Lab Assignment 7.4

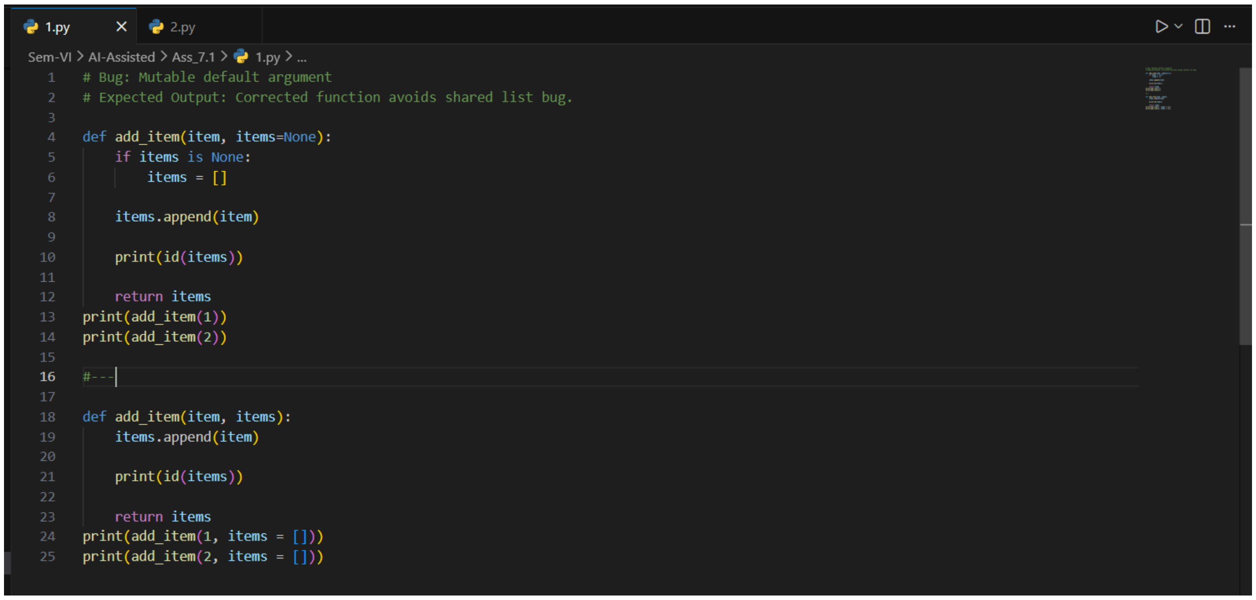
1. (Mutable Default Argument – Function Bug)

   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.



```
# Bug: Mutable default argument
# Expected Output: Corrected function avoids shared list bug.

def add_item(item, items=None):
    if items is None:
        items = []

    items.append(item)

    print(id(items))

    return items
print(add_item(1))
print(add_item(2))

#---

def add_item(item, items):
    items.append(item)

    print(id(items))

    return items
print(add_item(1, items = []))
print(add_item(2, items = []))
```

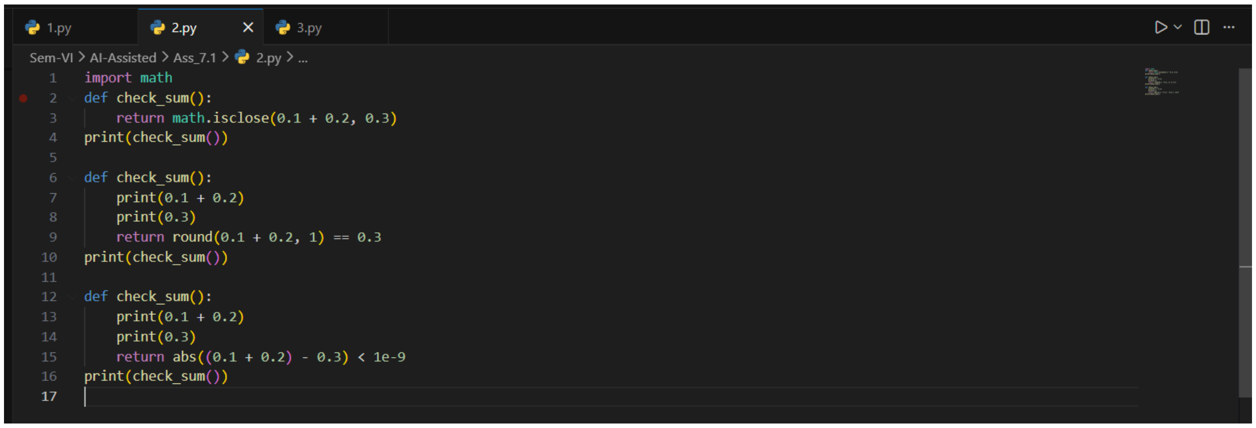
2. (Floating-Point Precision Error)

