

# Assignment-7.5

Name : K.Tejaswini

Ht.no : 2303A51425

Batch : No: 21

---

## Task 1 (Mutable Default Argument – Function Bug)

**Task:** Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

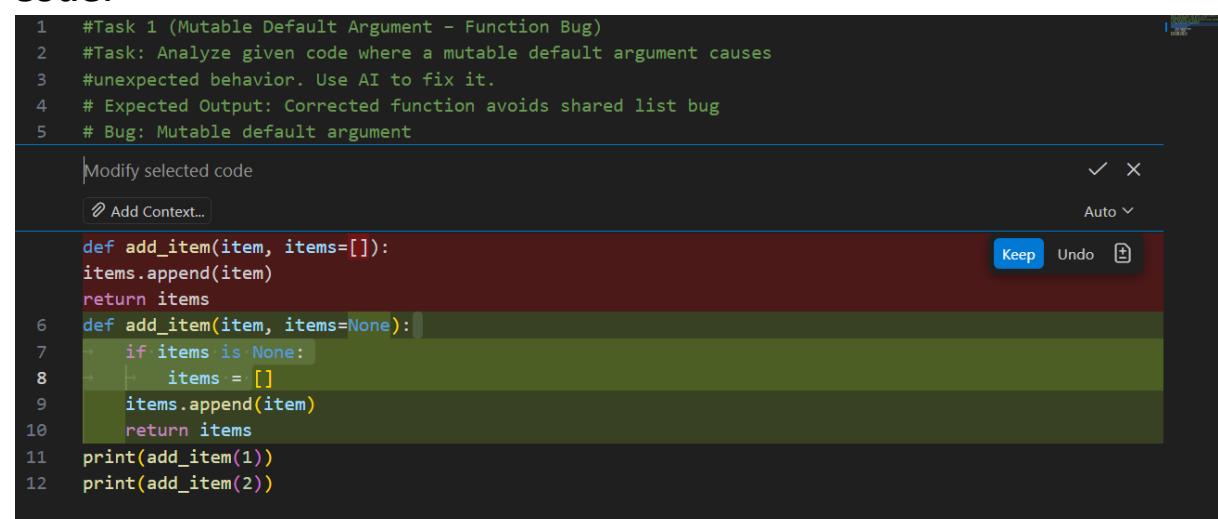
# Bug: Mutable default argument

```
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.

**Code:**



The screenshot shows a code editor interface with the following code:

```
1 #Task 1 (Mutable Default Argument - Function Bug)
2 #Task: Analyze given code where a mutable default argument causes
3 #unexpected behavior. Use AI to fix it.
4 # Expected Output: Corrected function avoids shared list bug
5 # Bug: Mutable default argument

Modify selected code
∅ Add Context... Auto ▾

def add_item(item, items=[]):
    items.append(item)
    return items
Keep Undo ⌂

6 def add_item(item, items=None):
7     if items is None:
8         items = []
9     items.append(item)
10    return items
11 print(add_item(1))
12 print(add_item(2))
```

The code editor has a dark theme. A modal window titled "Modify selected code" is open over the code. It contains a text input field with placeholder text "∅ Add Context..." and a dropdown menu labeled "Auto". At the bottom right of the modal are "Keep" and "Undo" buttons, along with a small icon. The code itself is numbered from 1 to 12. Lines 1 through 5 are comments explaining the task. Lines 6 through 10 define the corrected function, which uses a default argument of None and initializes items to an empty list if it is None. Lines 11 and 12 are print statements calling the function with arguments 1 and 2 respectively.

## Output:

```
1  #Task 1 (Mutable Default Argument - Function Bug)
2  #Task: Analyze given code where a mutable default argument causes
3  #unexpected behavior. Use AI to fix it.
4  # Expected Output: Corrected function avoids shared list bug
5  # Bug: Mutable default argument
6  def add_item(item, items=None):
7      if items is None:
8          items = []
9      items.append(item)
10     return items
11 print(add_item(1))
12 print(add_item(2))
```

The screenshot shows a terminal window with the following content:

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR SPELL CHECKER

(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> C:/Users/WINDOWS/anac
ctivate
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/an
exe "c:/Users/WINDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/task.py"
[1]
[2]
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

## Observation:

The default list `items=[]` is created only once, so all function calls share the same list.

Because the list is mutable, values added in earlier calls remain in later calls, causing unexpected accumulation.

Using `items=None` and creating a new list inside the function ensures each call works with a fresh list and fixes the bug.

## 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

### Code:

```
# Task 2 (Floating-Point Precision Error)
# Task: Analyze given code where floating-point comparison fails.
# Use AI to correct with tolerance.
#Expected Output: Corrected function
# Bug: Floating point precision issue

Modify selected code
∅ Add Context...

