Assignment-7.5

2303A51062
Batch:29

Task 1 (Mutable Default Argument – Function Bug)

Task:

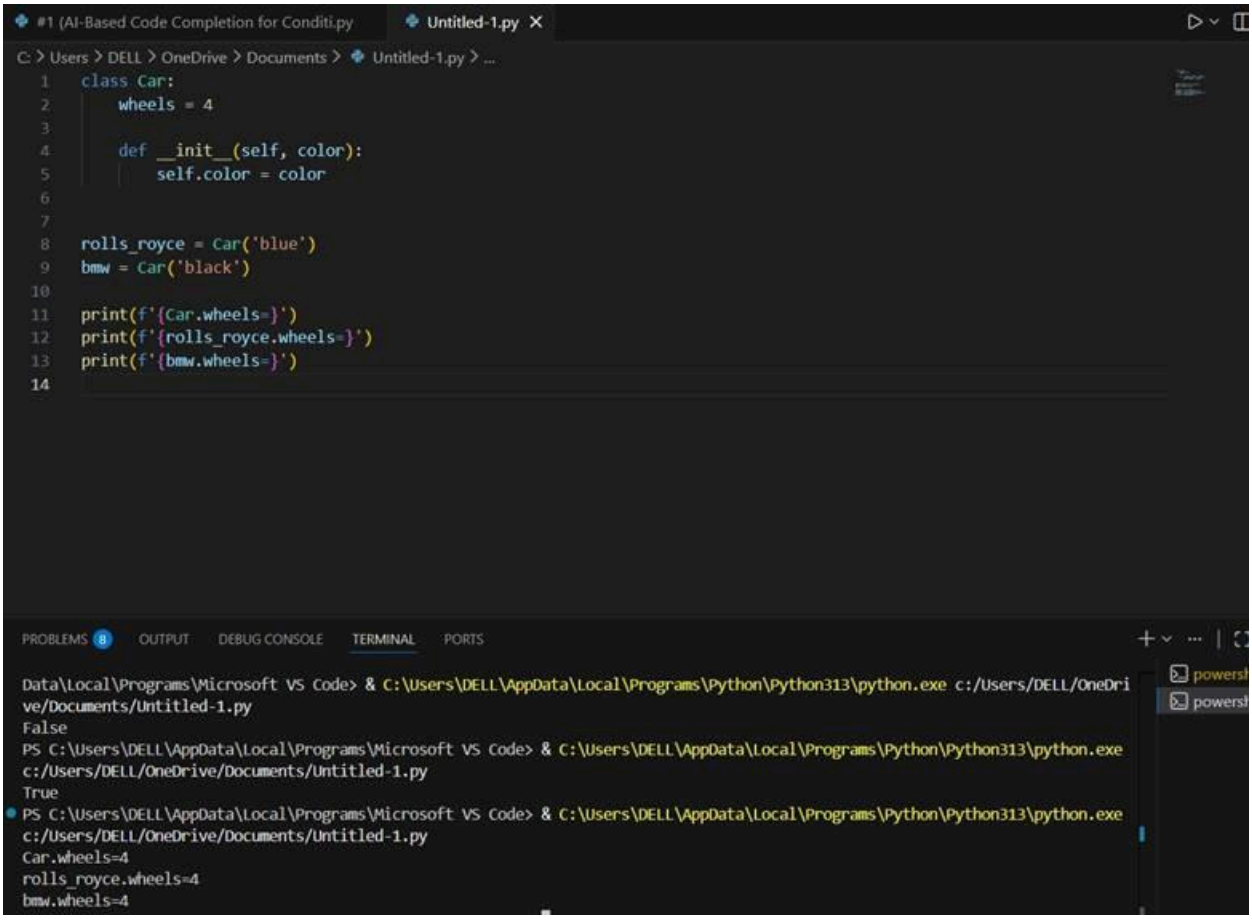
Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

```
# 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:



Justification:

wheels is a class variable, so it belongs to the class Car and is shared by all its objects. Accessing wheels using Car, rolls_royce, or bmw gives the same value because Python looks for the attribute in the instance first and, if not found, in the class.

Task 2 (Floating-Point Precision Error)

Task: Analyze given code where floating-point comparison fails. Use AI to correct with tolerance.

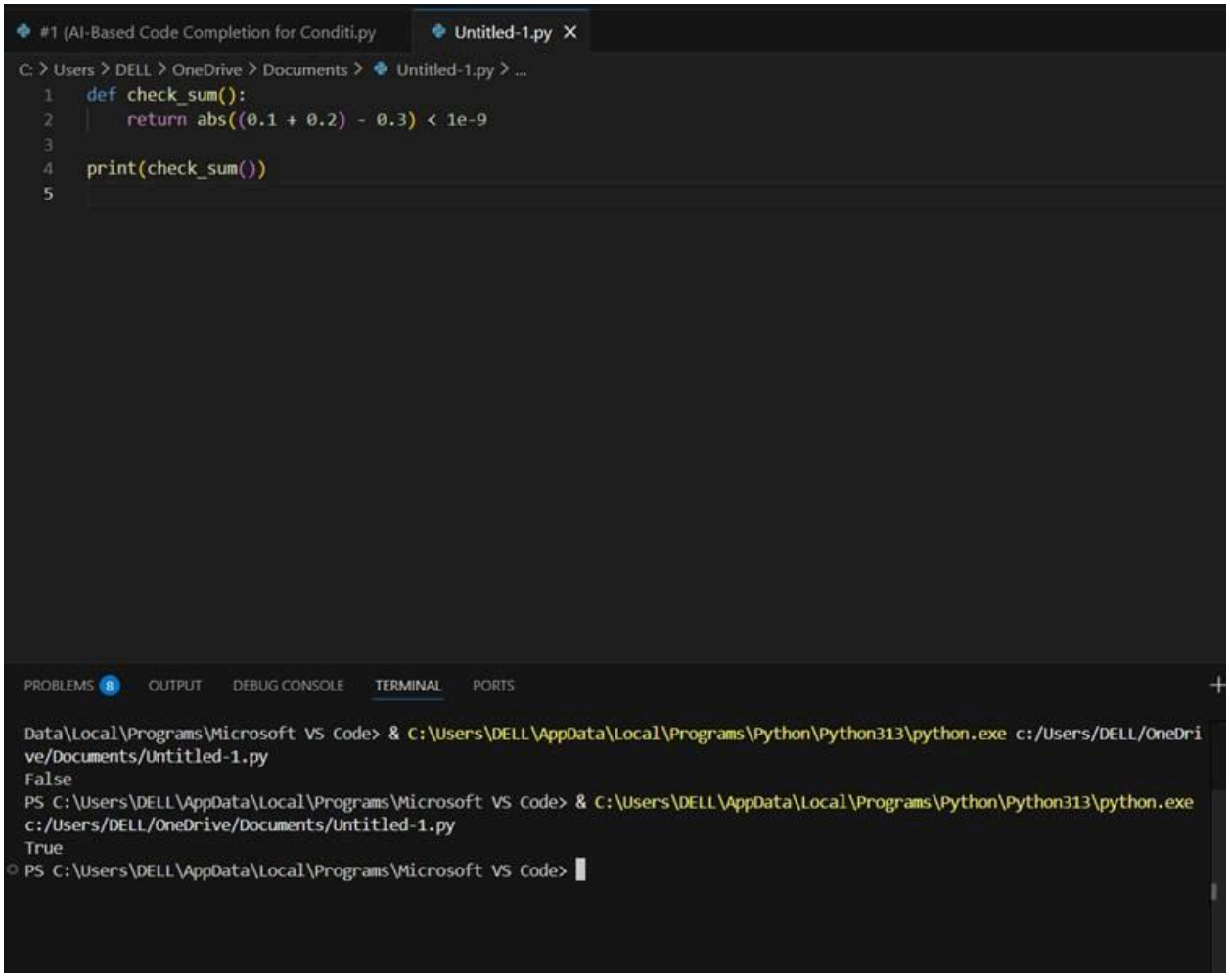
 # Bug: Floating point precision issue def

check_sum():

return (0.1 + 0.2) == 0.3 print(check_sum())

Expected Output: Corrected function

Code:



Justification:

Floating-point numbers are stored in binary, so 0.1 + 0.2 does not equal exactly 0.3.
