AI ASSITANT CODING

ASSIGNMENT-7.5

Name: B.Akhira Nandhini

Ht.no: 2303A51516

Batch:22

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)

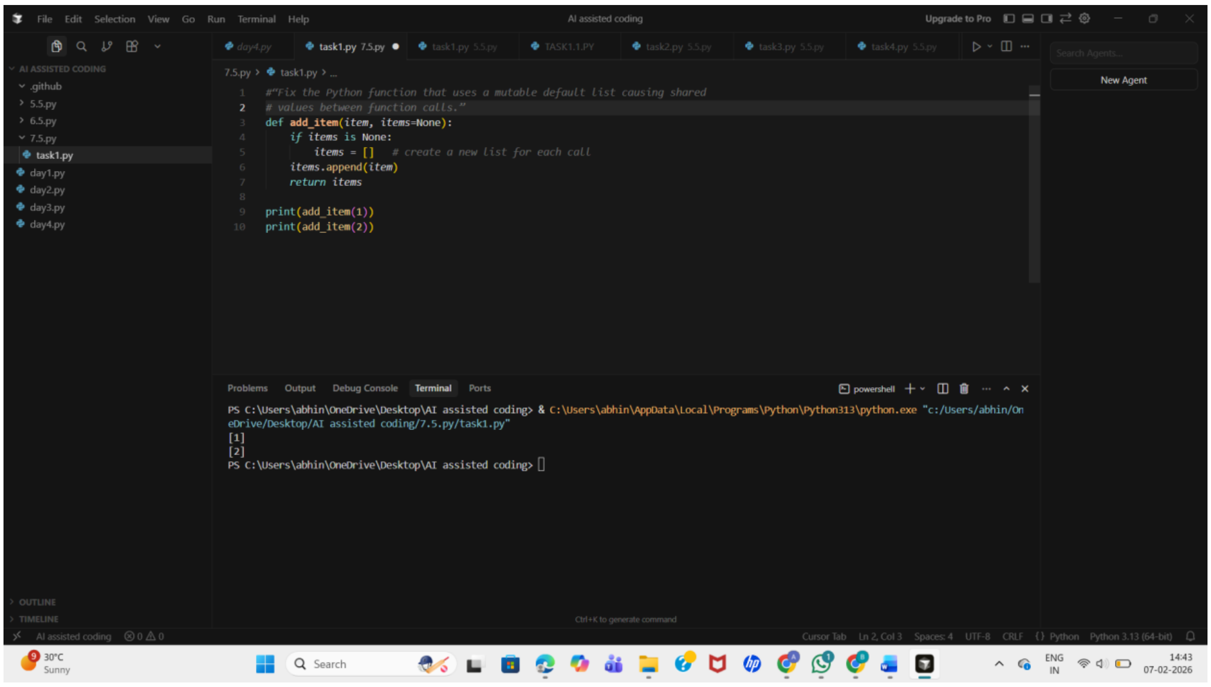
return items

print(add_item(1))

print(add_item(2))

Expected Output: Corrected function avoids shared list bug.

