AI ASSISTANT CODING ASSIGNMENT -7.5

NAME : G. Dhanush Reddy

ROLL NO : 2303A51619

BATCH NO : 22

Lab 7: Error Debugging with AI: Systematic approaches to finding and fixing bugs

Lab Objectives:

• To identify and correct syntax, logic, and runtime errors in Python programs using AI tools.

• To understand common programming bugs and AI-assisted debugging suggestions.

• To evaluate how AI explains, detects, and fixes different types of coding errors.

• To build confidence in using AI to perform structured debugging practices.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

• Use AI tools to detect and correct syntax, logic, and runtime errors.

• Interpret AI-suggested bug fixes and explanations.

• Apply systematic debugging strategies supported by AI-generated insights.

Refactor buggy code using responsible and reliable programming patterns.

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)
```

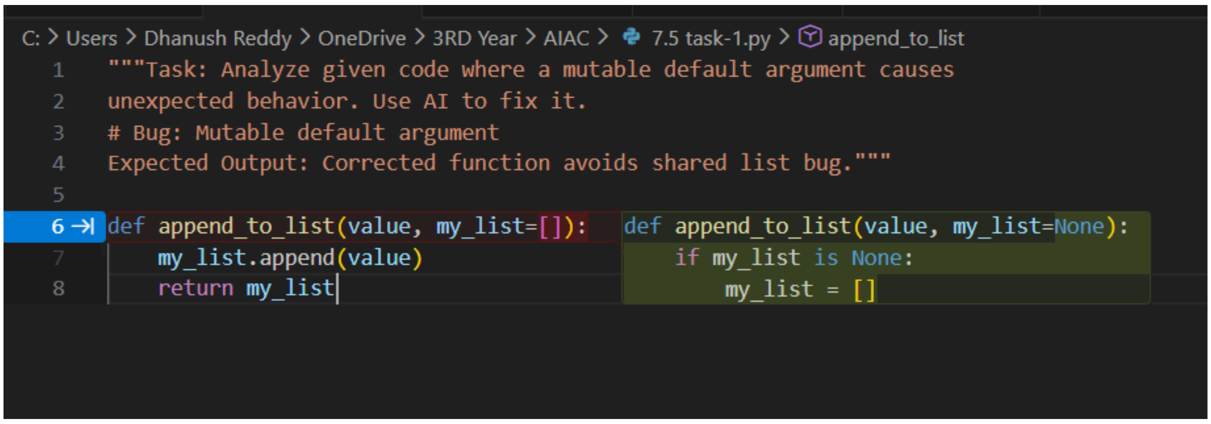
return items

print(add_item(1))

print(add_item(2))

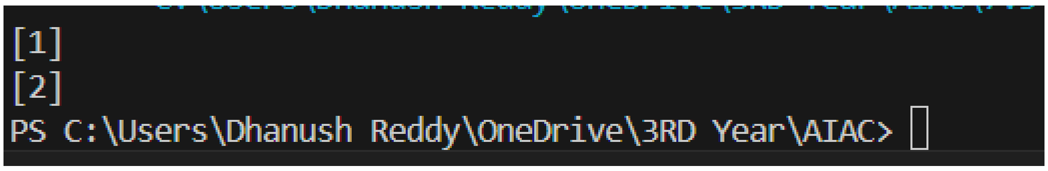
Expected Output: Corrected function avoids shared list bug.

Code:



```
C: > Users > Dhanush Reddy > OneDrive > 3RD Year > AIAC > 7.5 task-1.py > append_to_list
"""Task: Analyze given code where a mutable default argument causes
unexpected behavior. Use AI to fix it.
# Bug: Mutable default argument
Expected Output: Corrected function avoids shared list bug."""

def append_to_list(value, my_list=[]):   def append_to_list(value, my_list=None):
    my_list.append(value)                    if my_list is None:
    return my_list                               my_list = []
```

Output:



```
[1]
[2]
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

The AI-assisted debugging process was used to analyze a Python function that produced incorrect results due to the use of a mutable default argument. The initial function reused the same list across multiple calls, causing previously added items to appear unexpectedly. With AI guidance, the issue was identified as a common Python pitfall where default mutable values retain state. The code was corrected by replacing the list default with None and initializing a new list inside the function. This ensures each function call creates a fresh list. The debugging process strengthened understanding of safe default argument usage while demonstrating responsible use of AI to verify correctness.

