School of Computer Science and Artificial Intelligence

Lab Assignment-7.5

Course Title : AI Assistant Coding
Name of Student : Shaik Naved Ahmed
Enrollment No. : 2303A54053
Batch No. : 47-B

Task 1 (Mutable Default Argument – Function Bug)

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

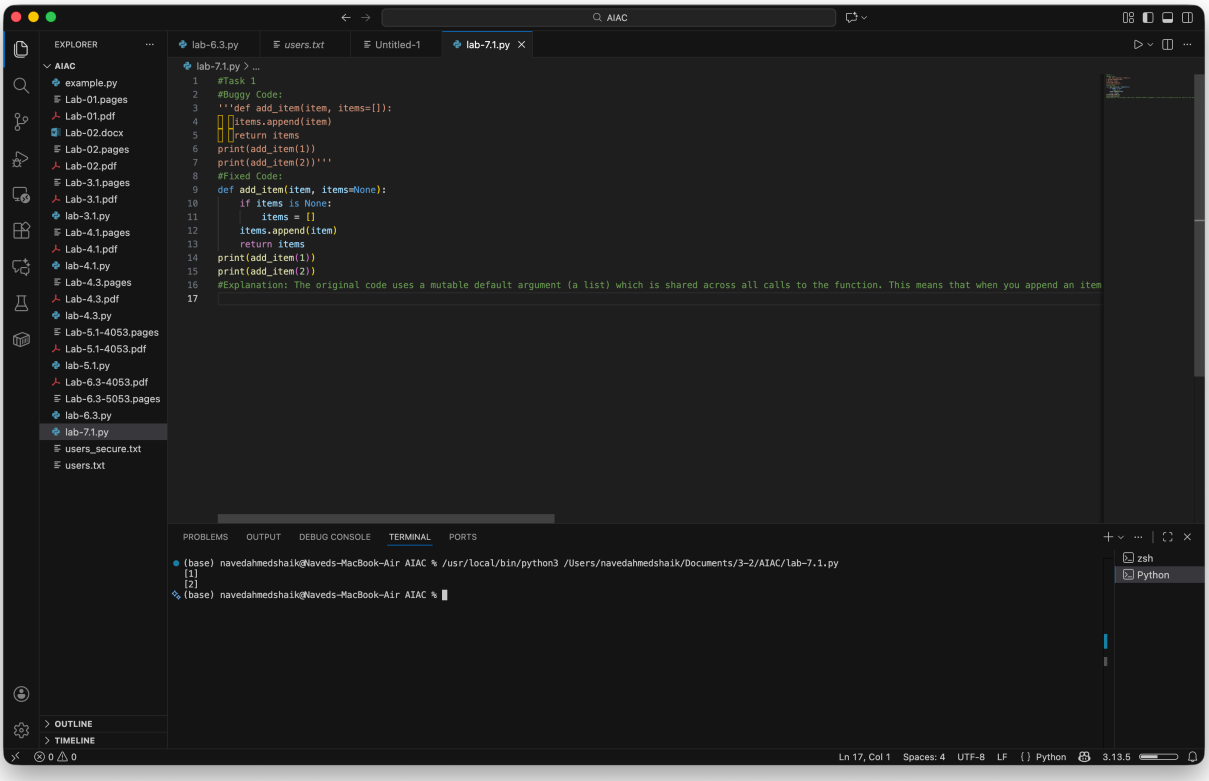
```
# Bug: Mutable default argument
def add_item(item, items=[]):
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
```

Expected Output: Corrected function avoids shared list bug.

EXPLANATION:

The original code uses a mutable default argument (a list) which is shared across all calls to the function. This means that when you append an item to the list in one call, it affects all subsequent calls that use the default list. In the fixed code, we use None as the default value and create a new list inside the function if items is None. This ensures that each call to add_item gets its own separate list.

SCREENSHOT OF GENERATED CODE:



Task 2 (Floating-Point Precision Error)

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

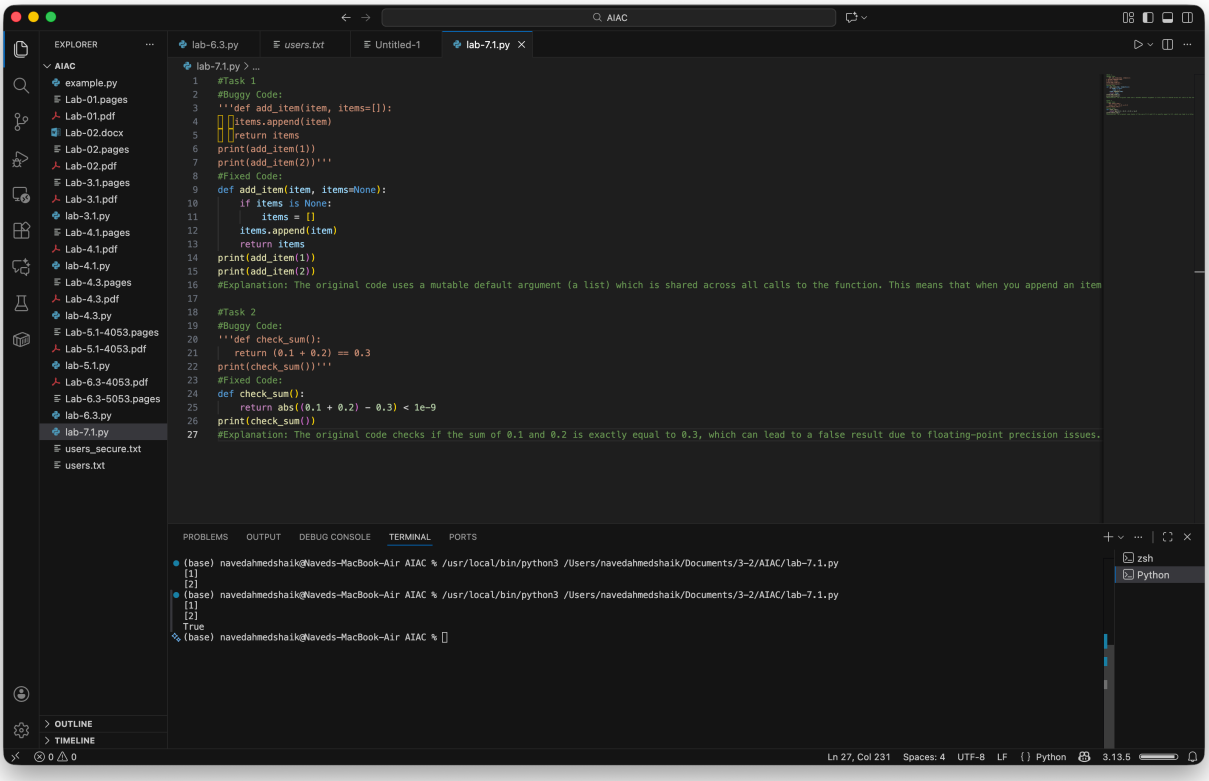
```
# Bug: Floating point precision issue
def check_sum():
  return (0.1 + 0.2) == 0.3
print(check_sum())
```

