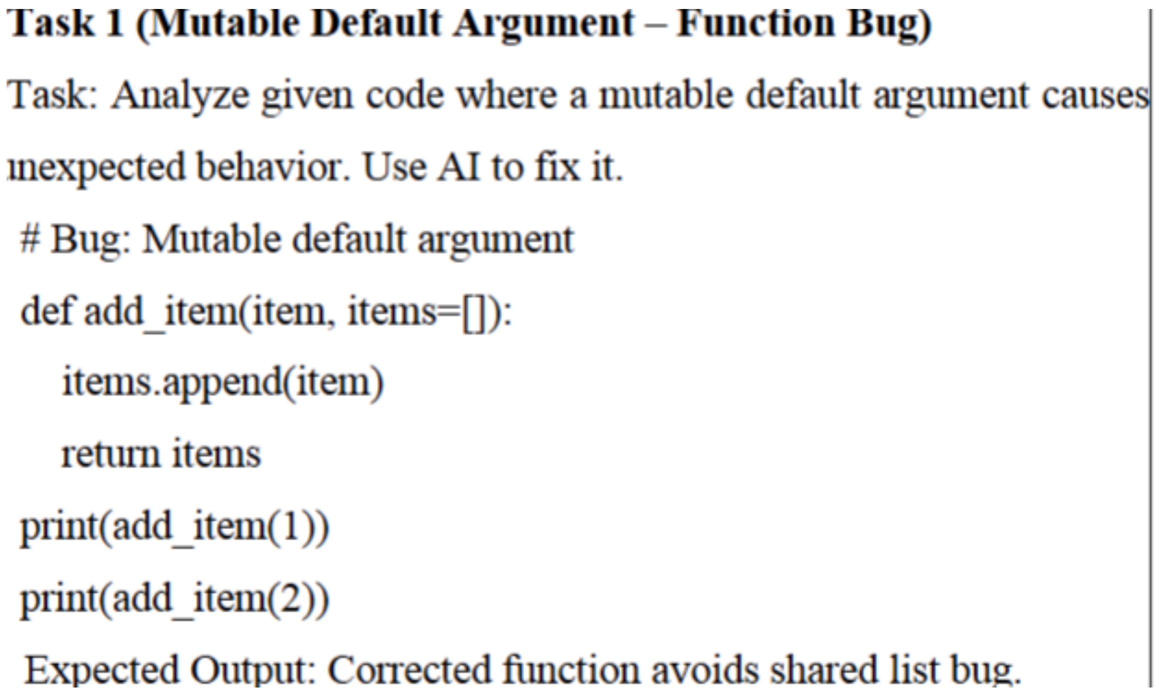
LAB 7: Error Debugging with AI

Course: AI Assisted Coding
Assignment Number: 7.5
Sushruth Reddy Chada
batch – 04
2303A51219

Objective: To identify and fix syntax, logic, and runtime errors using AI-assisted debugging.

Task 1: Mutable Default Argument – Function Bug



Task 1 (Mutable Default Argument – Function Bug)

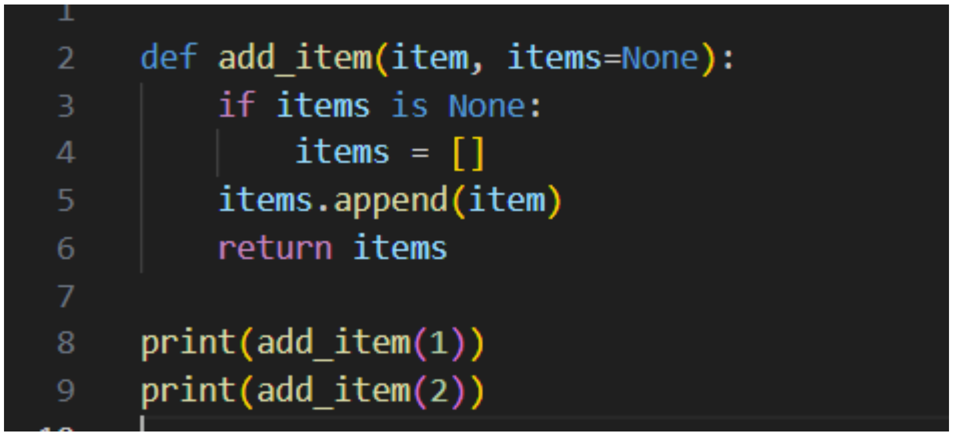
Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

