Assignment 7.5

Name: Keshireddy vedasri

Ht no: 2303A52382

Batch No: 45

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.

Prompt:Debug the error

Code:

def add\_item(item, items=None):

if items is None:

items = []

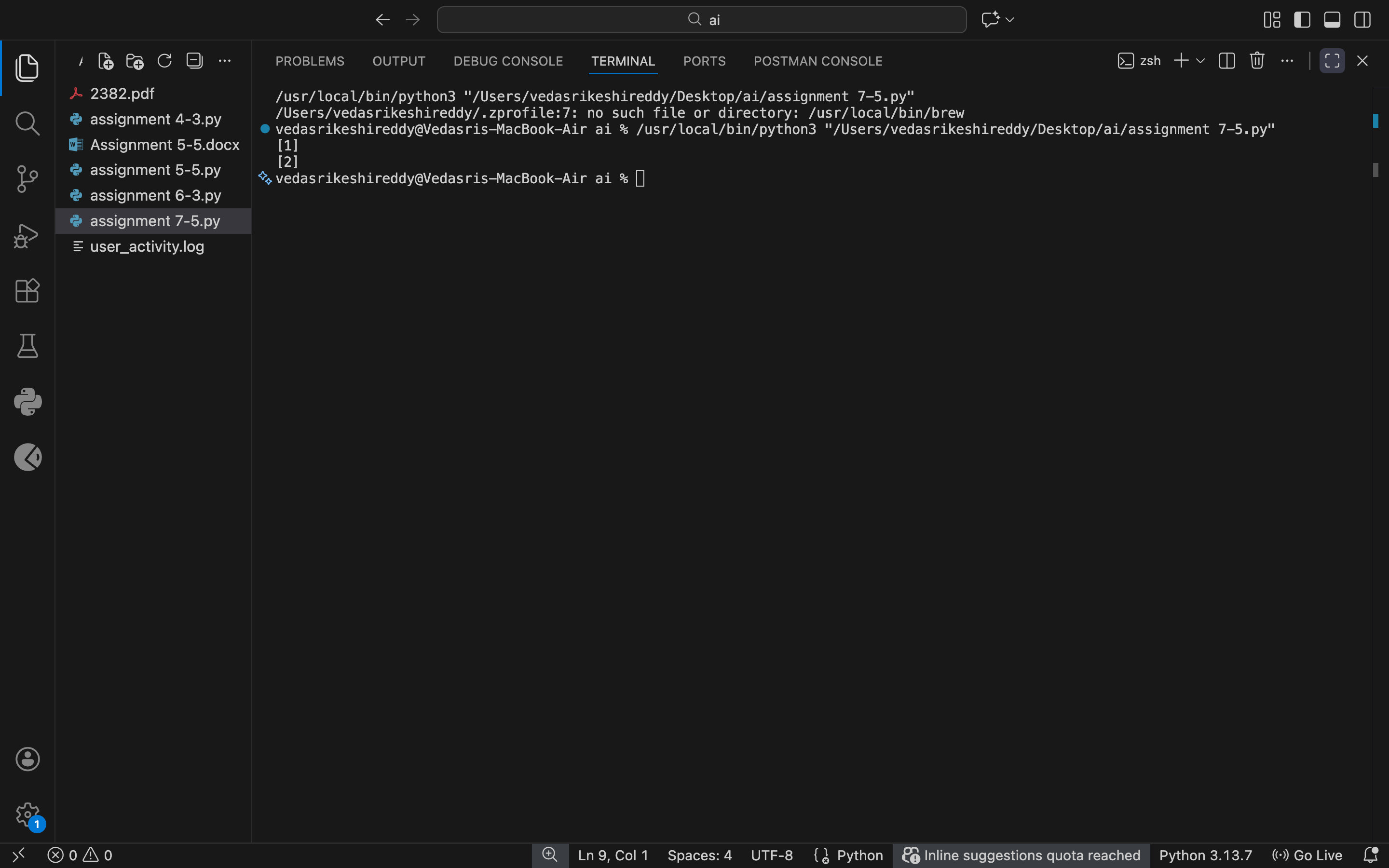
items.append(item)

return items

print(add\_item(1))

print(add\_item(2))

Output:



Justification:

The bug happens because the same list is reused every time the function is called. By using `None` and creating a new list inside the function, each call gets a fresh list. This prevents unexpected shared data.

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

Prompt:

Fix the floating point precision error

Code:

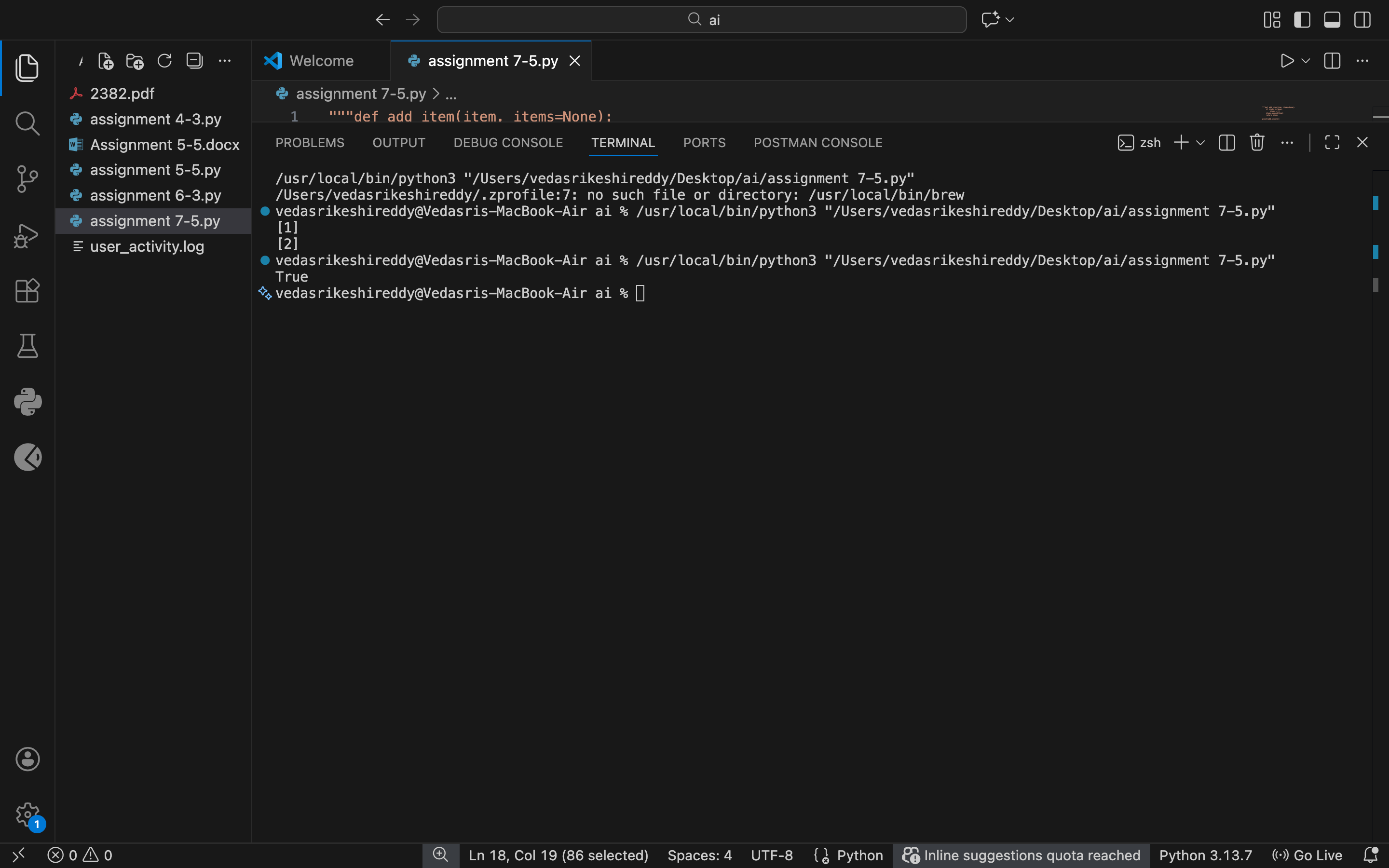
import math

def check\_sum():

return math.isclose(0.1 + 0.2, 0.3)

print(check\_sum())

Output:



Justification:

Floating-point numbers are not stored exactly in memory, which causes direct comparisons to fail. Using a small tolerance helps compare values safely and gives accurate logical results.

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**.**

Prompt: fix the recursion error

Code:

def countdown(n):

if n < 0:

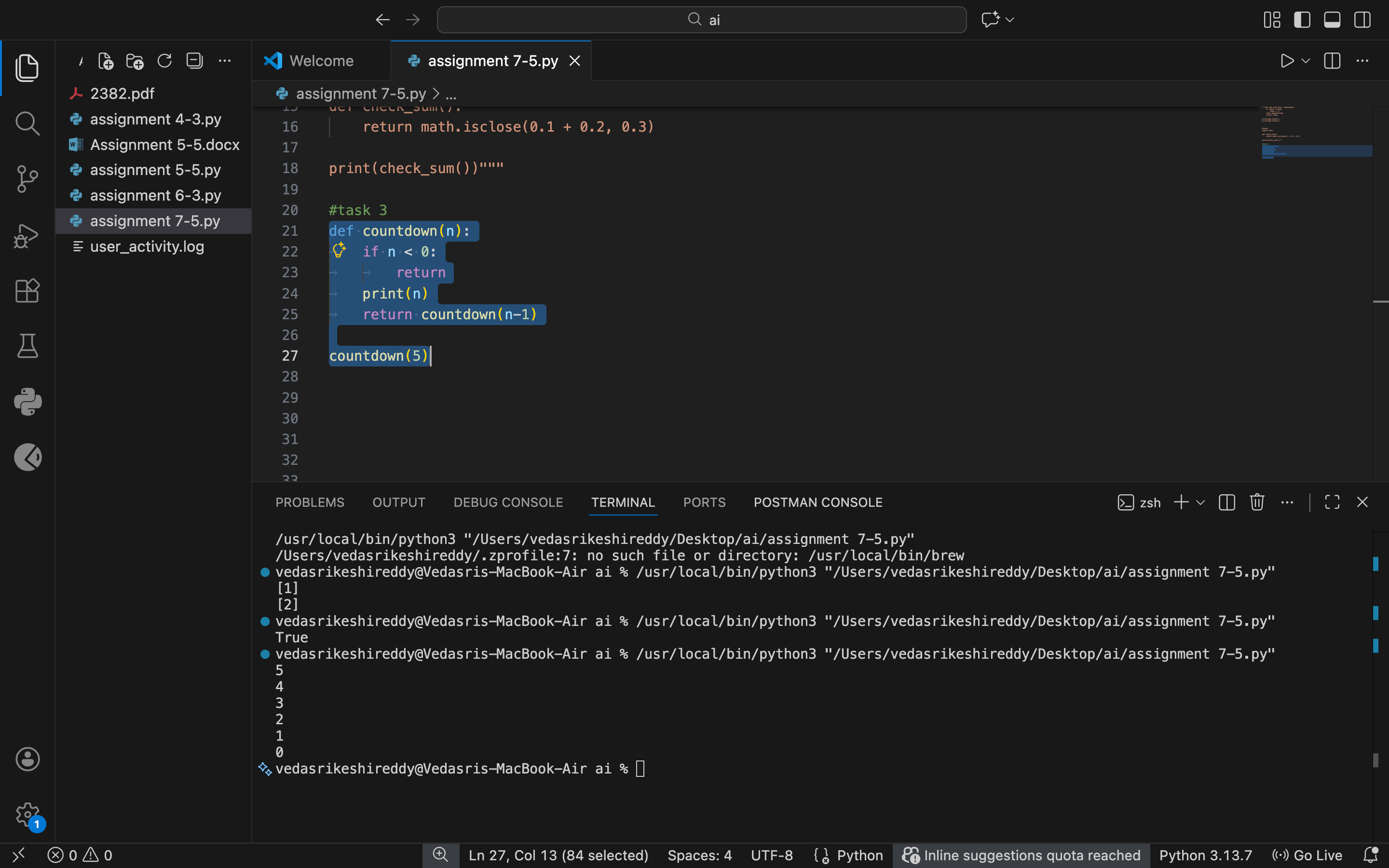
return

print(n)

return countdown(n-1)

countdown(5)

Output:



Justification:

Without a base case, the function keeps calling itself forever. Adding a stopping condition ensures the recursion ends properly and avoids stack overflow errors.

Task 4

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.

