Assignment 7.5

G SAIANIRUDH

2303A51891

Batch 04

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 :

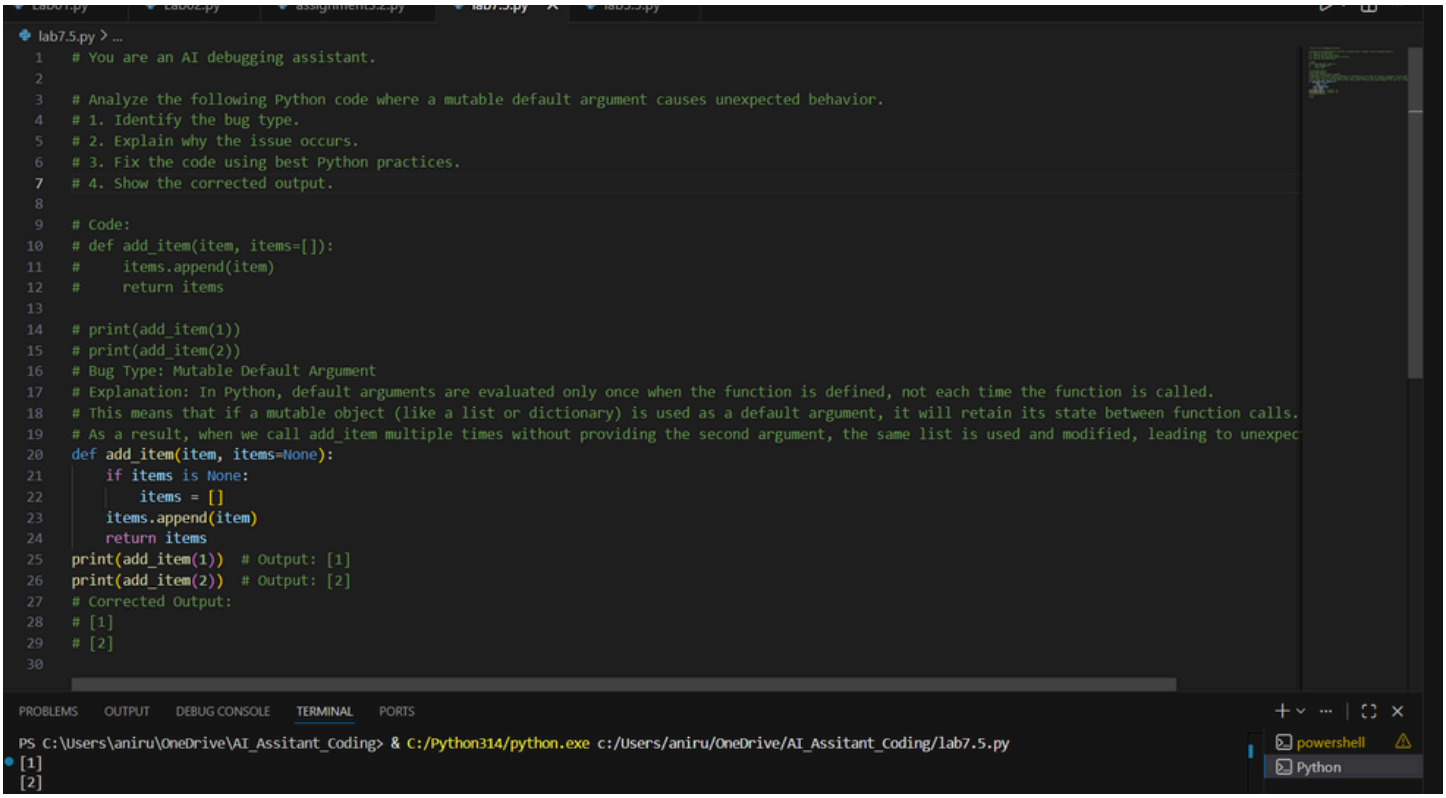
Task 1 (Mutable Default Argument – Function Bug)

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



Explanation / Justification:

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 previously added elements to persist unexpectedly. To avoid this, None is used as the default value and a new list is created inside the function for each call.