Code:



```python
#"Fix the Python function that uses a mutable default list causing shared
# values between function calls."
def add_item(item, items=None):
    if items is None:
        items = []   # create a new list for each call
    items.append(item)
    return items

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

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/task1.py"
[1]
[2]
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- A list is used as a default argument. In Python, default arguments are evaluated only once when the function is defined, not each time the function is called.
- Because lists are mutable, the same list is shared across all function calls, causing values from previous calls to persist.
- To fix this issue, None is used as the default value instead of a list. Inside the function, a new list is created when the argument is None.
- This ensures that each function call uses a fresh list and prevents the shared list bug.

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

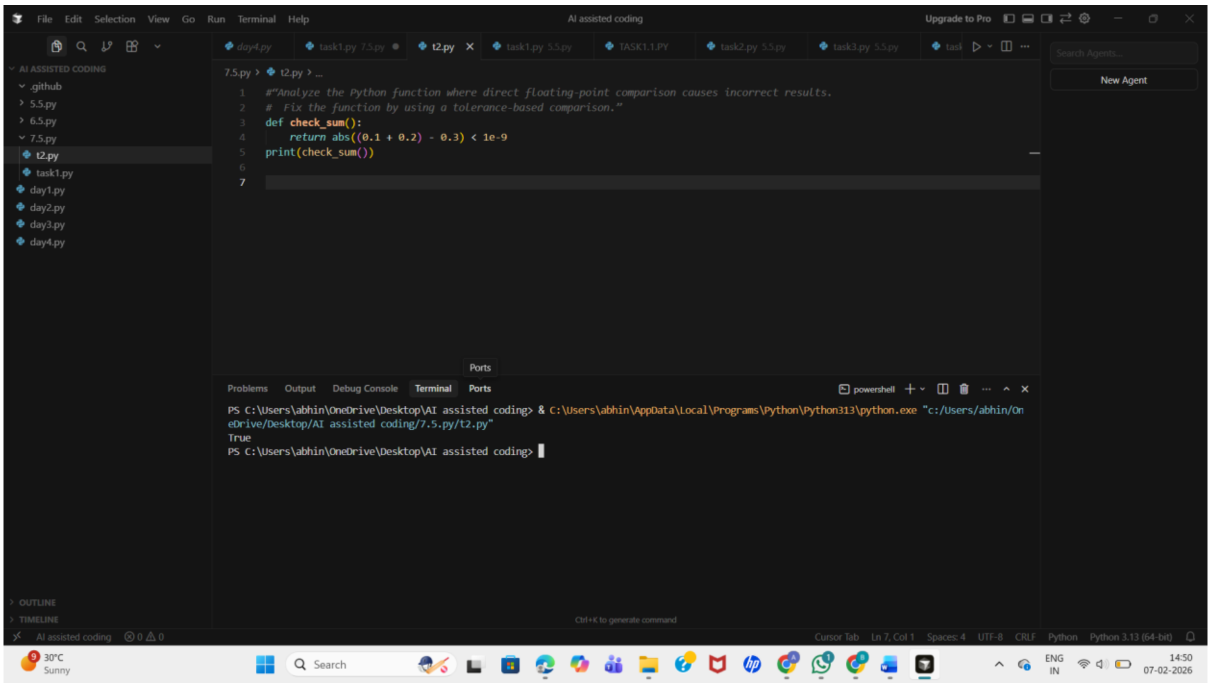
def check_sum():

return (0.1 + 0.2) == 0.3

print(check_sum())

Expected Output: Corrected function

Code:



```python
#"Analyze the Python function where direct floating-point comparison causes incorrect results.
#  Fix the function by using a tolerance-based comparison."
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9
print(check_sum())
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t2.py"
True
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- Floating-point numbers cannot always be represented exactly in binary format. As a result, the expression 0.1 + 0.2 does not produce an exact value of 0.3, causing direct comparison using == to fail.

- To correct this, a tolerance-based comparison is used. The absolute difference between the two values is checked to see if it is smaller than a very small number (1e-9).
- This approach accounts for minor precision errors and produces the correct result.

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):

