SHIVANAND RAMA
2303A51037
BATCH 01
LAB ASSIGNMENT 7.4
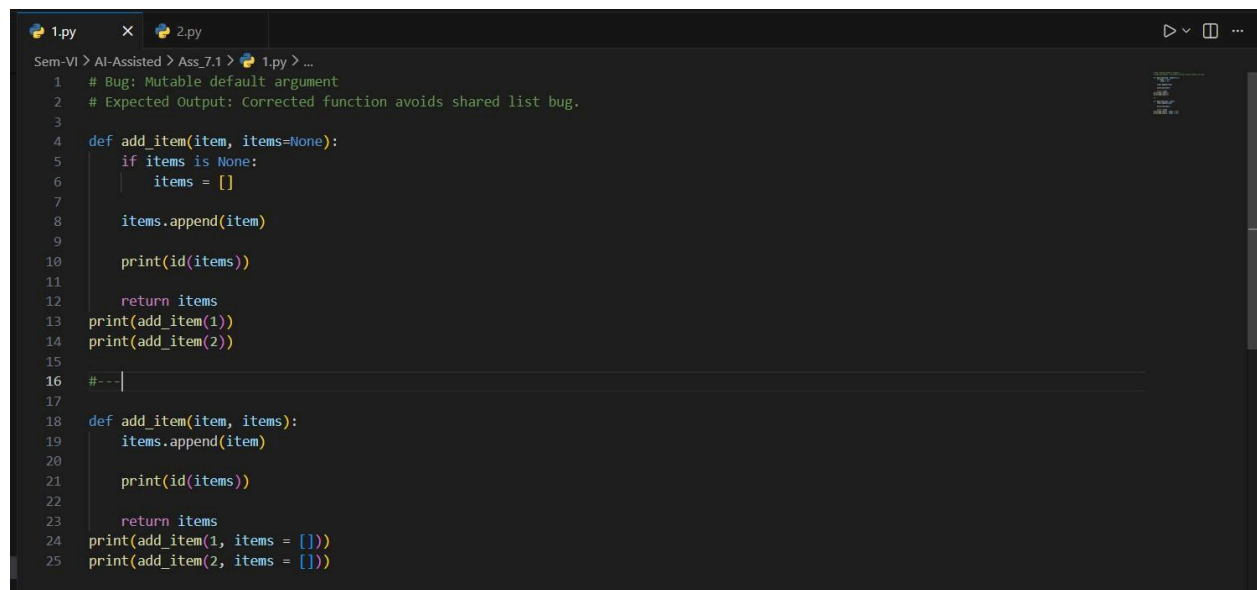
1. (Mutable Default Argument – Function Bug)