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:

```
#TASK 2
# You are an AI debugging assistant.

# Analyze the Python code where floating-point comparison fails.
# 1. Identify the logical issue.
# 2. Explain floating-point precision error.
# 3. Fix the code using tolerance or math methods.
# 4. Show the corrected output.

# Code:
# def check_sum():
#     return (0.1 + 0.2) == 0.3

# print(check_sum())
import math
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9)
print(check_sum())  # Output: True
# Corrected Output:
# True

```

Explanation / Justification:

Floating-point numbers are stored in binary format, which cannot represent some decimal values exactly. As a result, operations like 0.1 + 0.2 do not produce an exact 0.3, leading to incorrect comparisons. This issue is resolved by comparing numbers using a tolerance value or by using functions like math.isclose() to handle precision errors safely.

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:

```
#Task 3
You are an AI debugging assistant.

Analyze the recursive Python function that runs infinitely.
1. Identify the missing condition.
2. Explain why infinite recursion occurs.
3. Fix the function by adding a base case.
4. Show the corrected output.

Code:
def countdown(n):
    print(n)
    return countdown(n-1)

countdown(5)'''
def countdown(n):
    if n <= 0:  # Base case to stop recursion
        print("Countdown finished!")
        return
    print(n)
    return countdown(n-1)
countdown(5)  # Output: 5, 4, 3, 2, 1, Countdown finished!
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assitant_Coding/lab7.5.py
Countdown finished!
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assitant_Coding/lab7.5.py
5
4
3
2
1
Countdown finished!
```

Explanation / Justification:
Recursion requires a base case to stop further function calls. In the given code, the function keeps calling itself without any stopping condition, leading to infinite recursion and a runtime error. Adding a proper base case ensures that the recursion terminates when a specific condition is met.

