

# AI Assisted Coding

## Assignment 7.5

Name: P. Vineeth Kumar

Hall ticket no: 2303A51256

Batch no: 19

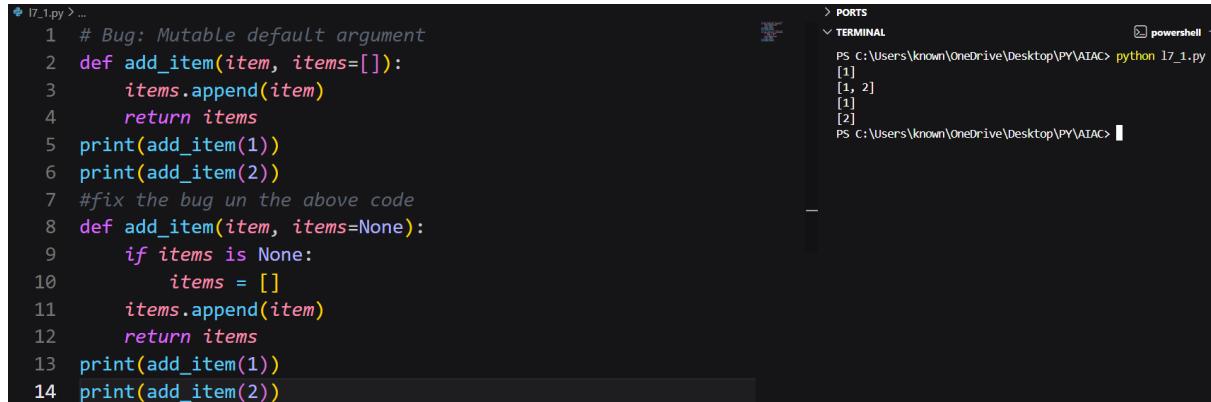
### Task 1: Mutable Default Argument – Function Bug

#### Prompt:

#Bug: Mutable default argument

#Fix the bug in the above code

#### Code & Output:



The screenshot shows a terminal window with two panes. The left pane displays a Python script named '17\_1.py' containing the following code:

```
1 # Bug: Mutable default argument
2 def add_item(item, items=[]):
3     items.append(item)
4     return items
5 print(add_item(1))
6 print(add_item(2))
7 #fix the bug un the above code
8 def add_item(item, items=None):
9     if items is None:
10         items = []
11     items.append(item)
12     return items
13 print(add_item(1))
14 print(add_item(2))
```

The right pane shows the terminal output of running the script with 'python 17\_1.py'. It shows two separate executions of the function, each printing a different list:

```
PS C:\Users\known\OneDrive\Desktop\PY\AIAC> python 17_1.py
[1]
[1, 2]
[1]
[2]
PS C:\Users\known\OneDrive\Desktop\PY\AIAC>
```

#### Explanation:

The AI correctly identified that using a mutable object (list) as a default argument leads to shared state across function calls. This results in unexpected accumulation of values. The AI-fixed version initializes the list inside the function when no argument is provided, ensuring a fresh list is created each time. This approach prevents side effects and follows best practices in Python function design.

### Task 2: Floating-Point Precision Error

#### Prompt:

#Bug: Floating point precision issue

#Fix the above code

## Code & Output:

The terminal window shows the execution of a Python script named 17\_1.py. The code contains a floating-point precision issue where it checks if the sum of 0.1 and 0.2 equals 0.3. It then fixes this by calculating the absolute difference between the sum and 0.3, ensuring it is less than 1e-9. The terminal output shows the original check failing (False) and the fixed version passing (True).

```
17_1.py > ...
1 # Bug: Floating point precision issue
2 def check_sum():
3     return (0.1 + 0.2) == 0.3
4 print(check_sum())
5 #fix the above code
6 def check_sum():
7     return abs((0.1 + 0.2) - 0.3) < 1e-9
8 print(check_sum())
```

```
> PORTS
  TERMINAL
  powershell
PS C:\Users\known\OneDrive\Desktop\PY\AIAC> python 17_1.py
False
True
PS C:\Users\known\OneDrive\Desktop\PY\AIAC>
```

## Explanation:

The AI identified that floating-point numbers cannot always be compared directly due to precision limitations in binary representation. Instead of using equality comparison, the AI suggested checking whether the difference between values lies within an acceptable tolerance. This solution improves reliability and accuracy in numerical computations.

## Task 3: Recursion Error – Missing Base Case

### Prompt:

#This code will cause a RecursionError: maximum recursion depth exceeded in comparison

#Fixed Code:

### Code & Output:

The terminal window shows the execution of a Python script named 17\_1.py. The code defines a recursive function countdown(n) that prints the value of n. It lacks a base case, leading to a RecursionError. The AI suggests adding a base case where if n is less than or equal to 0, it prints "STOP!". The terminal output shows the original code causing a stack overflow error, while the fixed code correctly prints the values from 5 down to 1, followed by "STOP!".

```
17_1.py > ...
1 # def countdown(n):
2 #     print(n)
3 #     return countdown(n-1)
4 # countdown(5)
5 # This code will cause a RecursionError: maximum recursion
# depth exceeded in comparison
6 #Fixed code:
7 def countdown(n):
8     if n <= 0:
9         print("STOP!")
10    else:
11        print(n)
12        countdown(n-1)
13 countdown(5)
```

```
> PORTS
  TERMINAL
  powershell
PS C:\Users\known\OneDrive\Desktop\PY\AIAC> python 17_1.py
PS C:\Users\known\OneDrive\Desktop\PY\AIAC> python 17_1.py
PS C:\Users\known\OneDrive\Desktop\PY\AIAC> python 17_1.py
5
4
3
2
1
STOP!
PS C:\Users\known\OneDrive\Desktop\PY\AIAC>
```

## Explanation:

The AI detected that the recursive function lacks a stopping condition, causing infinite recursion and stack overflow. By adding a proper base case, the AI ensured that recursion terminates correctly. This fix demonstrates the importance of defining exit conditions in recursive algorithms.

## Task 4: Dictionary Key Error

## Prompt:

#Fixed Code:

#Bug: Accessing non-existing key

## Code & Output:

The screenshot shows a code editor with a file named 17\_1.py containing Python code. The code defines a function get\_value() that returns a dictionary with keys 'a' and 'b'. It then prints the value of 'c', which is not in the dictionary. A comment '# Fixed code:' is present. Below the code editor is a terminal window showing the command 'python 17\_1.py' being run, followed by an error message 'Key not found'.

```
17_1.py > ...
1 # def get_value():
2 #     data = {"a": 1, "b": 2}
3 #     return data["c"]
4 # print(get_value())
5 #Fixed code:
6 # Bug: Accessing non-existing key
7 def get_value():
8     data = {"a": 1, "b": 2}
9     return data.get("c", "Key not found")
10 print(get_value())
```

```
> PORTS
> TERMINAL
powershell
PS C:\Users\known\OneDrive\Desktop\Py\AIAC> python 17_1.py
Key not found
PS C:\Users\known\OneDrive\Desktop\Py\AIAC>
```

## Explanation:

The AI recognized that directly accessing a missing dictionary key raises a `KeyError`. It suggested safer alternatives such as using the `.get()` method or handling the exception. This improves program robustness and prevents unexpected crashes during execution.

## Task 5: Infinite Loop – Wrong Condition

### Prompt:

#Bug: Infinite loop because 'i' is never incremented.

#Fixed Code:

## Code & Output:

The screenshot shows a code editor with a file named 17\_1.py containing Python code. The code defines a function loop\_example() that prints the value of 'i' and increments it by 1 in a while loop. A comment '#Bug: Infinite loop because 'i' is never incremented.' is present. Below the code editor is a terminal window showing the command 'python 17\_1.py' being run, displaying the values 0 through 4.

```
17_1.py > ...
1 # def loop_example():
2 #     i = 0
3 #     while i < 5:
4 #         print(i)
5 #Bug: Infinite loop because 'i' is never incremented.
6 #Fixed Code:
7 def loop_example():
8     i = 0
9     while i < 5:
10        print(i)
11        i += 1
12 loop_example()
```

```
> PORTS
> TERMINAL
powershell
PS C:\Users\known\OneDrive\Desktop\Py\AIAC> python 17_1.py
0
1
2
3
4
PS C:\Users\known\OneDrive\Desktop\Py\AIAC>
```

## Explanation:

The AI identified that the loop variable was never updated, causing an infinite loop. By adding the correct increment operation, the loop now progresses toward its termination condition. This fix highlights the importance of updating loop control variables correctly.

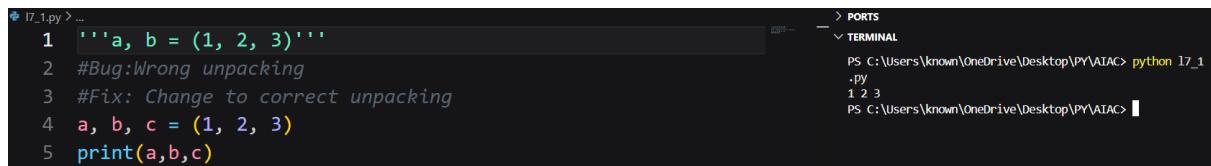
## Task 6: Unpacking Error – Wrong Variables

### Prompt:

#Bug: Wrong unpacking

#Fix : Change to correct unpacking

**Code:**



```
1 '''a, b = (1, 2, 3)'''
2 #Bug:Wrong unpacking
3 #Fix: Change to correct unpacking
4 a, b, c = (1, 2, 3)
5 print(a,b,c)
```

The screenshot shows a code editor window with a terminal tab open. The terminal shows the command `python 17_1.py` and its output: `1 2 3`. The code itself contains a tuple assignment with three variables but only two values, which is a common Python error.

**Explanation:**

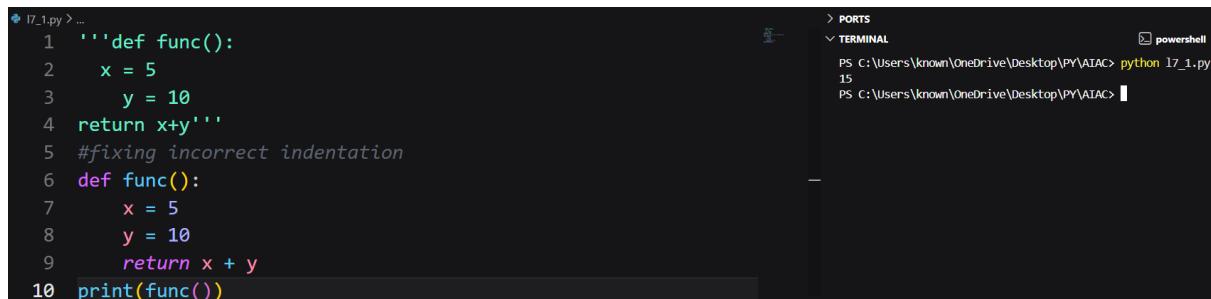
The AI detected a mismatch between the number of variables and values during tuple unpacking. It suggested either increasing the number of variables or using a placeholder variable `_` for unused values. This solution ensures correct unpacking without runtime errors.

### Task 7: Mixed Indentation – Tabs vs Spaces

**Prompt:**

#Fixing incorrect indentation

**Code:**



```
1 '''def func():
2     x = 5
3     y = 10
4     return x+y'''
5 #fixing incorrect indentation
6 def func():
7     x = 5
8     y = 10
9     return x + y
10 print(func())
```

The screenshot shows a code editor window with a terminal tab open. The terminal shows the command `python 17_1.py` and its output: `15`. The code uses a mix of tabs and spaces for indentation, which Python interprets differently, leading to an error.

**Explanation:**

The AI identified inconsistent indentation as the root cause of the error. Python relies strictly on indentation for block definition. By applying consistent spacing throughout the function, the AI restored correct program execution and improved code readability.

### Task 8: Import Error – Wrong Module Usage

**Prompt:**

#Bug: Wrong import module name

#Corrected Code:

**Code:**

The screenshot shows a code editor with a Python file named '17\_1.py' open. The code contains the following lines:

```
1 ...
2 import maths
3 print(maths.sqrt(16))
4 ...
5 #Bug:wrong import module name
6 #Corrected Code:
7 import math
8 print(math.sqrt(16))
```

To the right of the code editor is a terminal window titled 'powershell'. The terminal shows the command 'python 17\_1.py' being run, and the output '4.0' is displayed.

### Explanation:

The AI recognized that the imported module name was incorrect. It suggested importing the standard Python math module instead. This fix resolves the import error and allows the program to use mathematical functions correctly.