Using a small tolerance (epsilon) allows comparison within an acceptable error range, making the check reliable.

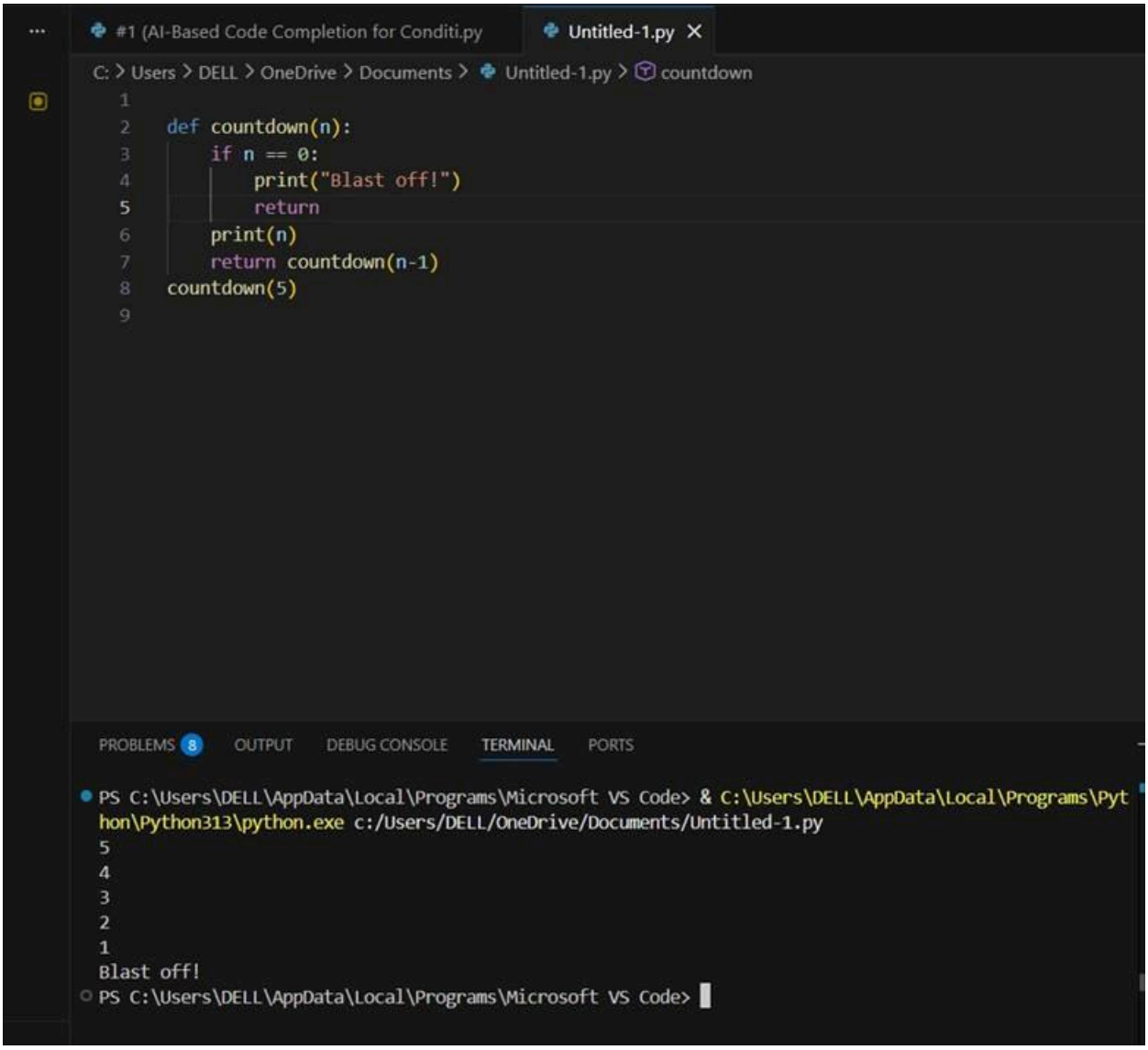
Task 3 (Recursion Error – Missing Base Case)

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 :

Code:



Justification:

The original function had **no base case**, so it kept calling itself endlessly and caused a recursion error.

Adding a base case (n == 0) gives the function a clear stopping condition, preventing infinite recursion.

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():

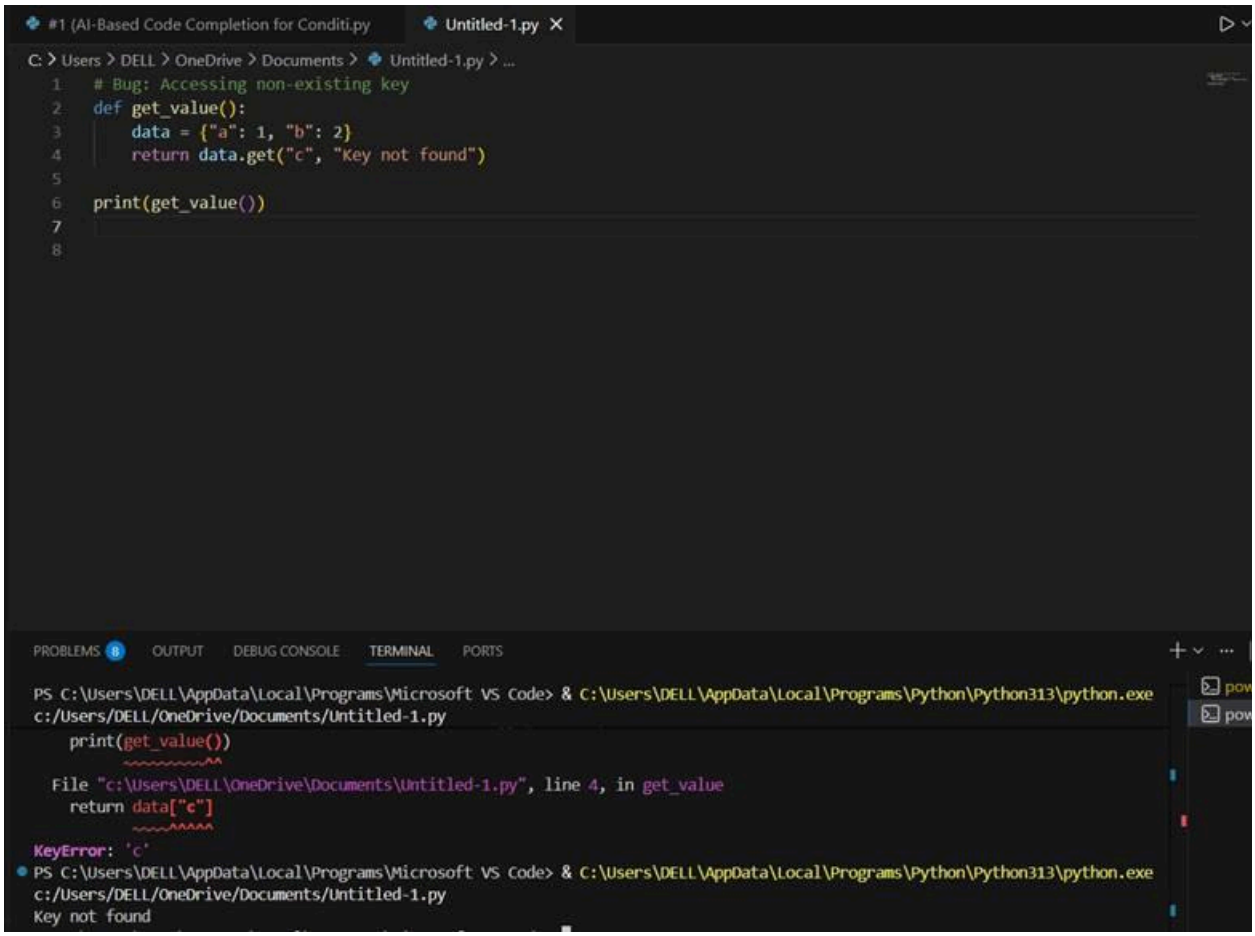
data = {"a": 1, "b": 2}

return data["c"]

print(get_value())

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

Code:



Justification:

Accessing a missing key with data["c"] raises a KeyError.

Using dict.get() returns a default value instead, preventing the runtime error.

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

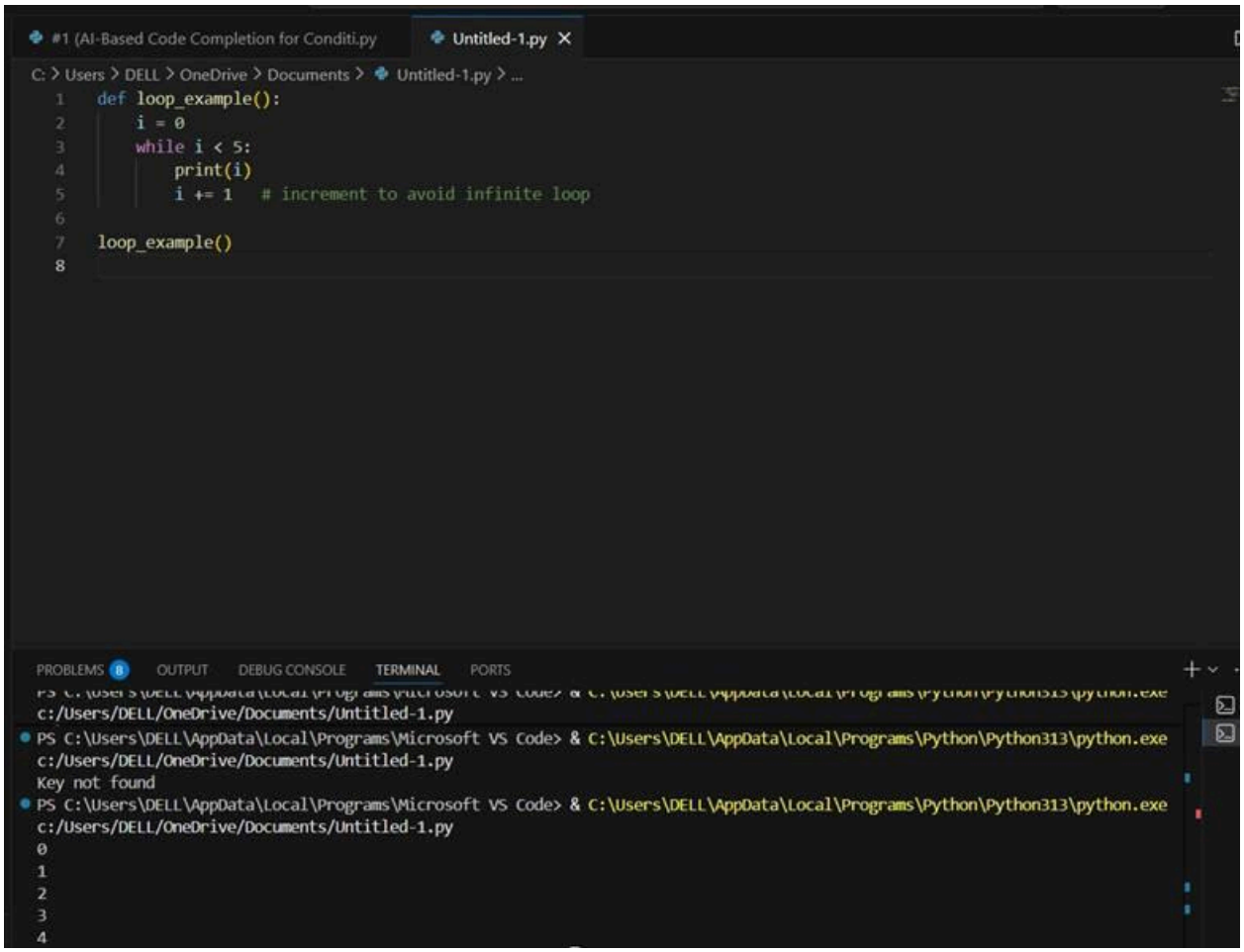
while