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 :

```
'''You are an AI debugging assistant.

Analyze the Python code where accessing a dictionary key causes an error.
1. Identify the runtime error.
2. Explain why the error occurs.
3. Fix the code using safe dictionary access or exception handling.
4. Show the corrected output.

Code:
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]

print(get_value())'''
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")  # Safe access with default message
print(get_value())  # Output: Key not found
# Corrected Output:
# Key not found
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users/aniru/OneDrive/AI_Assitant_Coding/la
Key not found
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

Explanation / Justification:

Accessing a key that does not exist in a dictionary raises a KeyError. In the given code, the key "c" is not present in the dictionary. This issue is fixed by using the .get() method or exception handling, which allows safe access to dictionary values without causing runtime errors.

Task 5 (Infinite Loop – Wrong Condition)

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

and fix it.

Bug: Infinite loop

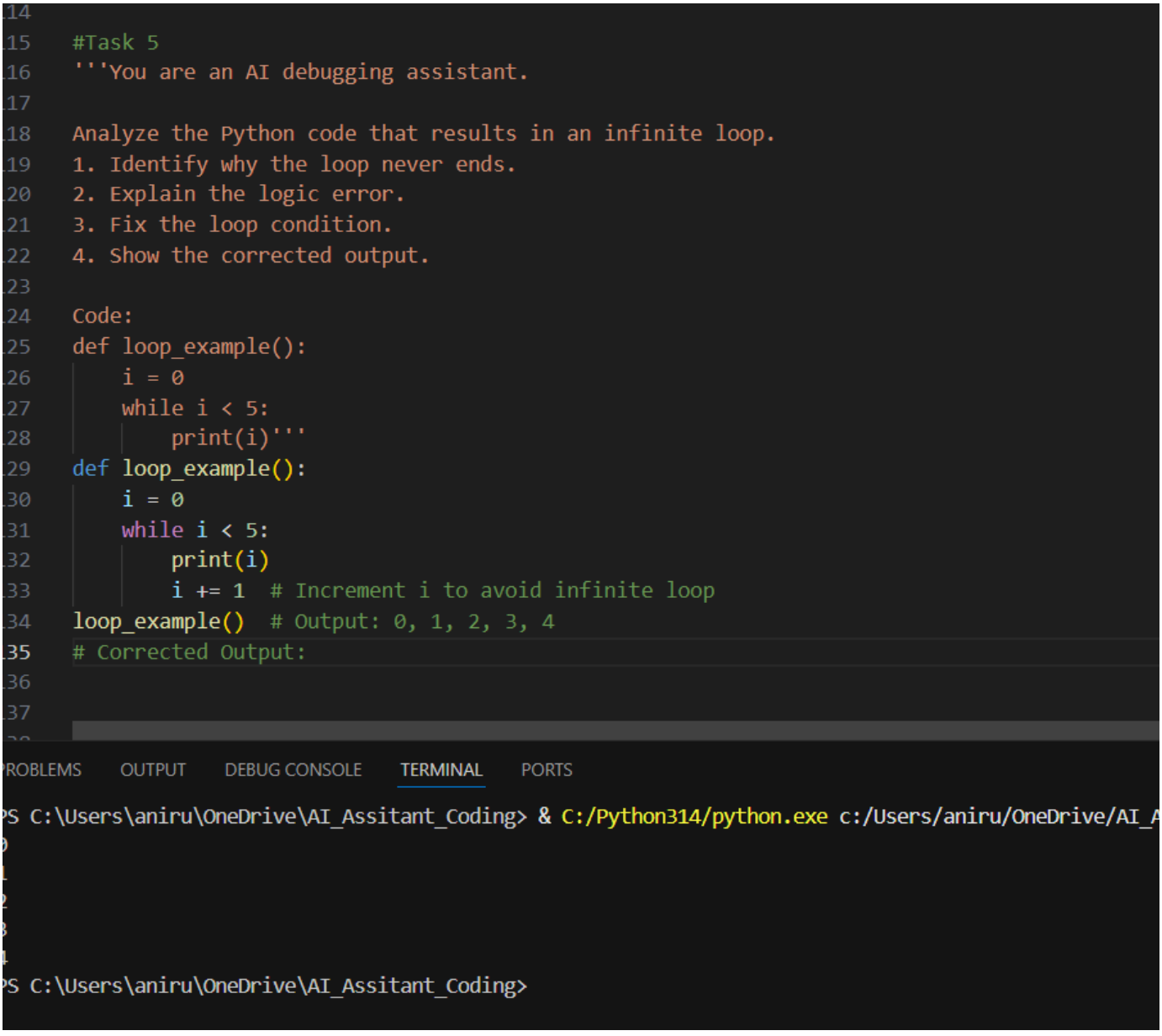
def loop_example():

i = 0

while i < 5:

print(i)

Expected Output: Corrected loop increments i.



Explanation / Justification:

An infinite loop occurs when the loop condition never becomes false. In this case, the loop variable i is not updated inside the loop, so the condition i < 5 remains true forever. Incrementing the loop variable ensures that the condition eventually becomes false and the loop terminates correctly.

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

```
#Task 6
'''You are an AI debugging assistant.

Analyze the Python code where tuple unpacking fails.
1. Identify the error type.
2. Explain why unpacking fails.
3. Fix the code using correct unpacking.
4. Show the corrected output.