Prompt:fix the Dictionary key error

Code:

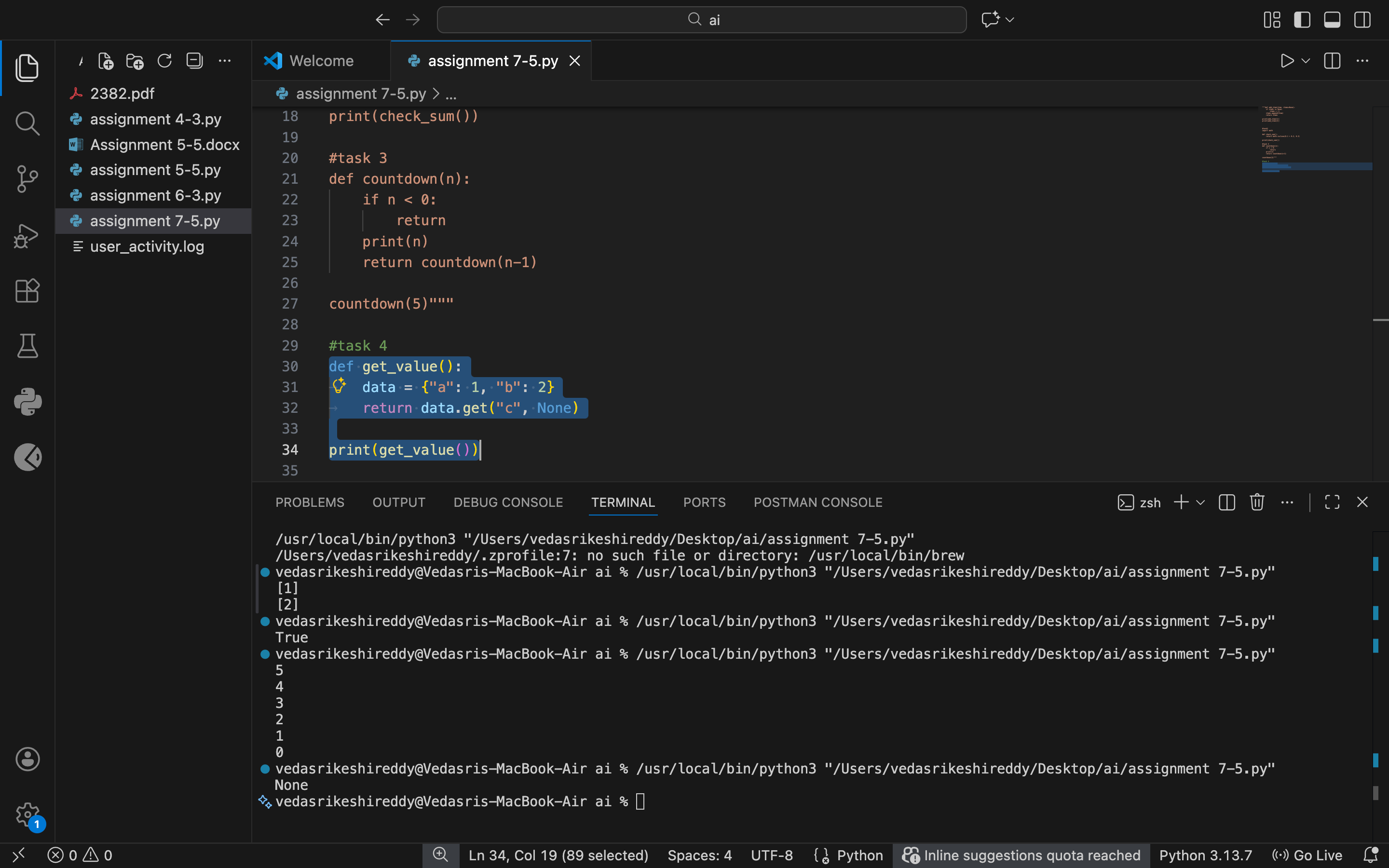
def get\_value():

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

return data.get("c", None)

print(get\_value())

Output:



Justification:

Accessing a key that doesn’t exist causes a runtime error. Using `.get()` safely handles missing keys and prevents the program from crashing.

Task 5

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.

Prompt: Fix the infinite loop.

Code:

def loop\_example():

i = 0

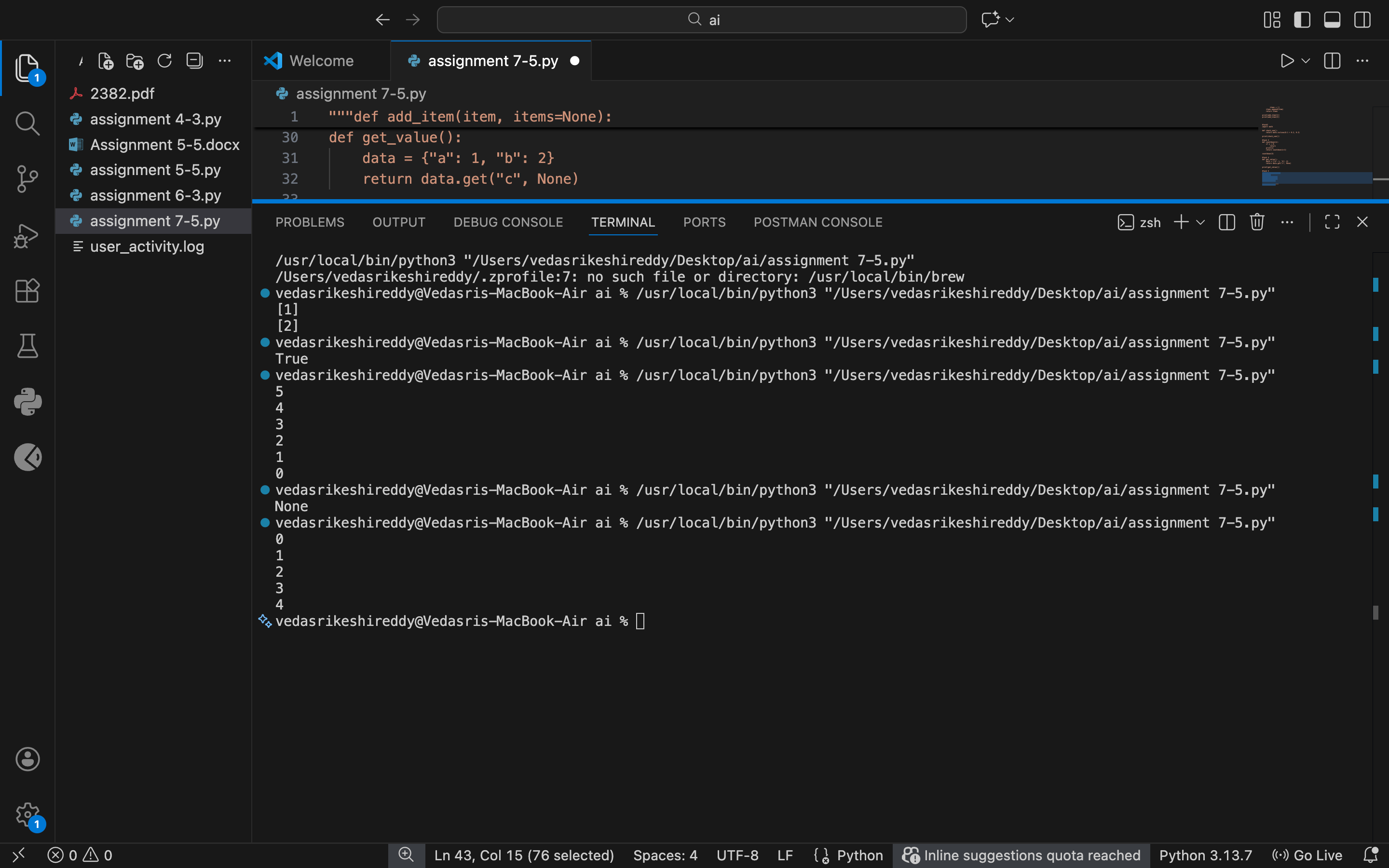
while i < 5:

print(i)

i += 1

loop\_example()

Output:



Justification:

The loop never ends because the counter variable is never updated. Incrementing the variable inside the loop allows the condition to eventually fail and stop execution.

Task 6

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.

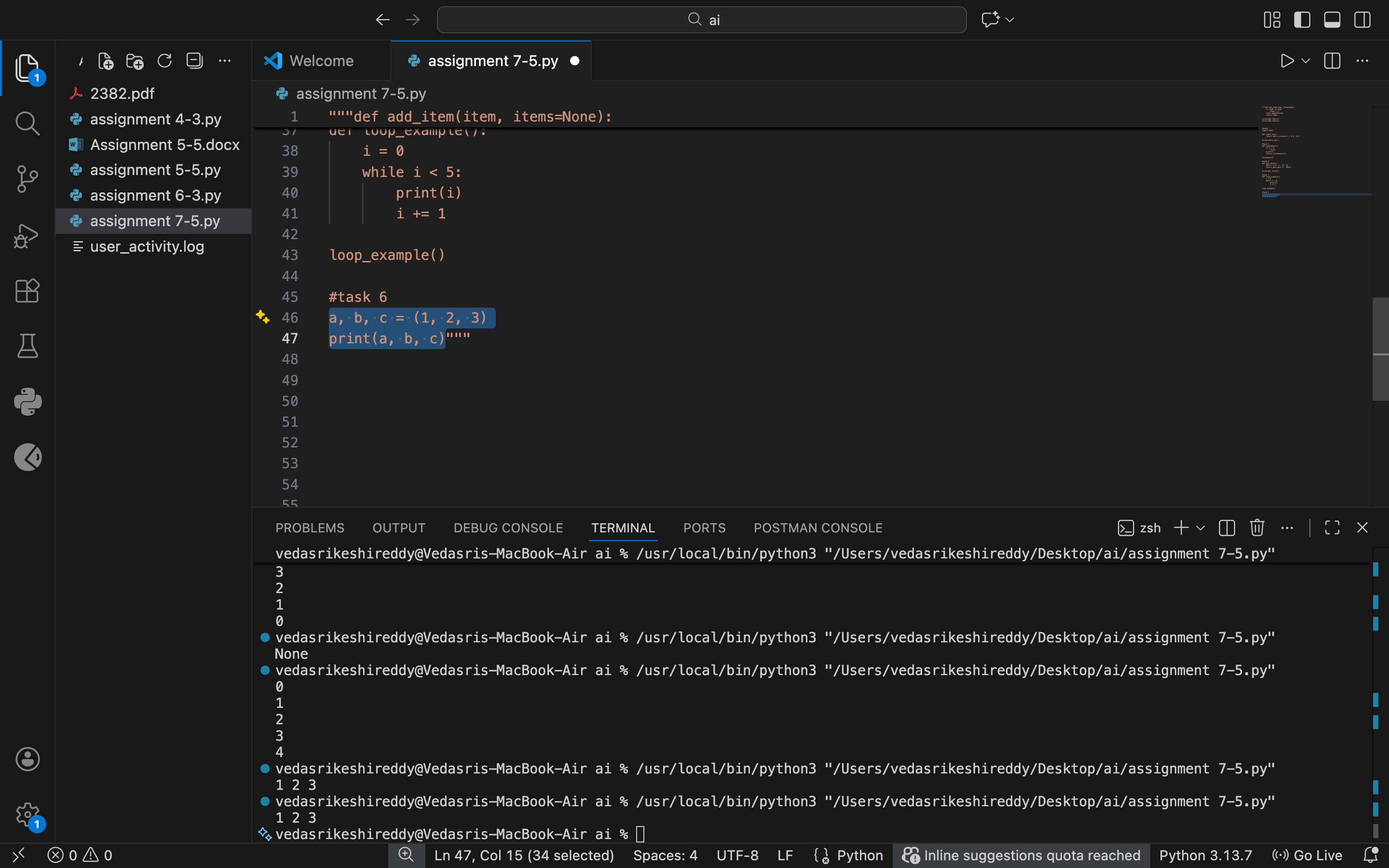
Prompt:Fix the unpacking error

Code:

a, b, c = (1, 2, 3)

print(a, b, c)

Output:



Justification:

The error occurs because there are more values than variables to unpack. Using an underscore or matching variable count correctly fixes the issue cleanly.

Task 7

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.

Prompt:fix the mixed indentation

Code:

def func():

x = 5

y = 10

return x+y

print(func())

Output:



Justification:

Python relies on consistent indentation for code blocks. Fixing the indentation using spaces throughout ensures the code runs without syntax errors.