Expected Output: Corrected function

EXPLANATION:

The original code checks if the sum of 0.1 and 0.2 is exactly equal to 0.3, which can lead to a false result due to floating-point precision issues. In the fixed code, we check if the absolute difference between (0.1 + 0.2) and 0.3 is smaller than a very small number (1e-9), which accounts for the precision error and gives a more accurate result.

SCREENSHOT OF GENERATED CODE:



Task 3 (Recursion Error – Missing Base Case)

Task: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.

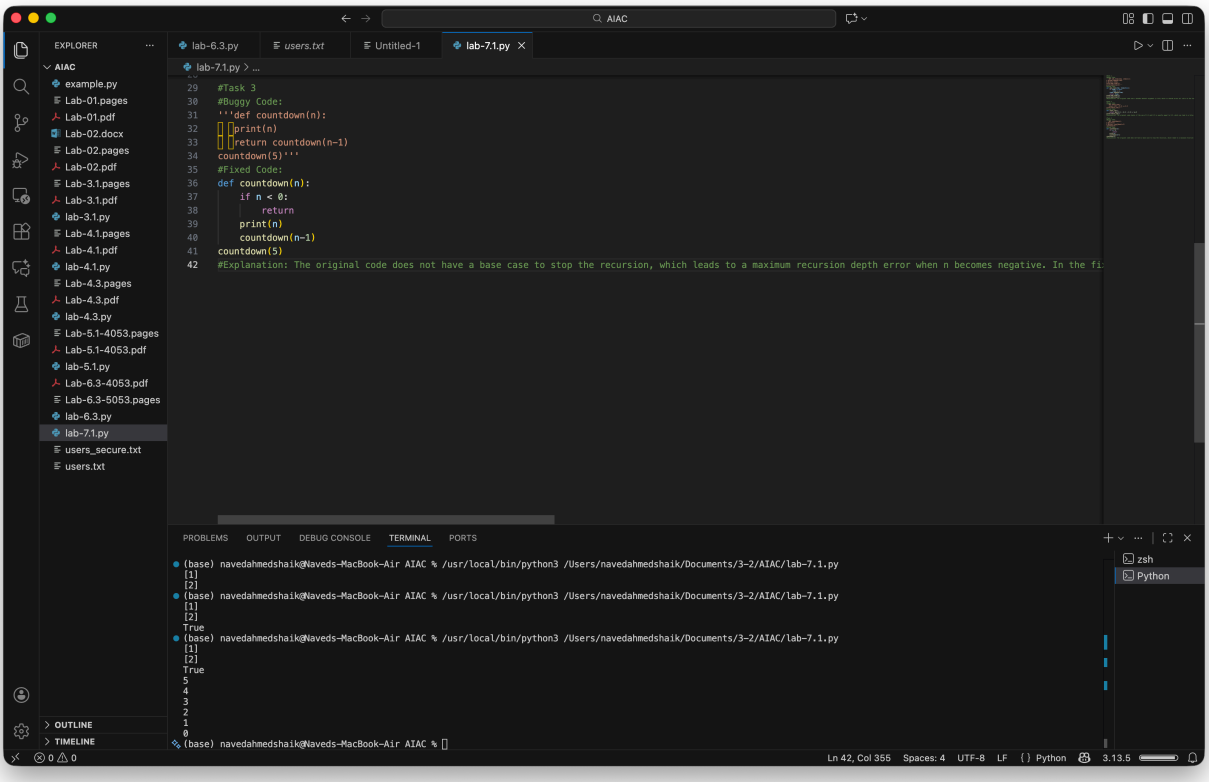
```
# Bug: No base case
def countdown(n):
  print(n)
  return countdown(n-1)
countdown(5)
```

Expected Output : Correct recursion with stopping condition.

EXPLANATION:

The original code does not have a base case to stop the recursion, which leads to a maximum recursion depth error when n becomes negative. In the fixed code, we add a base case that checks if n is less than 0, at which point the function returns and stops the recursion. This allows the countdown to work correctly without causing an error.

SCREENSHOT OF GENERATED CODE:



Task 4 (Dictionary Key Error)

