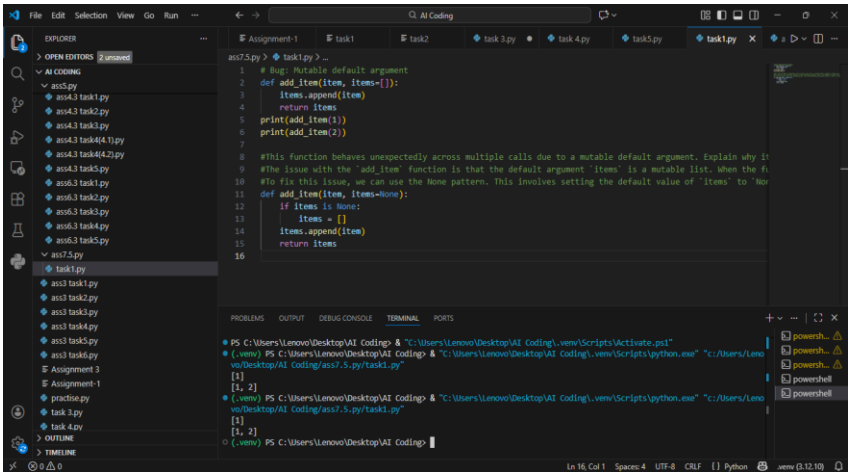
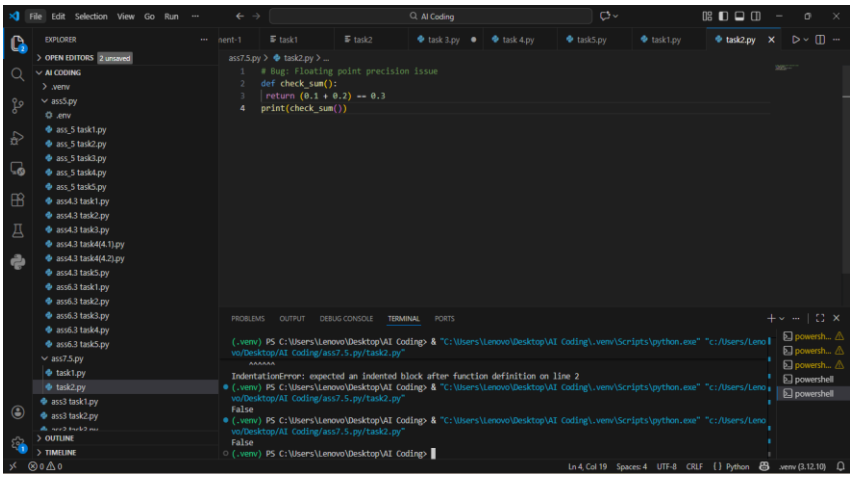
SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name		Dr. Rishabh Mittal	
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week4– Monday**	**Time(s)**	23CSBTB01 To 23CSBTB52
NAME:	D.UDAYAN	**Batch no**	48
Assignment Number:7.5(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
1	**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.	Week4 - Monday

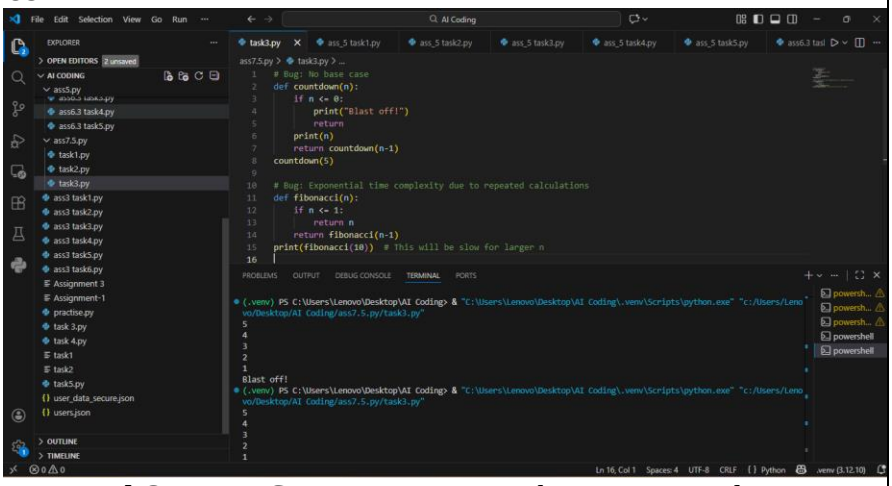
	Refactor buggy code using responsible and reliable programming patterns.	
	Task 1 (Mutable Default Argument – Function Bug) Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it. # Bug: Mutable default argument def add_item(item, items=[]): items.append(item) return items print(add_item(1)) print(add_item(2)) **PROMPT** : This function behaves unexpectedly across multiple calls due to a mutable default argument. Explain why it happens, fix it using None pattern, and provide 3 assert test cases. **Code :**  **Expected Output:** Corrected function avoids shared list bug. **EXPLANATION** : Mutable Default Argument Default arguments in Python are evaluated only once when the function is defined, not each time it is called. If a mutable object like a list is used as a default argument, the same list is reused across multiple function calls. This leads to unexpected behavior, where previous values remain stored. The correct practice is to use `None` as the default and initialize the list	