   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



1.py | 2.py | 3.py

Sem-VI > AI-Assisted > Ass_7.1 > 2.py > ...

```python
import math
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3)
print(check_sum())

def check_sum():
    print(0.1 + 0.2)
    print(0.3)
    return round(0.1 + 0.2, 1) == 0.3
print(check_sum())

def check_sum():
    print(0.1 + 0.2)
    print(0.3)
    return abs((0.1 + 0.2) - 0.3) < 1e-9
print(check_sum())

```

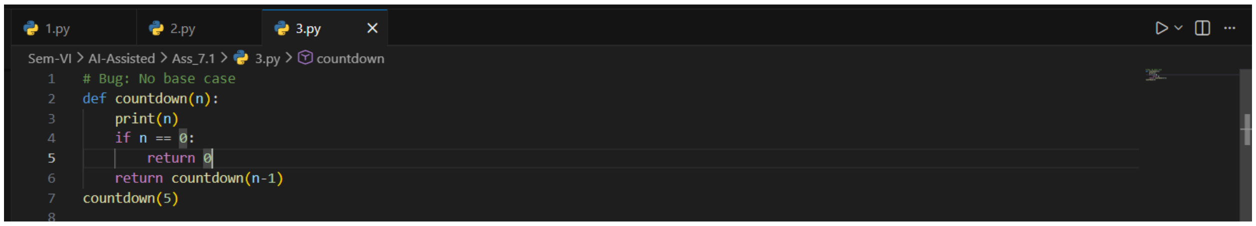
3. (Recursion Error – Missing Base Case)

Task:

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

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

Expected Output: Correct recursion with stopping condition.



1.py | 2.py | 3.py

Sem-VI > AI-Assisted > Ass_7.1 > 3.py > countdown

```python
# Bug: No base case
def countdown(n):
    print(n)
    if n == 0:
        return 0
    return countdown(n-1)
countdown(5)
```

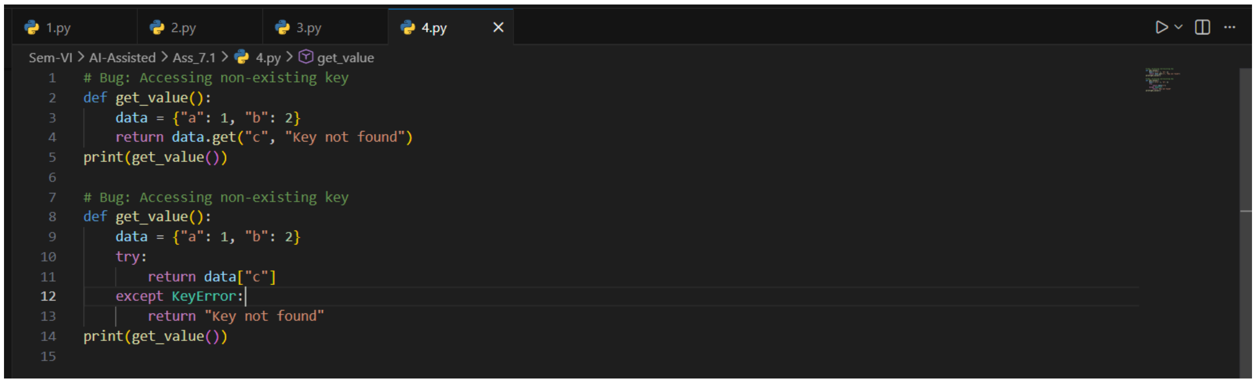
4. (Dictionary Key Error)

Task:

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

```python
# 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.



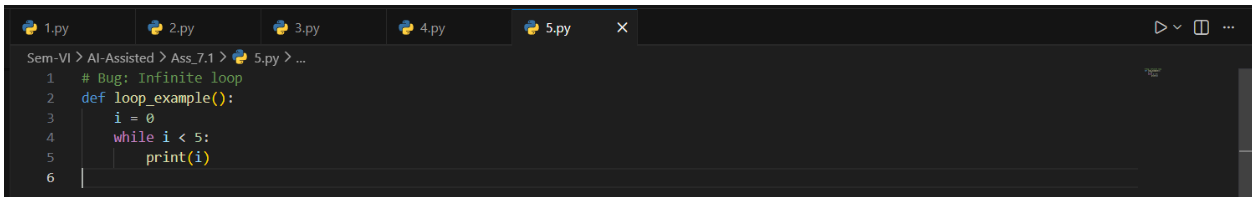
```python
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")
print(get_value())

# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    try:
        return data["c"]
    except KeyError:
        return "Key not found"
print(get_value())
```

5. (Infinite Loop – Wrong Condition)

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

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

Expected Output: Corrected loop increments i.



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

6. (Unpacking Error – Wrong Variables)

Task:

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

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

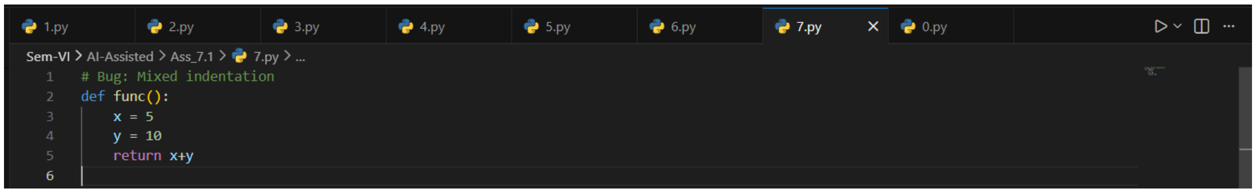
Expected Output: Correct unpacking or using _ for extra values.

7. (Mixed Indentation – Tabs vs Spaces)

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

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

Expected Output : Consistent indentation applied.

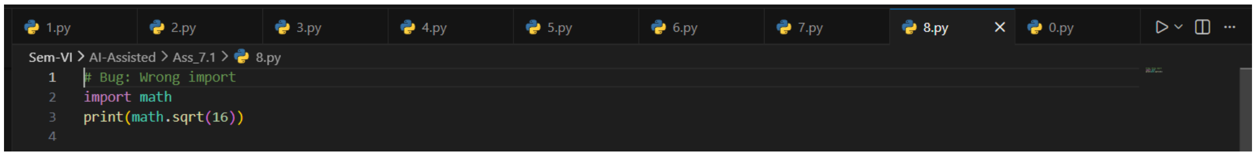


```
# Bug: Mixed indentation
def func():
    x = 5
    y = 10
    return x+y
```

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



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

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.

10. (Name Error – Undefined Variable)
    Task:
    Analyze given code where a variable is used before being defined. Let AI detect and fix the error.

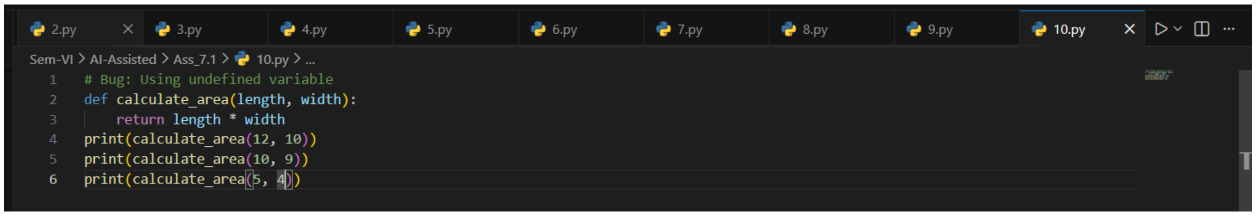
```
# Bug: Using undefined variable
def calculate_area():
    return length * width
print(calculate_area())
```

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.



```
# Bug: Using undefined variable
def calculate_area(length, width):
    return length * width
print(calculate_area(12, 10))
print(calculate_area(10, 9))
print(calculate_area(5, 4))
```

Explanation:

In tuple unpacking the values separated by commas are made into objects and assigned a variable but when the number of variables and the values separated by commas are different the assignment doesn't go by the plan and throws an error.

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.

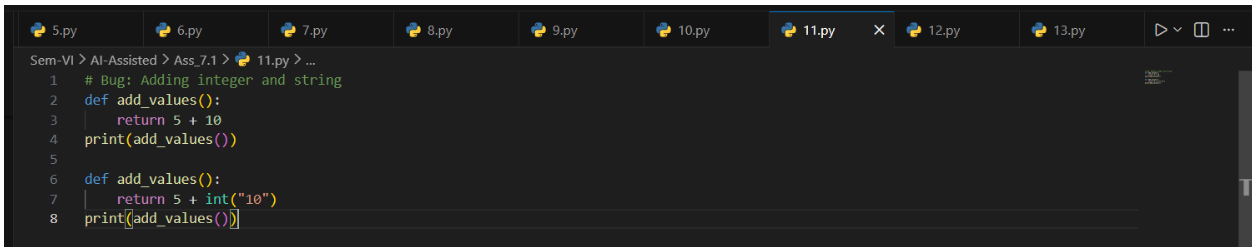
```
# Bug: Adding integer and string
def add_values():
    return 5 + "10"
print(add_values())
```

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.