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

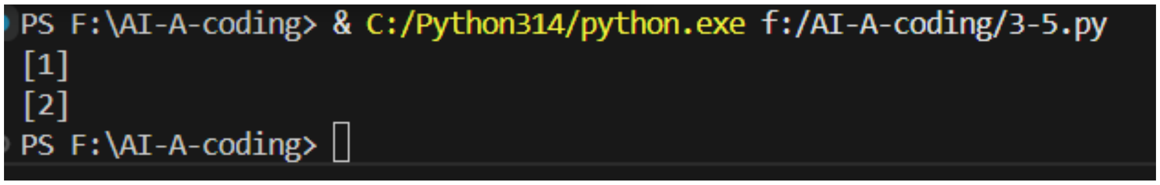
Expected Output: Corrected function avoids shared list bug.

AI Prompt:
Analyze the Python function and identify why the list is shared between calls. Fix the issue.

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

print(add_item(1))
print(add_item(2))
```



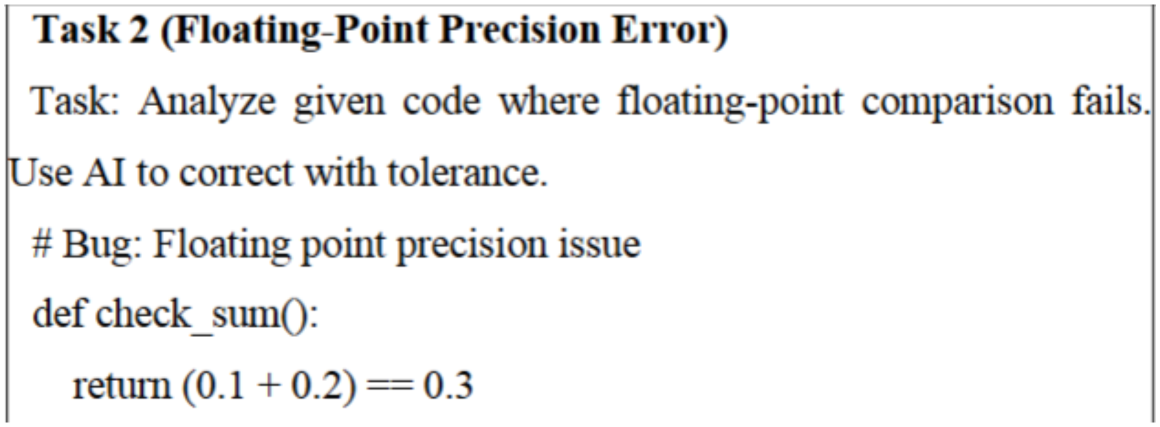
```python
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

print(add_item(1))
print(add_item(2))
```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/3-5.py
[1]
[2]
PS F:\AI-A-coding>
```

Explanation:
In Python, default arguments are evaluated only once when the function is defined. When a mutable object like a list is used as a default argument, the same list is shared across multiple function calls. This causes unexpected results. The issue is fixed by using None as the default value and creating a new list inside the function

