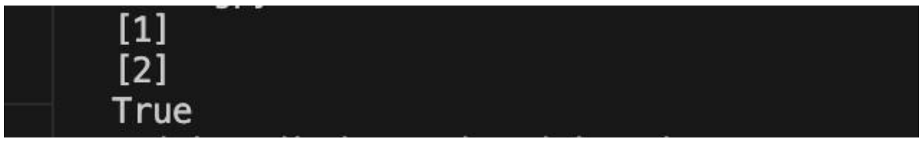
**Prompt**: #Fix the floating-point comparison issue

using tolerance **Code:**



```python
# Task 2 (Floating-Point Precision Error)
# Use math.isclose to compare with tolerance
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9, abs_tol=1e-9)
```

**Result:**



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

**Observation:**
The AI correctly addressed floating-point precision issues by using a tolerance-based comparison instead of direct equality, which is a recommended and reliable approach in numerical computing.

**Task 3:** Task 3: Recursion Error – Missing Base Case. The recursive function lacks a base case, resulting in infinite recursion.

**Prompt**: # Fix the recursion error caused by a missing base case.

# Bug: No base case

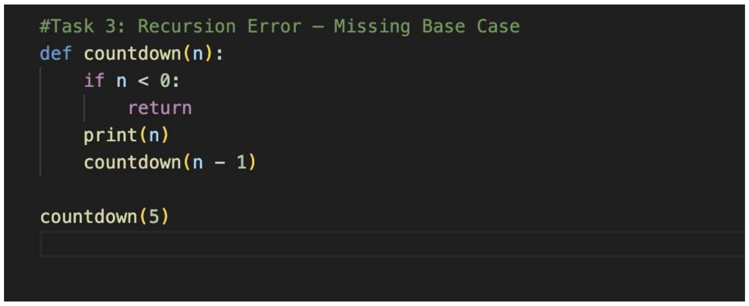
def countdown(n):

print(n)

return countdown(n-1)

countdown(5)

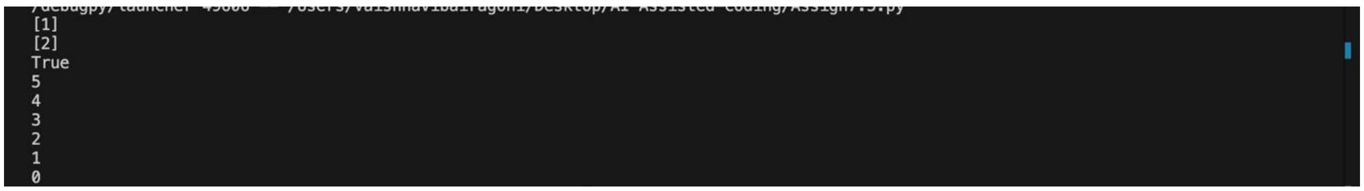
**Code:**



```
#Task 3: Recursion Error – Missing Base Case
def countdown(n):
    if n < 0:
        return
    print(n)
    countdown(n - 1)

countdown(5)
```

**Result:**



```
[illegible]
[1]
[2]
True
5
4
3
2
1
0
```

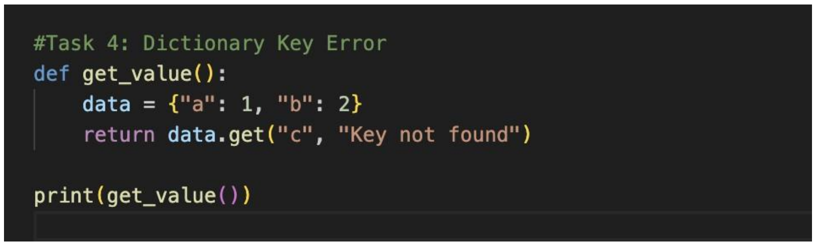
**Observation:**
The AI correctly identified the absence of a base condition and added a stopping condition, preventing infinite recursion and ensuring safe execution.