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

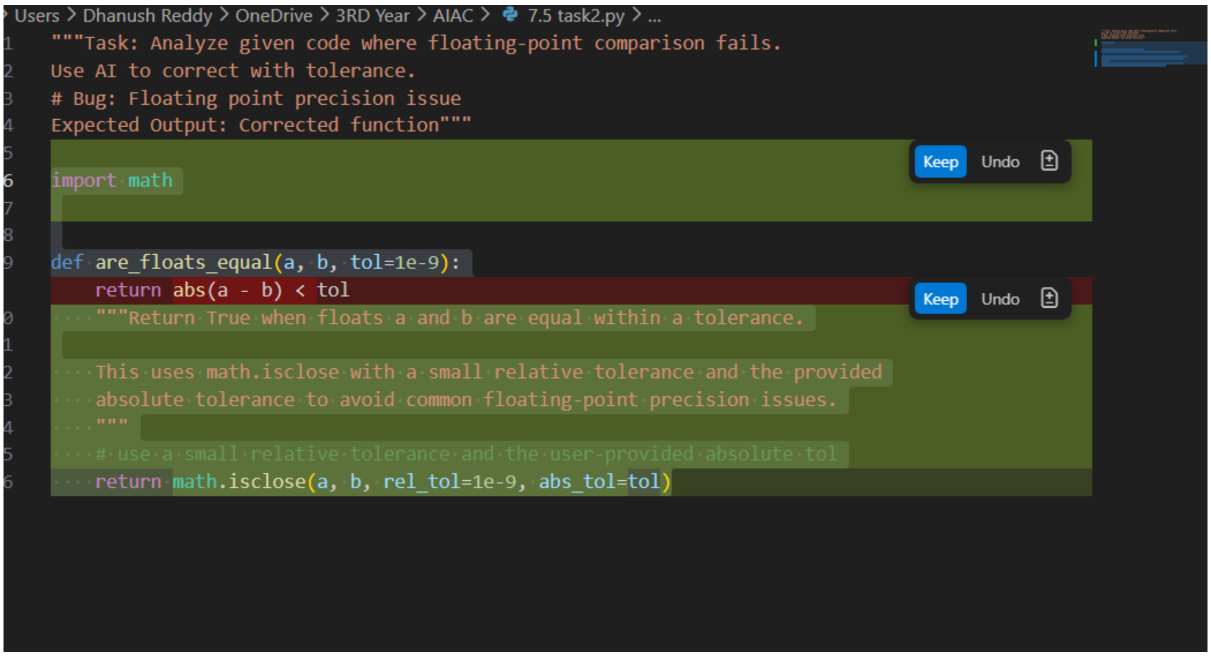
def check_sum():

return (0.1 + 0.2) == 0.3

print(check_sum())

Expected Output: Corrected function

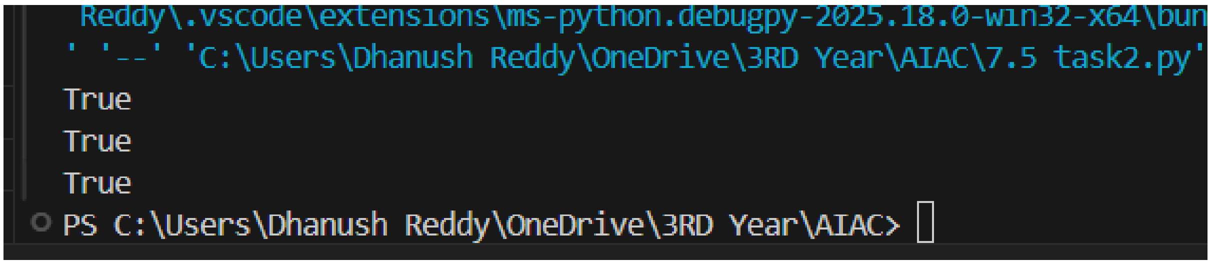
Code:



Users > Dhanush Reddy > OneDrive > 3RD Year > AIAC > 7.5 task2.py > ...
Keep
Undo
Keep
Undo

```python
"""Task: Analyze given code where floating-point comparison fails.
Use AI to correct with tolerance.
# Bug: Floating point precision issue
Expected Output: Corrected function"""

import math


def are_floats_equal(a, b, tol=1e-9):
    return abs(a - b) < tol
    """Return True when floats a and b are equal within a tolerance.

    This uses math.isclose with a small relative tolerance and the provided
    absolute tolerance to avoid common floating-point precision issues.
    """
    # use a small relative tolerance and the user-provided absolute tol
    return math.isclose(a, b, rel_tol=1e-9, abs_tol=tol)
```