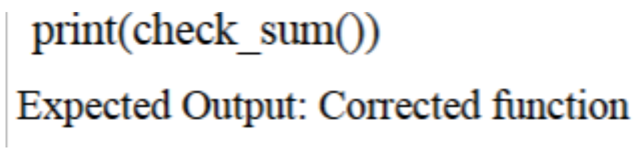
Task 2: Floating-Point Precision Error



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
```



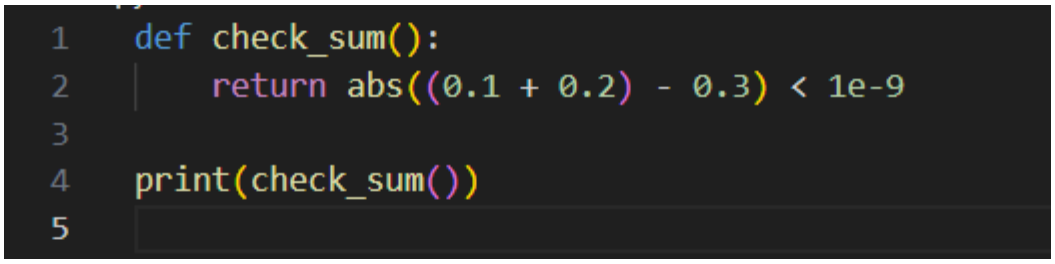
print(check_sum())

Expected Output: Corrected function

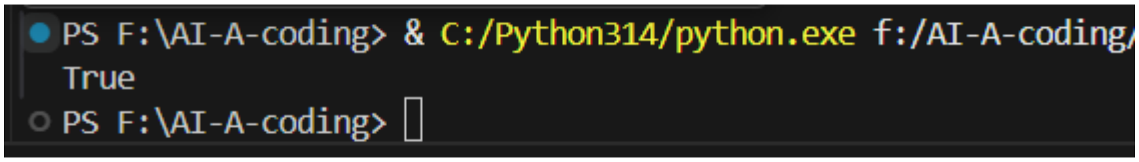
AI Prompt:
Why does floating-point comparison fail? Fix it using tolerance.

Solution:
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9

print(check_sum())



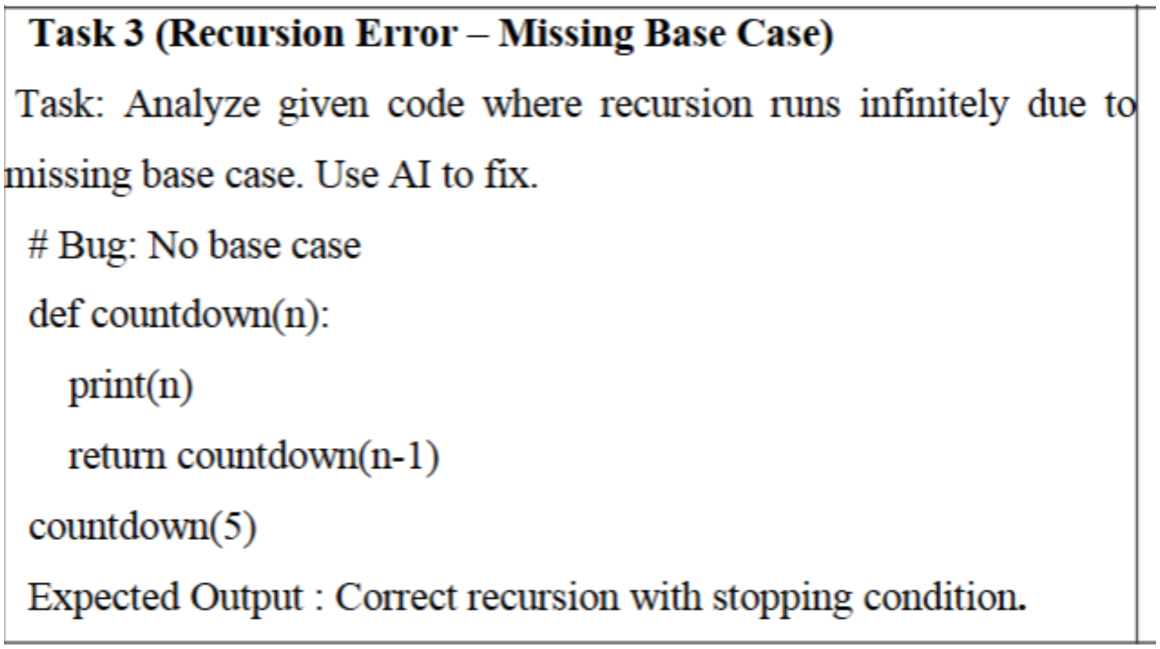
```python
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9

print(check_sum())

```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/
True
PS F:\AI-A-coding> 
```

Explanation:
Floating-point numbers are stored as approximations in memory, so exact comparisons using == may fail. The sum of 0.1 and 0.2 is not exactly equal to 0.3 due to precision limitations. This issue is corrected by comparing the difference using a small tolerance value

Task 3: Recursion Error – Missing Base Case



Task 3 (Recursion Error – Missing Base Case)

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

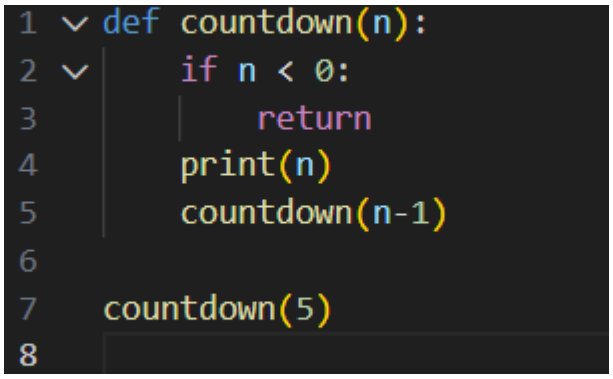
```
# Bug: No base case
def countdown(n):
    print(n)
    return countdown(n-1)
countdown(5)
```

Expected Output : Correct recursion with stopping condition.