Task: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

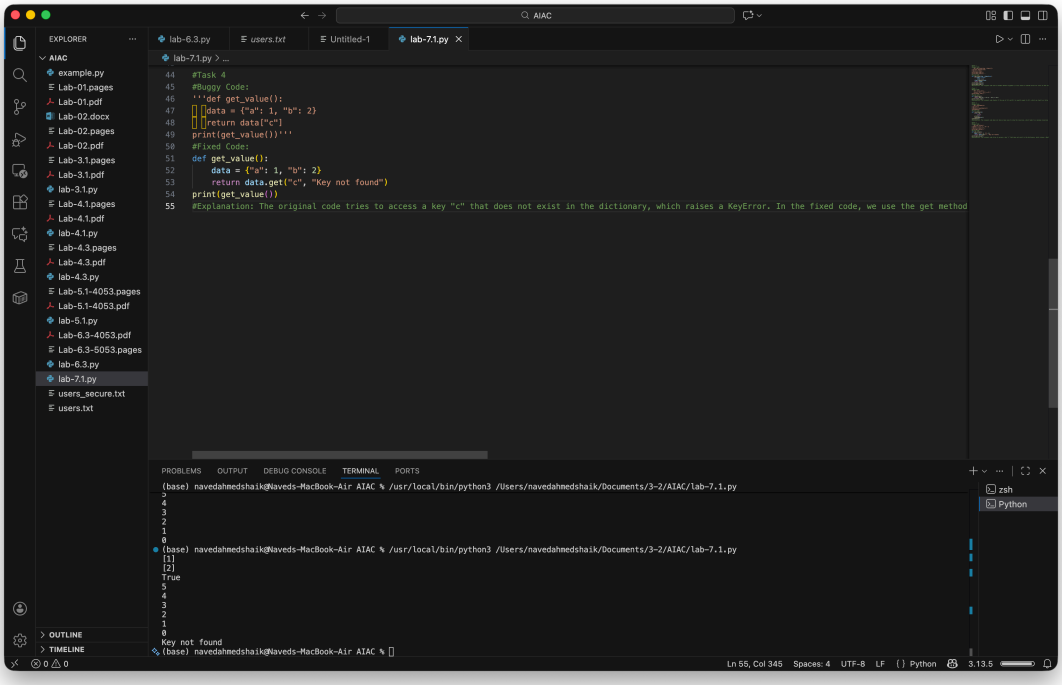
```
# Bug: Accessing non-existing key
def get_value():
  data = {"a": 1, "b": 2}
  return data["c"]
print(get_value())
```

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

EXPLANATION:

The original code tries to access a key "c" that does not exist in the dictionary, which raises a KeyError. In the fixed code, we use the get method of the dictionary, which allows us to specify a default value ("Key not found") to return if the key is not found. This prevents the error and provides a more user-friendly message.

SCREENSHOT OF GENERATED CODE:



Task 5 (Infinite Loop – Wrong Condition)

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

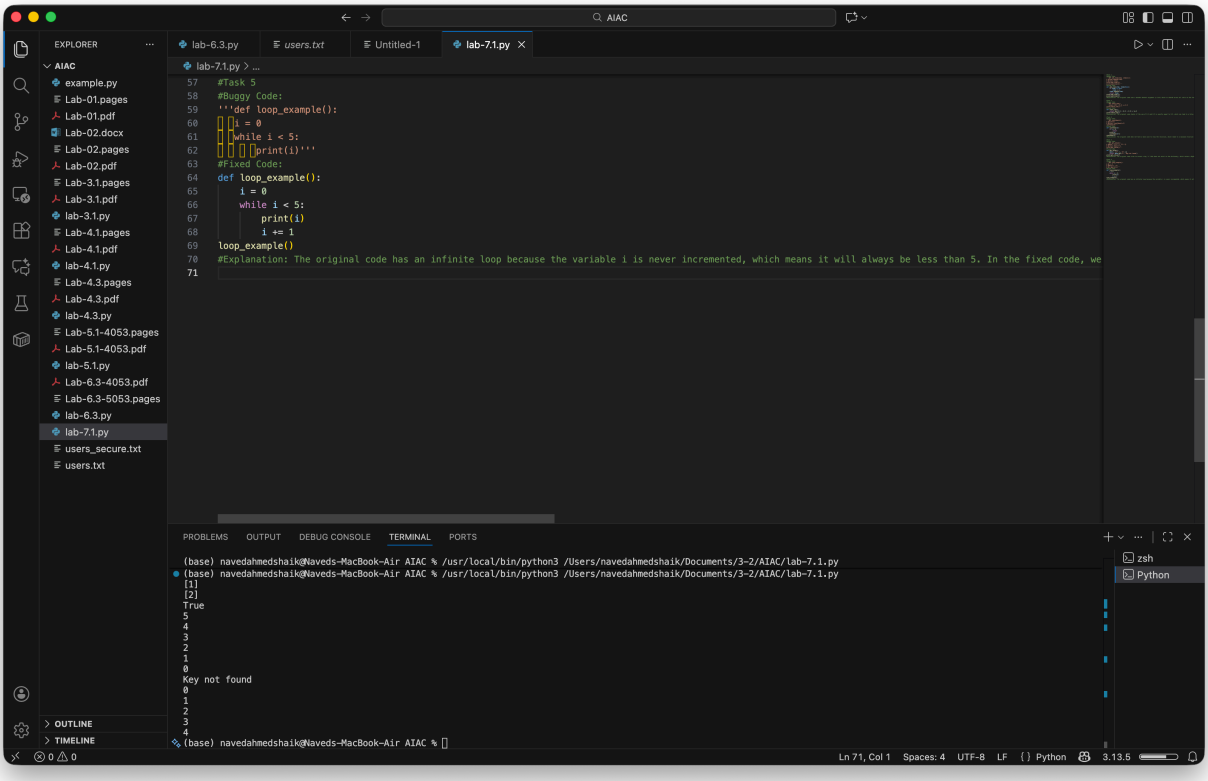
```
# Bug: Infinite loop
def loop_example():
  i = 0
  while i < 5:
     print(i)
```

Expected Output: Corrected loop increments i.

EXPLANATION:

The original code has an infinite loop because the variable i is never incremented, which means it will always be less than 5. In the fixed code, we add i += 1 inside the loop to ensure that i is incremented on each iteration, allowing the loop to eventually terminate when i reaches 5.