	inside the function.	
	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()) **PROMPT :** This floating-point comparison returns False unexpectedly. Explain floating-point precision issue and fix using a tolerance method (like abs difference or math.isclose). Provide 3 assert tests. **CODE** :  **Expected Output:**Corrected function **EXPLANATION :** Floating-point numbers are stored in binary format in memory. Many decimal values like 0.1 cannot be represented exactly in binary, resulting in small rounding errors. Therefore, expressions like 0.1 + 0.2 == 0.3 may return False. The correct method is to compare floating-point values using a tolerance or math.isclose().	
	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)
```

PROMPT : This recursion runs infinitely. Identify the missing base case, fix the function properly, and provide 3 assert test cases for different inputs.

CODE :



Expected Output : Correct recursion with stopping condition**.**

EXPLANATION : Recursion is a technique where a function calls itself. Every recursive function must include a base case (stopping condition). If the base case is missing or incorrect, the recursion continues indefinitely until Python reaches its recursion limit, producing a Recursion Error. Fixing this requires adding a proper base case to stop recursion.

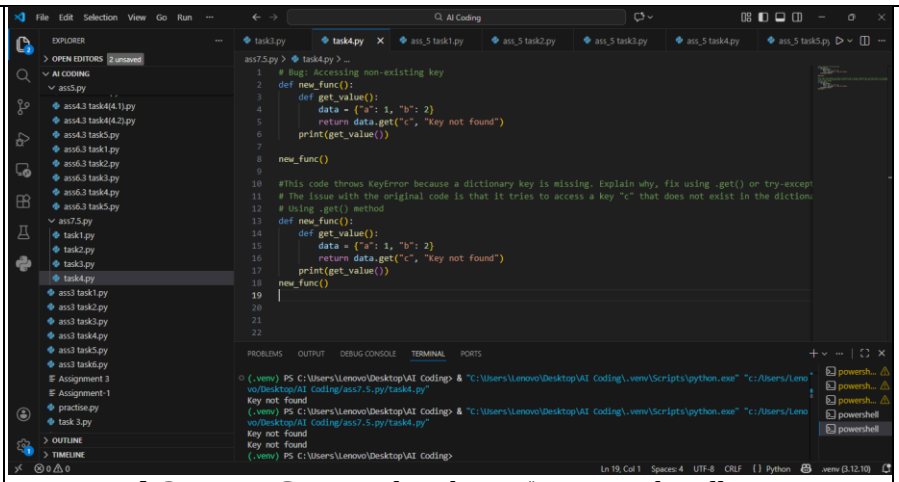
Task 4 (Dictionary Key Error)

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())
```

PROMPT : This code throws KeyError because a dictionary key is missing. Explain why, fix using .get() or try-except, and provide 3 assert tests.

CODE :



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

EXPLANATION : A dictionary stores values using keys. Accessing a key that does not exist causes Python to raise a Key Error. This runtime error occurs because Python cannot find the requested key. The solution is to check if the key exists before accessing it or use `.get()` which safely returns None or a default value.