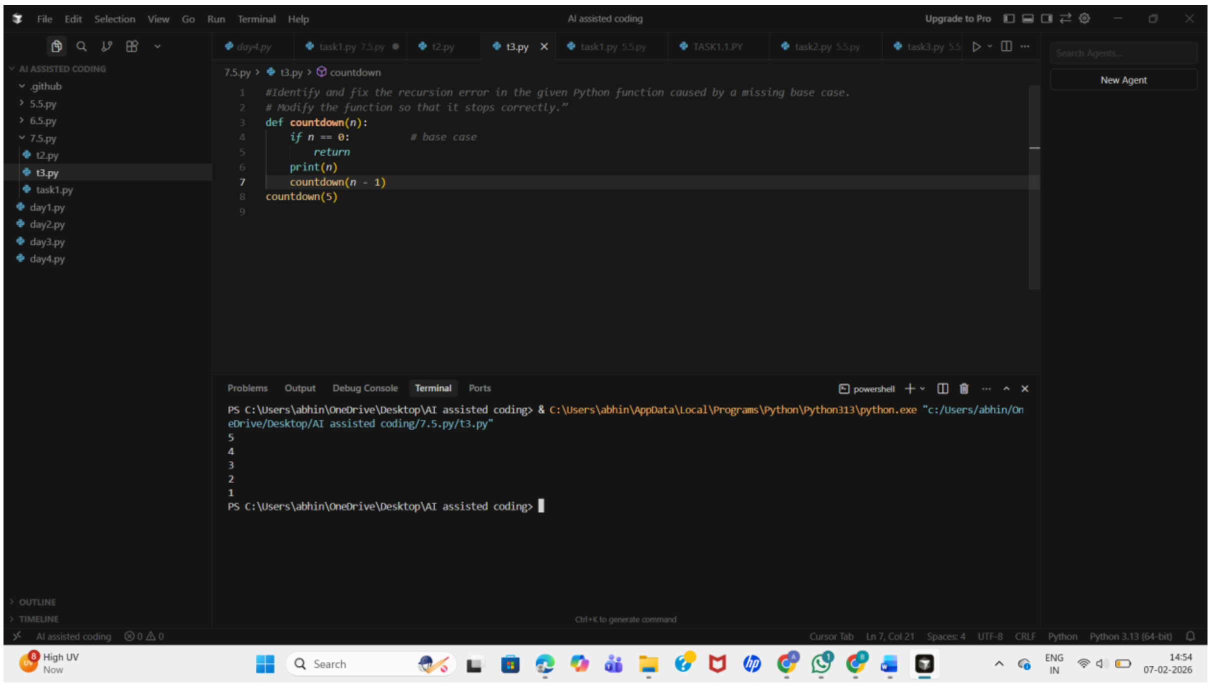
print(n)

return countdown(n-1)

countdown(5)

Expected Output : Correct recursion with stopping condition.

Code:



```python
#Identify and fix the recursion error in the given Python function caused by a missing base case.
# Modify the function so that it stops correctly.”
def countdown(n):
    if n == 0:          # base case
        return
    print(n)
    countdown(n - 1)
countdown(5)
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t3.py"
5
4
3
2
1
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- The original function does not contain a base case, so the recursive call continues indefinitely, eventually causing a recursion error.
- In recursion, a base case is required to stop further function calls.

- In the corrected version, a base case is added to stop the recursion when n becomes 0.
- Once this condition is met, the function returns without making another recursive call. This ensures that the recursion terminates properly.

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():

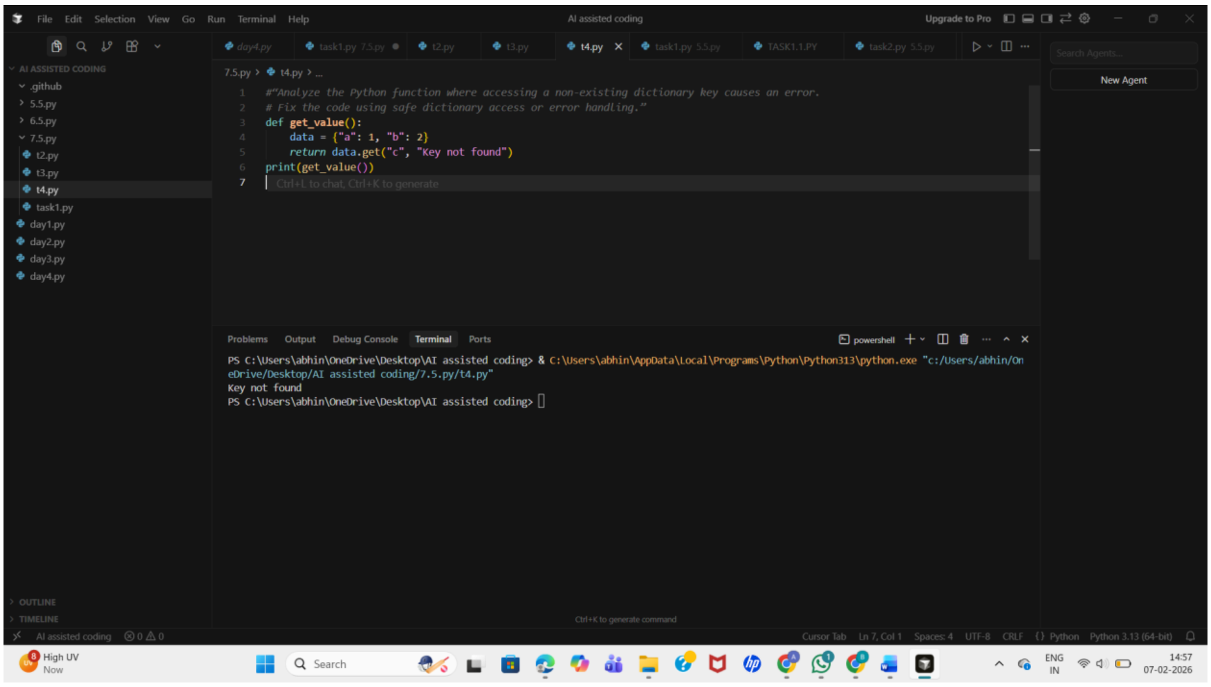
data = {"a": 1, "b": 2}

return data["c"]

print(get_value())

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

Code:



```python
#"Analyze the Python function where accessing a non-existing dictionary key causes an error.
# Fix the code using safe dictionary access or error handling."
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")
print(get_value())
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/On
eDrive/Desktop/AI assisted coding/7.5.py/t4.py"
Key not found
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- In the original code, the dictionary key "c" does not exist. Accessing a missing key using square brackets (data["c"]) raises a KeyError.
- To fix this, the .get() method is used. The .get() method safely returns a default value when the key is not found instead of raising an error.

- This prevents the program from crashing and handles missing keys gracefully.

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():

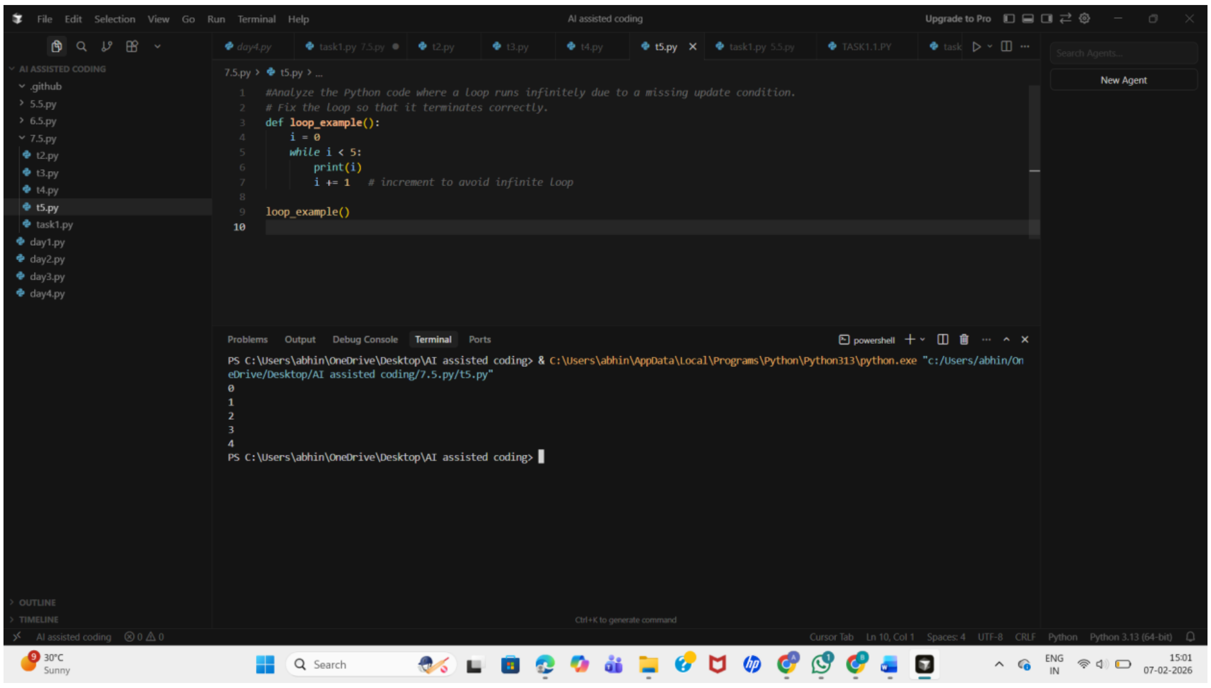
i = 0

while i < 5:

print(i)