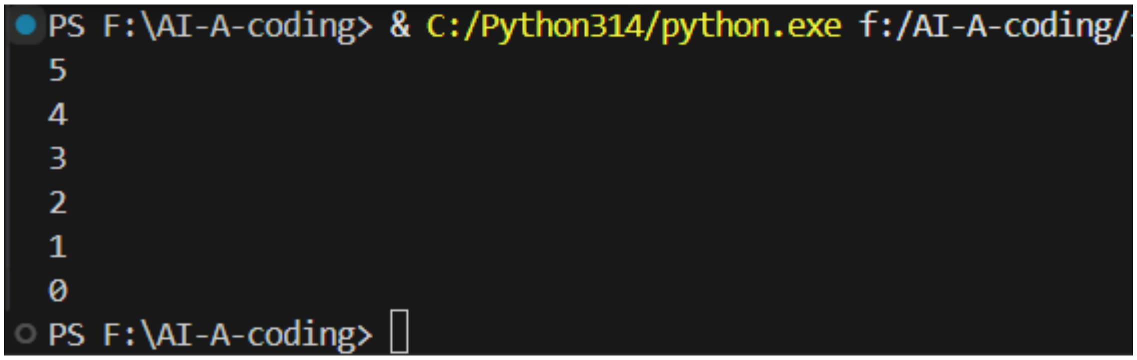
AI Prompt:
Identify the recursion error and add a proper base case.

Solution:

```
def countdown(n):
    if n < 0:
        return
    print(n)
    countdown(n-1)

countdown(5)
```



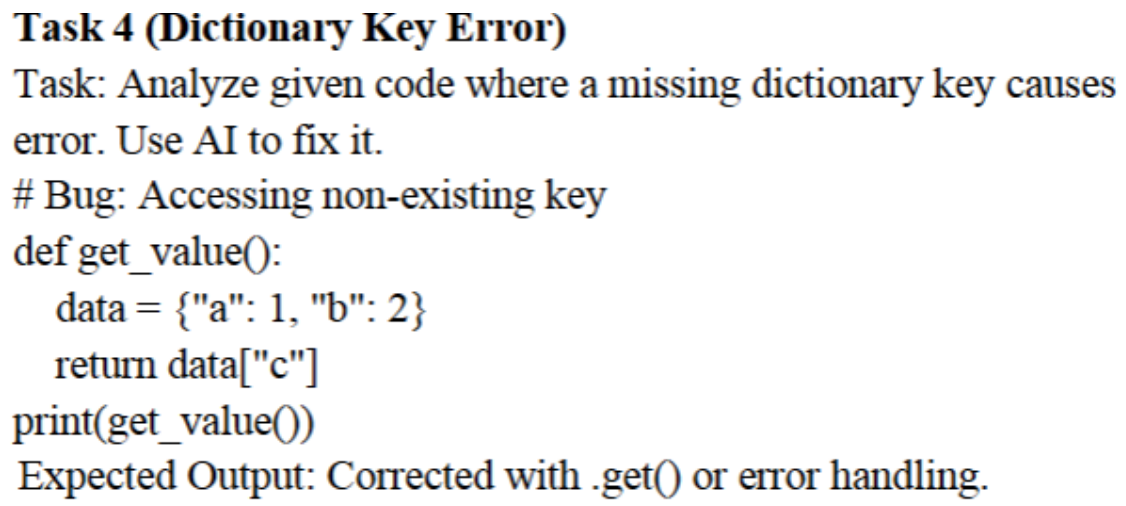
```
def countdown(n):
    if n < 0:
        return
    print(n)
    countdown(n-1)

countdown(5)
```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/
5
4
3
2
1
0
PS F:\AI-A-coding> 
```

Explanation:
Recursion must always have a base case to stop further function calls. Without a base case, the function keeps calling itself indefinitely, leading to a stack overflow error. Adding a base condition ensures the recursion stops at the correct point

Task 4: Dictionary Key Error



Task 4 (Dictionary Key Error)

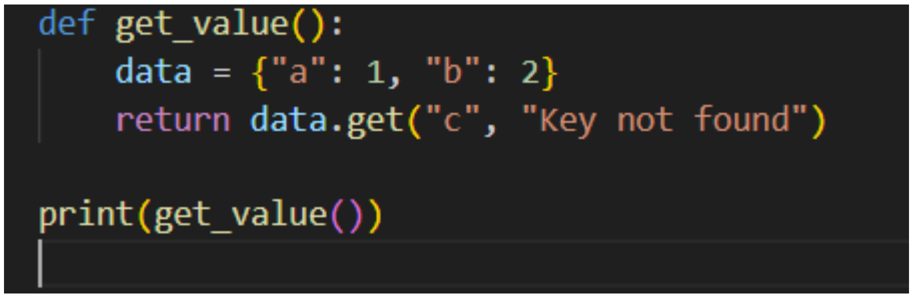
Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

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

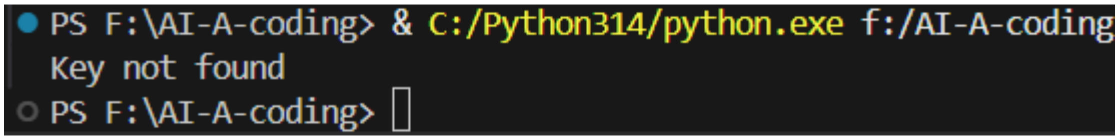
Expected Output: Corrected with .get() or error handling.

AI Prompt:
Fix the KeyError using safe dictionary access.

Solution:
```
def get_value():
   data = {"a": 1, "b": 2}
   return data.get("c", "Key not found")