SCREENSHOT OF GENERATED CODE:



Task 6 (Unpacking Error – Wrong Variables)

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

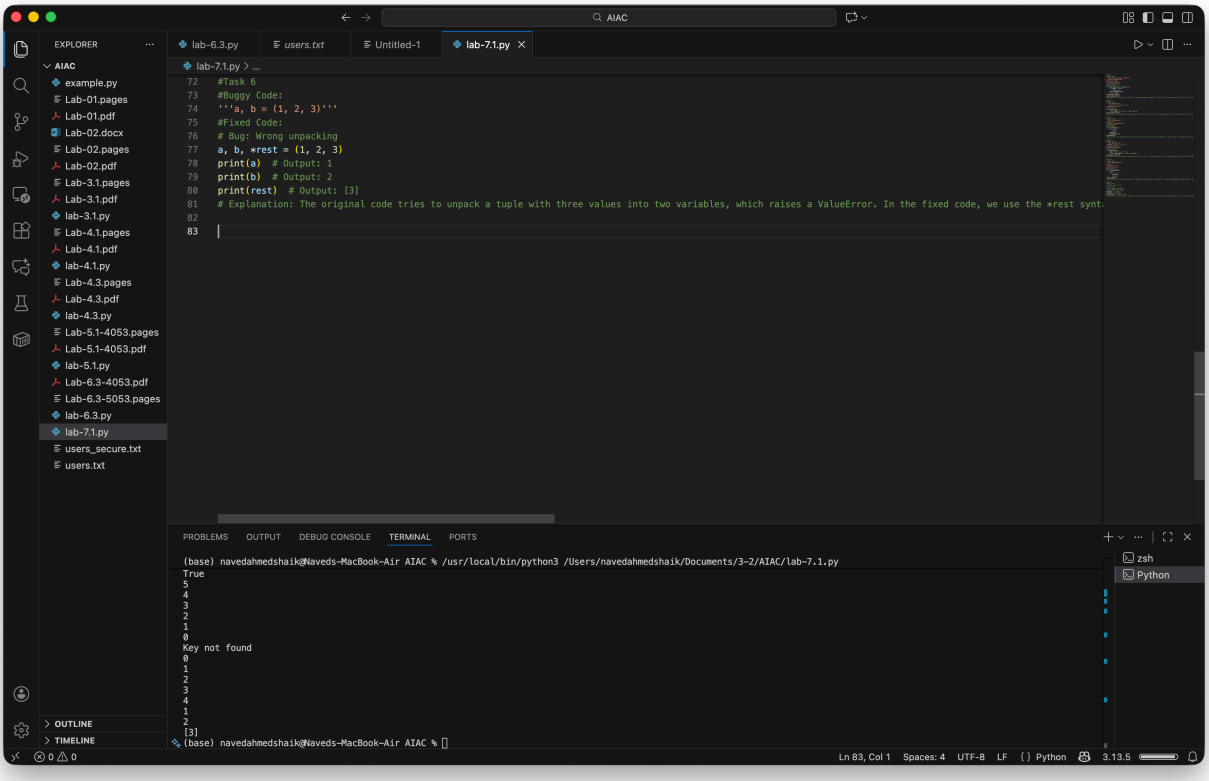
```
# Bug: Wrong unpacking
a, b = (1, 2, 3)
```

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

EXPLANATION:

The original code tries to unpack a tuple with three values into two variables, which raises a ValueError. In the fixed code, we use the *rest syntax to capture any extra values in a list called rest. This allows us to unpack the first two values into a and b while still keeping the remaining value in rest without causing an error.

SCREENSHOT OF GENERATED CODE:



Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.

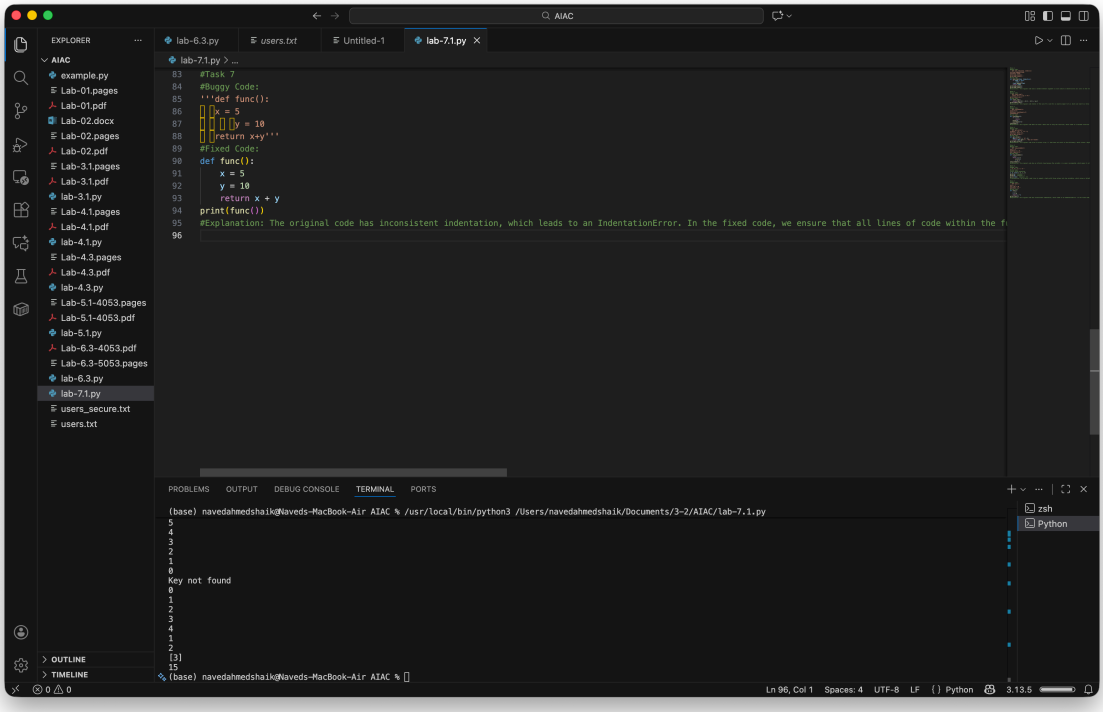
```
# Bug: Mixed indentation
def func():
  x = 5
     y = 10
  return x+y
```

Expected Output : Consistent indentation applied.

EXPLANATION:

The original code has inconsistent indentation, which leads to an IndentationError. In the fixed code, we ensure that all lines of code within the function have the same level of indentation. This allows the function to execute properly and return the correct result when called.