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 = []))
```
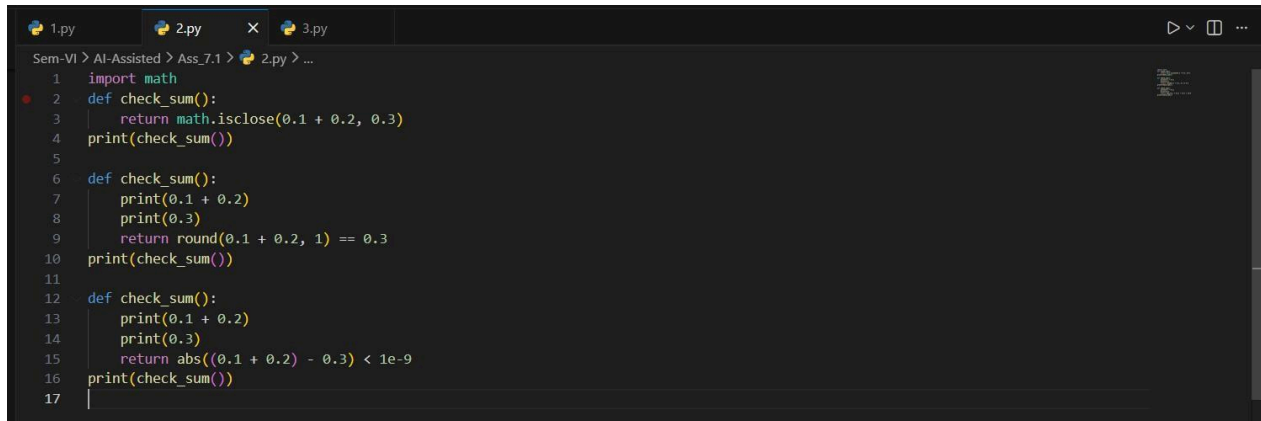
2. (Floating-Point Precision

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        X   3.py

Sem-VI > AI-Assisted > Ass_7.1 > 2.py > ...
  1    import math
● 2    def check_sum():
  3        return math.isclose(0.1 + 0.2, 0.3)
  4    print(check_sum())
  5
  6    def check_sum():
  7        print(0.1 + 0.2)
  8        print(0.3)
  9        return round(0.1 + 0.2, 1) == 0.3
  10   print(check_sum())
  11
  12   def check_sum():
  13       print(0.1 + 0.2)
  14       print(0.3)
  15       return abs((0.1 + 0.2) - 0.3) < 1e-9
  16   print(check_sum())
  17
```
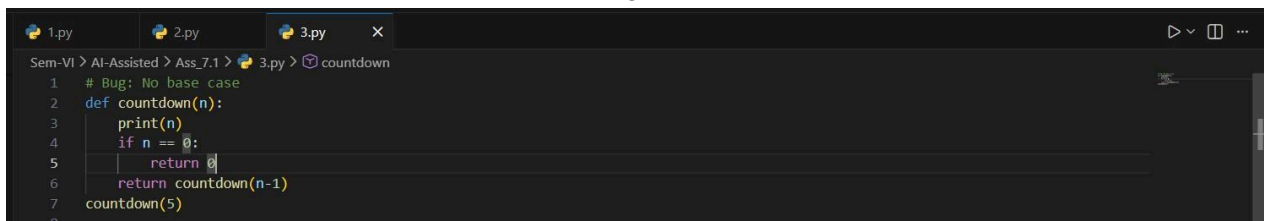
3. (Recursion Error – Missing Base
Case) Task:
Analyze given code where recursion runs infinitely due to
missing base case. Use AI to fix it.

```
# 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        X

Sem-VI > AI-Assisted > Ass_7.1 > 3.py > countdown
  1    # Bug: No base case
  2    def countdown(n):
  3        print(n)
  4        if n == 0:
  5            return 0
  6        return countdown(n-1)
  7    countdown(5)
  8
```
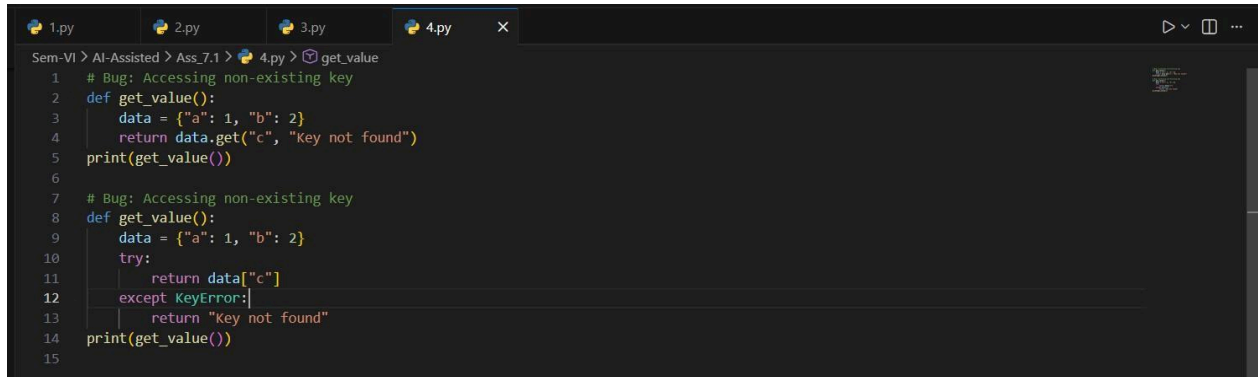
4. (Dictionary Key Error)
Task:
Analyze given code where a missing dictionary key causes error. Use AI to fix it.

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

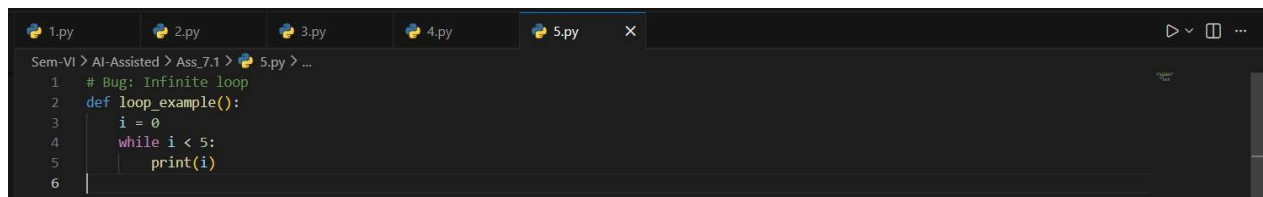Expected Output: Corrected with .get() or error handling.

```
1.py        2.py        3.py        4.py    ×                                        ▷∨  □  …
Sem-VI > AI-Assisted > Ass_7.1 > 4.py > get_value
  1    # Bug: Accessing non-existing key
  2    def get_value():
  3        data = {"a": 1, "b": 2}
  4        return data.get("c", "Key not found")
  5    print(get_value())
  6
  7    # Bug: Accessing non-existing key
  8    def get_value():
  9        data = {"a": 1, "b": 2}
 10        try:
 11            return data["c"]
 12        except KeyError:
 13            return "Key not found"
 14    print(get_value())
 15
```

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.

```
1.py        2.py        3.py        4.py        5.py    ×                            ▷∨  □  …
Sem-VI > AI-Assisted > Ass_7.1 > 5.py > ...
  1    # Bug: Infinite loop
  2    def loop_example():
  3        i = 0
  4        while i < 5:
  5            print(i)
  6
```