```
# Bug: Adding integer and string
def add_values():
    return 5 + 10
print(add_values())

def add_values():
    return 5 + int("10")
print(add_values())
```

Explanation:

Python requires explicit type conversion, it causes a TypeError, to solve this you can either convert the string 10 in int or remove the string conversion from 10

12. (Type Error – String + List Concatenation)

Task:

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

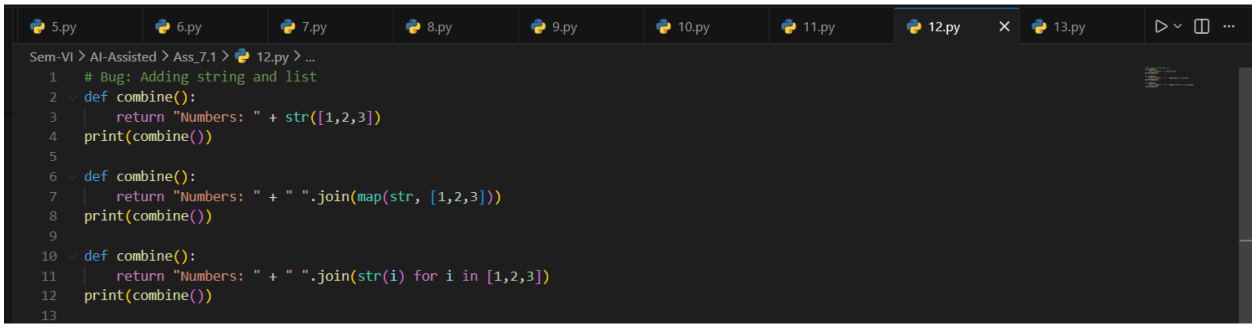
```python
# Bug: Adding string and list
def combine():
    return "Numbers: " + [1, 2, 3]
print(combine())
```

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



```python
# Bug: Adding string and list
def combine():
    return "Numbers: " + str([1,2,3])
print(combine())

def combine():
    return "Numbers: " + " ".join(map(str, [1,2,3]))
print(combine())

def combine():
    return "Numbers: " + " ".join(str(i) for i in [1,2,3])
print(combine())
```

Explanation:

Always explicitly convert data types before concatenation. Python prioritizes explicit operations over ambiguous implicit conversions.

13. (Type Error – Multiplying String by Float)

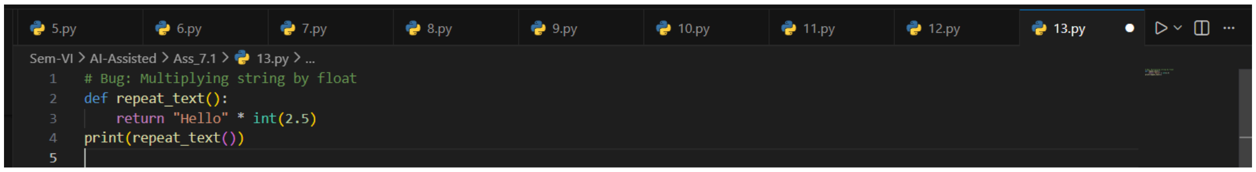
Task: Detect and fix code where a string is multiplied by a float.

```python
# 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



5.py 6.py 7.py 8.py 9.py 10.py 11.py 12.py 13.py

Sem-VI > AI-Assisted > Ass_7.1 > 13.py > ...

```python
# Bug: Multiplying string by float
def repeat_text():
    return "Hello" * int(2.5)
print(repeat_text())

```

Explanation:

- Strings can only be multiplied by integers (int).
- "Hello" * 3 → "HelloHelloHello" (valid repetition).
- "Hello" * 2.5 is invalid because:
- Repetition must be a whole number (you can’t repeat a string “2.5 times”).
- Python does not auto-convert float to int for safety and clarity.