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

print(check_sum())
Expected Output: Corrected function
```

### Output:

```
task.py > ...
1  # Task 2 (Floating-Point Precision Error)
2  # Task: Analyze given code where floating-point comparison fails.
3  # Use AI to correct with tolerance.
4  #Expected Output: Corrected function
5  # Bug: Floating point precision issue
6  def check_sum():
7      return abs((0.1 + 0.2) - 0.3) < 1e-9
8  print(check_sum())

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR

(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> C:/Users,
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> conda act
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/User
WINDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/task.py"
True
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

## **Observation:**

Floating-point numbers cannot always represent decimal values exactly, so  $(0.1 + 0.2)$  is not exactly equal to 0.3.

Direct comparison using `==` may therefore return `False` even when the values should logically be equal.

Using a tolerance (`epsilon`) and checking whether the absolute difference is smaller than that tolerance correctly handles floating-point precision errors.

## **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.

## Code:

The screenshot shows a code editor interface with a dark theme. At the top, there is a status bar with icons for file operations and a progress bar. Below the status bar is a toolbar with buttons for 'Modify selected code' (with a checkmark), 'X', 'Auto', and a dropdown menu. The main area contains the following Python code:

```
1 #Task 3 (Recursion Error - Missing Base Case)
2 # Task: Analyze given code where recursion runs infinitely due to
3 # missing base case. Use AI to fix.
4 #Expected Output : Correct recursion with stopping condition.

5 # Bug: No base case - FIXED
6 def countdown(n):
7     print(n)
8     return countdown(n-1)
9
10    if n < 0:
11        return
12    print(n)
13    return countdown(n-1)

14 countdown(5)
15
16 Expected Output : Correct recursion with stopping condition.
```

Line 5 is highlighted in red with the comment '# Bug: No base case - FIXED'. Lines 7 through 13 are highlighted in green, indicating they have been modified by the AI. Line 16 is also highlighted in red with the text 'Expected Output : Correct recursion with stopping condition.'

## Output:

The screenshot shows a terminal window with a dark theme. At the top, there is a navigation bar with tabs for 'PROBLEMS' (2), 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected), 'PORTS', 'SQL HISTORY', 'TASK MONITOR', and '...'. Below the navigation bar, the terminal shows the following command-line session:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> C:/Users/WINDO
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WIN
NDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/task.py"
5
4
3
2
1
0
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

The output shows the numbers 5, 4, 3, 2, 1, and 0 printed sequentially, indicating the function has reached its base case.

## **Observation:**

The original recursive function lacks a base case, so the function keeps calling itself indefinitely, leading to infinite recursion or a recursion depth error.

Adding a stopping condition such as if  $n == 0$ : return ensures that the recursion terminates properly.

After fixing, the function prints numbers from 5 down to 1 and then stops successfully.

## **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.

## Code:

```
1 # Task 4 (Dictionary Key Error)
2 # Task: Analyze given code where a missing dictionary key causes
3 # error. Use AI to fix it.
4 # Expected Output: Corrected with .get() or error handling.
5 # Bug: Accessing non-existing key
```

|Modify selected code ✓ X

∅ Add Context...

Auto ▾

```
6 def get_value():
7     data = {"a": 1, "b": 2}
8     return data["c"]
9
10    data = {"a": 1, "b": 2}
11    return data.get("c", None)
```

Keep Undo ⌂

## Output:

```
task.py > ⌂ get_value
1 # Task 4 (Dictionary Key Error)
2 # Task: Analyze given code where a missing dictionary key causes
3 # error. Use AI to fix it.
4 # Expected Output: Corrected with .get() or error handling.
5 # Bug: Accessing non-existing key
6 def get_value():
7     data = {"a": 1, "b": 2}
8     return data.get("c", None)
9 print(get_value())

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR ...
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> ^C
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WI
NDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/task.py"
None
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

## Observation:

Accessing a non-existing dictionary key using `data["c"]` raises a

`KeyError` because the key is not present in the dictionary.

Using `data.get("c", "Key not found")` safely returns a default value

instead of causing an error.

This fix ensures the program runs without crashing and handles missing keys gracefully.

## 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.

**Code:**

The screenshot shows a code editor interface with the following code:

```
1 #Task 5 (Infinite Loop - Wrong Condition)
2 #Task: Analyze given code where loop never ends. Use AI to detect and fix it.
3 #Expected Output: Corrected loop increments i.
4 # Bug: Infinite loop
5 # Fixed code
```

Below the code, there is a "Modify selected code" section with a "Keep" button and an "Undo" button. The code editor highlights several lines with red and green backgrounds, indicating changes made by the AI:

- Line 6: `def loop_example():` (highlighted in red)
- Line 7: `i = 0` (highlighted in red)
- Line 8: `while i < 5:` (highlighted in red)
- Line 9:  `print(i)` (highlighted in green)
- Line 10:  `i += 1 # Increment i to avoid infinite loop` (highlighted in green)
- Line 11: `loop_example()` (highlighted in green)

## Output:

The screenshot shows a code editor interface with a dark theme. At the top, there's a file navigation bar: task.py > loop\_example. Below this is the Python code:

```
task.py > loop_example
1 #Task 5 (Infinite Loop - Wrong Condition)
2 #Task: Analyze given code where loop never ends. Use AI to detect and fix it.
3 #Expected Output: Corrected loop increments i.
4 # Bug: Infinite loop
5 # Fixed code
6 def loop_example():
7     i = 0
8     while i < 5:
9         print(i)
10        i += 1    # Increment i to avoid infinite loop
11
12 loop_example()
13
```

Below the code editor is a terminal window showing the execution of the script:

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR SPELL CHECKER 2
1
2 ...
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/anaconda
coding/task.py"
0
1
2
3
4
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

The terminal output shows the values 0, 1, 2, 3, and 4 printed sequentially, indicating that the loop has terminated correctly.

## Observation:

The original loop does not update the value of `i`, so the condition `i < 5` always remains true, causing an infinite loop.

Adding the increment statement `i += 1` inside the loop ensures that `i` increases after each iteration.

With this fix, the loop prints values from 0 to 4 and then terminates correctly.

## 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.

**Code:**

```
1 # Task 6 (Unpacking Error - Wrong Variables)
2 # Task: Analyze given code where tuple unpacking fails. Use AI to fix it.
3 # Expected Output: Correct unpacking or using _ for extra values.
4 # Bug: Wrong unpacking
    Modify selected code
    ⚙ Add Context...
5 a, b ,_= (1, 2, 3)
6 print(a,b,c)
   print[a,b]
```

**Output:**

```
1 # Task 6 (Unpacking Error - Wrong Variables)
2 # Task: Analyze given code where tuple unpacking fails. Use AI to fix it.
3 # Expected Output: Correct unpacking or using _ for extra values.
4 # Bug: Wrong unpacking
5 a, b ,_= (1, 2, 3)
6 print[a,b]
```

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR SPELL CHECKER 2

(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> C:/Users/WINDOWS/anaconda3/Scr
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> conda activate base
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/anaconda3/p
coding/task.py"
1 2
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

**Observation:**

The tuple (1, 2, 3) contains three values, but only two variables (a, b) are provided, causing a “too many values to unpack” error.

Tuple unpacking requires the number of variables to match the

number of values unless a placeholder like `_` is used.

Fixing it as `a, b, _ = (1, 2, 3)` or using three variables correctly resolves the unpacking error.

## 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.

**Code:**

```
task7.py > func > Editor
1 # Task 7 (Mixed Indentation - Tabs vs Spaces)
2 # Task: Analyze given code where mixed indentation breaks
3 # execution. Use AI to fix it.
4 # Expected Output : Consistent indentation applied.
5 # Bug: Mixed indentation
Modify selected code
∅ Add Context...
Auto ▾
6 def func():
x = 5
y = 10
return x+y
7 + x = 5
8 + y = 10
9 + return x+y
10
```

## Output:

The screenshot shows a code editor interface with a terminal window below it. The code editor has tabs for 'task.py', 'lab7.5.py', and 'App.js'. The 'task.py' tab is active, displaying the following Python code:

```
task.py > ...
1 # Task 7 (Mixed Indentation - Tabs vs Spaces)
2 # Task: Analyze given code where mixed indentation breaks
3 # execution. Use AI to fix it.
4 # Expected Output : Consistent indentation applied.
5 # Bug: Mixed indentation
6 def func():
7     x = 5
8     y = 10
9     return x + y
10
11 print(func())
12
```

The terminal window below shows the command line history and the output of running the script:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> C:/U...
● (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> cond...
● (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/...
coding/task.py"
● 15
○ (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

## Observation:

The original code uses mixed indentation (tabs and spaces), which causes an indentation error and prevents the program from executing correctly.

Python requires consistent indentation within a block to define the function body properly.

Using the same indentation style (for example, four spaces for each line) fixes the issue and produces the correct output 15.

## 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

**Code:**

The screenshot shows a code editor interface. At the top, there is a status bar with the following text:  
1 # Task 8 (Import Error - Wrong Module Usage)  
2 # Task: Analyze given code with incorrect import. Use AI to fix.  
3 # Expected Output: Corrected to import math  
4 # Bug: Wrong import

Below the status bar is a dropdown menu with the following options:  
Modify selected code  
Add Context...

The main code area contains the following Python code:  
import maths  
print(maths.sqrt(16))  
5 import math  
6 print(math.sqrt(16))

**Output:**

The screenshot shows a code editor interface with a terminal tab active. The terminal window displays the following text:  
1 # Task 8 (Import Error - Wrong Module Usage)  
2 # Task: Analyze given code with incorrect import. Use AI to fix.  
3 # Expected Output: Corrected to import math  
4 # Bug: Wrong import  
5 import math  
6 print(math.sqrt(16))  
7

At the bottom of the terminal window, there is a navigation bar with the following tabs:  
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR

The terminal window also shows some command-line history:  
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3\_2\AI Assit coding> C:/Users/W  
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3\_2\AI Assit coding> conda acti  
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3\_2\AI Assit coding> & C:/Users/  
coding/task.py"  
4.0  
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3\_2\AI Assit coding>

**Observation:**

The original code fails because it uses mixed indentation, which Python does not allow within the same block.

This causes an indentation error and stops the program from running correctly.

Applying consistent indentation (such as four spaces for all lines inside the function) fixes the issue and allows the function to execute properly.