Task 8

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

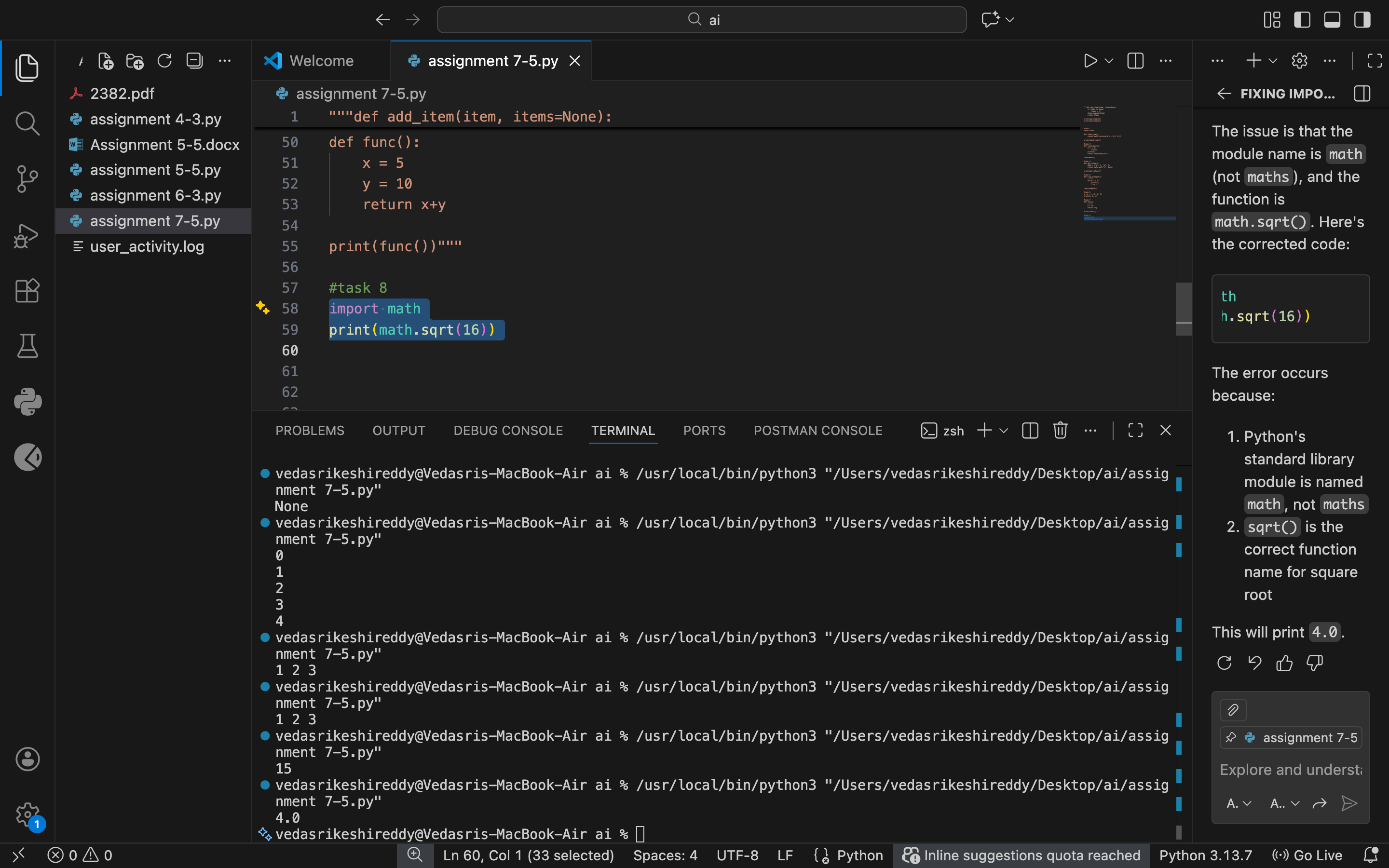
Prompt: fix the import error

Code:

import math

print(math.sqrt(16))

Output:



Justification:

The module name was incorrect, leading to an import failure. Correcting the module name allows Python to access the required functions properly.

Task 9

**Task:** Analyze given code where a return inside a loop prevents full iteration. Use AI to fix it.

# Bug: Early return inside loop

def total(numbers):

for n in numbers:

return n

print(total([1,2,3]))

**Expected Output:** Corrected code accumulates sum and returns after loop.

Prompt:fix the return inside a loop

Code:

def total(numbers):

sum\_total = 0

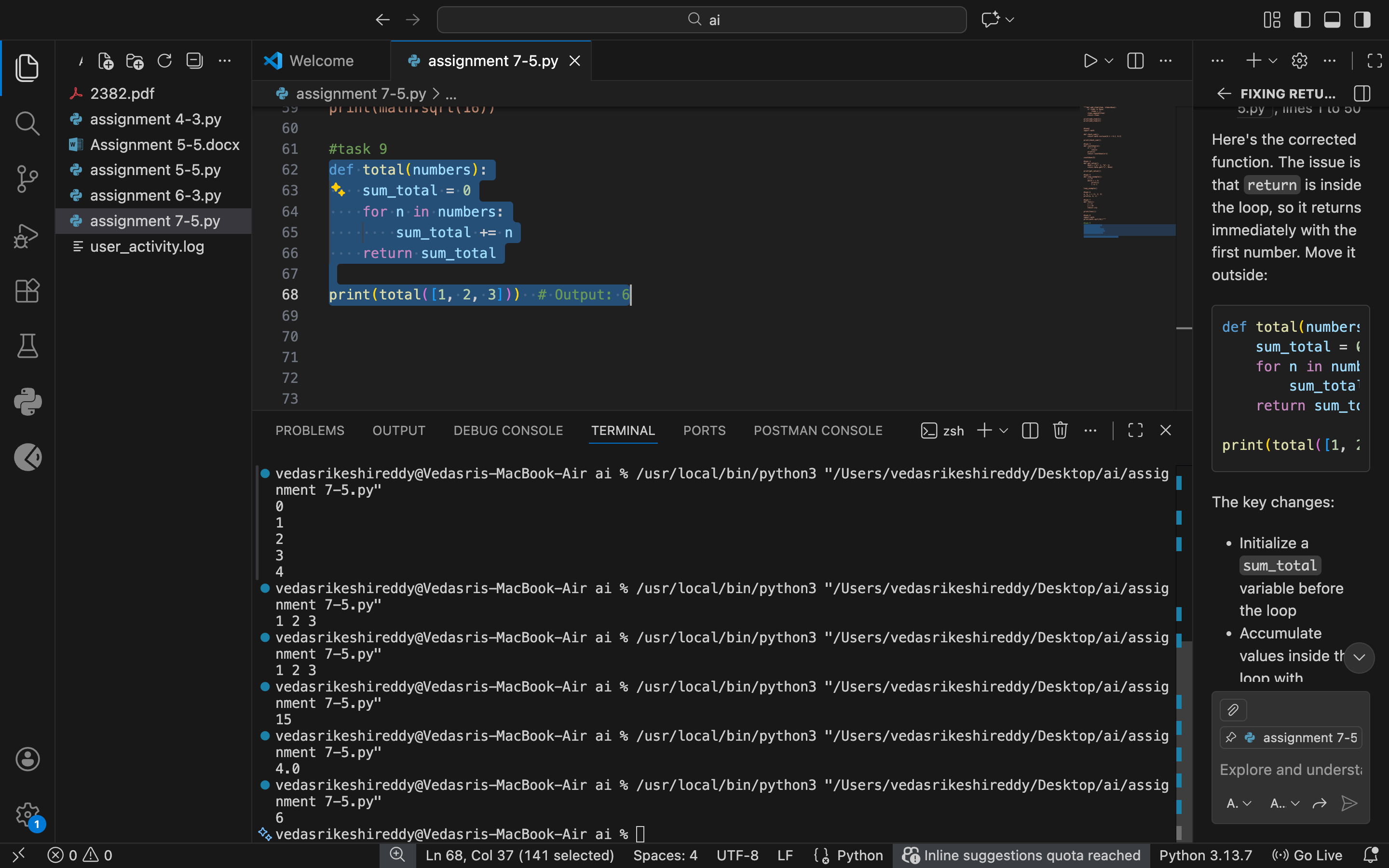
for n in numbers:

sum\_total += n

return sum\_total

print(total([1, 2, 3]))

Output:



Justification:

Returning inside the loop exits the function after the first iteration. Moving the return statement outside the loop ensures all values are processed.