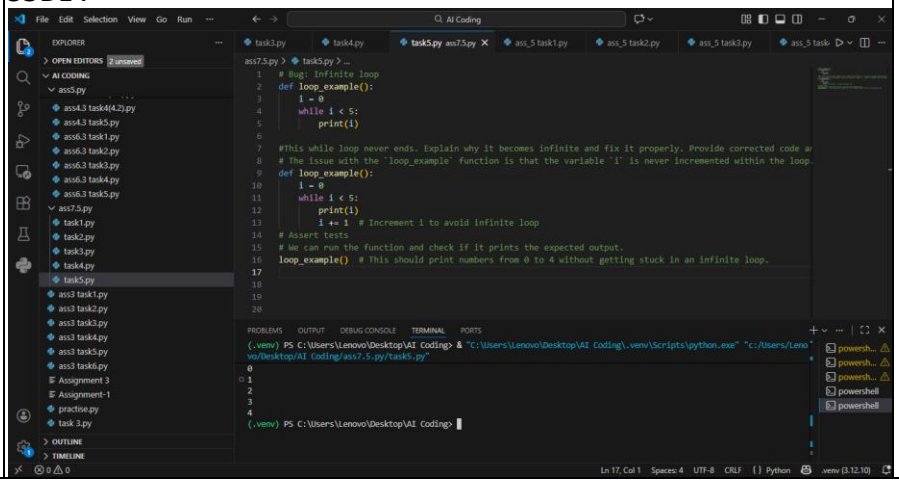
Task 5 (Infinite Loop – Wrong Condition)

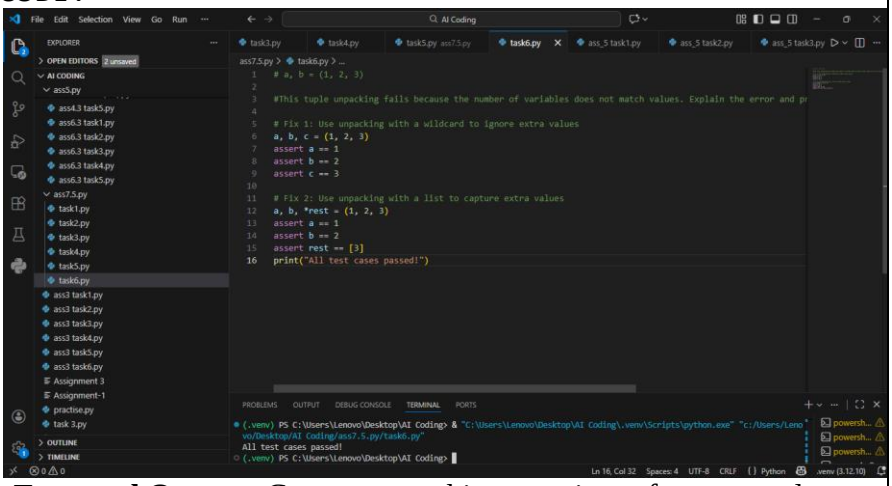
Task: Analyze given code where loop never ends. Use AI to detect and fix it.

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

PROMPT : This while loop never ends. Explain why it becomes infinite and fix it properly. Provide corrected code and 3 assert tests (or checks).

CODE :