i < 5:

print(i)

Expected Output: Corrected loop increments i

code:.



Justification:

In the original code, i was never updated, so the condition i < 5 was always true.

Incrementing i inside the loop ensures the condition eventually becomes false, stopping the loop.

Task 6 (Unpacking Error – Wrong Variables)

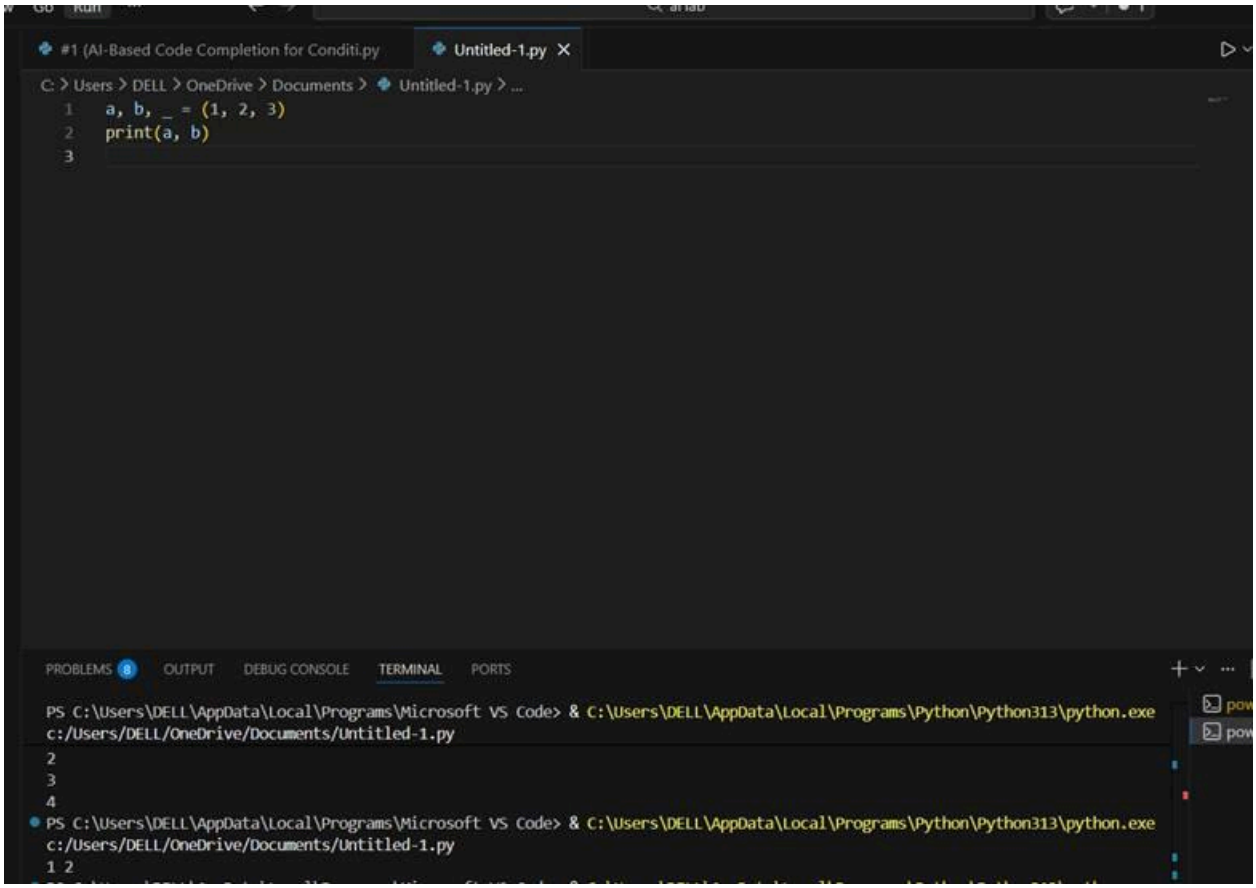
Task: Analyze given code where tuple unpacking fails. Use AI to fix it.

 # Bug: Wrong unpacking

a, b = (1, 2, 3)

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

Code:



Justification:

Tuple unpacking requires the **number of variables to match the number of values**.

Using _ allows you to intentionally ignore extra values without causing an error.

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():

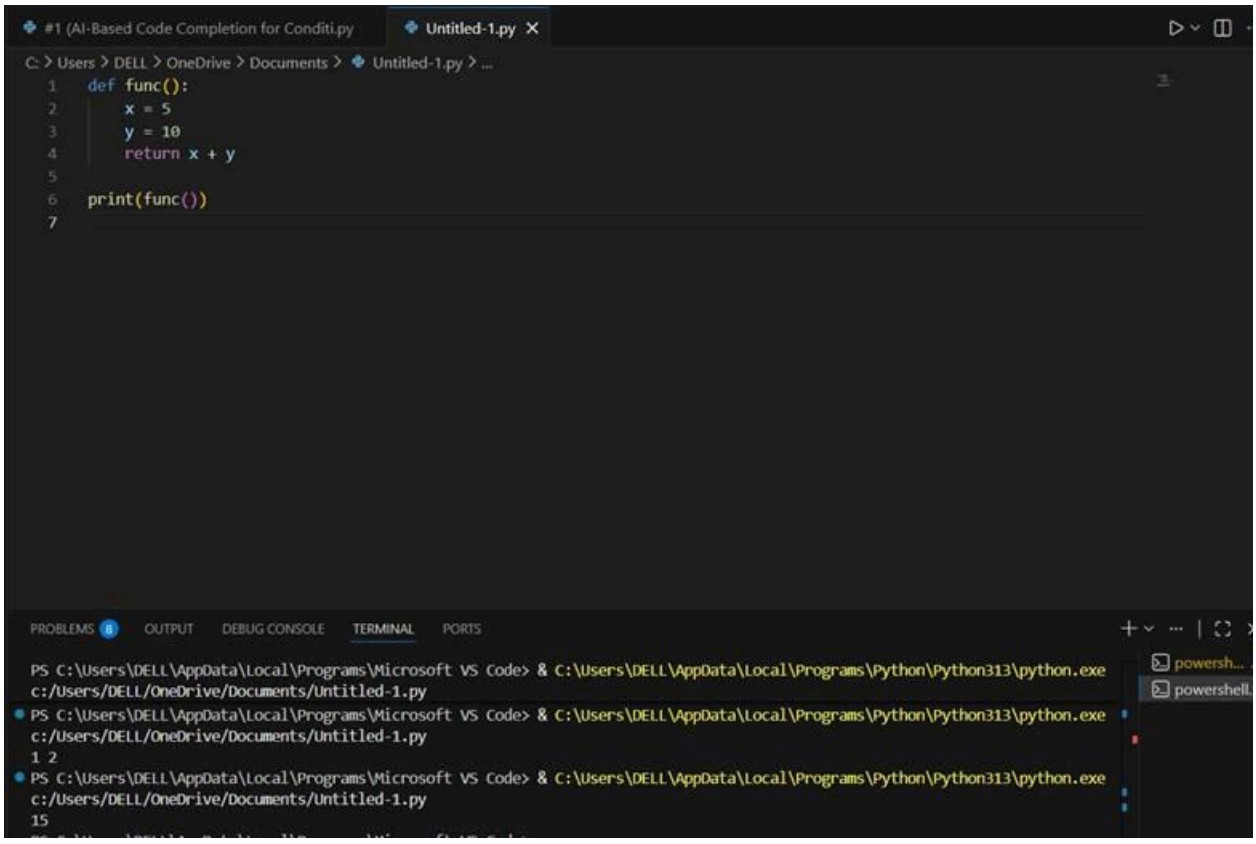
x = 5

y = 10

return x+y

Expected Output : Consistent indentation applied.

code:



Justification :