Task 10

Analyze given code where a variable is used before being defined. Let AI detect and fix the error.

# Bug: Using undefined variable

def calculate\_area():

return length \* width

print(calculate\_area())

Requirements:

* Run the code to observe the error.
* Ask AI to identify the missing variable definition.
* Fix the bug by defining length and width as parameters.
* Add 3 assert test cases for correctness.

Expected Output :

* Corrected code with parameters.
* AI explanation of the bug.

Successful execution of assertions.

Prompt:

fix the name error with 3 test cases

Code:

def calculate\_area(length, width):

return length \* width

# Test case 1

print(calculate\_area(5, 10)) # Output: 50

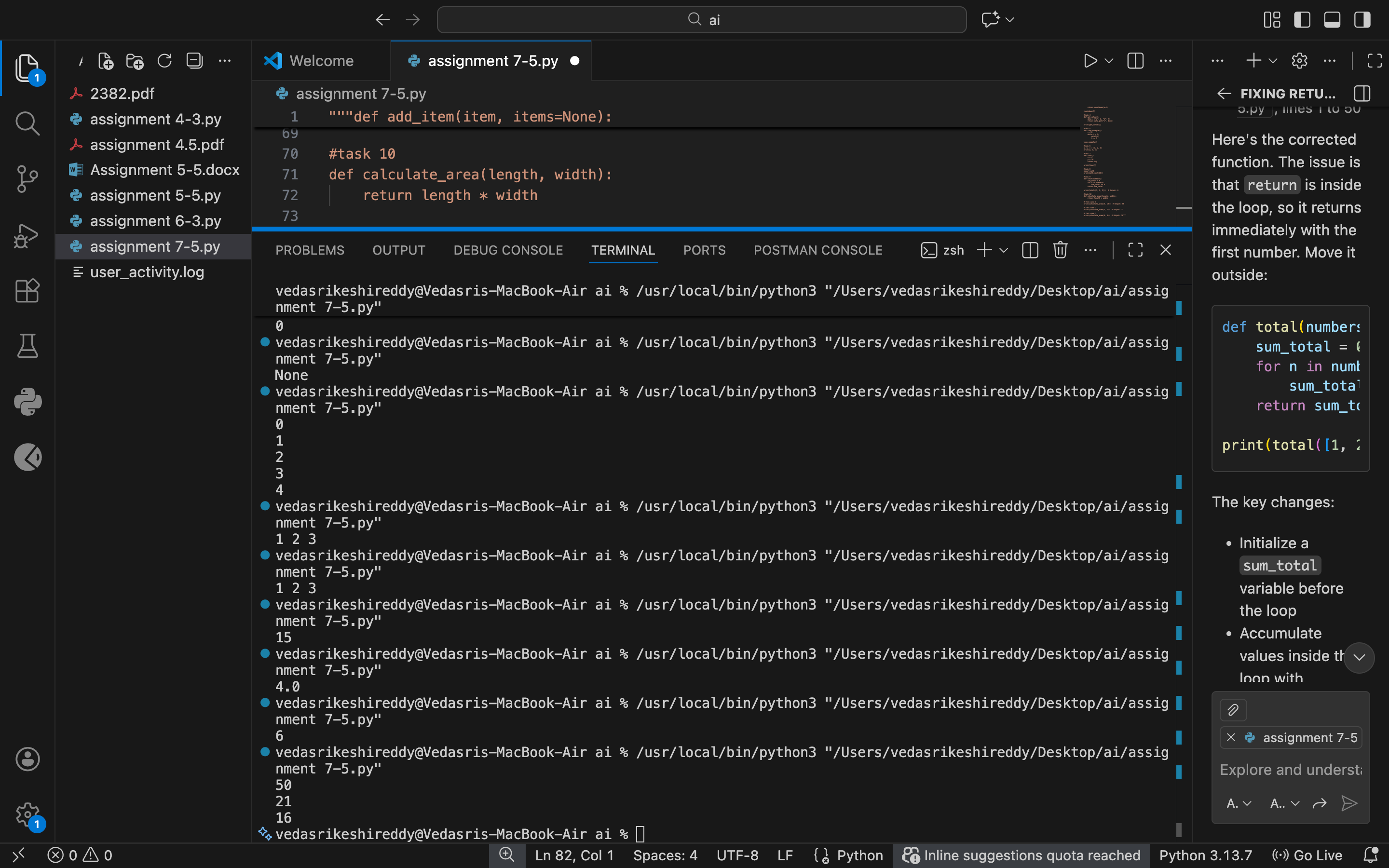
# Test case 2

print(calculate\_area(3, 7)) # Output: 21

# Test case 3

print(calculate\_area(4, 4)) # Output: 16

Output:



Justification:

The variables were used without being defined. Passing them as function parameters makes the function reusable and removes the name error.

Task 11

Task: Analyze given code where integers and strings are added incorrectly. Let AI detect and fix the error.

# Bug: Adding integer and string

def add\_values():

return 5 + "10"

print(add\_values())

Requirements:

* Run the code to observe the error.
* AI should explain why int + str is invalid.
* Fix the code by type conversion (e.g., int("10") or str(5)).
* Verify with 3 assert cases.

Expected Output #6:

* Corrected code with type handling.
* AI explanation of the fix.

Successful test validation.