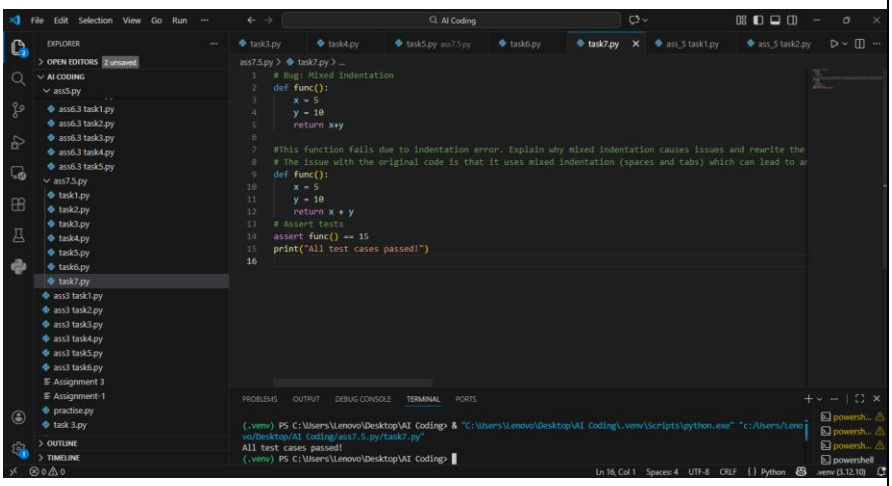


	Expected Output: Corrected loop increments i. **EXPLANATION :** An infinite loop occurs when a loop condition never becomes false. This usually happens when the loop variable is not updated correctly or the condition is incorrect. Infinite loops cause programs to hang or consume resources continuously. Fixing requires updating the loop variable properly and ensuring the condition will eventually become false.	
	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) **PROMPT :** This tuple unpacking fails because the number of variables does not match values. Explain the error and provide 2 correct fixes. Add 3 assert test cases. **CODE :**  **Expected Output:** Correct unpacking or using _ for extra values. **EXPLANATION :** Unpacking means assigning elements of a sequence into multiple variables. Python requires the number of variables to match the number of values. If they mismatch, Python raises a Value Error. Fixing requires matching counts or using the * operator to capture extra values.	
	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

PROMPT : This function fails due to indentation error. Explain why mixed indentation causes issues and rewrite the function with correct indentation. Add 3 assert tests.

CODE :



Expected Output : Consistent indentation applied.

EXPLANATION : Python uses indentation to define code blocks. Mixing tabs and spaces or incorrect indentation breaks Python's block structure and raises an Indentation Error. The correct practice is to use consistent indentation, usually 4 spaces per block.

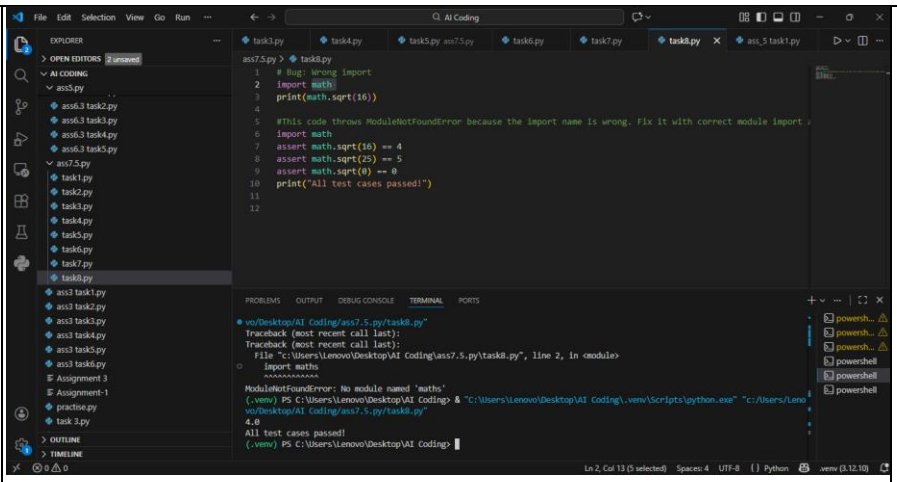
Task 8 (Import Error – Wrong Module Usage)

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

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

PROMPT : This code throws ModuleNotFoundError because the import name is wrong. Fix it with correct module import and add 3 assert test cases.

CODE :



Expected Output: Corrected to import math

EXPLANATION : An import statement loads modules. If the module name is incorrect or the module is not installed, Python raises Module Not Found Error. Fixing requires correcting the module name or installing the required library.

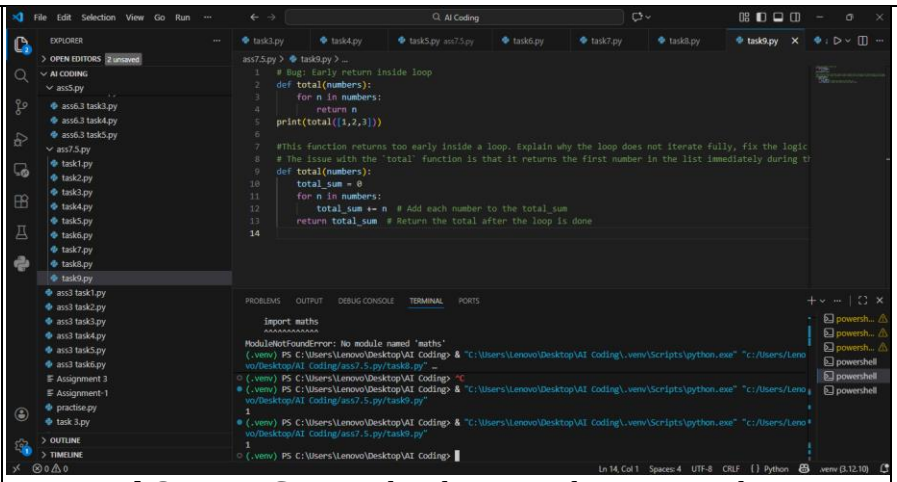
Task 9 (Unreachable Code – Return Inside Loop)