Expected Output: Corrected loop increments i.

Code:



```python
#Analyze the Python code where a loop runs infinitely due to a missing update condition.
# Fix the loop so that it terminates correctly.
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1   # increment to avoid infinite loop

loop_example()
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t5.py"
0
1
2
3
4
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- In the variable i is never updated inside the while loop. Because the condition i < 5 always remains true, the loop runs infinitely.
- The value of i is incremented during each iteration. This eventually makes the condition false, allowing the loop to stop correctly.

Task 6 (Unpacking Error – Wrong Variables)

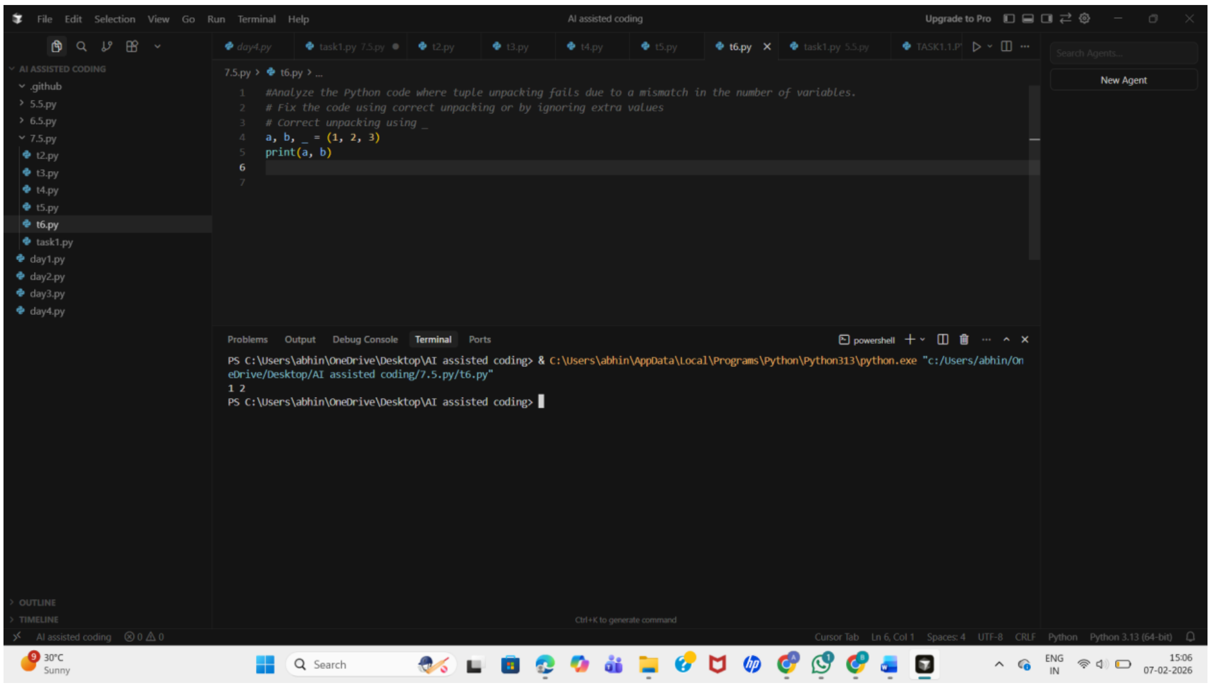
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

# Bug: Wrong unpacking

a, b = (1, 2, 3)

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

Code:



```python
#Analyze the Python code where tuple unpacking fails due to a mismatch in the number of variables.
# Fix the code using correct unpacking or by ignoring extra values
# Correct unpacking using _
a, b, _ = (1, 2, 3)
print(a, b)
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t6.py"
1 2
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- The tuple contains three values but only two variables are provided for unpacking. This causes a ValueError because Python expects the number of variables on the left side to match the number of values on the right side.
- The issue can be fixed either by using the correct number of variables or by using an underscore (_) to ignore unwanted values. This allows the unpacking operation to succeed without errors.

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():

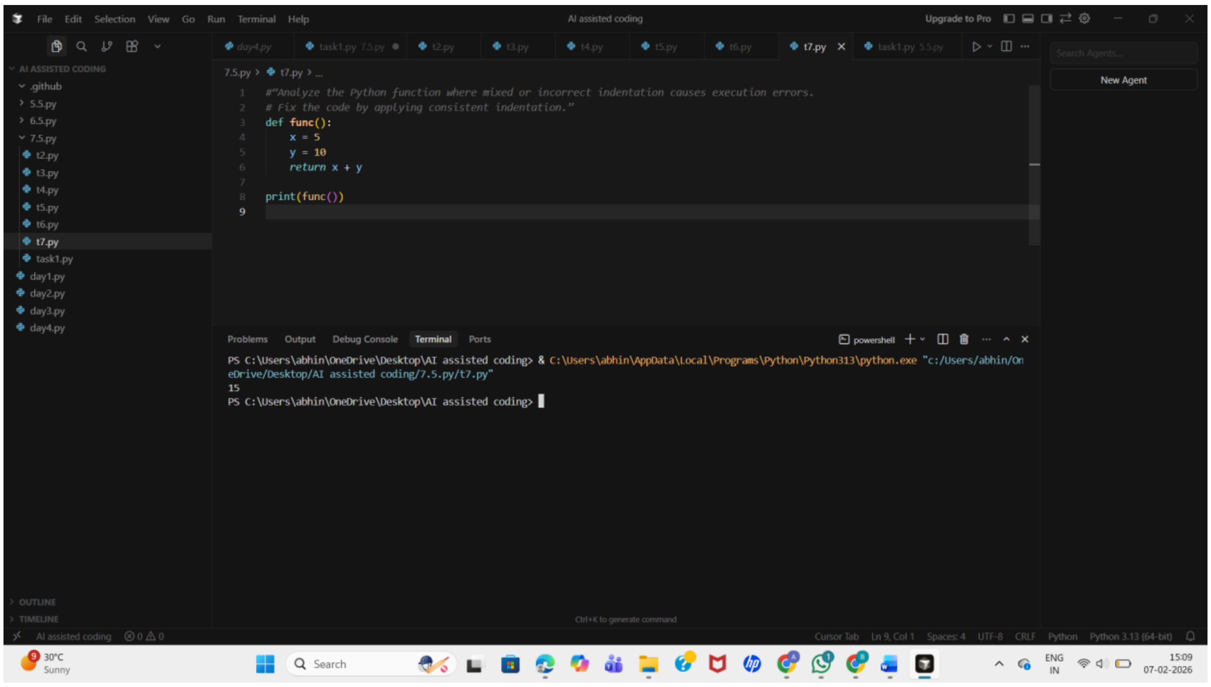
x = 5

y = 10

return x+y

Expected Output : Consistent indentation applied.