Mixed indentation causes an IndentationError because Python requires consistent use of spaces or tabs within the same block. Aligning all statements at the same indentation level using spaces fixes the execution error.

Task 8 (Import Error – Wrong Module Usage)

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

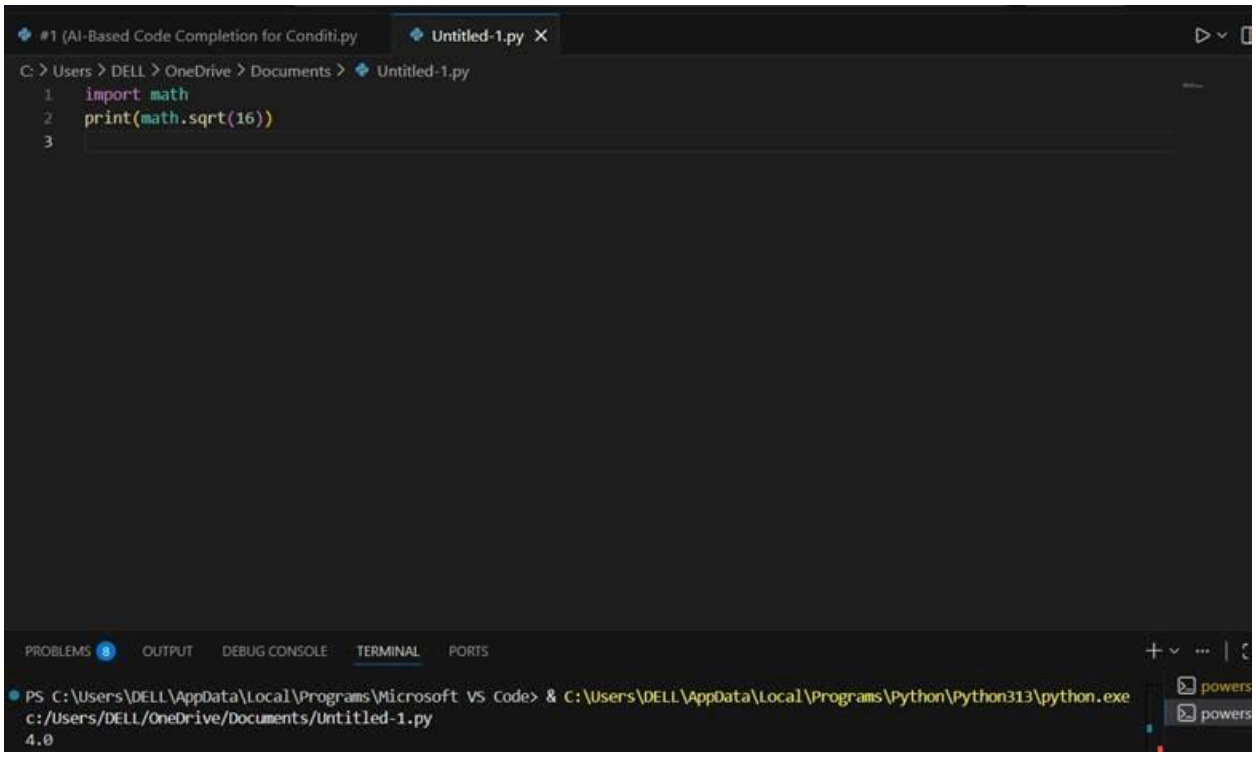
 # Bug: Wrong import

import maths

print(maths.sqrt(16))

Expected Output: Corrected to import math

Code:



Justification :

The module name is `math`, not `maths`. Importing the correct built-in module fixes the `ImportError` and allows access to functions like `sqrt()`.

Task 9 (Unreachable Code – Return Inside Loop)

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.

Code:

Justification :

The `return` statement inside the loop causes the function to exit after the first iteration. Moving the `return` outside the loop allows all elements to be processed and the total sum to be computed correctly.

Task 10 (Name Error – Undefined Variable)