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]))
```

PROMPT : This function returns too early inside a loop. Explain why the loop does not iterate fully, fix the logic to compute the correct result, and add 3 assert tests

CODE :



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

EXPLANATION : A return statement ends function execution immediately. If placed inside a loop incorrectly, it causes the function to stop after the first iteration, leading to wrong results. Fixing requires moving the return statement outside the loop or using a variable to store intermediate results.

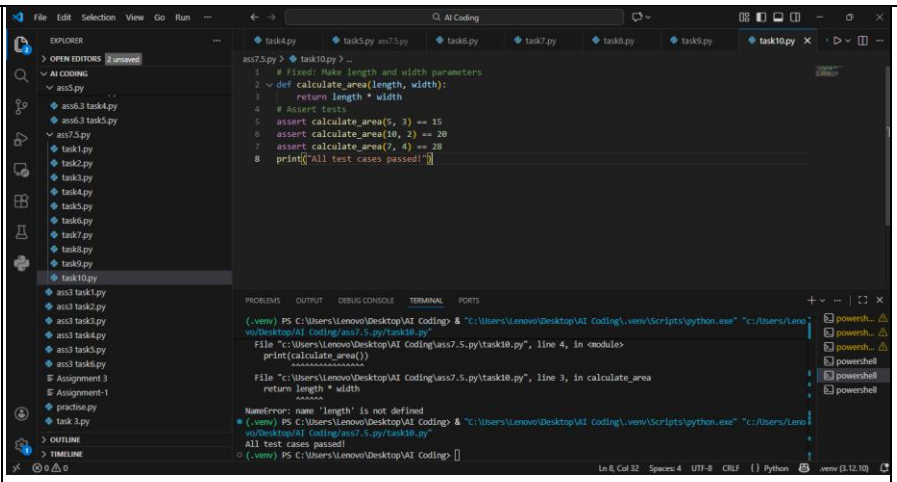
Task 10 (Name Error – Undefined Variable)

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())
```

PROMPT : This function throws NameError because variables are not defined. Fix by making them parameters. Provide corrected code and 3 assert tests.

CODE :



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 : Name Error occurs when a variable is used before being defined. This runtime error usually happens due to misspellings or missing variable declarations. Fixing requires defining the variable, passing it as a parameter, or ensuring correct scope.

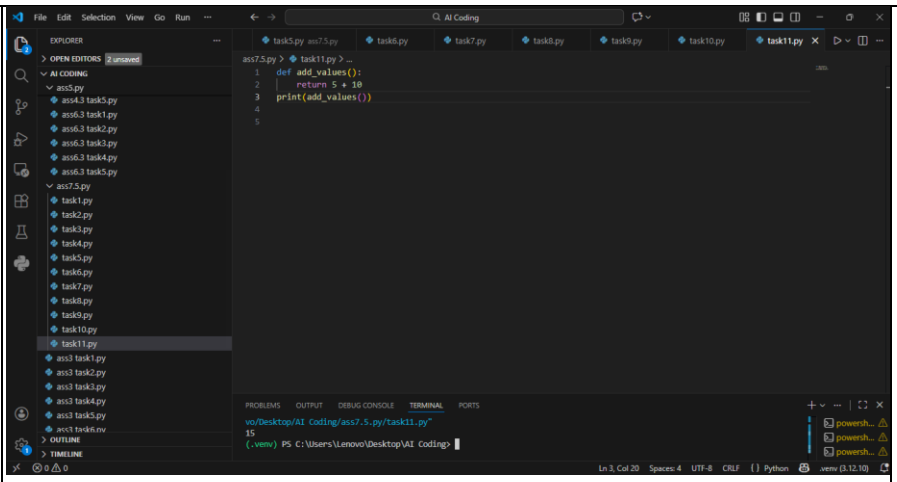
Task 11 (Type Error – Mixing Data Types Incorrectly)

