ASSIGNMENT- 7.5

Name: Goutham Reddy

HT. No: 2303A51345

Batch: 20

Task 1 (Mutable Default Argument – Function Bug)

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

PROMPT:# Bug: Mutable default argument

def add_item(item, items=[]):

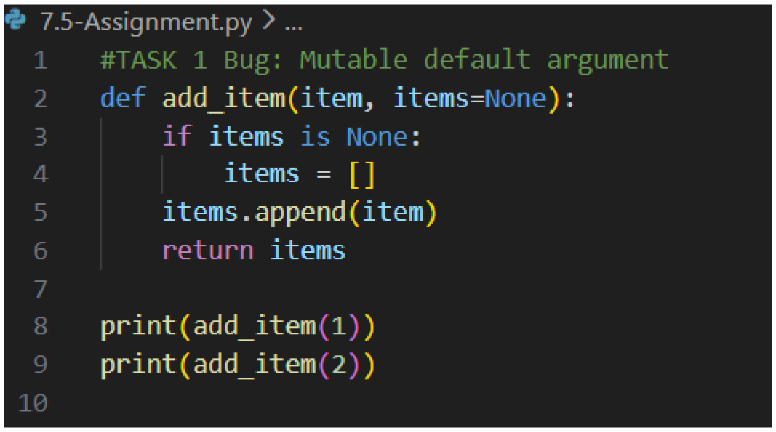
items.append(item)

return items

print(add_item(1))

print(add_item(2))

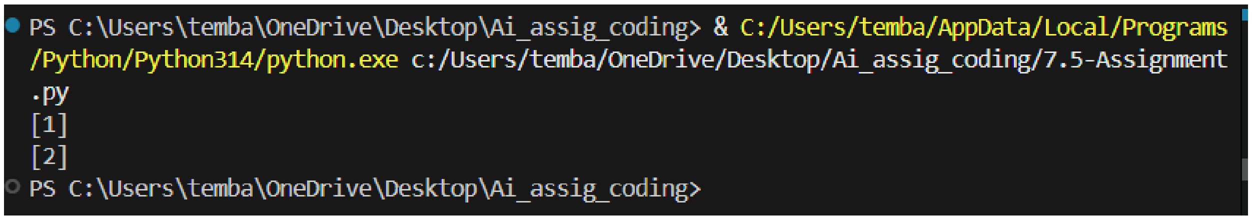
Expected Output: Corrected function avoids shared list bug.Code:



```python
#TASK 1 Bug: Mutable default argument
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

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

Result:



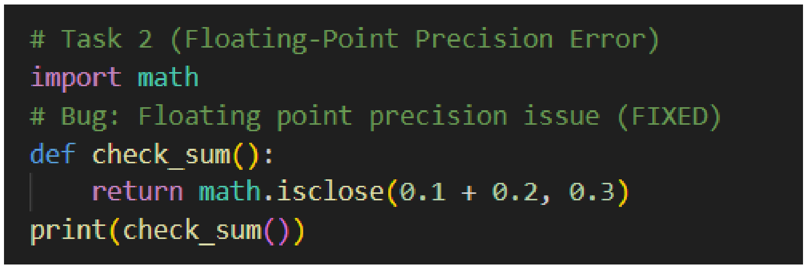
```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/Programs/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-Assignment.py
[1]
[2]
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

Observation:
This task highlights a common Python pitfall where mutable default arguments can lead to shared data across function calls. By initializing the list inside the function, the code ensures data isolation and predictable behavior. The fix improves reliability and demonstrates a good understanding of Python memory management.