Task: 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.

Code:

Justification:

The error occurs because length and width are referenced before being defined.

Defining them as function parameters ensures they are provided when the function is called, preventing the NameError and making the function logically correct and reusable.

Task 11 (Type Error – Mixing Data Types Incorrectly)

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.

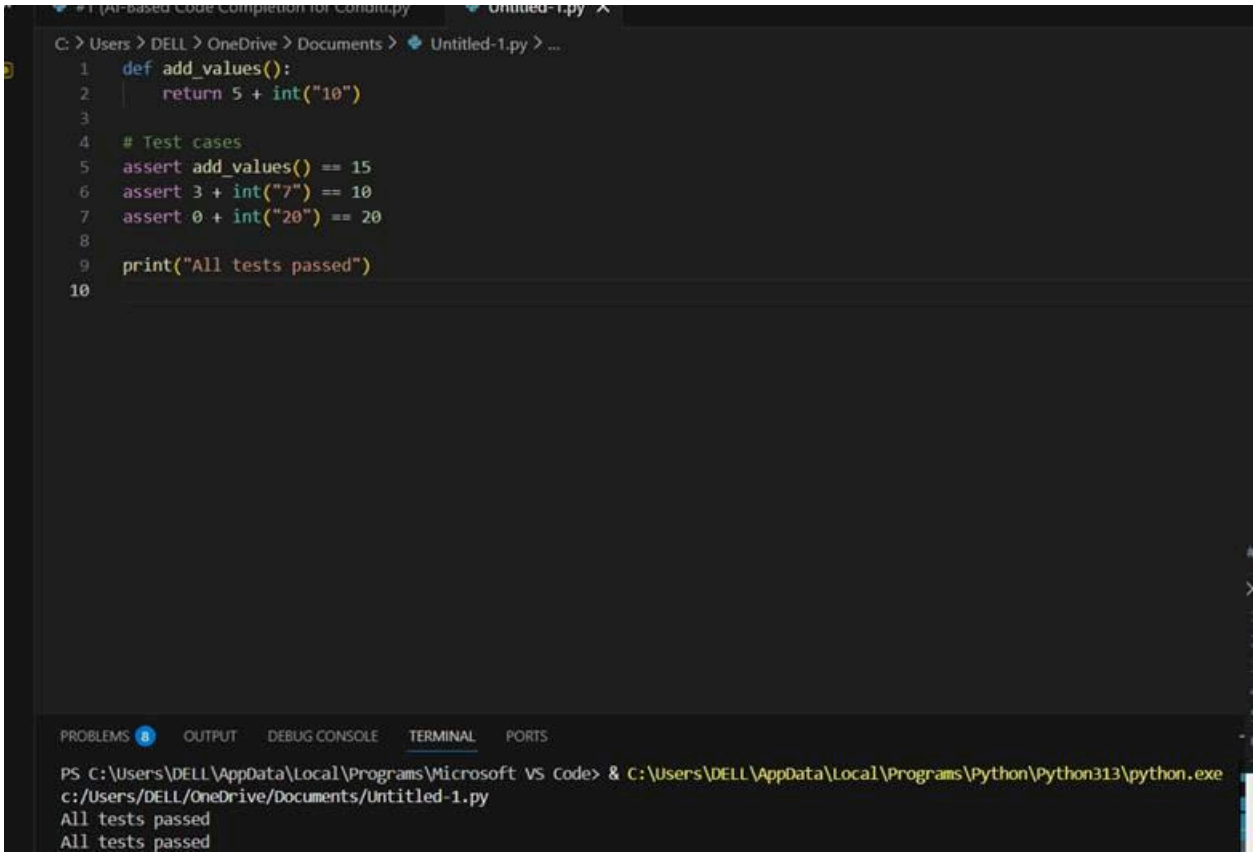
Expected Output #6:

· Corrected code with type handling.

· AI explanation of the fix.

Successful test validation.

code:



Justification:

Python does not support adding an integer and a string directly because they are different data types. Converting the string to an integer ensures both operands are compatible, preventing the TypeError and allowing correct arithmetic.

Task 12 (Type Error – String + List Concatenation)

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())
```

