

# ASSIGNMENT-7.5

P.Aneesh Reddy  
2303A51120

BATCH NO: 03

## TASK 1: Mutable Default Argument – Function Bug

ACTUAL CODE:

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

```
[7] ① 0s
    def add_item(item, items=[]):
        items.append(item)
        return items
    print(add_item(1))
    print(add_item(2))

...   File "/tmp/ipython-input-619917416.py", line 2
        items.append(item)
        ^
IndentationError: expected an indented block after function definition on line 1
```

Below the code, there is a button labeled "Next steps: Explain error".

**Prompt:** Analyze the Python function where a mutable default argument causes shared state between function calls. Fix the bug so each call uses a new list.

CORRECTED CODE:

The screenshot shows a Jupyter Notebook cell with the following code:

```
[1]  def add_item(item, items=None):
        if items is None:
            items = []
        items.append(item)
        return items

    print(add_item(1))
    print(add_item(2))

[2]
```

The cell has a green checkmark icon and the text "0s" next to it. A play button icon is also present. The output area below the cell shows the results of the code execution.

Explanation:

The issue occurs because a mutable object (list) is used as a default argument. In Python, default arguments are created once and reused across function calls, which leads to unexpected shared data. Each function call modifies the same list, causing incorrect results.

## TASK 2: Floating-Point Precision Error

ACTUAL CODE:

The screenshot shows a Jupyter Notebook cell with the following code:

```
[1]  def add_item(item, items=[]):
        items.append(item)
        return items
    print(add_item(1))
    print(add_item(2))

[2]
```

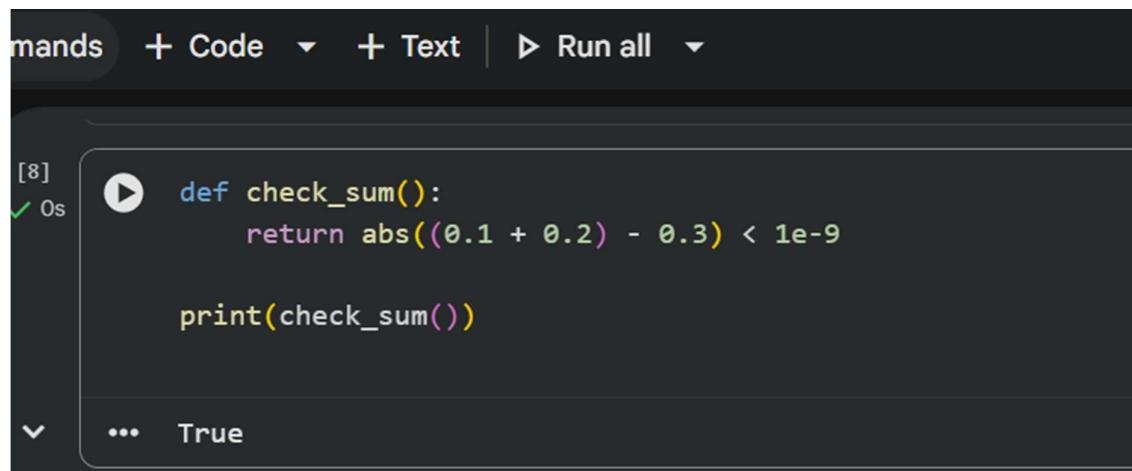
The code contains an indentation error on the second line of the function definition. The error message displayed is:

```
File "/tmp/ipython-input-619917416.py", line 2
    items.append(item)
          ^
IndentationError: expected an indented block after function definition on line 1
```

The "Next steps" button is visible at the bottom of the error message.

Prompt: Identify why direct floating-point comparison fails in Python and correct the function using an appropriate tolerance-based comparison.

CORRECTED CODE:



The screenshot shows a Jupyter Notebook cell with the following code:

```
[8] 0s
def check_sum():
    return abs(0.1 + 0.2) - 0.3) < 1e-9

print(check_sum())
...  True
```

The cell has a status bar indicating it took 0 seconds to run. The output shows the result of the function call as `True`.

Explanation:

Floating-point numbers are stored in binary form, which can cause small precision errors. Direct equality comparison fails because the computed result is not exactly equal to the expected value. Using a tolerancebased comparison avoids this issue.