SCREENSHOT OF GENERATED CODE:



Task 8 (Import Error – Wrong Module Usage)

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

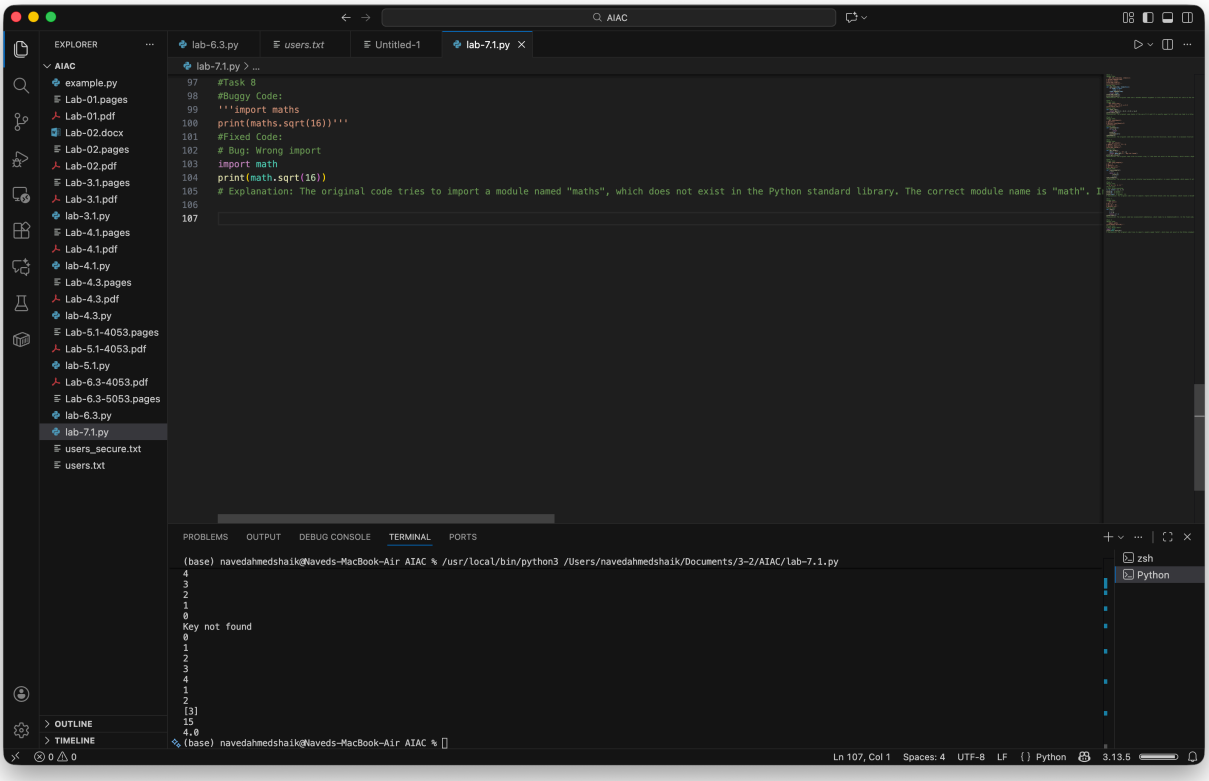
```
# Bug: Wrong import
import maths
print(maths.sqrt(16))
```

Expected Output: Corrected to import math

EXPLANATION:

The original code tries to import a module named "maths", which does not exist in the Python standard library. The correct module name is "math". In the fixed code, we change the import statement to import math, which allows us to use the sqrt function correctly and get the expected output of 4.0 when calculating the square root of 16.

SCREENSHOT OF GENERATED CODE:



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.

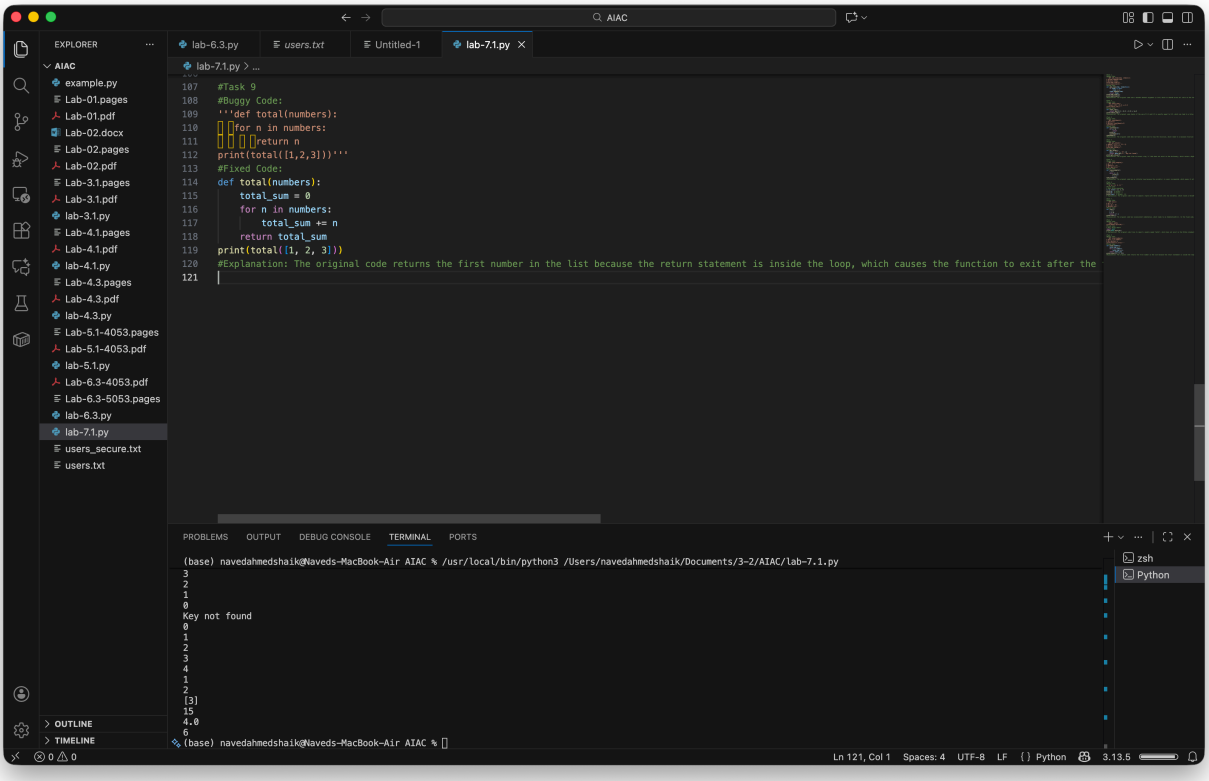
```
# Bug: Early return inside loop
def total(numbers):
    for n in numbers:
        return n
print(total([1,2,3]))
```