Code:



```python
#"Analyze the Python function where mixed or incorrect indentation causes execution errors.
# Fix the code by applying consistent indentation."
def func():
    x = 5
    y = 10
    return x + y

print(func())
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t7.py"
15
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- Python uses indentation to define code blocks. In the given code, inconsistent indentation (mixing tabs and spaces or missing indentation) breaks the function structure and causes an IndentationError.
- The corrected version uses consistent indentation (4 spaces) for all statements inside the function.
- This allows Python to correctly interpret the function body and execute it without errors.

Task 8 (Import Error – Wrong Module Usage)

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

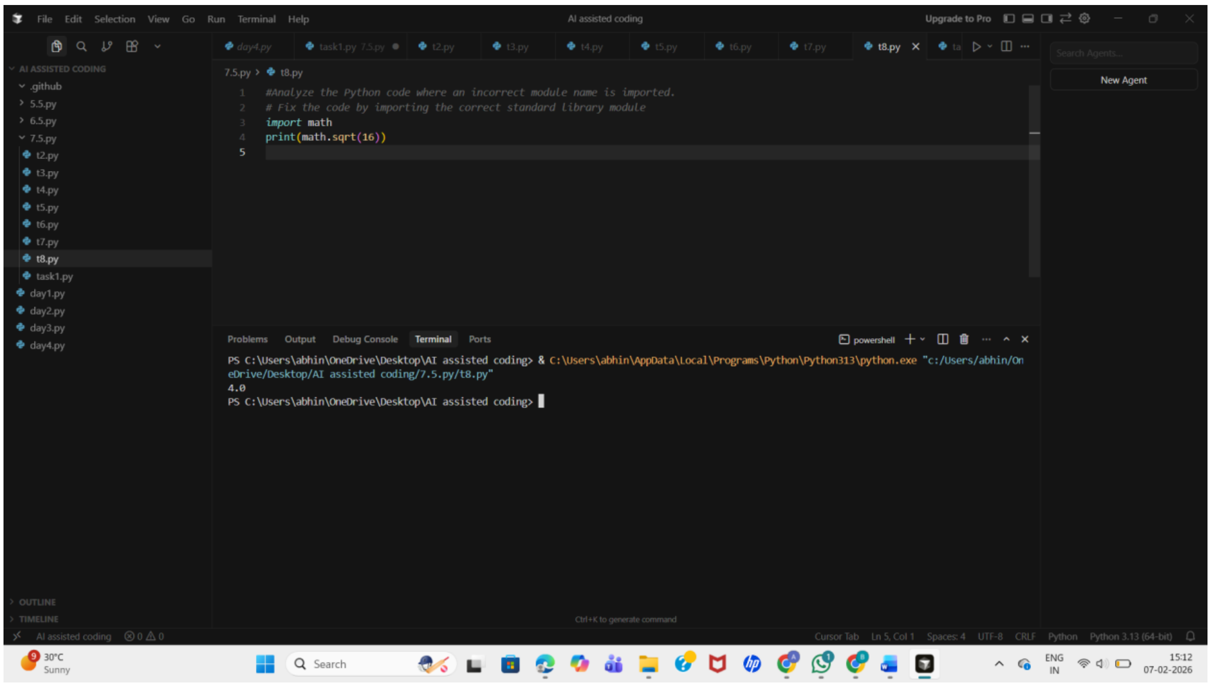
# Bug: Wrong import

import maths

print(maths.sqrt(16))

Expected Output: Corrected to import math

Code:



```python
#Analyze the Python code where an incorrect module name is imported.
# Fix the code by importing the correct standard library module
import math
print(math.sqrt(16))
```

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t8.py"
4.0
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- The corrected code imports the correct module, math, which provides mathematical functions such as sqrt().
- Using the proper module name fixes the error and allows the program to run successfully.