Prompt:

fix the type error with 3 test cases

Code:

def add\_values():

return 5 + int("10")

# Test case 1

print(add\_values()) # Output: 15

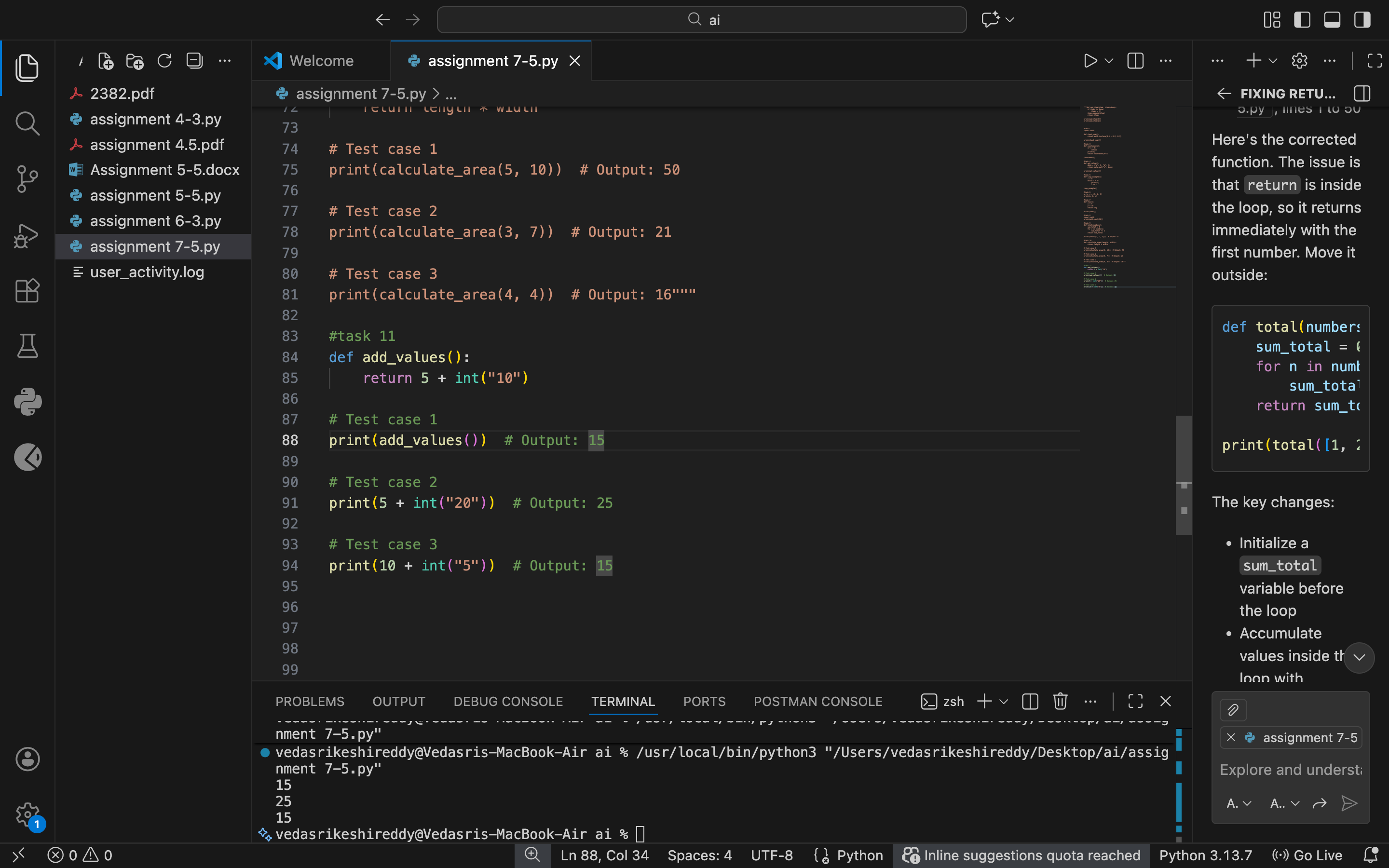
# Test case 2

print(5 + int("20")) # Output: 25

# Test case 3

print(10 + int("5")) # Output: 15

Output:



Justification:

Python does not allow adding integers and strings directly. Converting one type to match the other ensures valid operations and correct output.

Task 12

Task: Analyze code where a string is incorrectly added to a list.

# Bug: Adding string and list

def combine():

return "Numbers: " + [1, 2, 3]

print(combine())

Requirements:

* Run the code to observe the error.
* Explain why str + list is invalid.
* Fix using conversion (str([1,2,3]) or " ".join()).
* Verify with 3 assert cases.

Expected Output:

* Corrected code
* Explanation
* Successful test validation

Prompt:

fix the type error with 3 testcases

Code:

def combine():

return "Numbers: " + str([1, 2, 3])

# Test case 1

print(combine()) # Output: Numbers: [1, 2, 3]