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

EXPLANATION:

The original code returns the first number in the list because the return statement is inside the loop, which causes the function to exit after the first iteration. In the fixed code, we initialize a variable total_sum to 0 and add each number in the list to total_sum within the loop, ensuring that all numbers are summed correctly before returning the total.

SCREENSHOT OF GENERATED CODE:



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 :

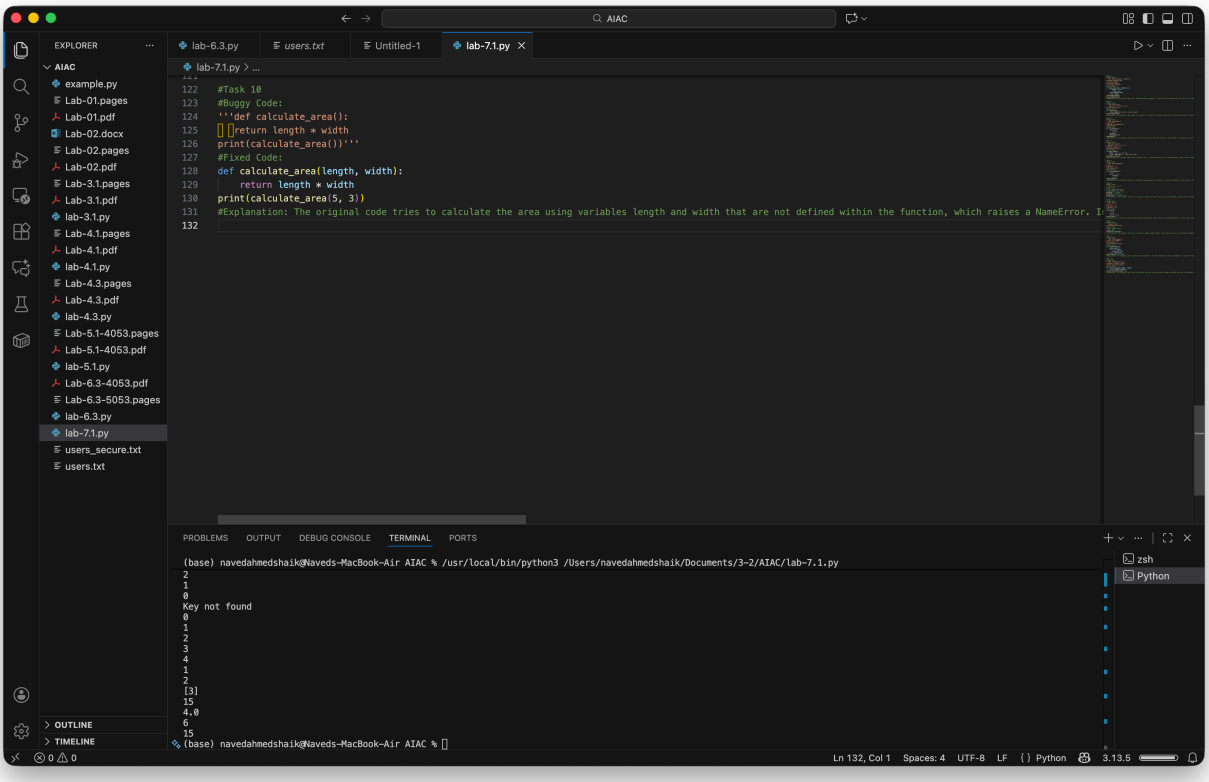
• Corrected code with parameters.
• AI explanation of the bug.

Successful execution of assertions.

EXPLANATION:

The original code tries to calculate the area using variables length and width that are not defined within the function, which raises a NameError. In the fixed code, we modify the function to accept length and width as parameters, allowing us to pass the necessary values when calling the function. This way, we can calculate the area correctly and get the expected output of 15 when calculating the area of a rectangle with length 5 and width 3.

SCREENSHOT OF GENERATED CODE:



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:

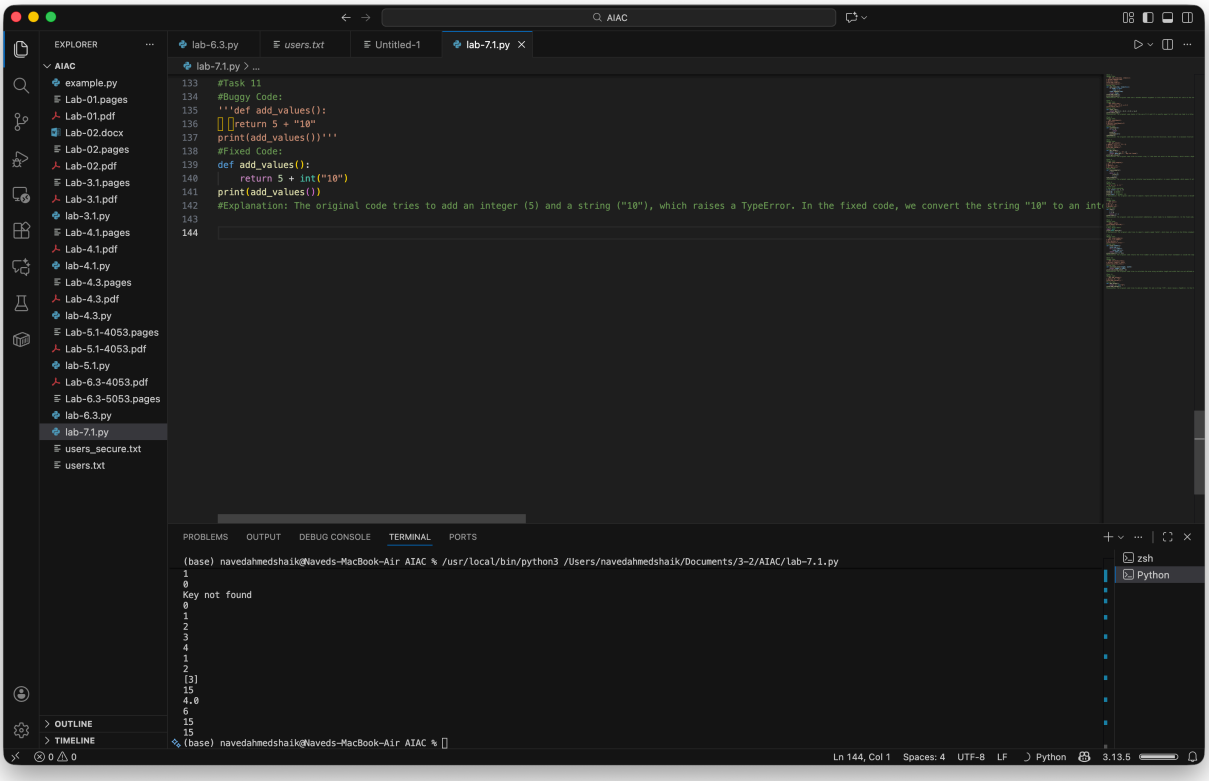
• Corrected code with type handling.
• AI explanation of the fix.

Successful test validation.

EXPLANATION:

The original code tries to add an integer (5) and a string ("10"), which raises a TypeError. In the fixed code, we convert the string "10" to an integer using the int() function before performing the addition. This allows us to successfully add the two values and get the expected output of 15.

SCREENSHOT OF GENERATED CODE:



Task 12 (Type Error – String + List Concatenation)

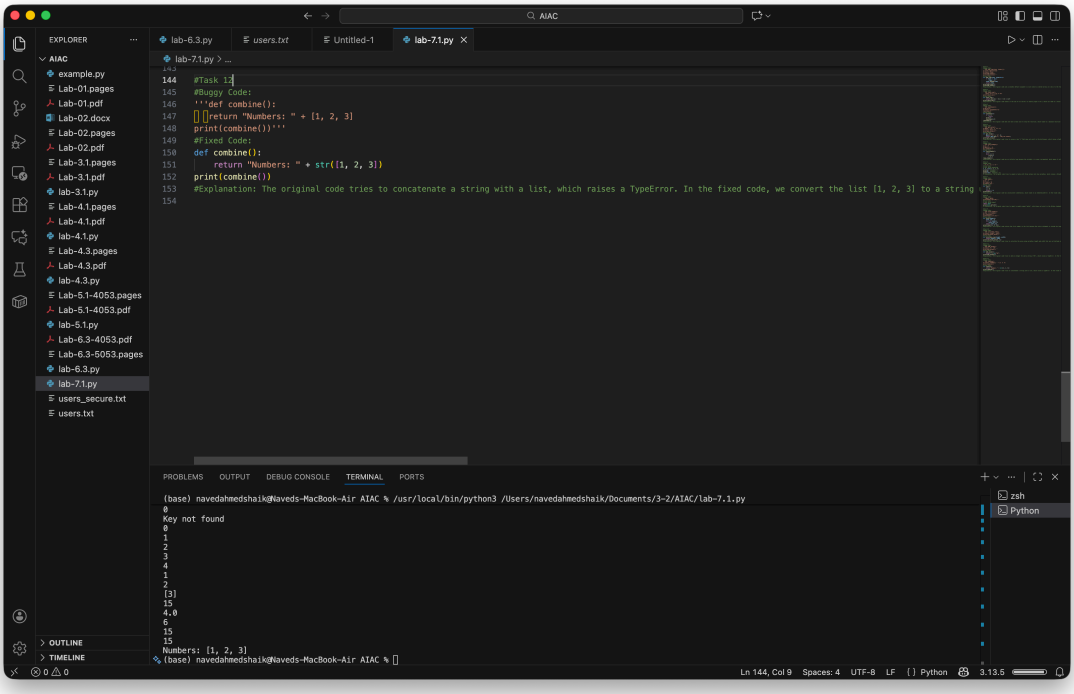
Task: Analyze code where a string is incorrectly added to a list.

```
# 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.

SCREENSHOT OF GENERATED CODE:



EXPLANATION:

The original code tries to concatenate a string with a list, which raises a TypeError. In the fixed code, we convert the list [1, 2, 3] to a string using the str() function before concatenating it with the "Numbers: " string. This allows us to successfully combine the string and the list representation, resulting in the output "Numbers: [1, 2, 3]".

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())
```