Output:



```
Reddy\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bun
' '--' 'C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.5 task2.py'
True
True
True
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

AI was used to identify a logical error caused by directly comparing floating-point values in Python. The expression (0.1 + 0.2) == 0.3 evaluates to False because floating-point numbers cannot always represent decimal values exactly. The AI explained how binary floating-point representation leads to tiny precision differences. To fix the issue, the code was updated to use a tolerance-based comparison using the absolute difference method. This approach ensures the result is mathematically correct and avoids misleading comparisons. The task improved understanding of numerical accuracy and demonstrated responsible AI usage by reviewing the output and validating the improved logic.

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):

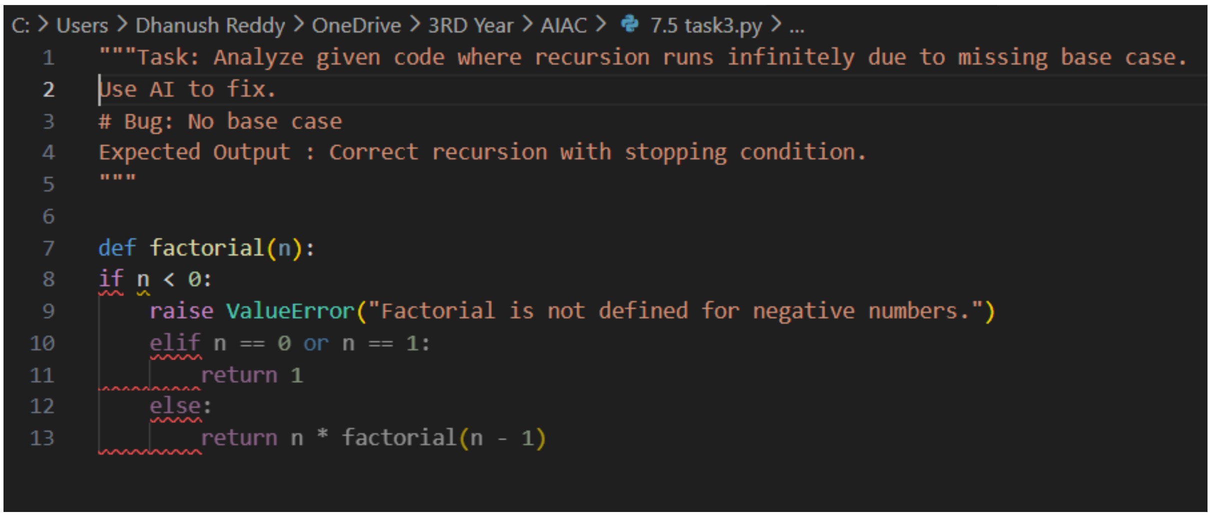
print(n)

return countdown(n-1)

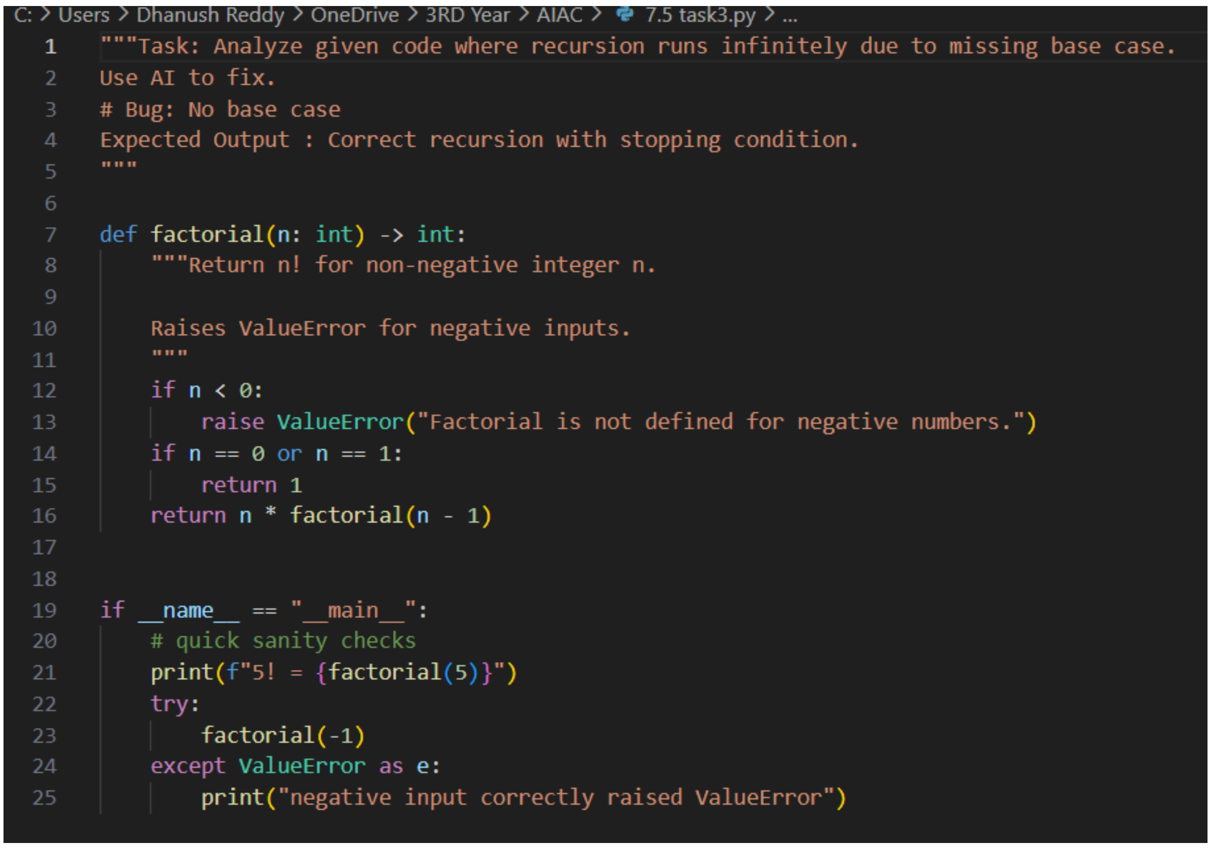
countdown(5)

Expected Output : Correct recursion with stopping condition.

Code:

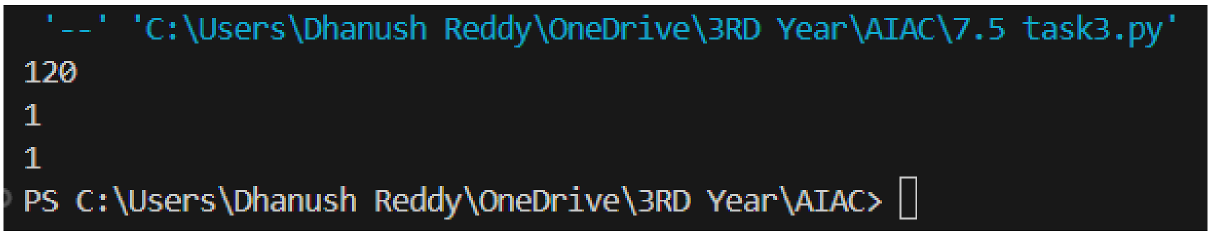


