AI ASSISTED CODING

LAB -7.5

N.GOUTHAM

2303A51209

BATCH – 04

Lab 7.5: Error Debugging with AI – Systematic Approaches to Finding and Fixing Bugs

Task 1: Mutable Default Argument – Function Bug

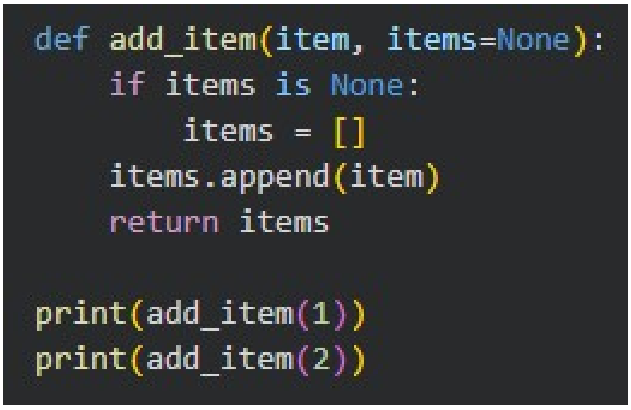
QUESTION

Analyze the given Python function where a mutable default argument causes unexpected behavior. Use AI assistance to fix the issue.

PROMPT

Identify the bug caused by using a mutable default argument in the function and modify the code so that the list is not shared between function calls.

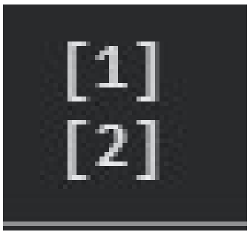
CODE



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

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

OUTPUT



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

EXPLANATION

In the original code, the default list was shared across multiple function calls, leading to unexpected results. By using None as the default value and creating a new list inside the function, each call gets its own independent list, avoiding shared state bugs.

Task 2: Floating-Point Precision Error

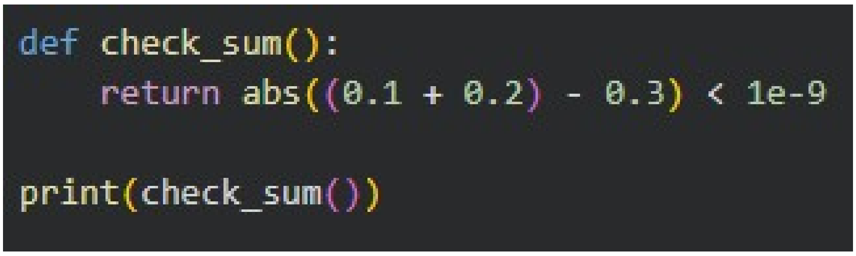
QUESTION

Analyze the given code where floating-point comparison fails due to precision issues. Use AI to correct it using tolerance.

PROMPT

Fix the floating-point comparison error by applying a tolerance-based comparison instead of direct equality.

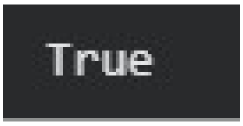
CODE



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

print(check_sum())
```

OUTPUT



```
True
```

EXPLANATION

Floating-point numbers cannot always be represented exactly in memory. Instead of direct comparison, a small tolerance value is used to check whether the numbers are close enough, ensuring reliable results.

Task 3: Recursion Error – Missing Base Case

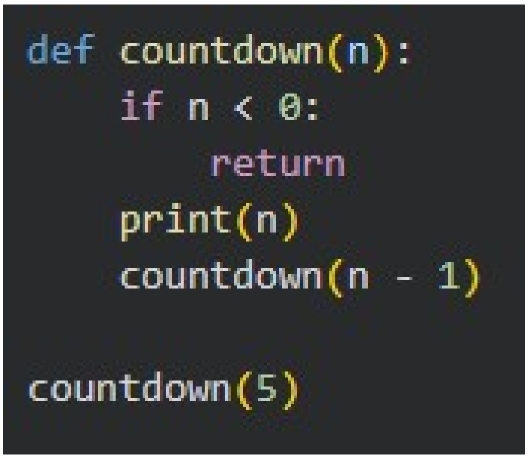
QUESTION

Analyze the recursive function that runs infinitely due to a missing base case and fix it using AI guidance.

PROMPT

Add an appropriate base condition to stop infinite recursion in the function.

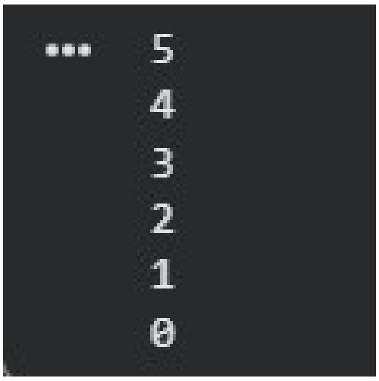
CODE



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

countdown(5)
```