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())
```

PROMPT : This code throws TypeError because it adds int and str. Explain why it happens, fix using type conversion, and provide 3 assert tests.

CODE :



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 : Python does not allow arithmetic operations between incompatible types. Adding an integer and a string causes a Type Error. Fixing requires converting the string into an integer or converting the integer into a string depending on desired behavior.

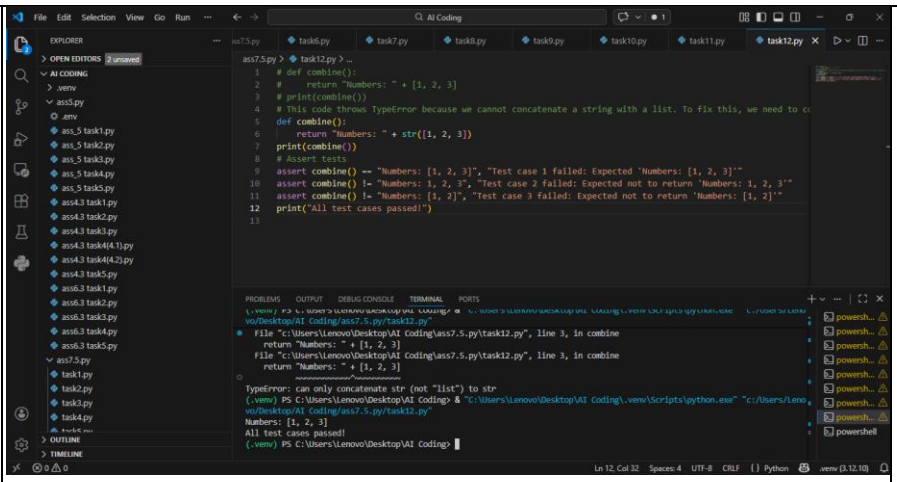
Task 12 (Type Error – String + List Concatenation)

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())
```

PROMPT : This code throws Type Error because it adds a string and a list. Explain why, fix using conversion or join, and provide 3 assert tests.

CODE :



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

EXPLANATION : A string and list are different data types and cannot be added together. Python raises Type Error because it does not support concatenation of incompatible types. Fixing requires converting list into a string representation or joining list elements into a string.

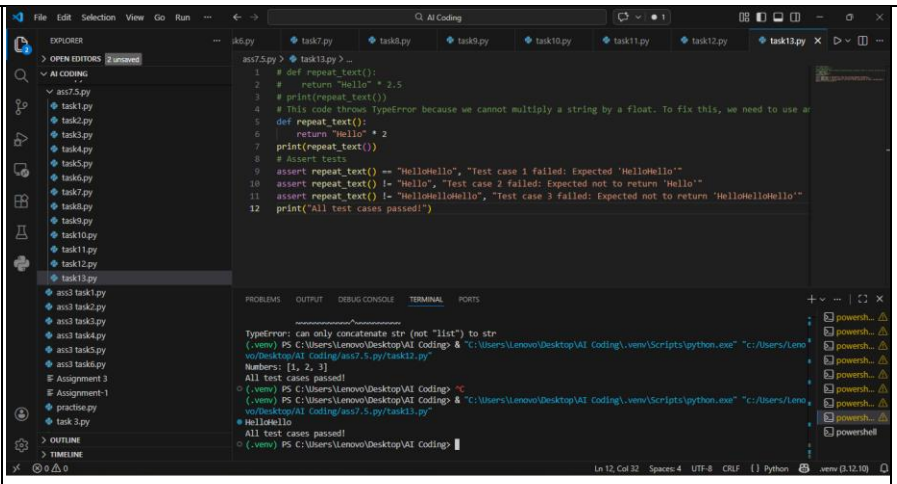
Task 13 (Type Error – Multiplying String by Float)