14. (Type Error – Adding None to Integer)

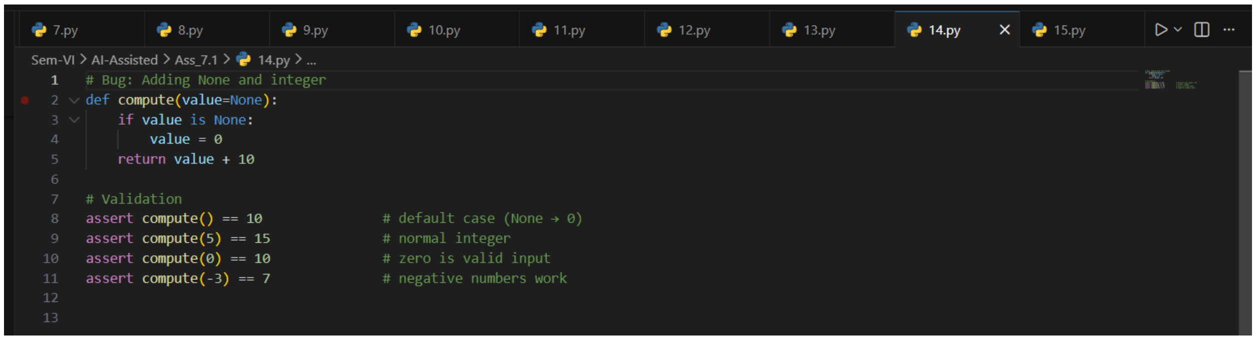
Task:

Analyze code where None is added to an integer.

```python
# 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.



7.py 8.py 9.py 10.py 11.py 12.py 13.py 14.py 15.py

Sem-VI > AI-Assisted > Ass_7.1 > 14.py > ...

```python
# Bug: Adding None and integer
def compute(value=None):
    if value is None:
        value = 0
    return value + 10

# Validation
assert compute() == 10              # default case (None → 0)
assert compute(5) == 15             # normal integer
assert compute(0) == 10             # zero is valid input
assert compute(-3) == 7             # negative numbers work


```

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

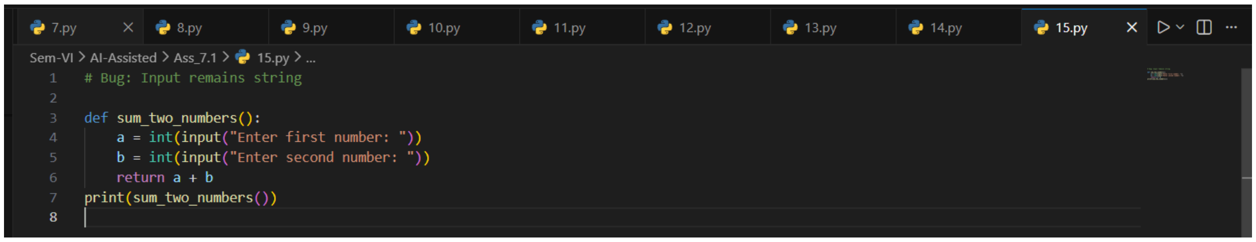
Task:

Fix code where user input is not converted properly.

```python
# 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:

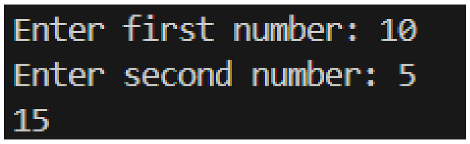
- Explain why input is always string.
- Fix using int() conversion.
- Verify with assert test cases.



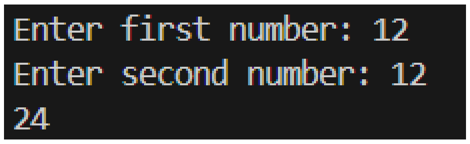
```python
# Bug: Input remains string

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

Explanation:

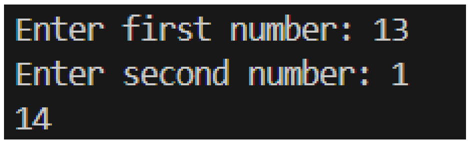
Input is always a string because, when used int() it converts the input string into integer.



```
Enter first number: 10
Enter second number: 5
15
```



```
Enter first number: 12
Enter second number: 12
24
```



```
Enter first number: 13
Enter second number: 1
14
```