OUTPUT



```
5
4
3
2
1
0
```

EXPLANATION

The original recursion lacked a stopping condition, causing infinite calls. Adding a base case ensures the function stops executing once the condition is met, preventing stack overflow errors.

Task 4: Dictionary Key Error

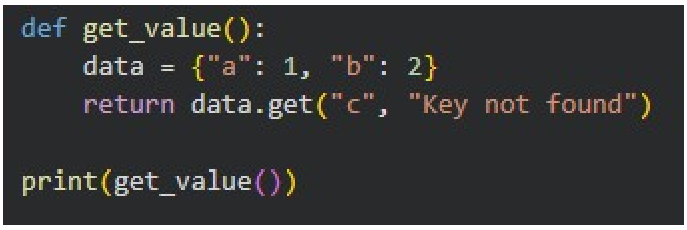
QUESTION

Analyze the code where accessing a non-existing dictionary key causes a runtime error and fix it.

PROMPT

Modify the code to safely access dictionary values using built-in methods or error handling.

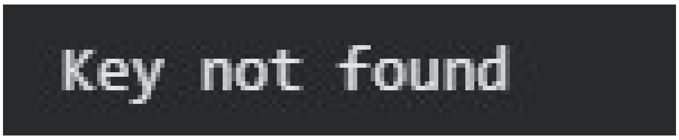
CODE



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

print(get_value())
```

OUTPUT



```
Key not found
```

EXPLANATION

Accessing a missing key using square brackets raises a KeyError. The .get() method safely returns a default value, preventing program crashes.

Task 5: Infinite Loop – Wrong Condition

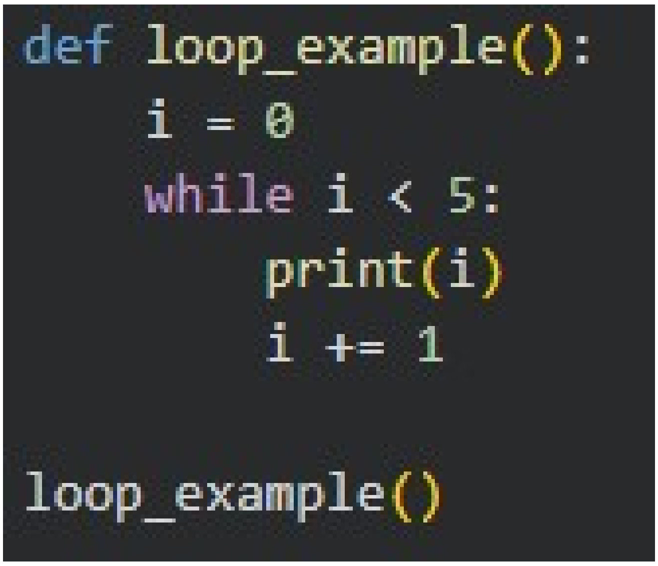
QUESTION

Analyze the loop that never terminates due to a missing increment and correct it.

PROMPT

Fix the infinite loop by updating the loop variable correctly.

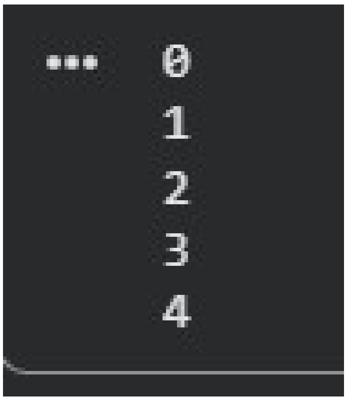
CODE



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

loop_example()
```

OUTPUT



```
0
1
2
3
4
```

EXPLANATION

The loop variable was not updated in the original code, causing an infinite loop. Incrementing the variable ensures that the loop condition eventually becomes false.

