

## AI ASSISTED CODING

### ASSIGNMENT-7.5

**HALLTICKET NO: 2303A52169**

**BATCH: 45**

#### **Lab 7: Error Debugging with AI: Systematic approaches to**

##### **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.

##### **Prompt:**

```
def add_item(item, items=[]):
```

```
    items.append(item)
```

```
    return items
```

```
print(add_item(1))
```

```
print(add_item(2)) debug the code
```

### Code:

```
ai assisted coding lab 7.5.py > add_item
1  #task1
2  def add_item(item, items=None):
3      if items is None:
4          items = []
5          items.append(item)
6          return items
7
8  print(add_item(1))      # [1]
9  print(add_item(2))      # [2]
10 print(add_item(3, [10])) # [10, 3]
11
```

### Output:

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/L
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
• [1]
  [2]
  [10, 3]
○ PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

### Justification:

The bug happens because the same list is reused every time the function is called. By using None and creating a new list inside the function, each call gets a fresh list. This prevents unexpected shared data.

### 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

### Prompt:

```
def check_sum():
    return (0.1 + 0.2) == 0.3
print(check_sum())
```

fix the code where the floating comparison fails

**Code:**

```
3 #task2
4 import math
5
6 def check_sum():
7     return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9, abs_tol=0.0)
8
9 print(check_sum()) # True
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
True
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

Floating-point numbers are not stored exactly in memory, which causes direct comparisons to fail. Using a small tolerance helps compare values safely and gives accurate logical results.

### 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.

**Prompt:**

```
def countdown(n):
    print(n)
```

```
return countdown(n-1)
```

```
countdown(5)
```

fix the recursion error

Code:

```
21 #task3
22 def countdown(n):
23     if n <= 0:
24         print(0)
25         return
26     print(n)
27     return countdown(n - 1)
28
29 countdown(5)
```

Output:

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/Programs/Python/Python310/python.exe C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py
5
4
3
2
1
0
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

Justification:

Without a base case, the function keeps calling itself forever. Adding a stopping condition ensures the recursion ends properly and avoids stack overflow errors.

#### 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.

**Prompt:**

```
def get_value():
```

```
    data = {"a": 1, "b": 2}
```

```
    return data["c"]
```

```
print(get_value())
```

fix the dictionary key with .get()

**Code:**

```
31
32 #task4
33 def get_value():
34     data = {"a": 1, "b": 2}
35     return data.get("c", 0) # default 0 if "c" not present
36
37 print(get_value())
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
0
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

Accessing a key that doesn't exist causes a runtime error. Using .get() safely handles missing keys and prevents the program from crashing.

**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.

**Prompt:**

```
def loop_example():
```

```
i = 0
```

```
while i < 5:
```

```
print(i)
```

fix the infinite loop

**Code:**

```
38
39 #task5
40 def loop_example():
41     i = 0
42     while i < 5:
43         print(i)
44         i += 1
45
46 loop_example()
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
0
1
2
3
4
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

The loop never ends because the counter variable is never updated. Incrementing the variable inside the loop allows the condition to eventually fail and stop execution.

**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.

**Prompt:**

a, b = (1, 2, 3)

fix the unpacking error and get values

**Code:**

```
47
48 #task6
49 a, b, c = (1, 2, 3)
50 print(a, b, c)
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local,
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
1 2 3
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

The error occurs because there are more values than variables to unpack. Using an underscore or matching variable count correctly fixes the issue cleanly.

### 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.

**Prompt:**

```
def func():
```

```
    x = 5
```

```
    y = 10
```

```
    return x+y fix the indentation error
```

**Code:**

```
51
52 #task7
53 def func():
54     x = 5
55     y = 10
56     return x + y
57
58 print(func())
```

**Output:**

```
PS C:\Users\matta& C:/Users/matta/AppData/Local/Programs/Python/
sted Coding/ai assisted coding lab 7.5.py"
15
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

Python relies on consistent indentation for code blocks. Fixing the indentation using spaces throughout ensures the code runs without syntax errors.

**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

**Prompt:**

```
import maths
```

```
print(maths.sqrt(16))
```

 fix the incorrect import

**Code:**

```
60 #task8
61 import math
62
63 print(math.sqrt(16))
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
4.0
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

The module name was incorrect, leading to an import failure. Correcting the module name allows Python to access the required functions properly.