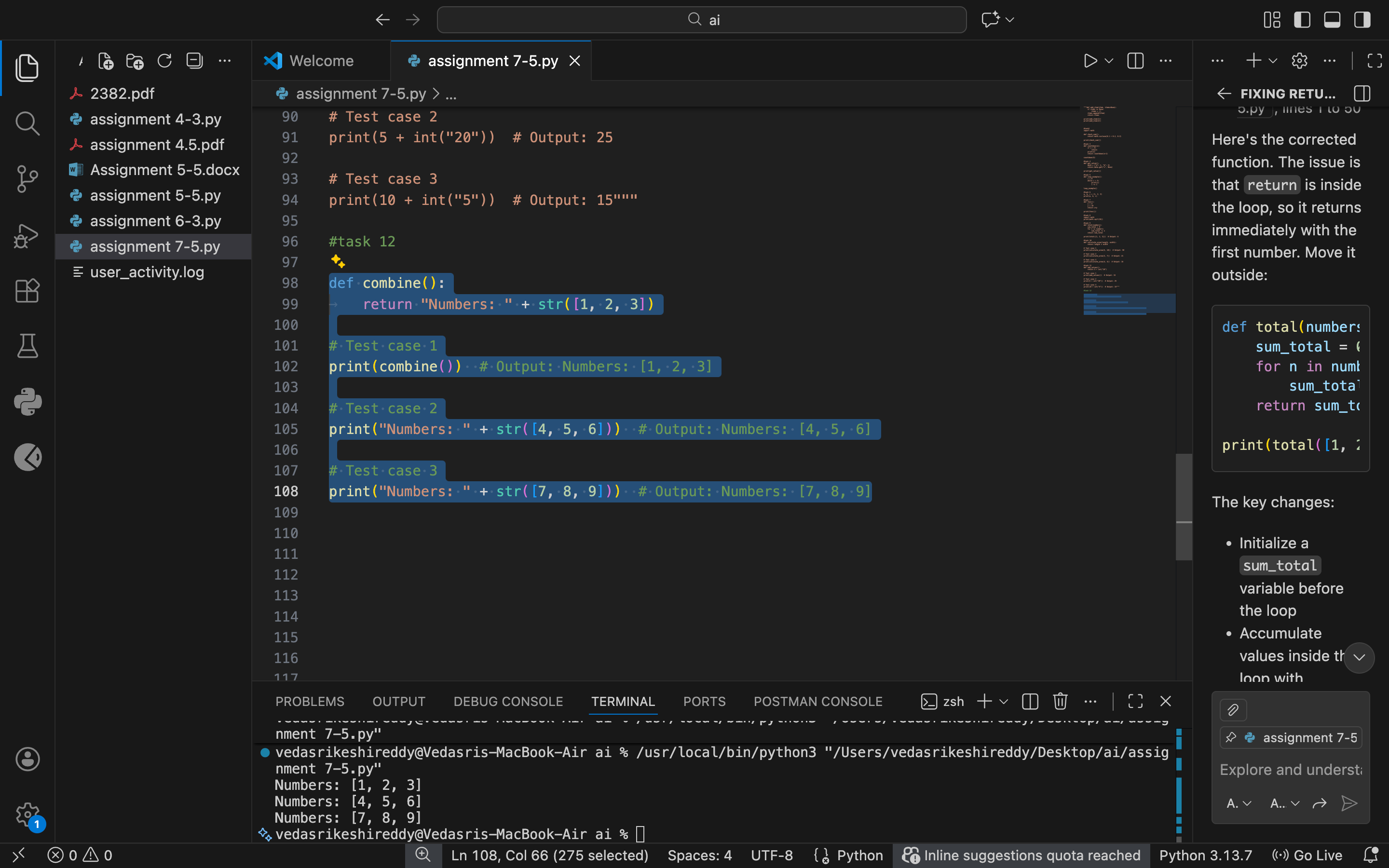
# Test case 2

print("Numbers: " + str([4, 5, 6])) # Output: Numbers: [4, 5, 6]

# Test case 3

print("Numbers: " + str([7, 8, 9])) # Output: Numbers: [7, 8, 9]

Output:



Justification:

Strings and lists cannot be concatenated directly. Converting the list to a string or joining its elements resolves the type mismatch.

Task 13

Task: Detect and fix code where a string is multiplied by a float.

# Bug: Multiplying string by float

def repeat\_text():

return "Hello" \* 2.5

print(repeat\_text())

Requirements:

* Observe the error.
* Explain why float multiplication is invalid for strings.
* Fix by converting float to int.
* Add 3 assert test cases.

Prompt:

fix the typeerror with 3 examples

Code:

ef repeat\_text():

return "Hello" \* int(2.5)

# Test case 1

print(repeat\_text()) # Output: HelloHello

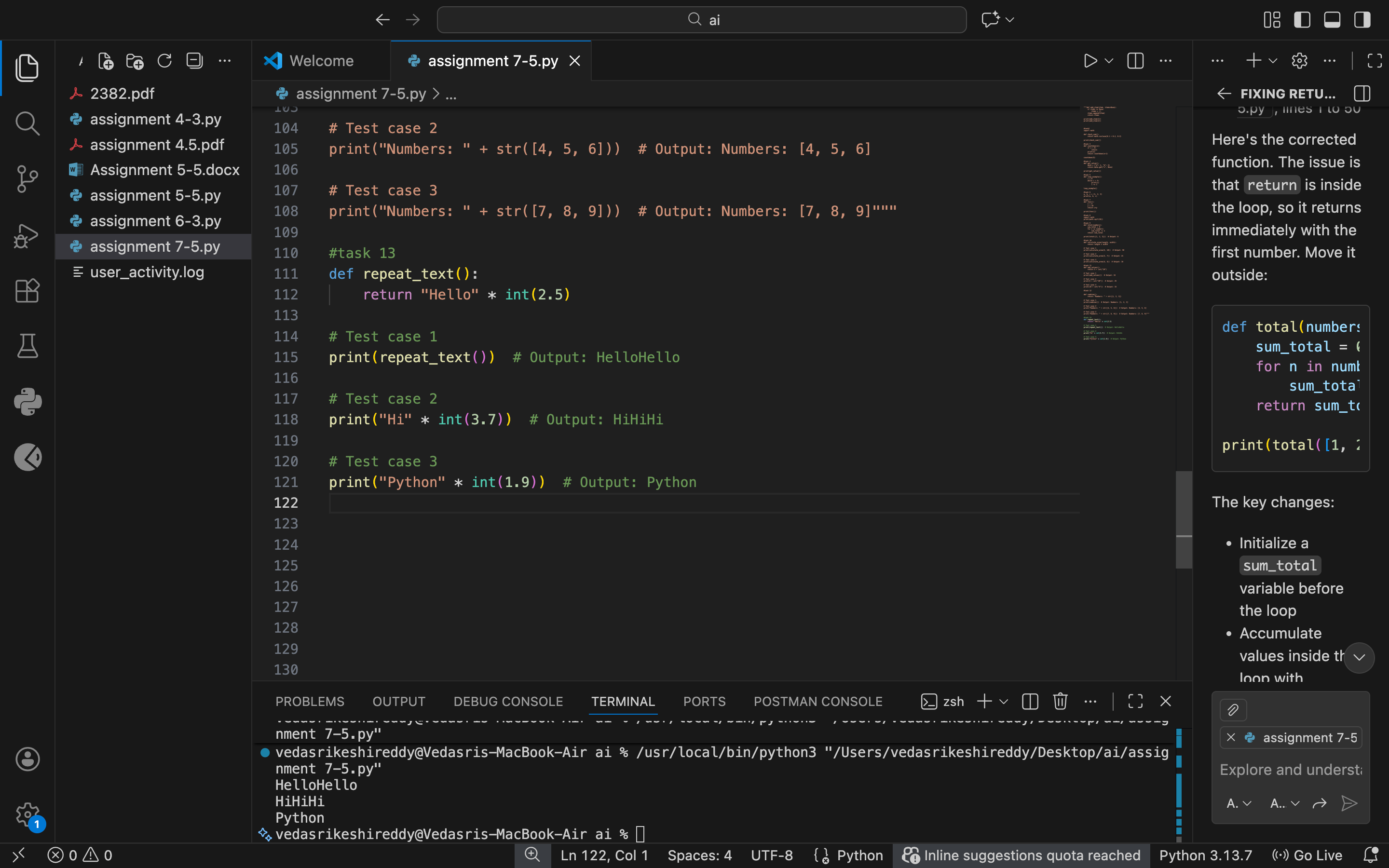
# Test case 2

print("Hi" \* int(3.7)) # Output: HiHiHi

# Test case 3

print("Python" \* int(1.9)) # Output: Python

Output:



Justification:

Strings can only be multiplied by integers, not floats. Converting the float to an integer makes repetition valid and predictable.