

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.

```
1.py 2.py
Sem-VI > AI-Assisted > Ass_7.1 > 1.py > ...
1  # Bug: Mutable default argument
2  # Expected Output: Corrected function avoids shared list bug.
3
4 def add_item(item, items=None):
5     if items is None:
6         items = []
7
8     items.append(item)
9
10    print(id(items))
11
12    return items
13 print(add_item(1))
14 print(add_item(2))
15
16 #---|
17
18 def add_item(item, items):
19     items.append(item)
20
21     print(id(items))
22
23
24     return items
25 print(add_item(1, items = []))
26 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 > 3.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
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.

```
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.

```
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.

Sem-VI > AI-Assisted > Ass_7.1 > 7.py > ...

```
1 # Bug: Mixed indentation
2 def func():
3     x = 5
4     y = 10
5     return x+y
6
```

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

Sem-VI > AI-Assisted > Ass_7.1 > 8.py

```
1 # Bug: Wrong import
2 import math
3 print(math.sqrt(16))
4
```

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.

```

Sem-VI > AI-Assisted > Ass_7.1 > 10.py > ...
1 # Bug: Using undefined variable
2 def calculate_area(length, width):
3     return length * width
4 print(calculate_area(12, 10))
5 print(calculate_area(10, 9))
6 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.

```

Sem-VI > AI-Assisted > Ass_7.1 > 11.py > ...
1 # Bug: Adding integer and string
2 def add_values():
3     return 5 + 10
4 print(add_values())
5
6 def add_values():
7     return 5 + int("10")
8 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.

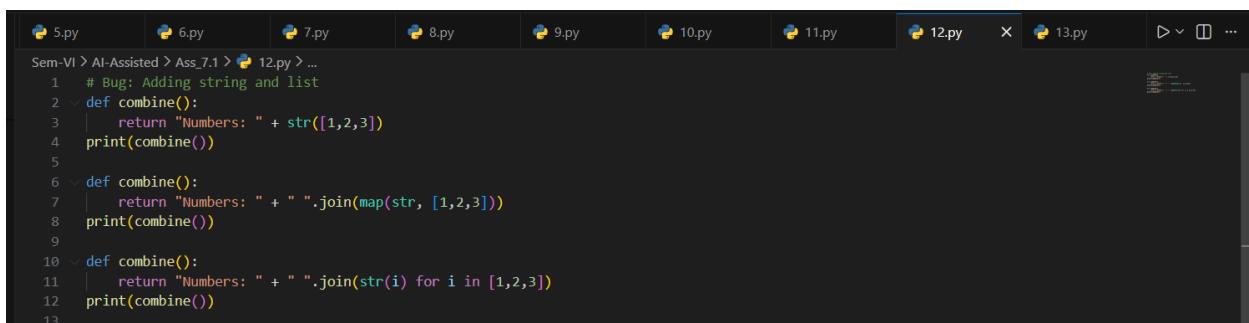
```
# Bug: Adding string and list
def combine():
    return "Numbers: " + [1, 2, 3]
print(combine())
```

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



```
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(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:

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.

```
# 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

The screenshot shows a code editor with multiple tabs at the top, each labeled with a Python file name: 5.py, 6.py, 7.py, 8.py, 9.py, 10.py, 11.py, 12.py, 13.py, and an ellipsis (...). The tab for 13.py is currently selected. The main editor area displays the following Python code:

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

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.