Task 6: Unpacking Error – Wrong Variables

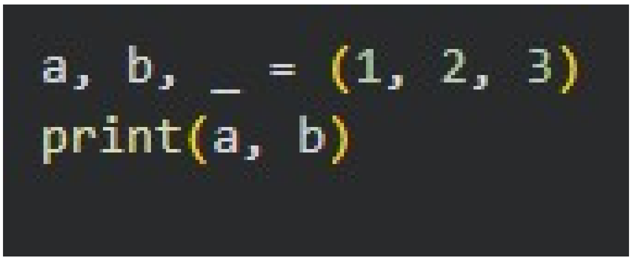
QUESTION

Analyze the tuple unpacking error where the number of variables does not match the tuple size.

PROMPT

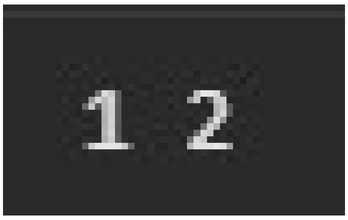
Correct the unpacking error by adjusting variables appropriately.

CODE



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

OUTPUT



```
1 2
```

EXPLANATION

Tuple unpacking requires matching the number of variables with values. The underscore _ is used to ignore extra values safely.

Task 7: Mixed Indentation – Tabs vs Spaces

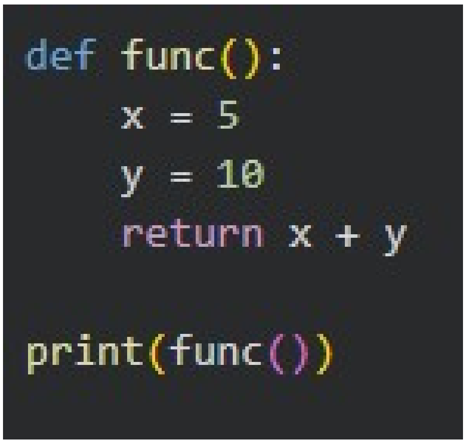
QUESTION

Analyze the code where inconsistent indentation breaks execution and fix it.

PROMPT

Correct the indentation to follow consistent spacing rules.

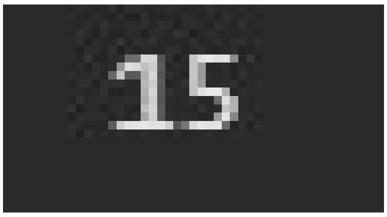
CODE



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

print(func())
```

OUTPUT



```
15
```

EXPLANATION

Python is indentation-sensitive. Mixing tabs and spaces leads to syntax errors. Consistent indentation improves readability and prevents execution issues.

Task 8: Import Error – Wrong Module Usage

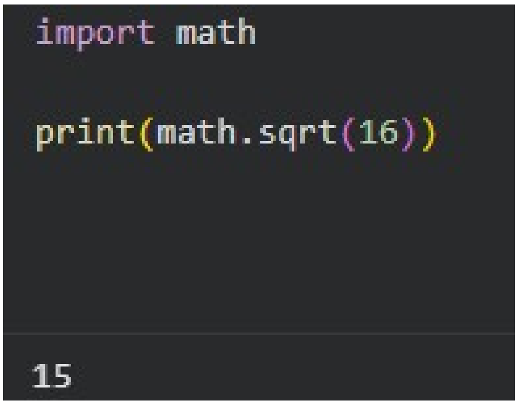
QUESTION

Analyze the code with an incorrect module import and fix it using AI assistance.

PROMPT

Correct the module name to ensure successful import and execution.

CODE

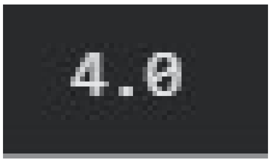


```
import math

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

```
15
```

OUTPUT



```
4.0
```

EXPLANATION

The module name was incorrect in the original code. Importing the correct built-in module resolves the error and allows the function to work properly.