print(get_value())
```



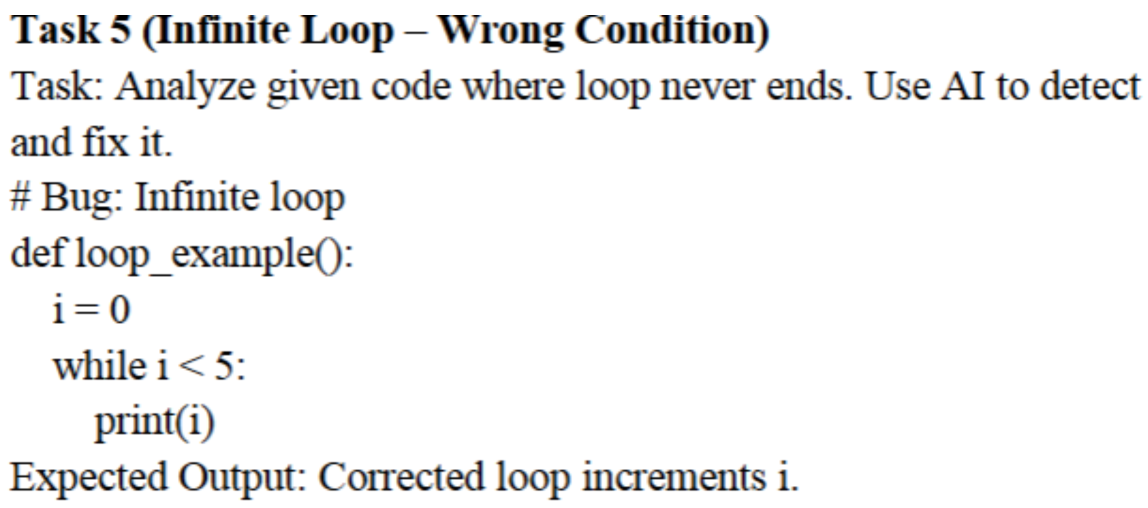
```python
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")

print(get_value())
```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding
Key not found
PS F:\AI-A-coding>
```

Explanation:
Accessing a dictionary key that does not exist raises a KeyError. This issue occurs when the program directly tries to access a missing key. Using the .get() method or exception handling prevents the error and allows the program to continue safely

Task 5: Infinite Loop – Wrong Condition



Task 5 (Infinite Loop – Wrong Condition)

Task: Analyze given code where loop never ends. Use AI to detect and fix it.

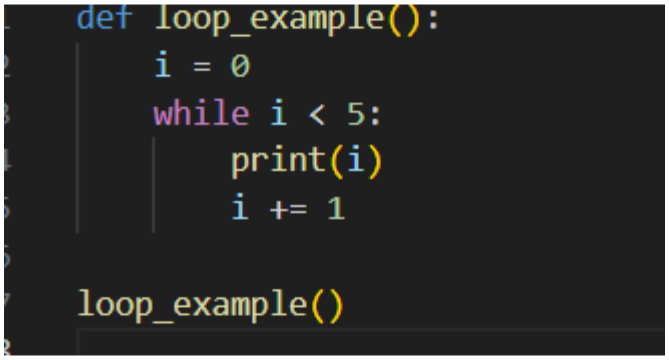
```
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)
```

Expected Output: Corrected loop increments i.