Code:
a, b = (1, 2, 3)
'''
a, b, c = (1, 2, 3)  # Correct unpacking with three variables
print(a, b, c)  # Output: 1, 2, 3
# Corrected Output:
# 1 2 3
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\aniru\OneDrive\AI_Assitant_Coding> & C:/Python314/python.exe c:/Users
1 2 3
PS C:\Users\aniru\OneDrive\AI_Assitant_Coding>
```

Explanation / Justification:

Tuple unpacking requires the number of variables on the left-hand side to match the number of elements in the tuple. In the given code, three values are assigned to only two variables, causing a ValueError. This can be fixed by using the correct number of variables or by ignoring extra values using _ or the * operator.

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():

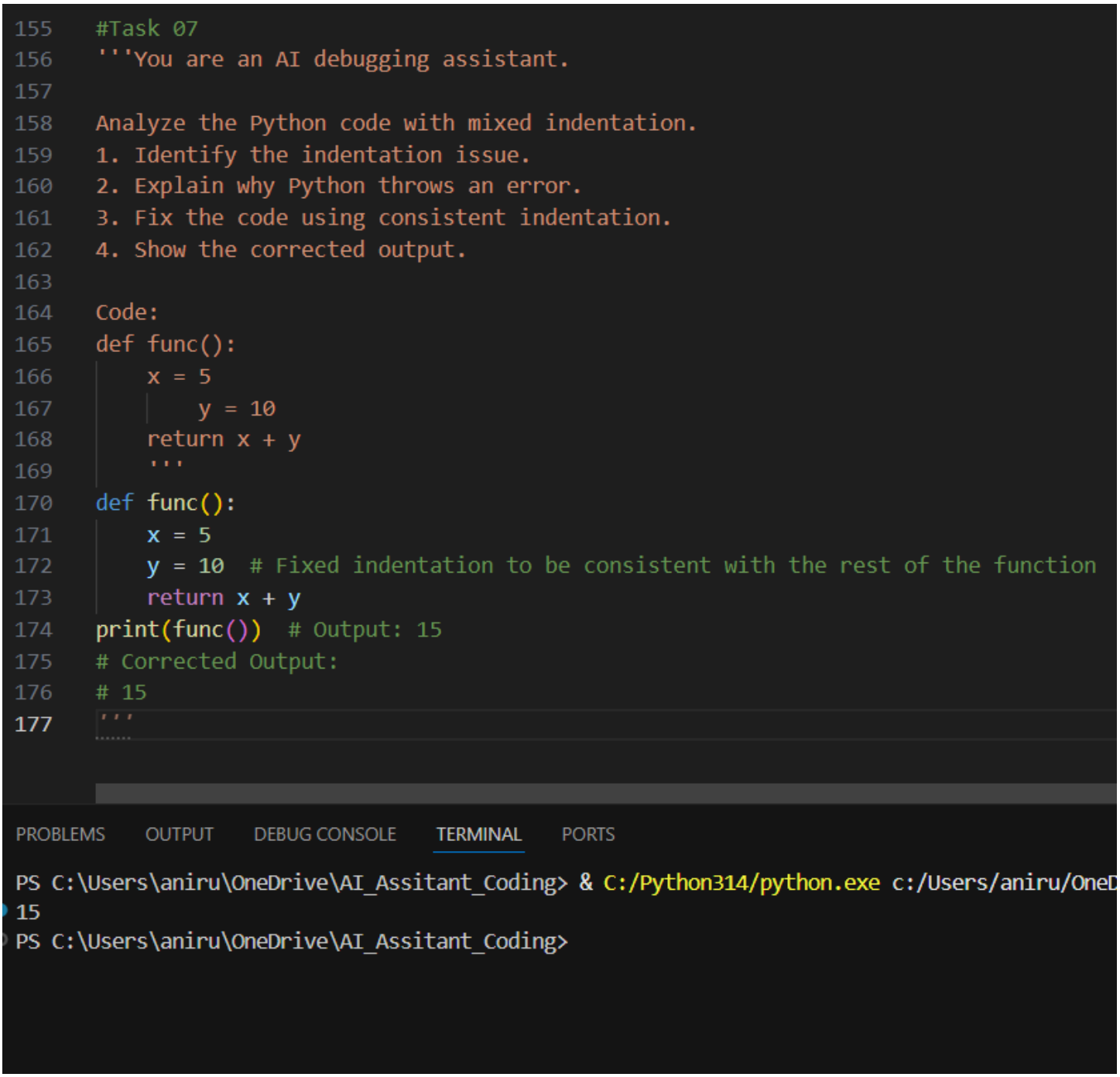
x = 5

y = 10

return x+y

Expected Output : Consistent indentation applied.

Code



Explanation / Justification:
Python uses indentation to define code blocks. Mixing tabs and spaces or incorrect indentation leads to syntax or indentation errors. In the given code, inconsistent indentation breaks execution. Using a consistent indentation style (preferably 4 spaces) ensures proper block structure and avoids indentation-related errors.

Task 8 (Import Error – Wrong Module Usage)

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

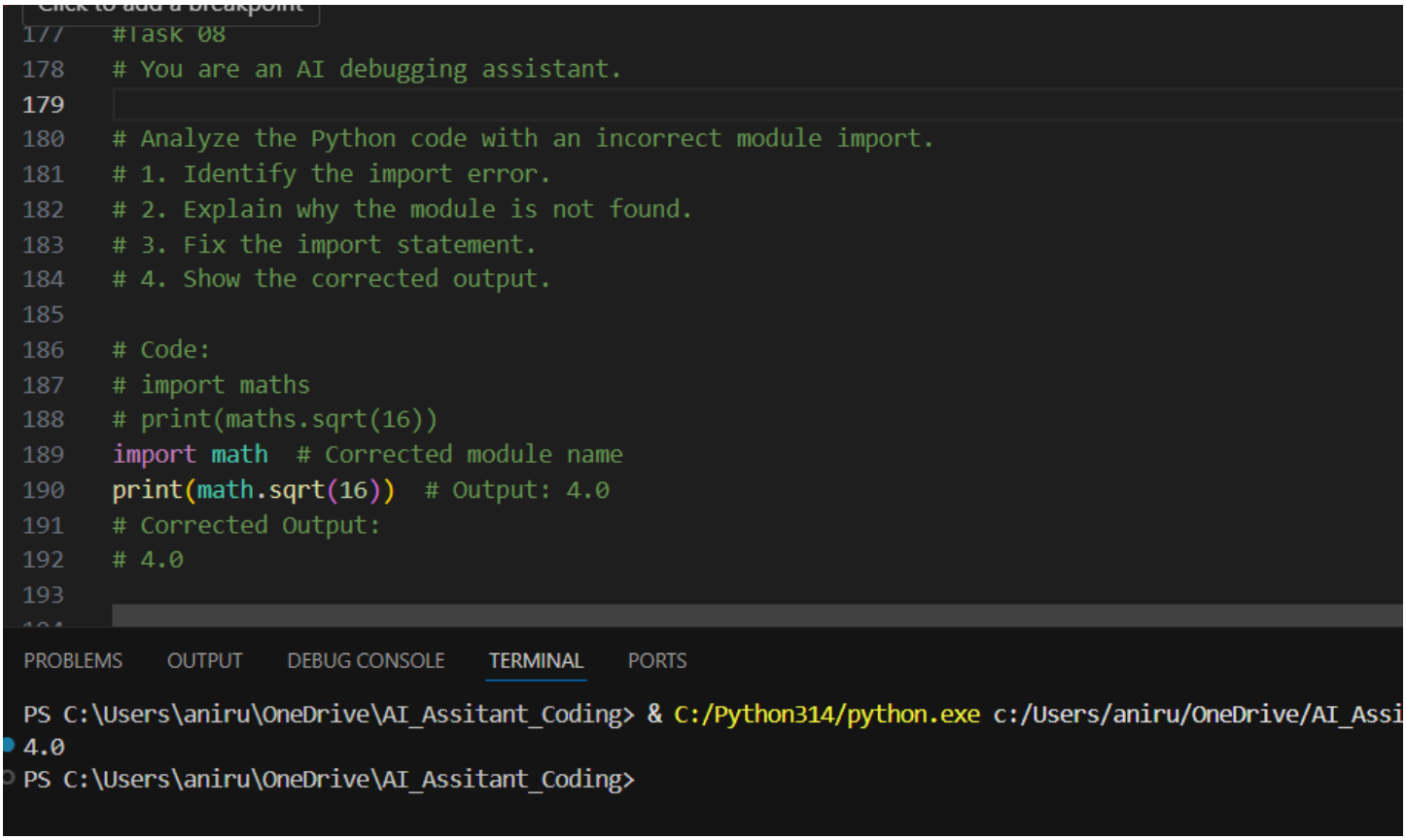
Bug: Wrong import

import maths

print(maths.sqrt(16))

Expected Output: Corrected to import math

Code



Explanation / Justification:

Python raises an ImportError when a module that does not exist or is incorrectly named is imported. In the given code, maths is not a valid standard Python module. Correcting the module name to math resolves the issue and allows access to mathematical functions like sqrt().

-- THANK YOU --