TASK 3: Recursion Error – Missing Base Case

## ACTUAL CODE:

The screenshot shows a code editor interface with a dark theme. In the top left corner, there is a status bar with '[8]' and '0s'. Below it is a code editor pane containing the following Python code:

```
[8] 0s
def check_sum():
    return (0.1 + 0.2) == 0.3
print(check_sum())
```

Below the code, an error message is displayed:

```
...   File "/tmp/ipython-input-3438411890.py", line 2
        return (0.1 + 0.2) == 0.3
               ^
IndentationError: expected an indented block after function definition on line 1
```

At the bottom of the code editor, there is a button labeled "Next steps: Explain error".

Prompt: Debug the recursive function that runs infinitely due to a missing base case. Add a proper stopping condition

## CORRECTED CODE:

The screenshot shows a code editor interface with a dark theme. In the top left corner, there is a status bar with '[9]' and '0s'. Below it is a code editor pane containing the following Python code:

```
[9] 0s
def countdown(n):
    if n < 0:
        return
    print(n)
    countdown(n - 1)

countdown(5)
```

Below the code, the output of the function execution is shown:

```
... 5
4
3
2
1
0
```

Explanation: The recursive function lacks a base case, so it keeps calling itself indefinitely. This leads to infinite recursion and eventually a runtime error. Adding a proper stopping condition ensures safe termination.

## TASK 4: Dictionary Key Error

ACTUAL CODE:

The screenshot shows a Jupyter Notebook cell with the following code:

```
[6] In [1]: def countdown(n):
    print(n)
    return countdown(n-1)
countdown(5)
```

Below the code, an error message is displayed:

```
... File "/tmp/ipython-input-782688475.py", line 3
        return countdown(n-1)
               ^
IndentationError: unexpected indent
```

At the bottom of the cell, there are two buttons: "Next steps:" and "Explain error".

Prompt: Fix the function that raises a KeyError when accessing a non-existing dictionary key by using safe access or error handling.

CORRECTED CODE:

The screenshot shows a Jupyter Notebook interface. At the top, there are tabs for 'Commands', '+ Code' (with a dropdown arrow), '+ Text', and 'Run all'. Below the tabs is a code cell with the following content:

```
[10] ✓ 0s
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")

print(get_value())

```

The output of the cell is:

```
... Key not found
```

Explanation: Accessing a key that does not exist in a dictionary raises a `KeyError`. Using safe access methods or handling missing keys prevents the program from crashing.

## TASK 5: Infinite Loop – Wrong Condition

### ACTUAL CODE:

The screenshot shows a Jupyter Notebook interface. At the top, there are tabs for 'Commands', '+ Code' (with a dropdown arrow), '+ Text', and 'Run all'. Below the tabs is a code cell with the following content:

```
[11] ⓘ 0s
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]
print(get_value())

```

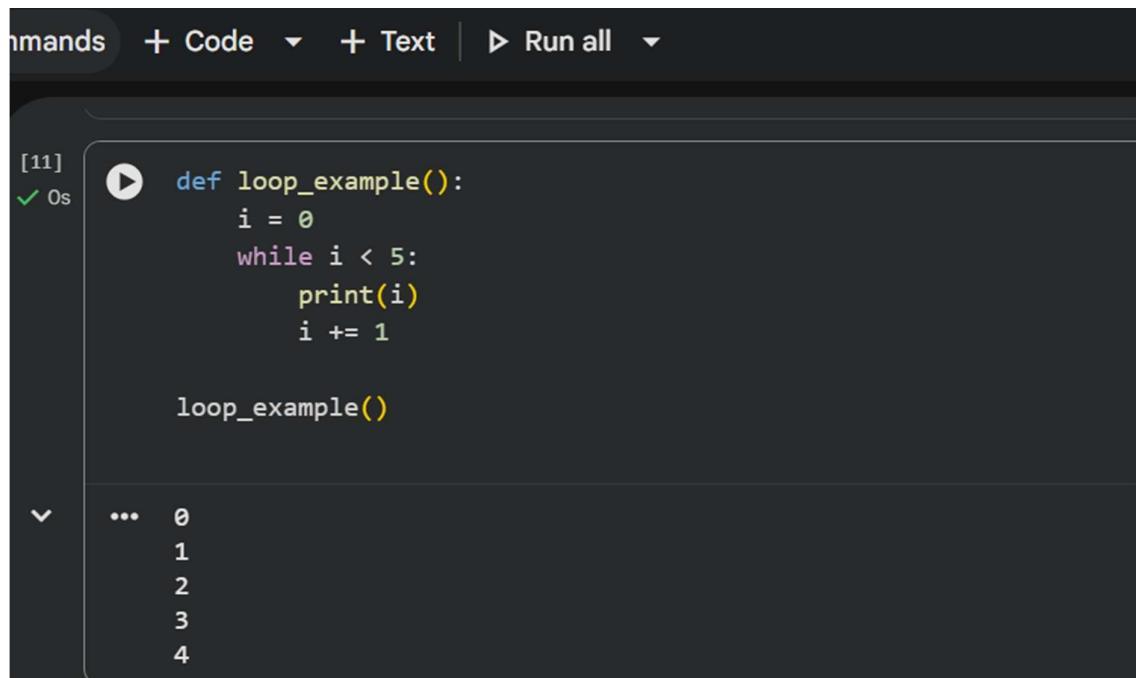
The output of the cell is:

```
... File "/tmp/ipython-input-3600671670.py", line 2
      data = {"a": 1, "b": 2}
      ^
IndentationError: expected an indented block after function definition on line 1
```