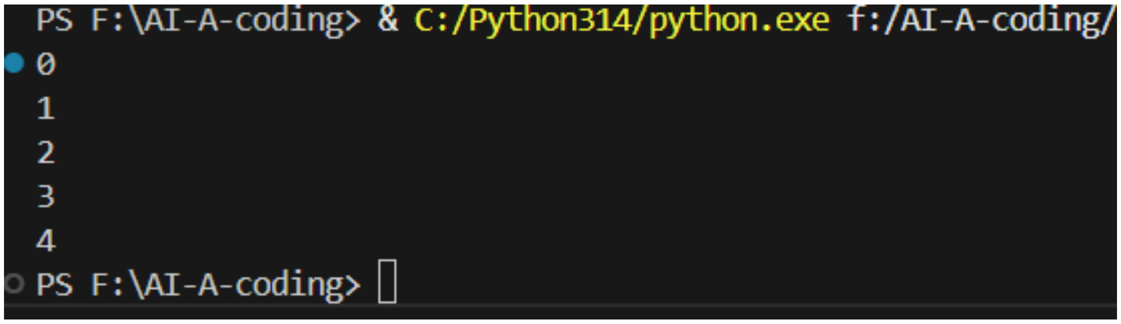
AI Prompt:
Why does the loop never stop? Fix the loop.

Solution:

```python
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1

loop_example()
```



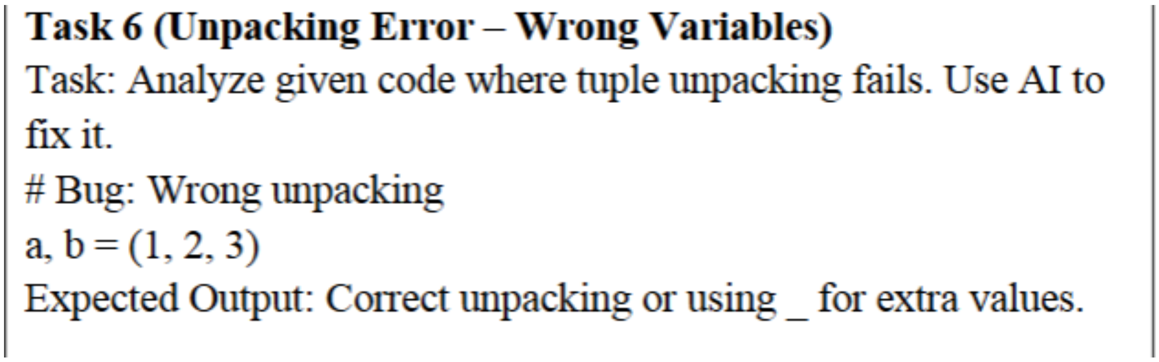
```
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1

loop_example()
```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/
0
1
2
3
4
PS F:\AI-A-coding> 
```

Explanation:
An infinite loop occurs when the loop condition never becomes false. In this case, the loop variable is not updated, so the condition always remains true. Incrementing the loop variable inside the loop ensures proper termination

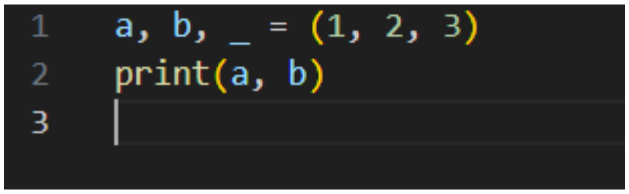
Task 6: Unpacking Error – Wrong Variables



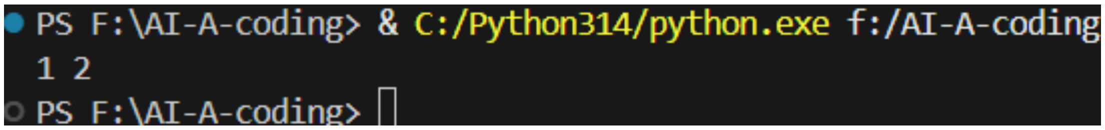
Task 6 (Unpacking Error – Wrong Variables)
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.
Bug: Wrong unpacking
a, b = (1, 2, 3)
Expected Output: Correct unpacking or using _ for extra values.

AI Prompt:
Fix the tuple unpacking error.

Solution:
a, b, _ = (1, 2, 3)
print(a, b)



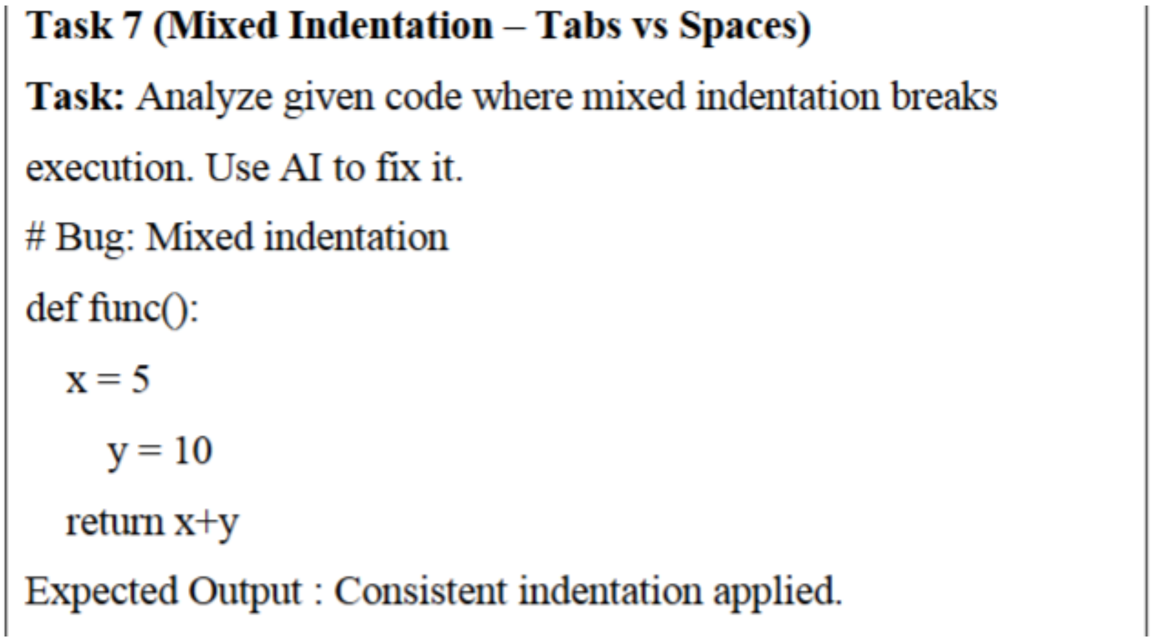
```
a, b, _ = (1, 2, 3)
print(a, b)

```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding
1 2
PS F:\AI-A-coding> 
```