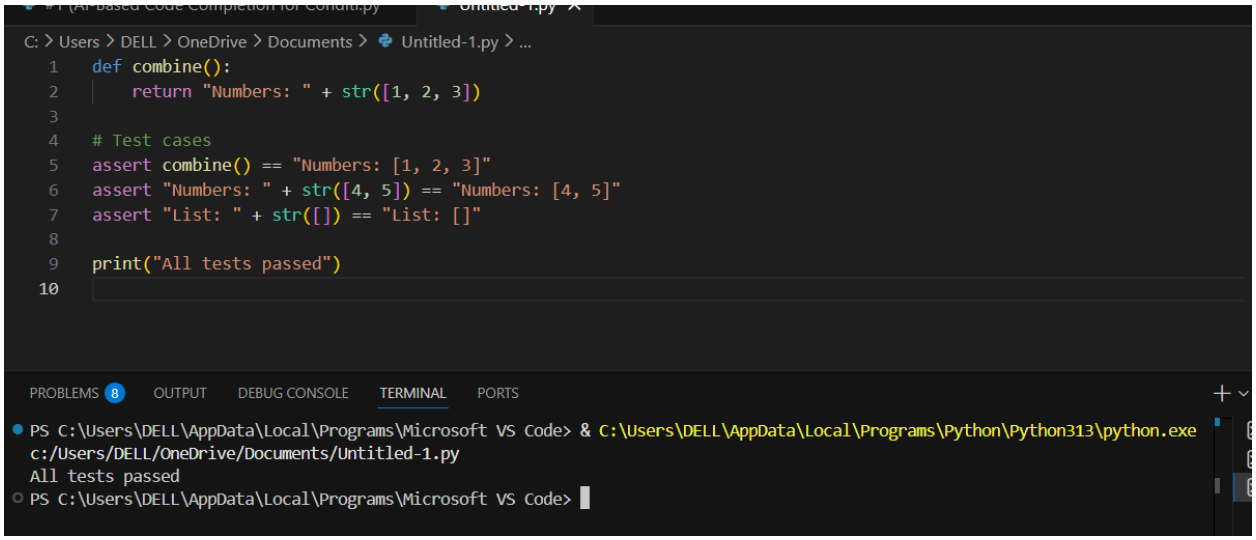
Requirements:

- 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

Code:



Justification:

Type conversion ensures compatibility between operands. Converting the list to a string prevents the `TypeError` and allows correct string concatenation, while assertions confirm the fix works as expected.

Task 13 (Type Error – Multiplying String by Float)

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:

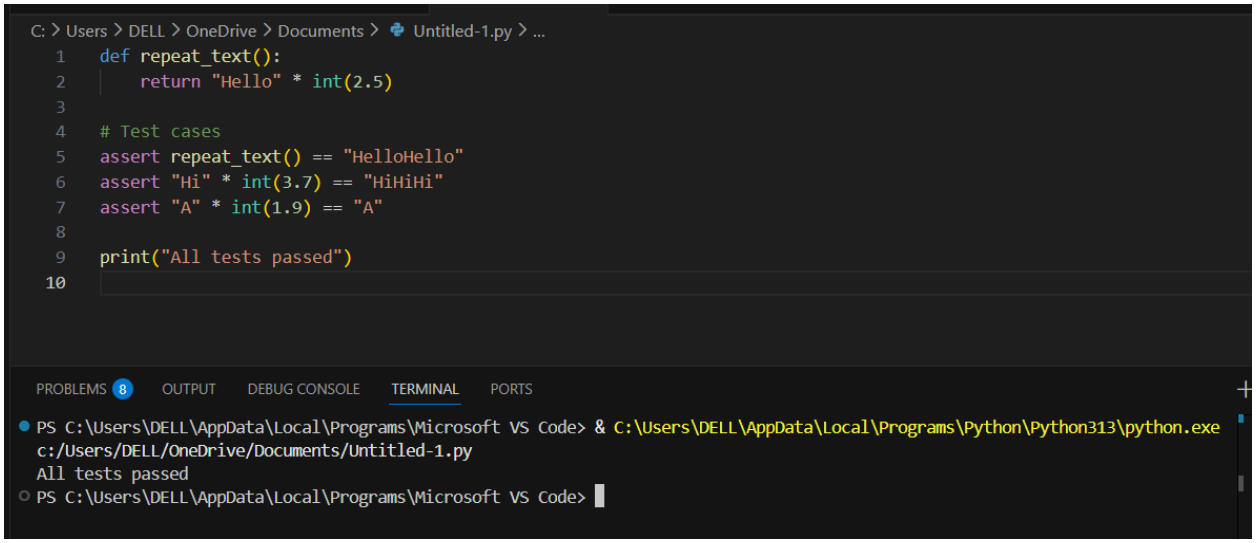
· Observe the error.