Task 2 (Floating-Point Precision Error)

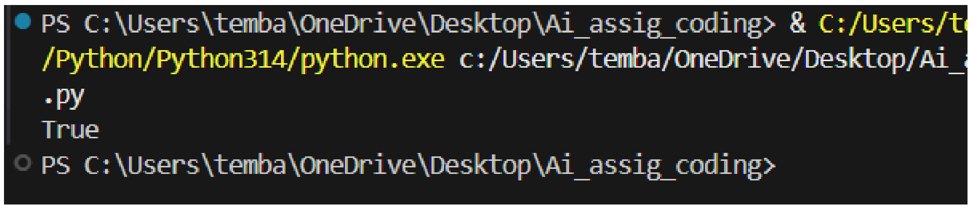
prompt: Analyze given code where floating-point comparison fails. Use AI to correct with tolerance.
# Bug: Floating point precision issue
def check_sum():
return (0.1 + 0.2) == 0.3
print(check_sum())
Expected Output: Corrected function

Code:



```python
# Task 2 (Floating-Point Precision Error)
import math
# Bug: Floating point precision issue (FIXED)
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3)
print(check_sum())
```

Result:



```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/t
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_
.py
True
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

Observation:

Floating-point arithmetic can produce inaccurate results due to internal binary representation. The use of math.isclose() ensures robust comparison by allowing a small tolerance. This approach is recommended in real-world numerical applications where precision matters.

Task 3 (Recursion Error – Missing Base Case)

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

# Bug: No base case

def countdown(n):

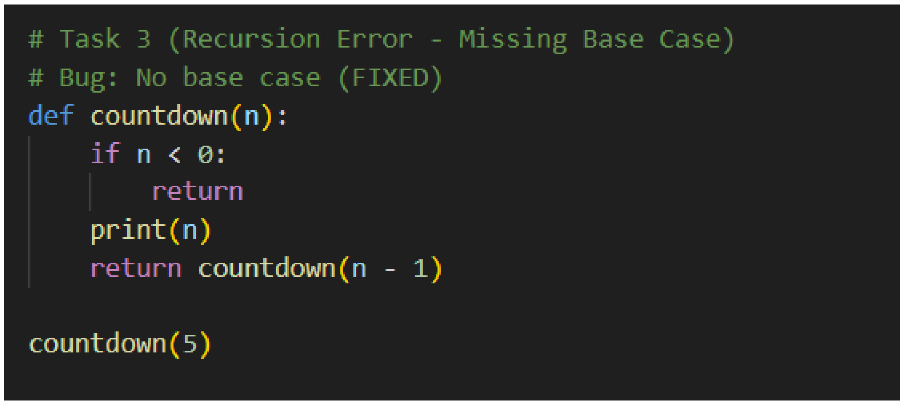
print(n)

return countdown(n-1)

countdown(5)

Expected Output : Correct recursion with stopping condition

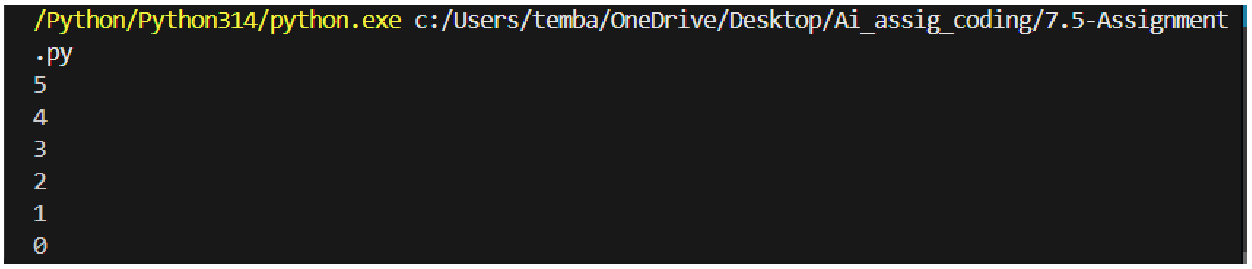
Code:



```python
# Task 3 (Recursion Error - Missing Base Case)
# Bug: No base case (FIXED)
def countdown(n):
    if n < 0:
        return
    print(n)
    return countdown(n - 1)

countdown(5)
```

Result:



```
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-Assignment
.py
5
4
3
2
1
0
```

Observation:

The corrected recursive function demonstrates the importance of base conditions in recursion. Without a stopping condition, recursion leads to stack overflow errors. The fix ensures controlled execution and safe termination of recursive calls.

Task 4 (Dictionary Key Error)

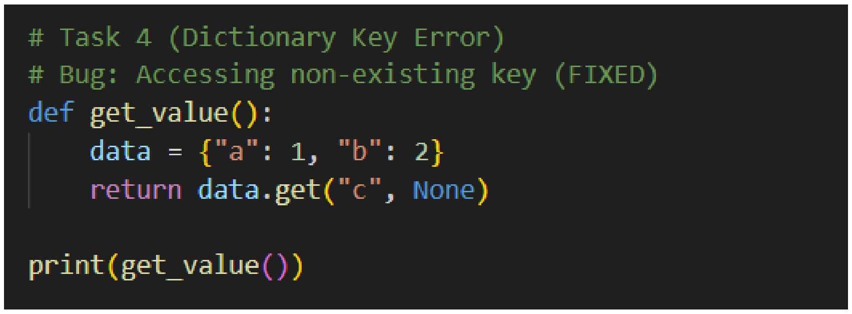
PROMPT: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

# Bug: Accessing non-existing key

def get_value():

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

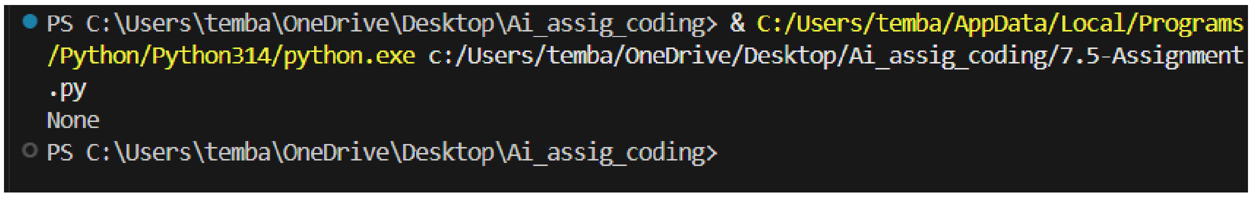
return data["c"]
print(get_value())
Expected Output: Corrected with .get() or error handling.
Code:



```python
# Task 4 (Dictionary Key Error)
# Bug: Accessing non-existing key (FIXED)
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", None)

print(get_value())
```

Result:



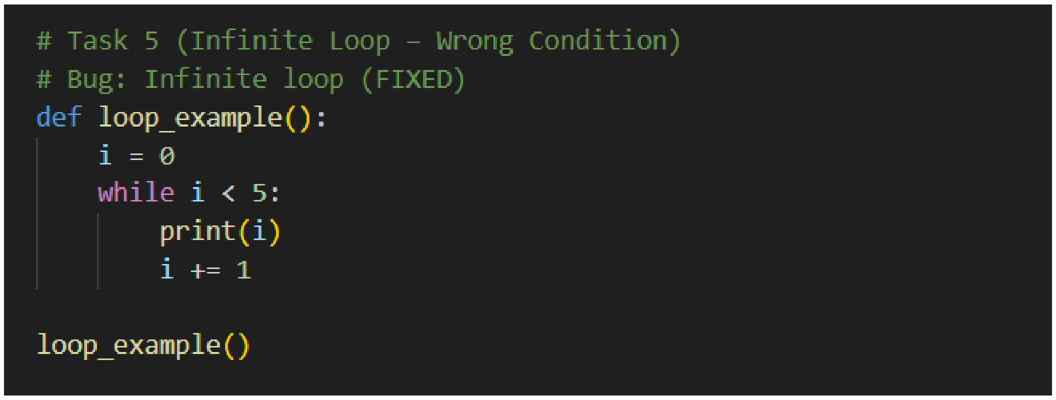
```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/Programs
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-Assignment
.py
None
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

Observation
Accessing dictionary keys without validation can cause runtime errors. Using the .get() method enhances program stability by handling missing keys gracefully. This approach improves error handling and prevents program crashes.