Explanation:
Tuple unpacking requires the number of variables to match the number of values. When there are extra values, Python raises an error. This issue is fixed by using an underscore (_) to ignore unwanted values during unpacking.

Task 7: Mixed Indentation – Tabs vs Spaces



Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.

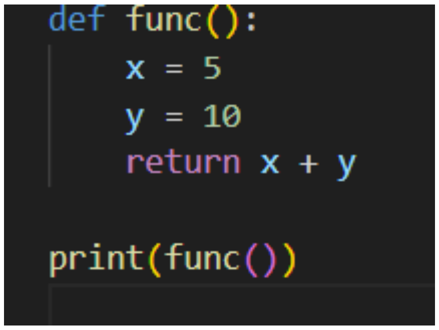
```
# Bug: Mixed indentation
def func():
    x = 5
        y = 10
    return x+y
```

Expected Output : Consistent indentation applied.

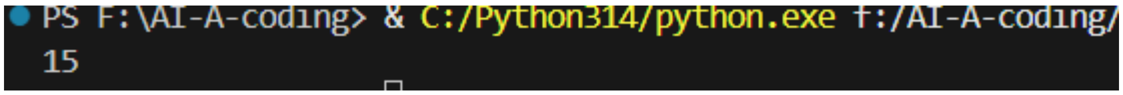
AI Prompt:
Fix the indentation error using consistent spacing.