**Task 4:** Task 4: Dictionary Key Error. Accessing a non-existent key in a dictionary causes a runtime `KeyError`.

# Bug: Accessing non-existing key
def get_value(): data = {"a": 1,
"b": 2} return data["c"]
print(get_value())

**Prompt:** #Fix the dictionary KeyError using safe access methods..

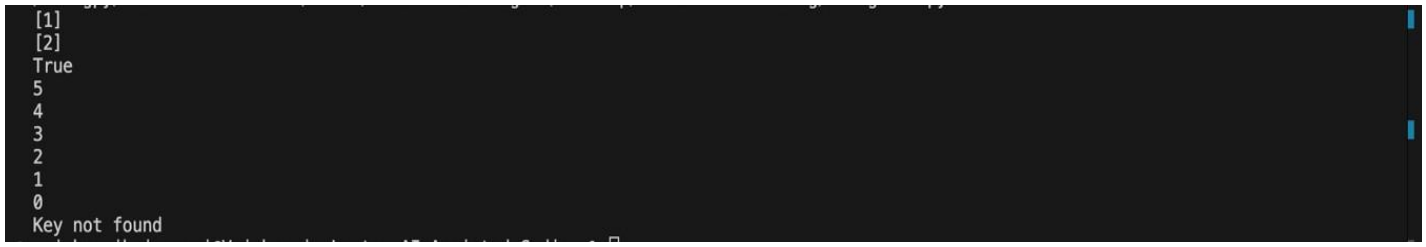
**Code:**



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

print(get_value())
```

**Result:**



```
[1]
[2]
True
5
4
3
2
1
0
Key not found
```

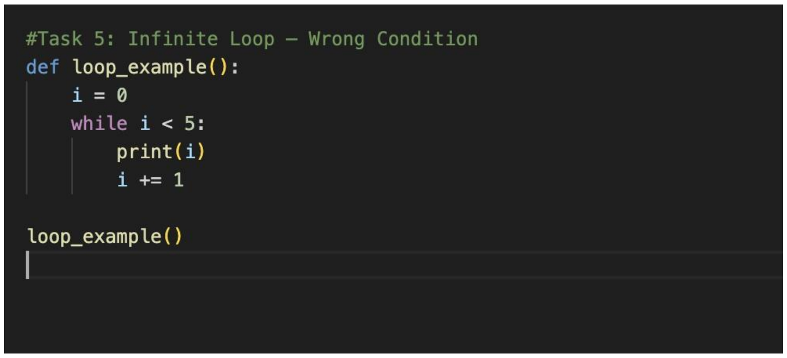
## Observation
The AI resolved the issue by using the `.get()` method, which safely handles missing keys and prevents runtime errors.

**Task 5:** Task 5: Infinite Loop – Wrong Condition. The loop never terminates because the loop variable is not updated.

# Bug: Infinite loop def
loop_example():
i = 0 while
i < 5:
print(i)

**Prompt: #Fix the infinite loop by correcting the loop condition.**

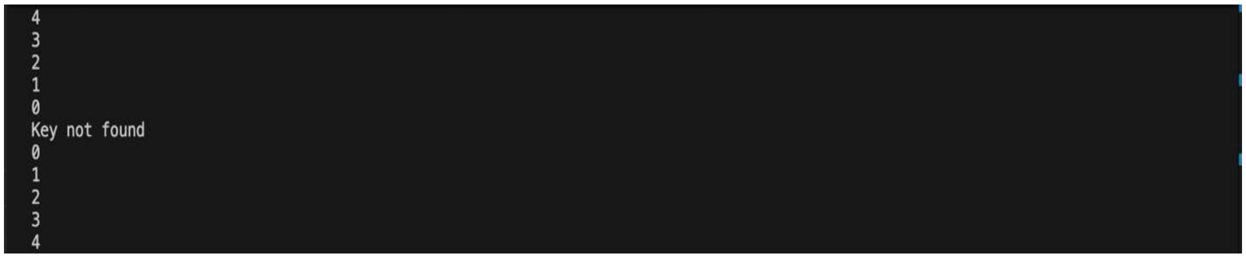
**Code:**



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

loop_example()
```

**Result:**



```
4
3
2
1
0
Key not found
0
1
2
3
4
```

**Observation:**
The AI correctly identified the missing increment statement and fixed the infinite loop by updating the loop variable inside the loop
**Task 6:** Task 6: Unpacking Error – Wrong Variables

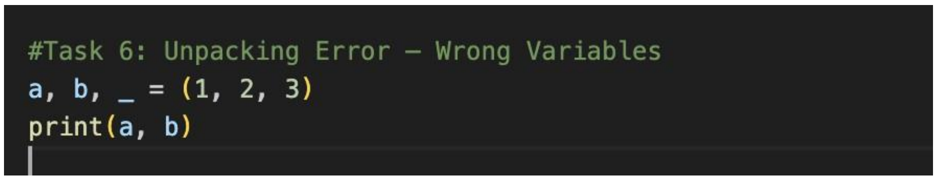
Tuple unpacking fails because the number of variables does not match the tuple size

# Bug: Wrong unpacking

a, b = (1, 2, 3)

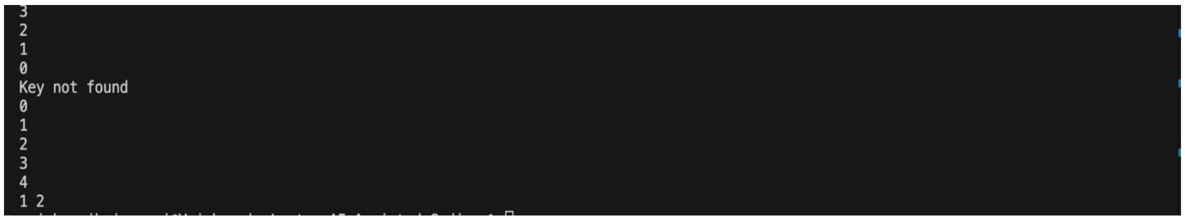
**Prompt: # Fix the tuple unpacking error caused by mismatched variables.**