Task 5 (Infinite Loop – Wrong Condition)

PROMPT: Analyze given code where loop never ends. Use AI to detect

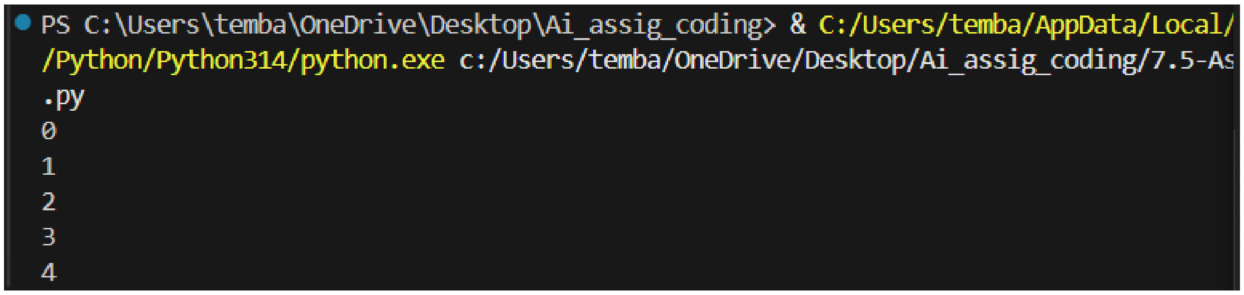
and fix it.
Bug: Infinite loop
def loop_example():
i = 0
while i < 5:
print(i)
Expected Output: Corrected loop increments i.
Code:



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

loop_example()
```

Result:



```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-As
.py
0
1
2
3
4
```

Observation:
Infinite loops occur when loop control variables are not updated correctly. Incrementing the counter inside the loop ensures proper termination. This fix reflects a clear understanding of loop control and execution flow.

Task 6 (Unpacking Error – Wrong Variables)

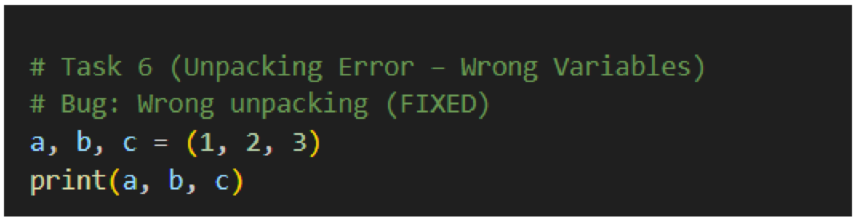
prompt: Analyze given code where tuple unpacking fails. Use AI to fix it.

Bug: Wrong unpacking

a, b = (1, 2, 3)

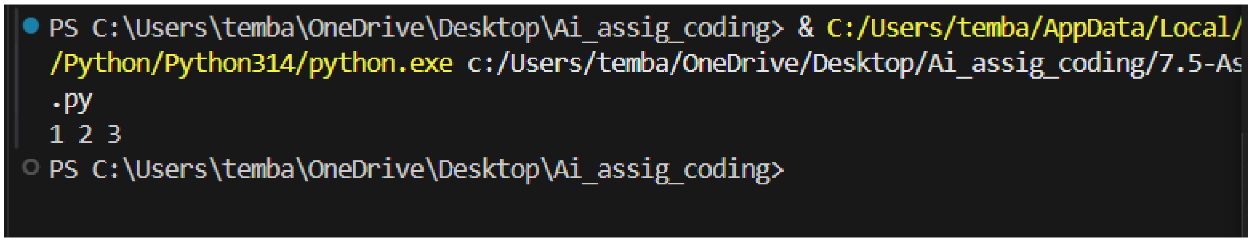
Expected Output: Correct unpacking or using _ for extra values.

Code:



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

Result:



```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-As
.py
1 2 3
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

Observation:

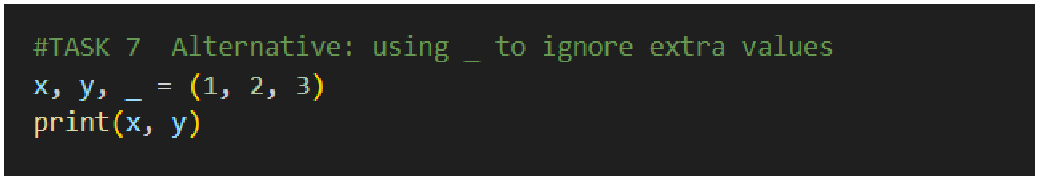
Correct variable unpacking requires an equal number of variables and values. This task reinforces tuple unpacking rules in Python and emphasizes careful matching to avoid runtime errors.