Solution:

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

print(func())
```



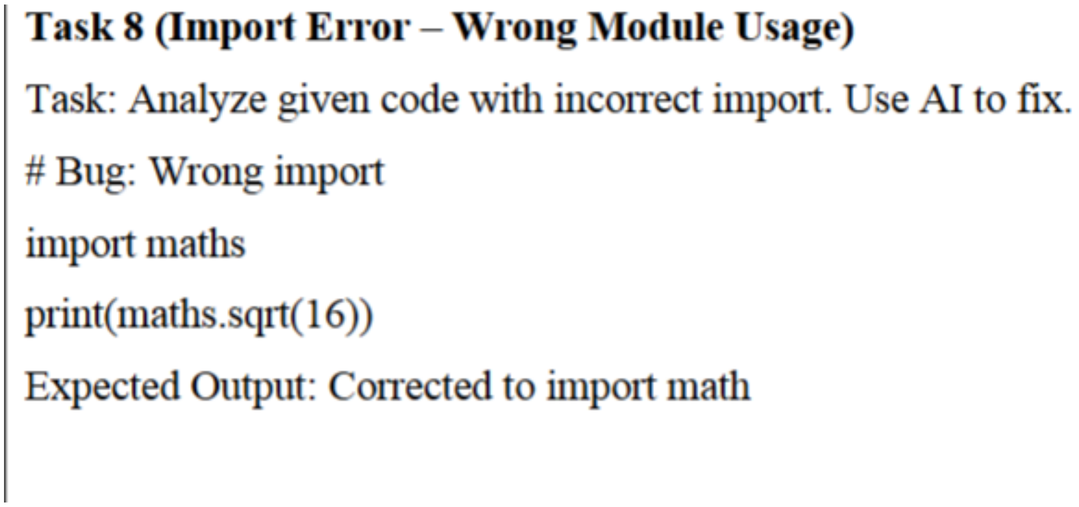
```python
def func():
    x = 5
    y = 10
    return x + y

print(func())
```



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/
15
```

Explanation:
Python relies on indentation to define code blocks. Mixing tabs and spaces causes indentation errors and breaks execution. The problem is fixed by using consistent indentation, usually four spaces, throughout the code

Task 8: Import Error – Wrong Module Usage



Task 8 (Import Error – Wrong Module Usage)

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

Bug: Wrong import

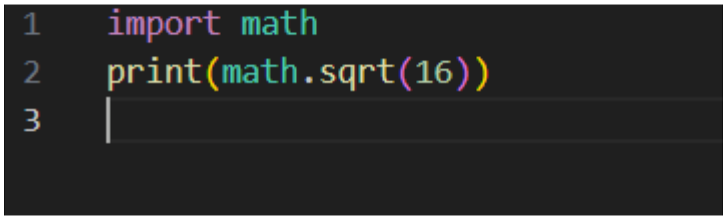
import maths

print(maths.sqrt(16))

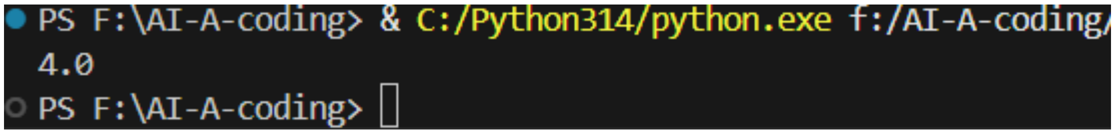
Expected Output: Corrected to import math

AI Prompt:
Correct the module import error.

Solution:
import math
print(math.sqrt(16))



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



```
PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/
4.0
PS F:\AI-A-coding> 
```

Explanation:
Import errors occur when a module name is incorrect or does not exist. In this case, the wrong module name is used. Correcting the module name ensures the required functions are imported properly

Lab Conclusion (Task 1 to Task 8)

This lab provided a comprehensive understanding of common programming errors in Python and demonstrated how AI-assisted debugging can effectively identify and resolve them. Across Tasks 1 to 8, various types of bugs were analyzed, including design flaws, logical errors, runtime exceptions, and syntax-related issues. Each task highlighted how minor mistakes in code structure, data handling, or logic can lead to incorrect results or program failure.

The lab emphasized the importance of following best coding practices such as avoiding mutable default arguments, handling floating-point precision carefully, defining proper base cases in recursive functions, and safely accessing data structures like dictionaries. It also reinforced the need for correct loop control to prevent infinite loops, proper variable matching during unpacking, consistent indentation in Python, and accurate module imports.

By using AI tools to detect errors, suggest fixes, and explain the root causes, the debugging process became more systematic and efficient. This lab enhanced problem-solving skills, improved code reliability, and built confidence in using AI as a supportive tool for debugging and refactoring Python programs. Overall, the lab successfully achieved its objectives by combining practical debugging scenarios with AI-driven insights to promote responsible and error-free programming