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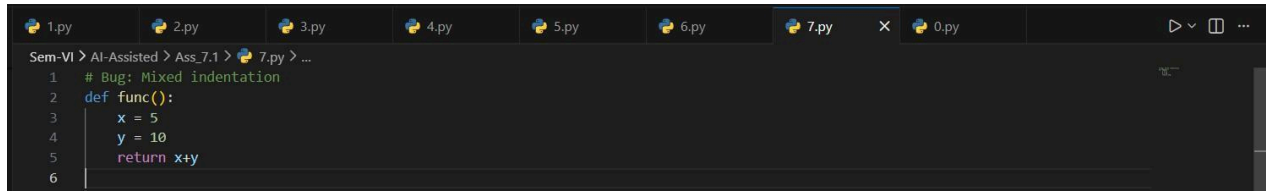
Expected Output: Correct unpacking or using _ for extra values.

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.

```
# Bug: Mixed indentation
def func():
    x = 5
    y = 10
    return x+y
```
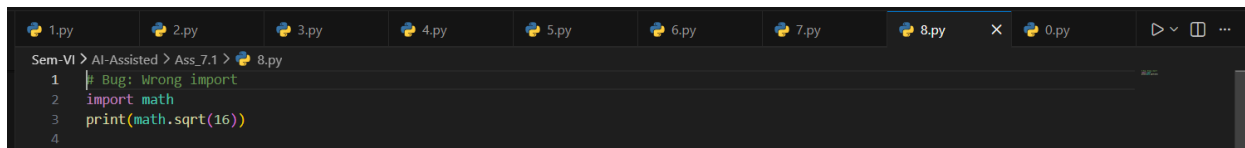
8. (Import Error – Wrong Module Usage)

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:
During python tuple unpacking the number of variables and number of values should be same

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:
TypeError because of the different data types one should be converted

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():
```
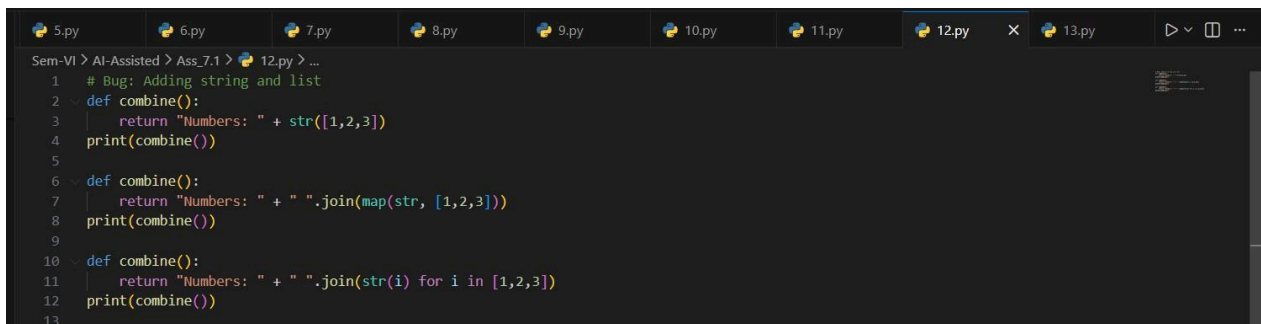
```
    return "Numbers: " + [1, 2, 3]
print(combine())
```

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

```
 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 >  12.py > …
 1    # Bug: Adding string and list
 2  ∨ def combine():
 3        return "Numbers: " + str([1,2,3])
 4    print(combine())
 5
 6  ∨ def combine():
 7        return "Numbers: " + " ".join(map(str, [1,2,3]))
 8    print(combine())
 9
10  ∨ def combine():
11        return "Numbers: " + " ".join(str(i) for i in [1,2,3])
12    print(combine())
13
```

Explanation:
String and list cannot be concatenated together it should be explicitly changed
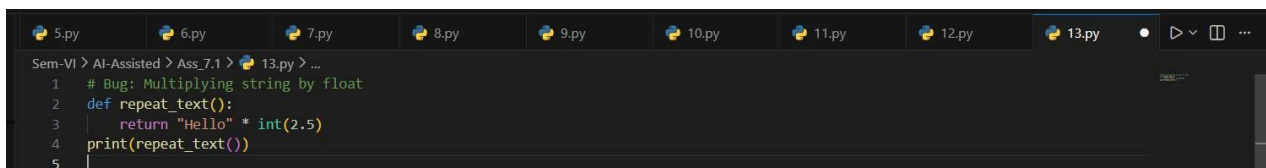
    13. (Type Error – Multiplying String by Float)
Task: Detect and fix code where a string is multiplied by a float.

```
# 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.

```
 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 > …
 1    # Bug: Multiplying string by float
 2  ∨ def repeat_text():
 3        return "Hello" * int(2.5)
 4    print(repeat_text())
 5
```

Explanation:
String repetition requires an integer count. Always convert floats explicitly using int() or round() based on your intent—never rely on implicit conversion.