At the bottom of the cell, there is a button labeled 'Next steps: Explain error'.

Prompt: Detect and correct the infinite loop caused by an incorrect loop condition so the loop terminates properly.

## CORRECTED CODE:



The screenshot shows a Jupyter Notebook interface with a dark theme. At the top, there are tabs for 'Commands', '+ Code' (which is active), '+ Text', and 'Run all'. Below the tabs, a code cell is displayed with the following content:

```
[11] ✓ 0s
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1

loop_example()
...
```

The output of the code is shown below the cell, displaying the numbers 0, 1, 2, 3, and 4, each on a new line.

Explanation: The loop condition is correct, but the loop variable is never updated. This causes the loop to run endlessly. Incrementing the loop variable allows proper termination.

## TASK 6: Unpacking Error – Wrong Variables

### ACTUAL CODE:

```
[12] ① 0s
  def loop_example():
    i = 0
    while i < 5:
        print(i)

...
File "/tmp/ipython-input-3417722996.py", line 2
    i = 0
          ^
IndentationError: expected an indented block after function definition on line 1

Next steps: Explain error
```

Prompt: Analyze the tuple unpacking error caused by mismatched variables and fix it using proper unpacking.

CORRECTED CODE:

```
Commands + Code ▾ + Text ▾ ▶ Run all ▾

[12] ② 0s
  a, b, _ = (1, 2, 3)
  print(a, b)

...
1 2
```

Explanation: Tuple unpacking fails when the number of variables does not match the number of values. Correct unpacking or ignoring extra values resolves the error.

TASK 7: Mixed Indentation – Tabs vs Spaces

ACTUAL CODE:

The screenshot shows a code editor interface with a dark theme. In the top left corner, there is a status bar with '[14]' and '0s'. Below it, a code editor window displays the following Python code:

```
[14] 0s
def func():
    x = 5
    y = 10
    return x+y

... File "/tmp/ipython-input-1176682017.py", line 2
        x = 5
        ^
IndentationError: expected an indented block after function definition on line 1
```

At the bottom of the code editor, there is a button labeled 'Next steps: Explain error'.

Prompt: Correct the Python function that fails due to mixed or incorrect indentation by applying consistent indentation.

CORRECTED CODE:

The screenshot shows a code editor interface with a dark theme. In the top left corner, there is a status bar with '[13]' and '0s'. Below it, a code editor window displays the following Python code:

```
[13] 0s
def func():
    x = 5
    y = 10
    return x + y

print(func())
... 15
```

Explanation: Python relies on indentation to define code blocks. Mixed or incorrect indentation causes syntax errors. Using consistent spacing fixes the issue

TASK 8: Import Error – Wrong Module Usage

ACTUAL CODE:

The screenshot shows a Jupyter Notebook cell with the following code:

```
[15] 0s ⏪ print(maths.sqrt(16))
```

When run, it produces the following traceback:

```
... -----
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-3375551128.py in <cell line: 0>()
----> 1 print(maths.sqrt(16))

NameError: name 'maths' is not defined
```

Below the code cell, there is a button labeled "Next steps: Explain error".

Prompt: Correct the Python function that fails due to mixed or incorrect indentation by applying consistent indentation.

### CORRECTED CODE:

The screenshot shows a Jupyter Notebook cell with the following corrected code:

```
[15] 0s ⏪ import math  
      print(math.sqrt(16))
```

When run, it produces the output:

```
...   4.0
```

Explanation: The error occurs due to importing a nonexistent module. Using the correct standard library module name resolves the import issue.