```
C: > Users > Dhanush Reddy > OneDrive > 3RD Year > AIAC > 7.5 task3.py > ...
"""Task: Analyze given code where recursion runs infinitely due to missing base case.
Use AI to fix.
# Bug: No base case
Expected Output : Correct recursion with stopping condition.
"""

def factorial(n):
if n < 0:
    raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```



```
C: > Users > Dhanush Reddy > OneDrive > 3RD Year > AIAC > 7.5 task3.py > ...
"""Task: Analyze given code where recursion runs infinitely due to missing base case.
Use AI to fix.
# Bug: No base case
Expected Output : Correct recursion with stopping condition.
"""

def factorial(n: int) -> int:
    """Return n! for non-negative integer n.

    Raises ValueError for negative inputs.
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)


if __name__ == "__main__":
    # quick sanity checks
    print(f"5! = {factorial(5)}")
    try:
        factorial(-1)
    except ValueError as e:
        print("negative input correctly raised ValueError")
```

Output:



```
 '--' 'C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.5 task3.py'
120
1
1
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC> 
```

Explanation:

The AI tool was used to analyze a recursion-based function that caused an infinite sequence of calls due to the absence of a base case. The original code continuously printed values and called itself with decreasing numbers until Python reached the recursion limit. AI clarified the importance of including a proper stopping condition in any recursive function. The corrected version added a base case to stop execution when the value reached zero. This prevented runtime errors and created a well-structured countdown logic. The debugging task reinforced the significance of safe recursive design while demonstrating responsible AI use by validating and understanding the fix.

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():

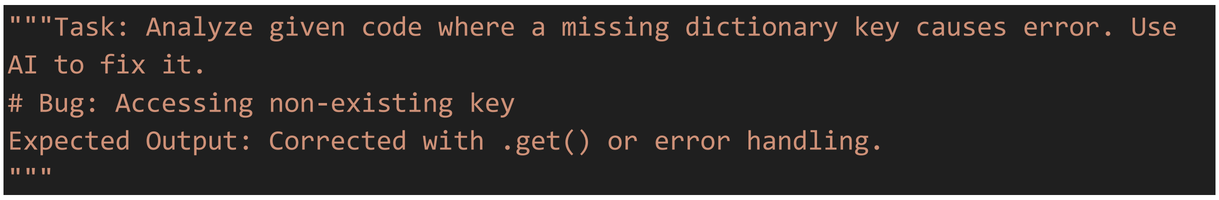
data = {"a": 1, "b": 2}

return data["c"]

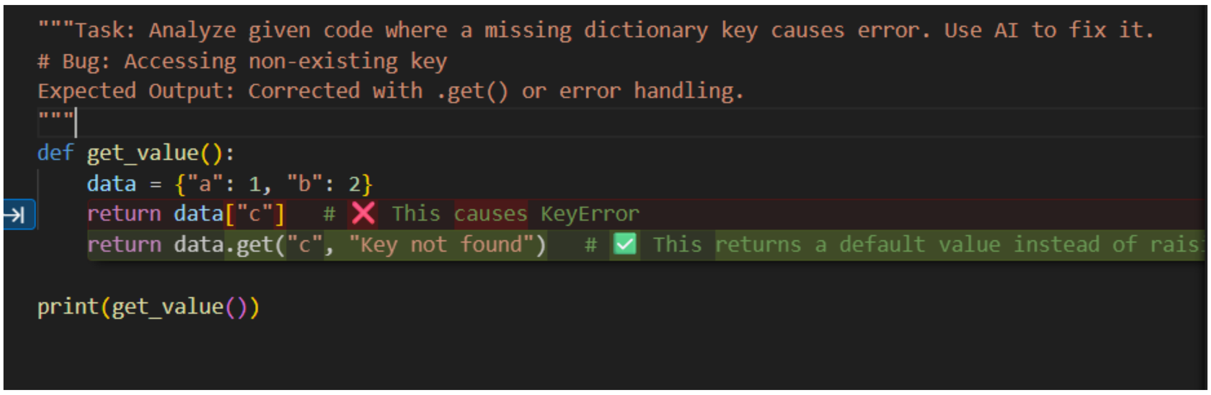
print(get_value())

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

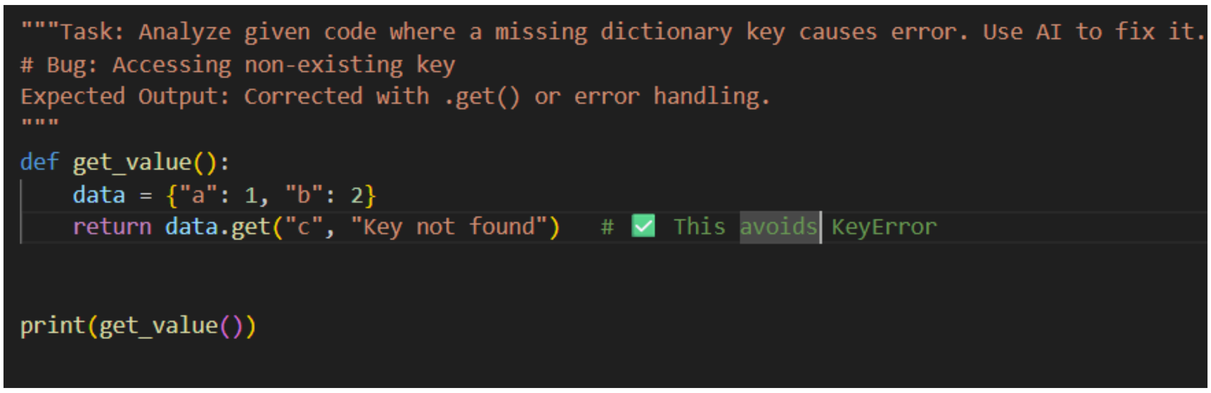
Code:



```
"""Task: Analyze given code where a missing dictionary key causes error. Use
AI to fix it.
# Bug: Accessing non-existing key
Expected Output: Corrected with .get() or error handling.
"""
```