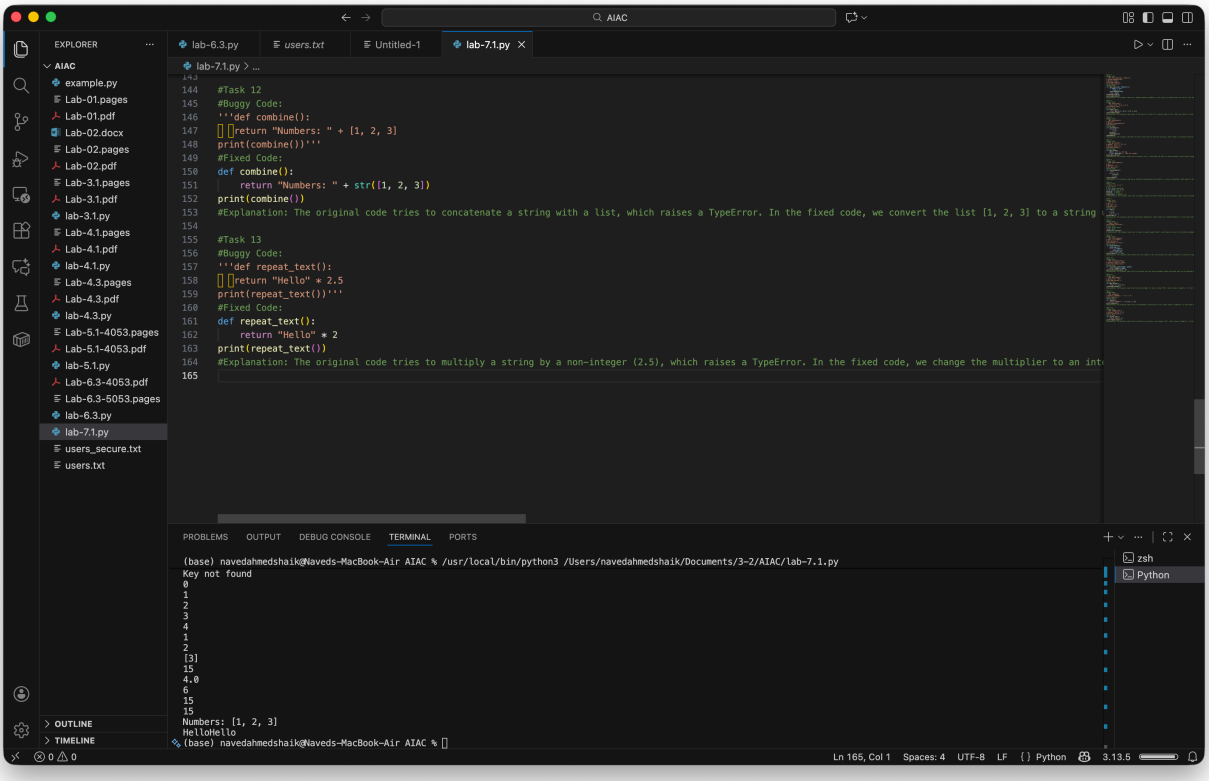
Requirements:

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

EXPLANATION:

The original code tries to multiply a string by a non-integer (2.5), which raises a TypeError. In the fixed code, we change the multiplier to an integer (2), which allows us to repeat the string "Hello" twice successfully, resulting in the output "HelloHello".

SCREENSHOT OF GENERATED CODE:



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:

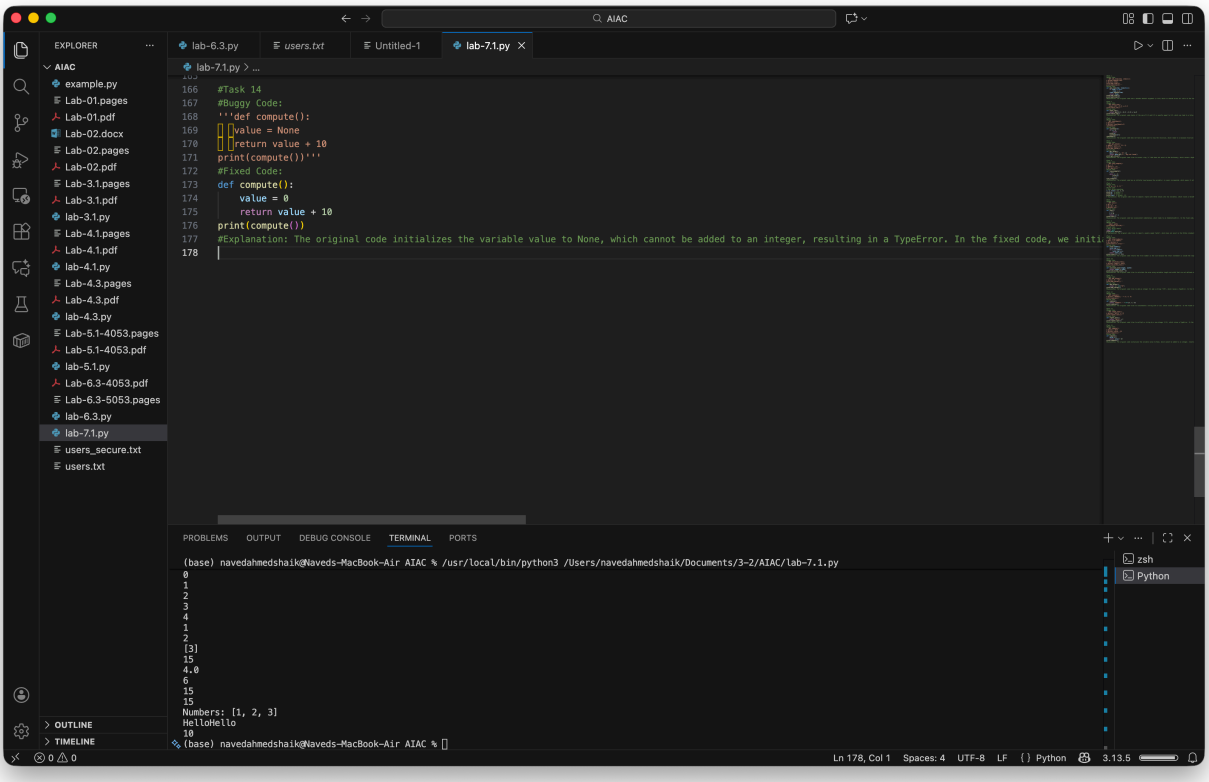
- Run and identify the error.

• Explain why NoneType cannot be added.
• Fix by assigning a default value.
• Validate using asserts.

EXPLANATION:

The original code initializes the variable value to None, which cannot be added to an integer, resulting in a TypeError. In the fixed code, we initialize value to 0, which allows us to perform the addition without any errors. The function now returns 10, which is the expected output when adding 10 to the initial value of 0.

SCREENSHOT OF GENERATED CODE:



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.

EXPLANATION:

The original code takes user input as strings, and when it tries to add them together, it concatenates the strings instead of performing numerical addition, which is not the intended behavior. In the fixed code, we convert the input strings to floats using the float() function before performing the addition. This allows us to correctly sum the two numbers and return the expected numerical result.

SCREENSHOT OF GENERATED CODE: