

# AI ASSITANT CODING

## ASSIGNMENT-7.5

**Name:** B.Akhira Nandhini

**Ht.no:** 2303A51516

**Batch:**22

### 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:**

```
task1.py > ...
1  #Fix the Python function that uses a mutable default list causing shared
2  # values between function calls.
3  def add_item(item, items=None):
4      if items is None:
5          items = []
6      items.append(item)
7      return items
8
9  print(add_item(1))
10 print(add_item(2))
```

PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/task1.py"
[1]
[2]
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>

30°C  
Sunny

**Explanation:**

- A list is used as a default argument. In Python, default arguments are evaluated only once when the function is defined, not each time the function is called.
- Because lists are mutable, the same list is shared across all function calls, causing values from previous calls to persist.
- To fix this issue, None is used as the default value instead of a list. Inside the function, a new list is created when the argument is None.
- This ensures that each function call uses a fresh list and prevents the shared list 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:

The screenshot shows a dark-themed IDE window titled "AI assisted coding". On the left, a sidebar lists files: "7.5.py", "t2.py" (which is selected), "task1.py", "task2.py", "task3.py", and "task4.py". The main editor area contains the following Python code:

```
7.5.py > t2.py > ...
1  """Analyze the Python function where direct floating-point comparison causes incorrect results.
2  # Fix the function by using a tolerance-based comparison."""
3  def check_sum():
4      return abs((0.1 + 0.2) - 0.3) < 1e-9
5  print(check_sum())
6
7
```

Below the editor is a terminal window showing the command-line output:

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & c:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t2.py"
True
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

The bottom status bar shows the weather as "30°C Sunny", system icons, and the date/time as "07-02-2026 14:50".

Explanation:

- Floating-point numbers cannot always be represented exactly in binary format. As a result, the expression  $0.1 + 0.2$  does not produce an exact value of 0.3, causing direct comparison using `==` to fail.

- To correct this, a tolerance-based comparison is used. The absolute difference between the two values is checked to see if it is smaller than a very small number (1e-9).
- This approach accounts for minor precision errors and produces the correct result.

### 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 software interface for AI-assisted coding. At the top, there's a menu bar with File, Edit, Selection, View, Go, Run, Terminal, Help, and an 'AI assisted coding' section. Below the menu is a toolbar with icons for file operations like Open, Save, and Print. The main workspace is titled 'AI assisted coding' and contains a code editor with Python code. The code is as follows:

```
1 #Identify and fix the recursion error in the given Python function caused by a missing base case.
2 # Modify the function so that it stops correctly."
3 def countdown(n):
4     if n == 0:          # base case
5         return
6     print(n)
7     countdown(n - 1)
8 countdown(5)
```

Below the code editor is a terminal window titled 'Terminal'. It shows the command being run: 'python.exe "c:/Users/abhin/OneDrive/Desktop/AT assisted coding/7.5.py/t3.py"'. The terminal output shows the numbers 5, 4, 3, 2, and 1 printed sequentially, indicating the function is working correctly. The bottom of the screen shows a taskbar with various application icons and system status indicators like battery level and date/time.

Explanation:

- The original function does not contain a base case, so the recursive call continues indefinitely, eventually causing a recursion error.
- In recursion, a base case is required to stop further function calls.

- In the corrected version, a base case is added to stop the recursion when n becomes 0.
- Once this condition is met, the function returns without making another recursive call. This ensures that the recursion terminates properly.

#### 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:

```
File Edit Selection View Go Run Terminal Help
AI assisted coding
Upgrade to Pro Search Agents... New Agent
AI ASSISTED CODING
github
5.5.py
6.5.py
7.5.py
t2.py
t3.py
t4.py
task1.py
day1.py
day2.py
day3.py
day4.py
task1.py 7.5.py t2.py t3.py t4.py task1.py 5.5.py TASK1.LPY task2.py 5.5.py
7.5.py > t4.py > ...
1  #Analyze the Python function where accessing a non-existing dictionary key causes an error.
2  #Fix the code using safe dictionary access or error handling.
3  def get_value():
4      data = {"a": 1, "b": 2}
5      return data.get("c", "Key not found")
6  print(get_value())
7  | Ctrl+L to chat, Ctrl+K to generate

Problems Output Debug Console Terminal Ports
powershell + ×
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/users/abhin/OneDrive/Desktop/AI assisted coding/7.5.py/t4.py"
Key not found
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding>
```

Explanation:

- In the original code, the dictionary key "c" does not exist. Accessing a missing key using square brackets (data["c"]) raises a KeyError.
- To fix this, the .get() method is used. The .get() method safely returns a default value when the key is not found instead of raising an error.

- This prevents the program from 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():
```

j = 0

while j < 5:

```
print(i)
```

**Expected Output:** Corrected loop increments i.

Code:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** On the left, it lists files under "AI ASSISTED CODING". The file "t5.py" is currently selected.
- Code Editor:** The main area displays the Python code for "t5.py".

```
1  #Analyze the Python code where a Loop runs infinitely due to a missing update condition.
2  # Fix the loop so that it terminates correctly.
3  def loop_example():
4      i = 0
5      while i < 5:
6          print(i)
7          i += 1 # increment to avoid infinite loop
8
9  loop_example()
10
```
- Terminal:** At the bottom, the terminal window shows the command being run and its output.

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & C:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/t5.py"
0
1
2
3
4
```
- Bottom Status Bar:** Shows "AI assisted coding" as the active workspace, along with system status icons for battery, signal, and date.

## Explanation:

- In the variable  $i$  is never updated inside the while loop. Because the condition  $i < 5$  always remains true, the loop runs infinitely.
  - The value of  $i$  is incremented during each iteration. This eventually makes the condition false, allowing the loop to stop correctly.

## Task 6 (Unpacking Error – Wrong Variables)

Task: Analyze given code where tuple unpacking fails. Use AI to

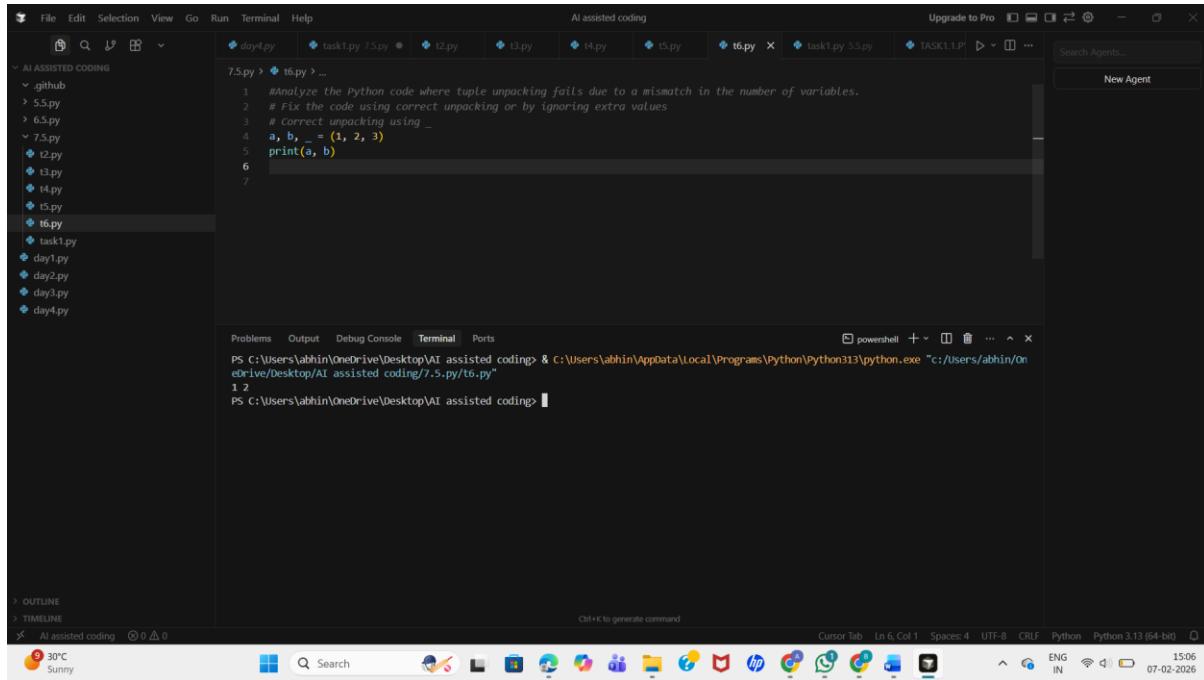
fix it.

# Bug: Wrong unpacking

```
a, b = (1, 2, 3)
```

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

Code:



Explanation:

- The tuple contains three values but only two variables are provided for unpacking. This causes a `ValueError` because Python expects the number of variables on the left side to match the number of values on the right side.
- The issue can be fixed either by using the correct number of variables or by using an underscore (`_`) to ignore unwanted values. This allows the unpacking operation to succeed without errors.

### 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:

The screenshot shows the Visual Studio Code interface with the 'AI assisted coding' extension open. The left sidebar shows a tree view of files under 'AI ASSISTED CODING'. The main editor area contains a Python script named 't7.py' with the following code:

```
1  #Analyze the Python function where mixed or incorrect indentation causes execution errors.
2  # Fix the code by applying consistent indentation.
3  def func():
4      x = 5
5      y = 10
6      return x + y
7
8  print(func())
9
```

The terminal at the bottom shows the command being run: 'python t7.py'. The output is: '15'. The status bar at the bottom right indicates the file is 'Python 3.13 (64-bit)'.

Explanation:

- Python uses indentation to define code blocks. In the given code, inconsistent indentation (mixing tabs and spaces or missing indentation) breaks the function structure and causes an `IndentationError`.
- The corrected version uses consistent indentation (4 spaces) for all statements inside the function.
- This allows Python to correctly interpret the function body and execute it without 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

Code:

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

- File Structure:** On the left, there's a sidebar titled "AI ASSISTED CODING" containing a tree view of files. The "t8.py" file is selected.
- Code Editor:** The main area displays the content of "t8.py".

```
1 #Analyze the Python code where an incorrect module name is imported.
2 # Fix the code by importing the correct standard library module
3 import math
4 print(math.sqrt(16))
```
- Terminal:** Below the code editor is a terminal window titled "Terminal". It shows the command being run:

```
PS C:\Users\abhin\OneDrive\Desktop\AI assisted coding> & c:\Users\abhin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/abhin/OneDrive/Desktop/AI assisted coding/t8.py"
```
- Bottom Status Bar:** The status bar at the bottom provides system information: "powerhell", "cursor tab", "ln 5, col 1", "spaces: 4", "utf-8", "crlf", "python", "python 3.13 (64-bit)", "eng", "in", "15:12", and "07-02-2026".

### Explanation:

- The corrected code imports the correct module, `math`, which provides mathematical functions such as `sqrt()`.
- Using the proper module name fixes the error and allows the program to run successfully.