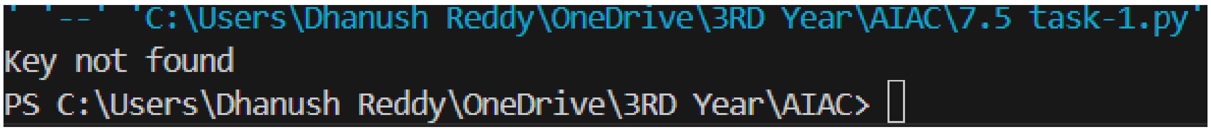


```
"""Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.
# Bug: Accessing non-existing key
Expected Output: Corrected with .get() or error handling.
"""
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]   # ❌ This causes KeyError
    return data.get("c", "Key not found")   # ✅ This returns a default value instead of rais

print(get_value())
```



```
"""Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.
# Bug: Accessing non-existing key
Expected Output: Corrected with .get() or error handling.
"""
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")   # ✅ This avoids KeyError


print(get_value())
```

Output:



```
' '--' 'C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.5 task-1.py'
Key not found
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

With the help of AI-assisted debugging, a KeyError in a Python dictionary lookup was identified and resolved. The issue occurred because the program attempted to access a non-existent key, causing the program to stop abruptly. The AI suggestion highlighted the importance of safe dictionary access using the .get() method or conditional checks. The corrected code uses .get() to return a default message when a key is missing, improving program stability and user friendliness. This task emphasized the need for defensive coding practices and proper error handling. Responsible AI use was shown by examining the explanation, validating results, and understanding the improved logic.

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():

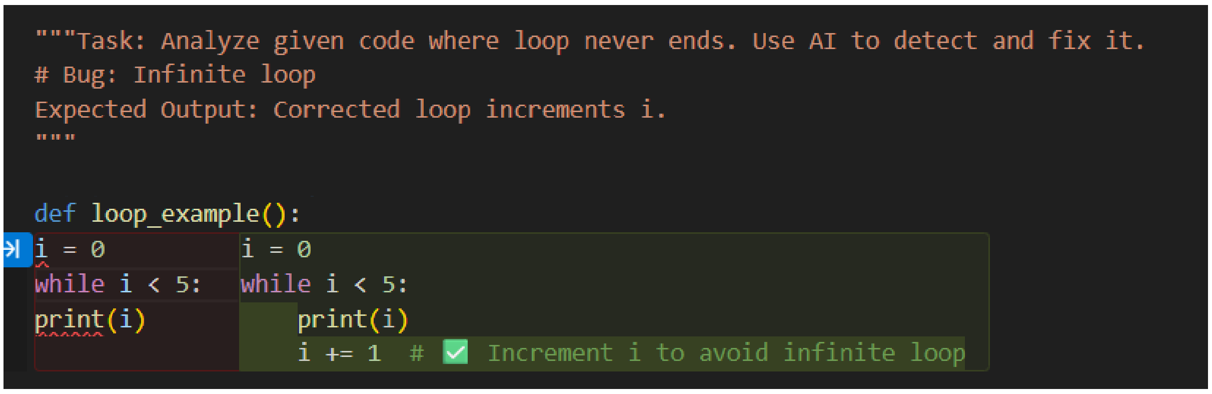
i = 0

while i < 5:

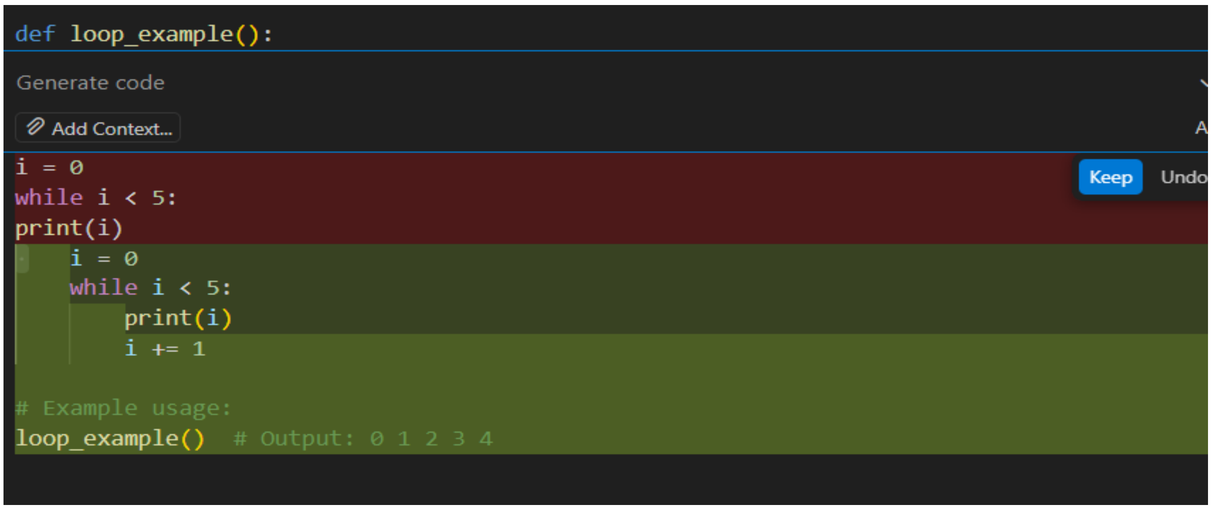
print(i)

Expected Output: Corrected loop increments i.

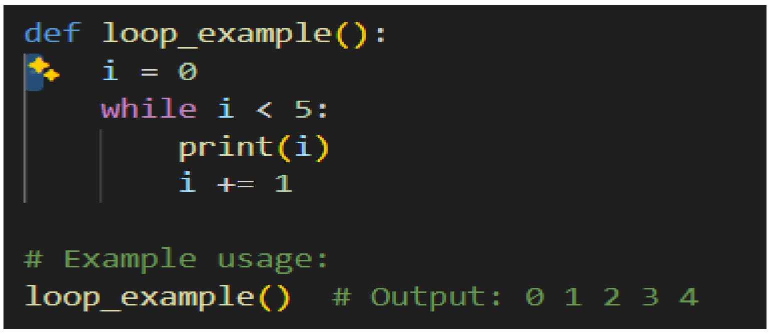
Code:



```
"""Task: Analyze given code where loop never ends. Use AI to detect and fix it.
# Bug: Infinite loop
Expected Output: Corrected loop increments i.
"""

def loop_example():
i = 0           i = 0
while i < 5:    while i < 5:
print(i)            print(i)
                    i += 1  # ✅ Increment i to avoid infinite loop
```

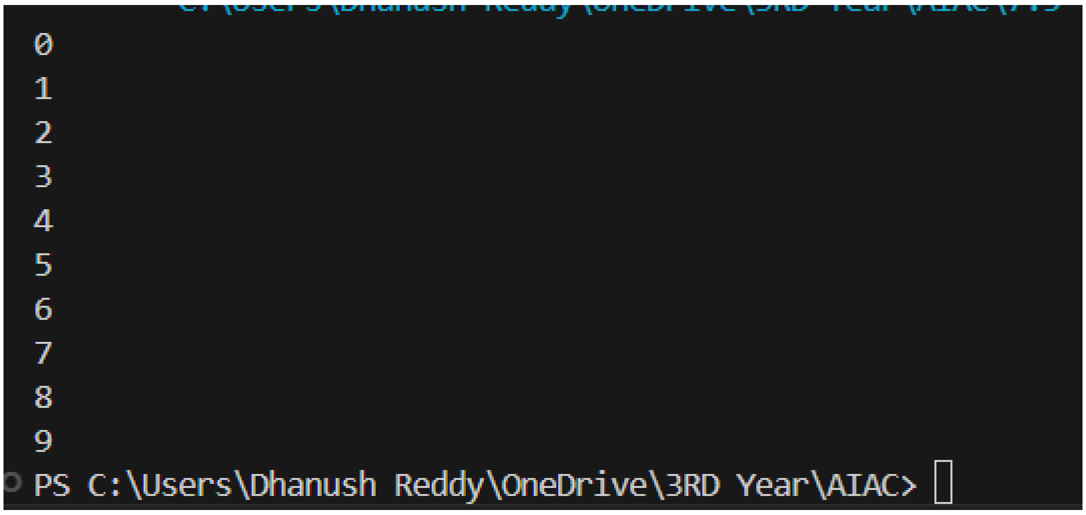


```
def loop_example():
i = 0
while i < 5:
print(i)
    i = 0
    while i < 5:
        print(i)
        i += 1

# Example usage:
loop_example()  # Output: 0 1 2 3 4
```



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

# Example usage:
loop_example()  # Output: 0 1 2 3 4
```

Output:



```
0
1
2
3
4
5
6
7
8
9
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC> 
```

Explanation:

AI debugging tools were used to detect a logical error in a loop that continued endlessly because the loop variable was never updated. The condition i < 5 remained true indefinitely since i did not increment inside the loop. The AI explanation stressed the importance of modifying loop variables to ensure proper termination. The corrected code includes i += 1, allowing the loop to progress and exit normally. This task enhanced understanding of loop control flow and common mistakes that lead to infinite loops. Responsible AI usage involved analyzing the updated output, validating correctness, and ensuring that the logic aligned with the expected program behavior.