**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.

**Prompt:**

```
def total(numbers):
```

for n in numbers:

return n

print(total([1,2,3])) fix the return inside a loop

**Code:**

```
64
65 #task9
66 def total(numbers):
67     s = 0
68     for n in numbers:
69         s += n
70     return s
71
72 print(total([1, 2, 3])) # 6
```

**Output:**

```
PS C:\Users\matta& C:/Users/matta/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe C:/Users/matta/AppData/Local/Programs/Python/Python39-64/Scripts/python.exe C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py
6
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

Returning inside the loop exits the function after the first iteration. Moving the return statement outside the loop ensures all values are processed.

### 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

### Prompt:

```
def calculate_area():
    return length * width

print(calculate_area()) fix the name error with 3 test cases
```

### Code:

```
74 #task10
75 def calculate_area(length, width):
76     return length * width
77
78 # Test cases
79 print(calculate_area(5, 3)) # 15
80 print(calculate_area(10, 2)) # 20
81 print(calculate_area(7, 4)) # 28
```

### Output:

```
PS C:\Users\matta& C:/Users/matta/AppData/Local/Programs/Python/Python310/Scripts/Python.exe: The term 'calculate_area' is not recognized as the name of a cmdlet, function, script file, or executable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
15
20
28
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

### Justification:

The variables were used without being defined. Passing them as function parameters makes the function reusable and removes the name error.

### 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.

**Prompt:**

```
def add_values():  
    return 5 + "10"  
  
print(add_values()) fix the type error with 3 test cases
```

**Code:**

```
82  
83 #task11  
84 def add_values(a, b):  
85     # Convert both to integers before adding  
86     return int(a) + int(b)  
87  
88 # 3 test cases  
89 print(add_values(5, "10")) # 15  
90 print(add_values("3", "7")) # 10  
91 print(add_values(0, "0")) # 0
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local,
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py"
15
10
0
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

### Justification:

Python does not allow adding integers and strings directly. Converting one type to match the other ensures valid operations and correct output.

### 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.

Expected Output:

- Corrected code
- Explanation
- Successful test validation

### Prompt:

```
def combine():
```

```
    return "Numbers: " + [1, 2, 3]
```

```
print(combine())
```

fix the type error with 3 testcases

### Code:

```
92
93 #task12
94 def combine(numbers):
95     # Convert the list of numbers to a comma-separated string
96     return "Numbers: " + ", ".join(str(n) for n in numbers)
97
98 # 3 test cases
99 print(combine([1, 2, 3]))      # Numbers: 1, 2, 3
100 print(combine([10, 20, 30]))  # Numbers: 10, 20, 30
101 print(combine([]))            # Numbers:
```

### Output:

```
PS C:\Users\matta& C:/Users/matta/AppData/Local/Programs/Python/Python313/python.
sted Coding/ai assisted coding lab 7.5.py"
Numbers: 1, 2, 3
Numbers: 10, 20, 30
Numbers:
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

### Justification:

Strings and lists cannot be concatenated directly. Converting the list to a string or joining its elements resolves the type mismatch.

### 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.

### Prompt:

```
def repeat_text():  
    return "Hello" * 2.5  
print(repeat_text()) fix the typeerror with 3 examples
```

#### Code:

```
02  
03 #task13  
04 def repeat_text(text, times):  
05     # Ensure times is an integer to avoid TypeError  
06     return text * int(times)  
07  
08 # 3 examples  
09 print(repeat_text("Hello", 2.5)) # HelloHello  
10 print(repeat_text("Hi", 3))     # HiHiHi  
11 print(repeat_text("Bye", 1.2))  # Bye
```

#### Output:

```
Bye  
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData  
:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.  
HelloHello  
HiHiHi  
Bye  
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

#### Justification:

Strings can only be multiplied by integers, not floats. Converting the float to an integer makes repetition valid and predictable.

#### 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.

**Prompt:**

```
def compute():
```

```
    value = None
```

```
    return value + 10
```

```
print(compute()) fix the type error with 3 testcases
```

**Code:**

```
112
113 #task14
114 def compute(value=None):
115     # If value is None, treat it as 0
116     if value is None:
117         value = 0
118     return value + 10
119
120 # 3 test cases
121 print(compute())      # 10
122 print(compute(5))    # 15
123 print(compute(-3))   # 7
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/Programs/Python/Python39-6/Scripts/python.exe C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py
10
15
7
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

None represents the absence of a value and cannot be used in arithmetic. Assigning a default numeric value prevents runtime errors.

**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.

**Prompt:**

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

print(sum_two_numbers())
```

fix the input error with 3 test cases

**Code:**

```
125 #task15
126 def sum_two_numbers(a, b):
127     return int(a) + int(b)
128
129 # 3 test cases
130 print(sum_two_numbers("2", "3")) # 5
131 print(sum_two_numbers("10", "20")) # 30
132 print(sum_two_numbers(7, "8")) # 15
```

**Output:**

```
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding> & C:/Users/matta/AppData/Local/Programs/Python/Python39-64/Python.exe -i C:/Users/matta/OneDrive/Desktop/AI Assisted Coding/ai assisted coding lab 7.5.py
5
30
15
PS C:\Users\matta\OneDrive\Desktop\AI Assisted Coding>
```

**Justification:**

Python's input() always returns a string by default. Converting inputs to integers allows correct numerical addition instead of string concatenation.

