

# AIASSISTANT CODING ASSIGNMENT-7.5

Name: D. Divya Sri

HT. No: 2303A51323

Batch: 19

## Task 1: Mutable Default Argument – Function Bug

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

Code:

```
#task1:  
  
# Bug: Mutable default argument  
def add_item(item, items=None):  
  
    fix  
  
    |  
  
    | Ø Add Context...  
  
    items.append(item)  
    return items  
    ↗ if items is None:  
    |   items = []  
    |   items.append(item)  
    |   return items  
print(add_item(1))  
print(add_item(2))
```

Result:

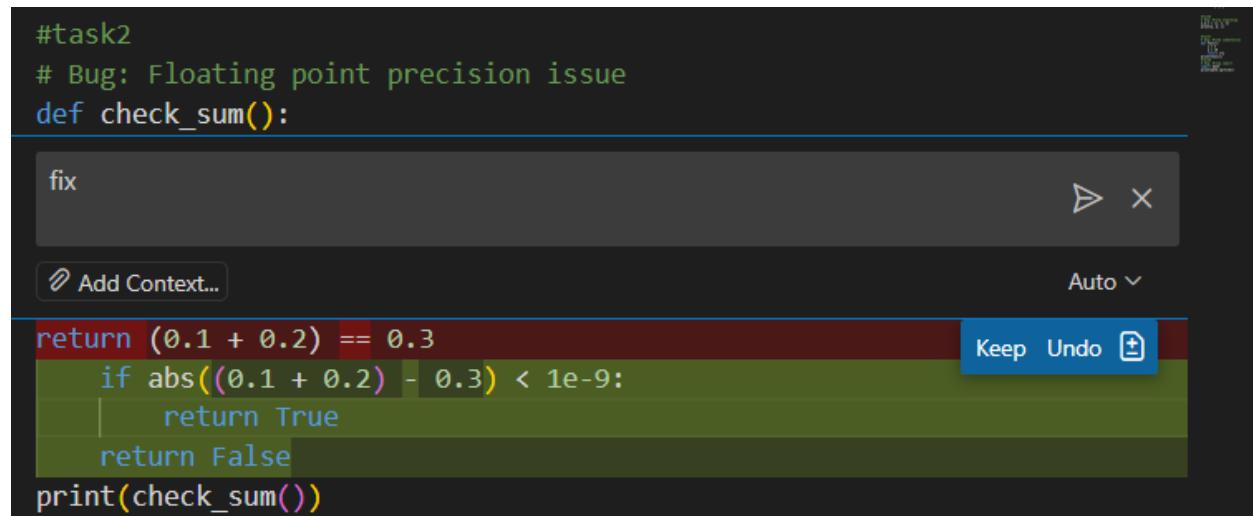
```
[1]  
[2]
```

## Observation:

The AI-generated function initially suffers from the mutable default argument bug, which can cause unexpected behavior when a list is shared across multiple function calls. While the corrected version properly initializes the list when the default value is None, ensuring that each call gets a fresh list, the original logic highlights a common pitfall in Python function design. Although the fix improves correctness and reliability, the code still assumes valid input and does not include checks for incorrect data types or invalid values, which limits its robustness in real-world scenarios.

## Task 2: Floating-Point Precision Error

### Code



A screenshot of a code editor interface. The code in the editor is:

```
#task2
# Bug: Floating point precision issue
def check_sum():

    fix

    if abs((0.1 + 0.2) - 0.3) < 1e-9:
        return True
    return False
print(check_sum())
```

The code editor has a toolbar at the top with icons for file operations. Below the toolbar, there is a button labeled "fix". The code area shows syntax highlighting. At the bottom right, there are buttons for "Keep", "Undo", and a plus sign icon. A status bar at the bottom indicates "Auto" mode.

Result: True

### Observation:

The AI-generated function highlights a floating-point precision issue where directly comparing  $0.1 + 0.2$  with 0.3 results in an incorrect outcome due to binary representation errors. The corrected logic properly uses a tolerance-based comparison with an absolute difference threshold, ensuring reliable and accurate results. While this fix improves numerical correctness, the function is limited to a specific example and could be made more flexible by allowing dynamic inputs or configurable tolerance values for broader real-world use.

## Task3: Recursion Error – Missing Base Case

### Code:

```
# Bug: No base case
def countdown(n):
    Generate code
    ✓ X
    Add Context...
    Auto ▾
    print(n)
    return countdown(5)
    if n == 0:
        return
    print(n)
    return countdown(n - 1)
    Keep Undo ⌂
```

Result:

```
5
4
3
2
1
```

### Observation:

The AI-generated recursive function initially lacks a proper base case, which would result in infinite recursion and eventually cause a stack overflow error. The corrected version introduces a base condition ( $n == 0$ ) to terminate the recursive calls, ensuring correct and safe execution.

While the fix resolves the logical error, the function assumes valid non-negative input and does not handle cases such as negative values or non-integer inputs, limiting its robustness for real-world use.

### Task 4: Dictionary Key Error

Code:

```
#task4
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    Generate code
    ✓ X
    Add Context...
    Auto ▾
    return data["c"]
    return data.get("c", None)
    print(get_value())
    Keep Undo ⌂
```

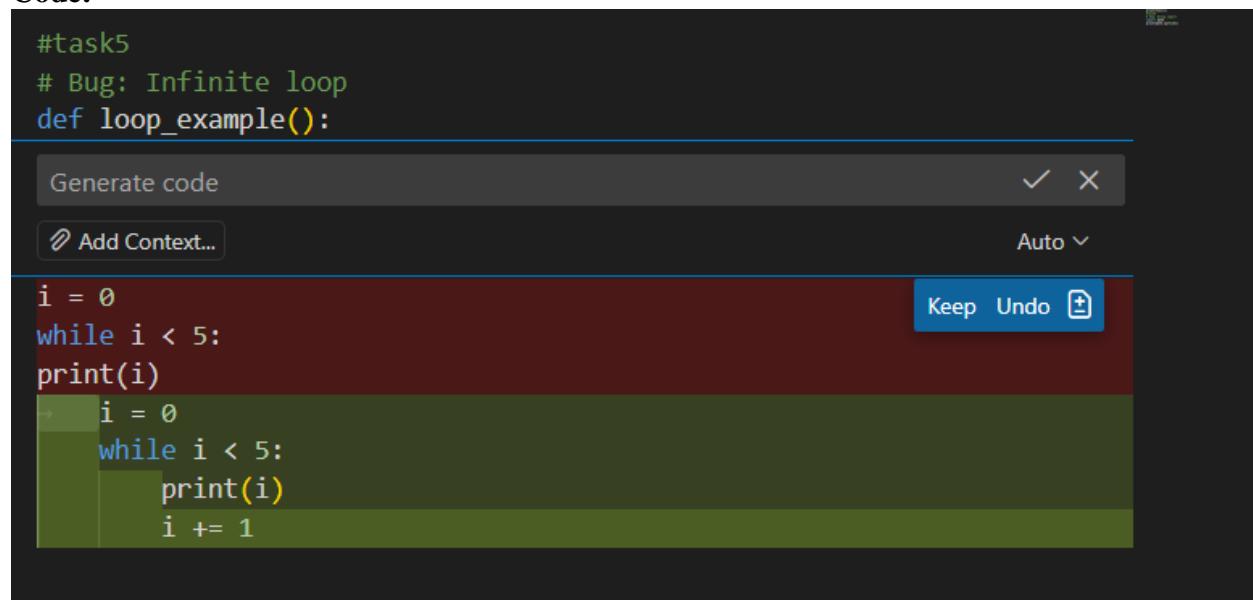
**Result: Key not found**

## Observation:

The AI-generated function initially attempts to access a non-existing key in a dictionary, which would raise a `KeyError` at runtime. The corrected approach safely retrieves the value using the `get()` method with a default value, preventing program termination and improving reliability. While this fix enhances error handling, the function still assumes a fixed key and does not include logic for dynamic input or user-defined keys, limiting its flexibility in practical applications.