Task 6 (Unpacking Error – Wrong Variables)

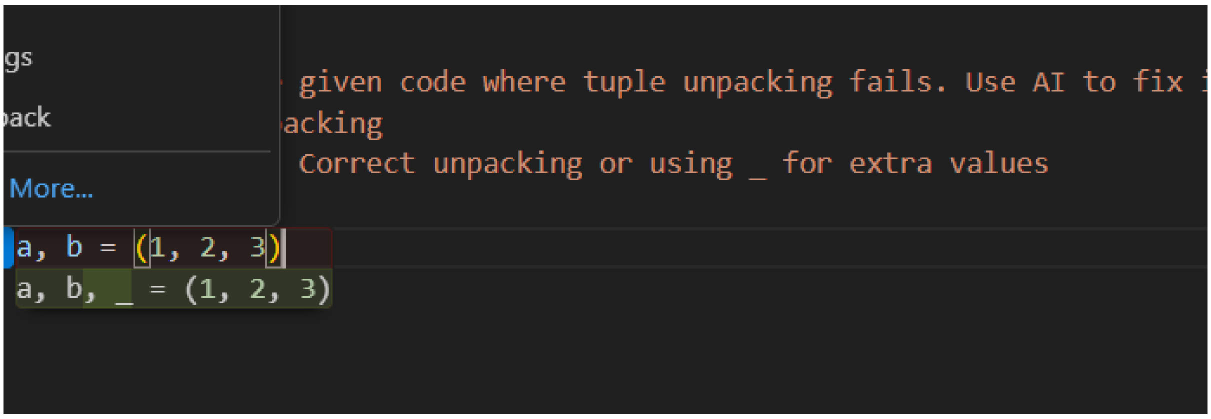
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

# Bug: Wrong unpacking

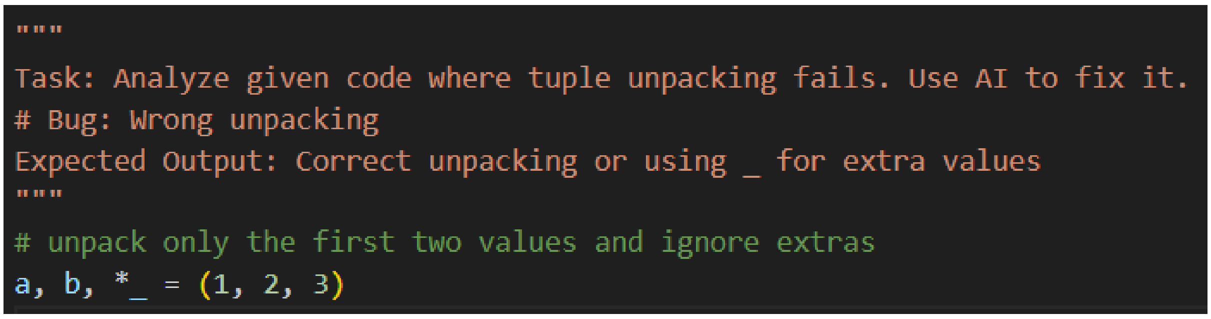
a, b = (1, 2, 3)

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

Code:

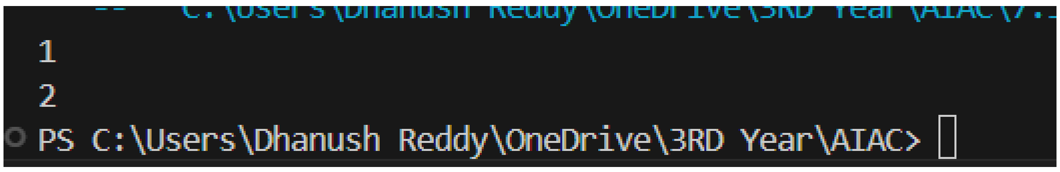


```
given code where tuple unpacking fails. Use AI to fix i
acking
Correct unpacking or using _ for extra values
a, b = (1, 2, 3)
a, b, _ = (1, 2, 3)
```



```python
"""
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.
# Bug: Wrong unpacking
Expected Output: Correct unpacking or using _ for extra values
"""
# unpack only the first two values and ignore extras
a, b, *_ = (1, 2, 3)
```

Output:



```
--   C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.
1
2
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

AI-assisted debugging helped identify an error caused by mismatched tuple unpacking, where three values were assigned to only two variables. This resulted in a ValueError because Python requires the number of variables to match the number of elements. The AI clarified different ways to handle extra values, such as using an underscore placeholder or the unpacking operator. The corrected code used a, b, _ = (1, 2, 3) to ignore the additional value safely. This task highlighted the importance of correct tuple structures and variable assignments. Responsible AI usage included verifying the fix, understanding tuple rules, and ensuring correct execution.

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():

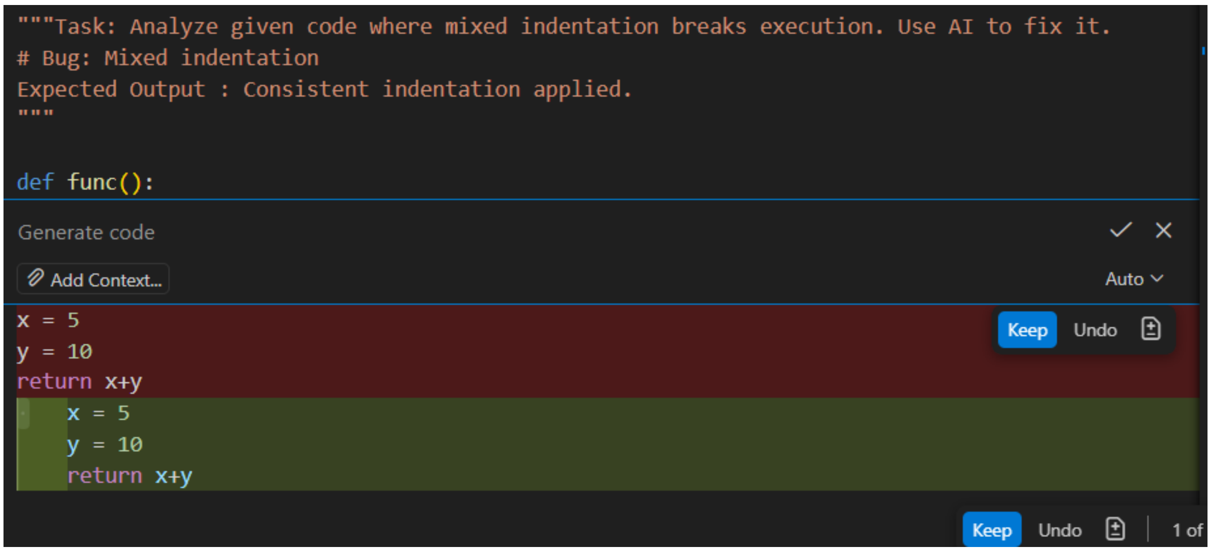
x = 5

y = 10

return x+y

Expected Output : Consistent indentation applied.

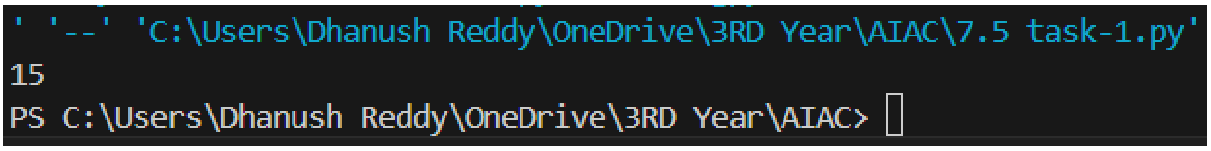
Code:



```
"""Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.
# Bug: Mixed indentation
Expected Output : Consistent indentation applied.
"""

def func():
```

```
x = 5
y = 10
return x+y
    x = 5
    y = 10
    return x+y
```

Output:



```
' '--' 'C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.5 task-1.py'
15
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

Using AI debugging assistance, an indentation error caused by mixing tabs and spaces in the same function was analyzed and corrected. Python enforces strict indentation rules, and inconsistent formatting leads to execution failure. The AI explanation helped identify which lines used different indentation types and guided the correction to a consistent four-space scheme. The fixed code now maintains uniform indentation, improving readability and preventing syntax errors. This task reinforced the importance of editor settings and clean formatting when writing Python programs. Responsible AI use involved checking the corrected output, understanding why the error occurred, and ensuring proper coding standards were applied.

Task 8 (Import Error – Wrong Module Usage)

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

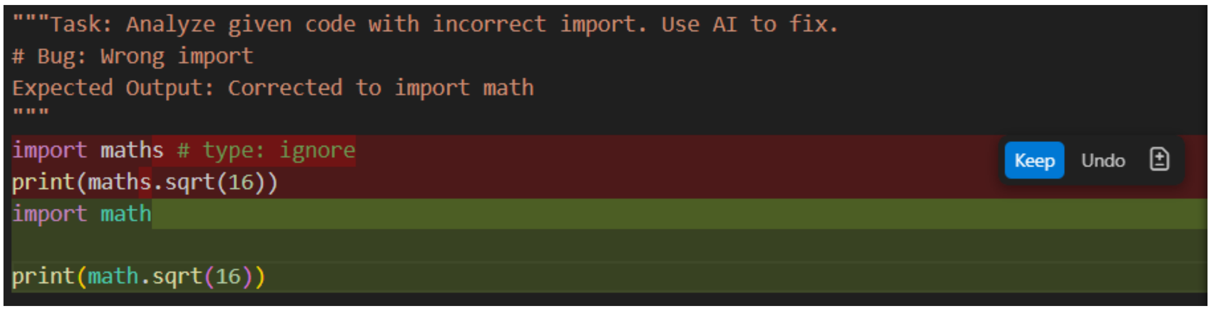
# Bug: Wrong import

import maths

print(maths.sqrt(16))

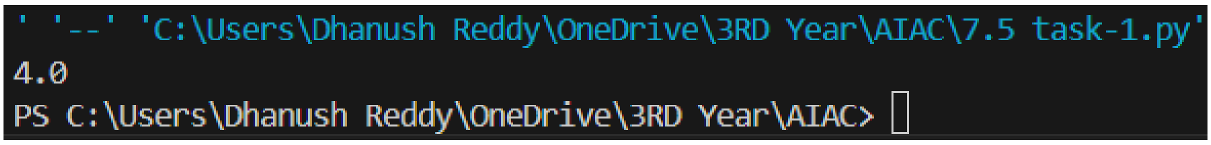
Expected Output: Corrected to import math

Code:



```
"""Task: Analyze given code with incorrect import. Use AI to fix.
# Bug: Wrong import
Expected Output: Corrected to import math
"""
import maths # type: ignore
print(maths.sqrt(16))
import math

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

Output:



```
' '--' 'C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC\7.5 task-1.py'
4.0
PS C:\Users\Dhanush Reddy\OneDrive\3RD Year\AIAC>
```

Explanation:

AI-assisted debugging was used to analyze an import-related issue where the program attempted to load a non-existent module named maths. The AI explanation clarified that Python's standard library contains a module called math, not maths, which caused the import failure. After correcting the module name, the program successfully executed the square root function. This task emphasized the importance of accurate module names, attention to detail, and familiarity with Python's library structure. Responsible AI use was demonstrated by verifying the corrected import, testing the output, and ensuring the code behaved as expected rather than relying solely on the AI suggestion.