**Code:**



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

**Result:**



```
3
2
1
0
Key not found
0
1
2
3
4
1 2
```

**Observation:**

The AI fixed the unpacking issue by using an underscore (_) to ignore extra values, which is a Pythonic and safe practice
**Task 7:** Task 7: Mixed Indentation – Tabs vs Spaces. Inconsistent indentation causes syntax or runtime errors in Python. # Bug: Mixed indentation

def func():

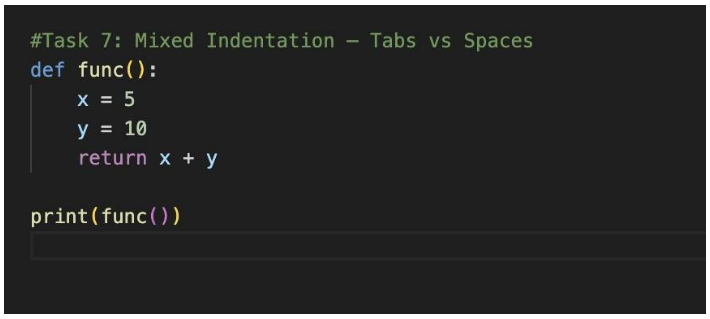
x = 5

y = 10

return x+y

**Prompt**: # Fix the Python code with mixed indentation.

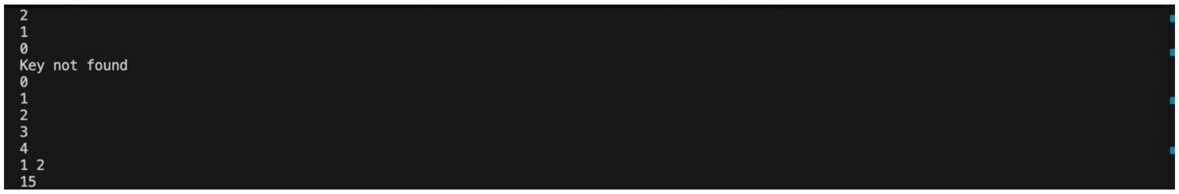
**Code:**



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

print(func())
```

**Result:**



```
2
1
0
Key not found
0
1
2
3
4
1 2
15
```

**Observation:**
The AI resolved the issue by applying consistent indentation using spaces, which is the recommended Python coding standard.

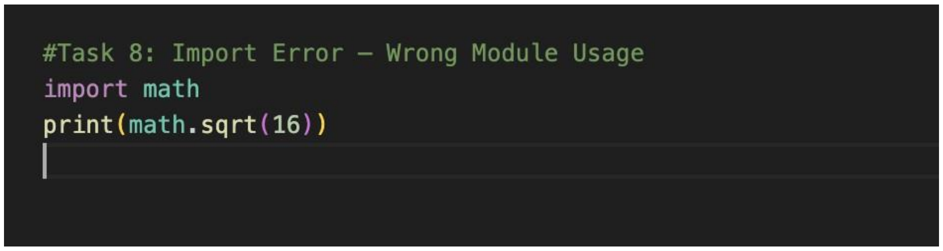
**Task 8:** Task 8: Import Error – Wrong Module Usage. The code attempts to import a nonexistent module, causing an import error.

**Prompt**: # Fix the incorrect module import in the Python code.
# Bug: Wrong import

import maths

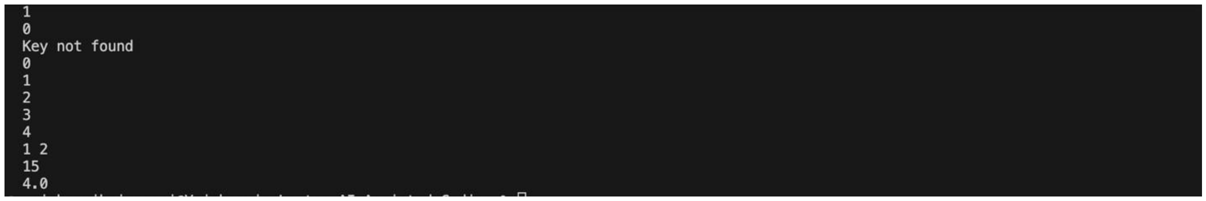
print(maths.sqrt(16))

**Code:**



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

**Result:**



```
1
0
Key not found
0
1
2
3
4
1 2
15
4.0
```

**Observation:**

The AI correctly identified the incorrect module name and replaced it with the standard `math` module, resolving the import error.