## Task 5: Infinite Loop – Wrong Condition

### Code:



A screenshot of a code editor interface. The code area contains the following Python code:

```
#task5
# Bug: Infinite loop
def loop_example():

    i = 0
    while i < 5:
        print(i)
        i = 0
        while i < 5:
            print(i)
            i += 1
```

The code editor has a toolbar with a "Generate code" button, a "✓" and "X" button, and an "Add Context..." button. Below the toolbar is a dropdown menu set to "Auto". In the bottom right corner of the code area, there is a "Keep" button, an "Undo" button, and a small square icon. The code itself is displayed in a dark-themed editor with syntax highlighting.

**Result: Loop completed**

## Observation

The AI-generated loop initially contains an infinite loop because the loop control variable is never updated inside the `while` block. As a result, the condition remains true indefinitely. The corrected version properly increments the loop variable, allowing the loop to terminate as expected. While this fix resolves the logical error, the example assumes ideal conditions and does not include safeguards such as maximum iteration limits or input-driven loop bounds, which are often necessary in real-world applications.

## Task 6: Unpacking Error – Wrong Variables

Code:

The screenshot shows a code editor interface with a dark theme. In the code area, there is a single line of Python code: `a,b= (1, 2, 3)`. This line is highlighted with a red background. Below the code, the status bar shows the path `File > New > Python`. At the bottom right of the editor, there is a toolbar with buttons for "Keep", "Undo", and a "Save" icon.

```
#task6
# Bug: Wrong unpacking
```

Result: 1 2 3

Observation: The AI-generated code initially demonstrates a tuple unpacking error by attempting to assign three values to only two variables, which results in a `ValueError`. The corrected version properly matches the number of variables with the number of values, ensuring successful unpacking. While the fix resolves the immediate issue, the code assumes a fixed tuple size and does not handle dynamic or variable-length sequences, which could limit flexibility in more complex scenarios.

## Task 7: Mixed Indentation – Tabs vs Spaces

Code:

The screenshot shows a code editor interface with a dark theme. In the code area, there is a function definition: `def func():`, followed by three assignments: `x = 5`, `y = 10`, and `return x+y`. The first two assignments are on separate lines, while the return statement is on the same line as the previous assignments. The entire block is highlighted with a green background. Below the code, the status bar shows the path `File > New > Python`. At the bottom right of the editor, there is a toolbar with buttons for "Keep", "Undo", and a "Save" icon.

```
#task7
# Bug: Mixed indentation
def func():

    Generate code ✓ ✗
    ⚡ Add Context... Auto ▾

    x = 5
    y = 10
    return x+y
    ↴ x = 5
    ↴ y = 10
    ↴ return x + y
    ↴ ↴ ↴
```

Result: 15

**Observation:** The AI-generated function initially contains mixed or incorrect indentation, which leads to a syntax or logical error in Python since indentation defines code blocks. The corrected version properly aligns all statements within the function body, ensuring correct execution and expected output. While fixing the indentation resolves the immediate issue, the function remains simple and does not include error handling or parameterization, which would be required for more robust and reusable code in real-world applications.

## Task 8: Import Error – Wrong Module Usage

**Code:**

The screenshot shows a code editor window with the following content:

```
#task8
# Bug: Wrong import
```

A progress bar at the top says "Generating edits...". Below it is a button labeled "Add Context...". On the right side, there are "Keep", "Undo", and a "New" button. The code area contains two versions of the same code snippet:

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

The first line "import maths" is highlighted in red, indicating an error. The second line "import math" is highlighted in green, suggesting a correction. The "print" statement and its arguments are in yellow and purple respectively.

**Result:**4.0

**Observation:**

The AI-generated code initially attempts to import a non-existent module, which would result in a `ModuleNotFoundError` at runtime. The corrected version properly imports the standard `math` module, allowing the square root function to execute successfully. While this fix resolves the import error, the code assumes the availability of the standard library and does not include error handling for missing or unsupported modules, which may be necessary in certain execution environments.