· Explain why float multiplication is invalid for strings.

· Fix by converting float to int.

· Add 3 assert test cases.

Code:



Justification:

Converting the float to an integer ensures a valid repetition count, prevents the `TypeError`, and allows the code to execute correctly with predictable results.

Task 14 (Type Error – Adding None to Integer)

Task: Analyze code where None is added to an integer.

 # Bug: Adding None and integer

def compute():

```
    value = None

    return value + 10

print(compute())
```

Requirements:

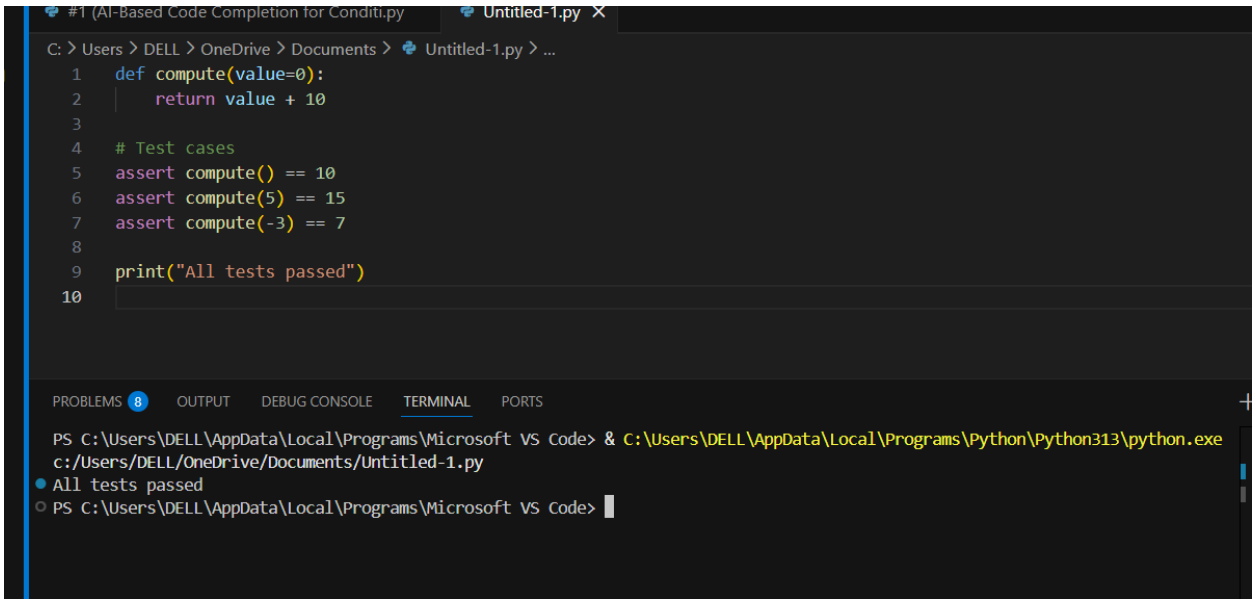
· Run and identify the error.

· Explain why NoneType cannot be added.

· Fix by assigning a default value.

· Validate using asserts.

code:



Justification:

Assigning a default numeric value ensures valid arithmetic, prevents the `TypeError`, and makes the function safe and predictable.

Task 15 (Type Error – Input Treated as String Instead of Number)

Task: Fix code where user input is not converted properly.

 # Bug: Input remains string

```
def sum_two_numbers():
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    return a + b
print(sum_two_numbers())
```

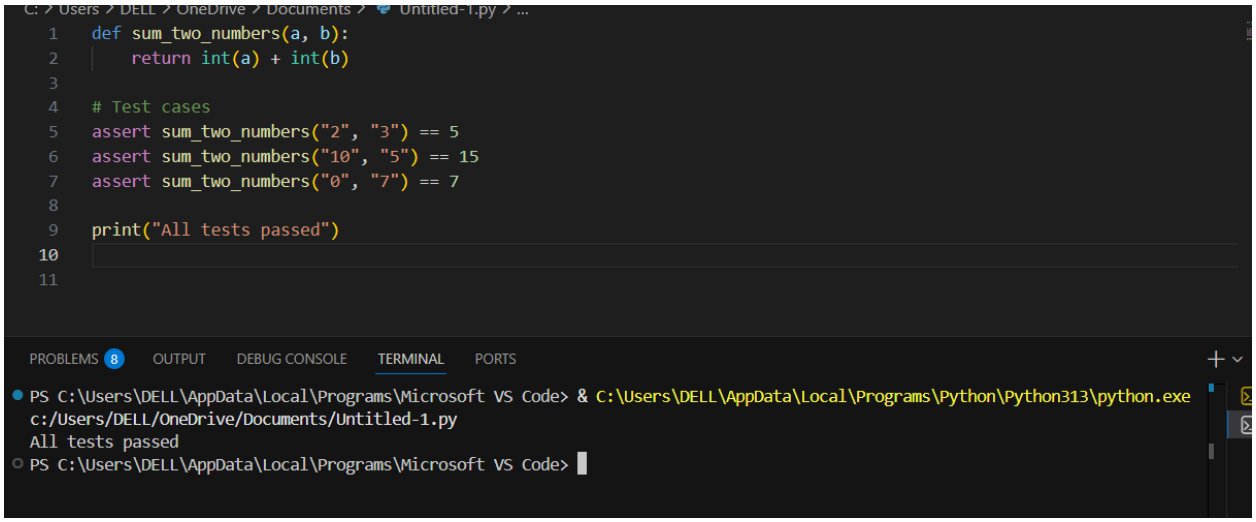
Requirements:

· Explain why input is always string.

· Fix using int() conversion.

· Verify with assert test cases

code:



Justification:

Type conversion changes string inputs into numbers, preventing logical errors and enabling correct arithmetic. Assertions confirm the fix works correctly for multiple cases.