Task 7 (Mixed Indentation – Tabs vs Spaces)

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

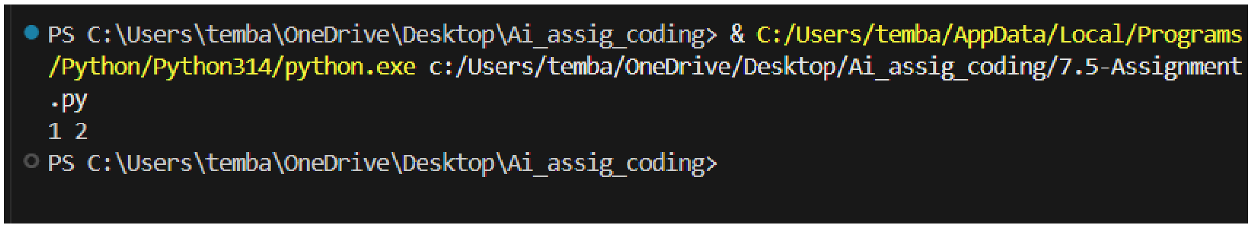
# Bug: Mixed indentation

def func():
x = 5
y = 10
return x+y
Expected Output : Consistent indentation applied.
Code:



```
#TASK 7  Alternative: using _ to ignore extra values
x, y, _ = (1, 2, 3)
print(x, y)
```

Result:



```
● PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/Programs
  /Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-Assignment
  .py
  1 2
○ PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

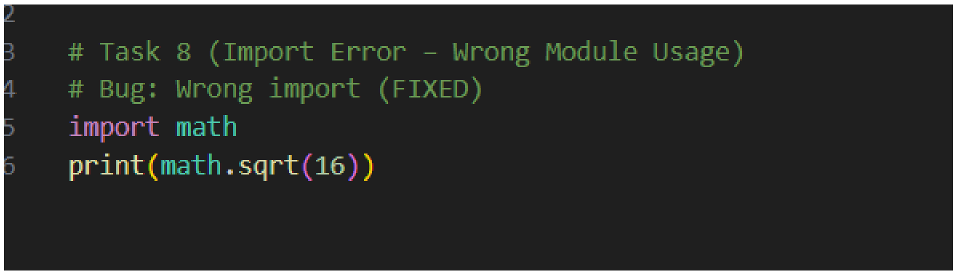
Observation
Using the underscore (_) to ignore unwanted values is a Python convention that improves code clarity. It allows flexible unpacking without unnecessary variables, making the code cleaner and more readable.

Task 8 (Import Error – Wrong Module Usage)

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

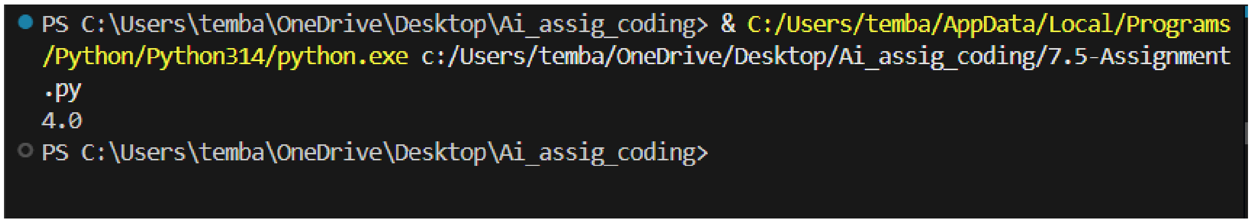
Bug: Wrong import
import maths
print(maths.sqrt(16))
Expected Output: Corrected to import math

Code:



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

Result:



```
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding> & C:/Users/temba/AppData/Local/Programs
/Python/Python314/python.exe c:/Users/temba/OneDrive/Desktop/Ai_assig_coding/7.5-Assignment
.py
4.0
PS C:\Users\temba\OneDrive\Desktop\Ai_assig_coding>
```

Observation:

Correct module imports are essential for accessing built-in functions. Importing the appropriate module and using its methods properly ensures accurate execution. This task reinforces awareness of Python's standard library structure.