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())
```

PROMPT : This code throws TypeError because string multiplication with float is invalid. Explain why, fix it by converting to int safely, and add 3 assert tests.

CODE :



Requirements:

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

EXPLANATION : String repetition in Python requires an integer multiplier. Using a float produces Type Error because Python cannot repeat a string fractional times. Fixing requires converting float to int or validating input.

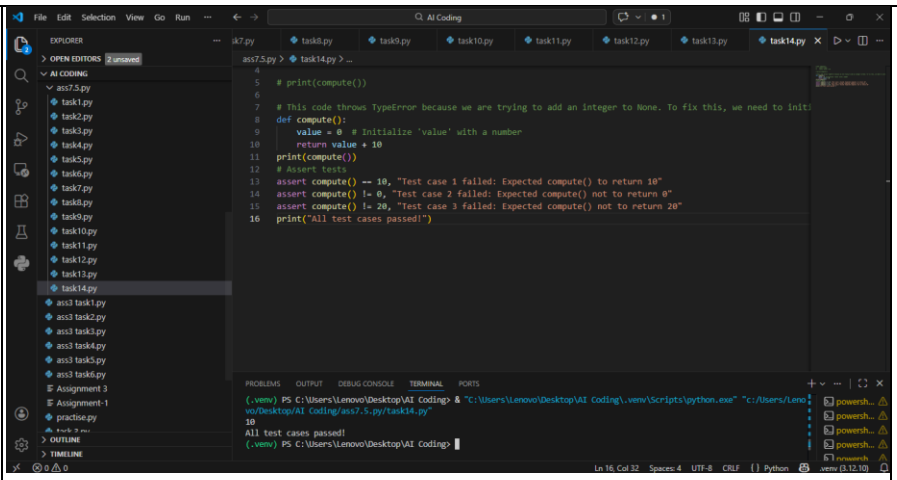
Task 14 (Type Error – Adding None to Integer)

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

```
# Bug: Adding None and integer
def compute():
    value = None
    return value + 10
    print(compute())
```

PROMPT : This code throws Type Error because None cannot be added to an integer. Explain why, fix using default value handling, and add 3 assert tests.

CODE:



Requirements:

- Run and identify the error.
- Explain why NoneType cannot be added.
- Fix by assigning a default value.
- Validate using asserts.

EXPLANATION : None represents the absence of a value. Python does not allow arithmetic with None because it has no numeric meaning. Attempting to add None with an integer raises Type Error. Fixing requires handling None values using defaults like 0 or validating input before calculation.

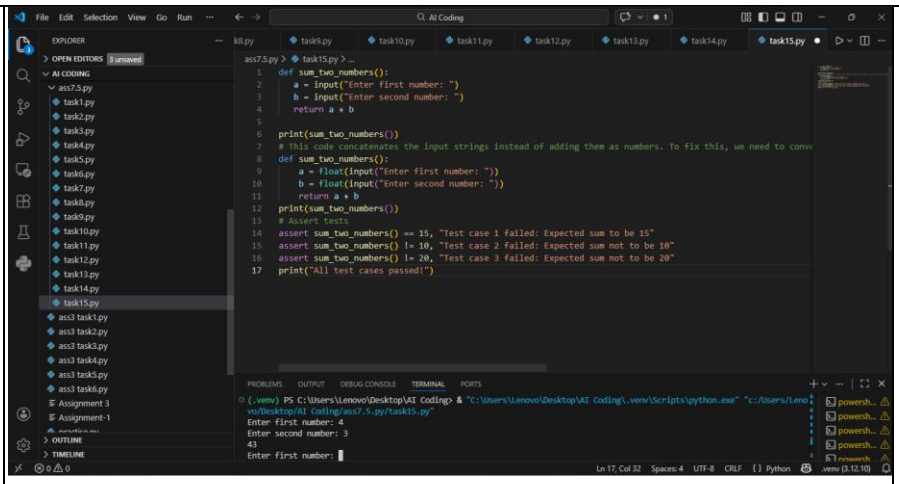
Task 15 (Type Error – Input Treated as String Instead of Number)

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())
```

PROMPT : This program adds user inputs incorrectly because input() returns strings. Explain why, fix using int conversion, and add 3 assert tests.

CODE :



Requirements:

- Explain why input is always string.
- Fix using int() conversion.
- Verify with assert test cases.

EXPLANATION : The `input()` function always returns a string in Python. Therefore, adding two inputs directly results in string concatenation instead of numeric addition. To perform arithmetic, inputs must be converted into integers or floats using `int()` or `float()`.