ASSIGNMENT-7.5

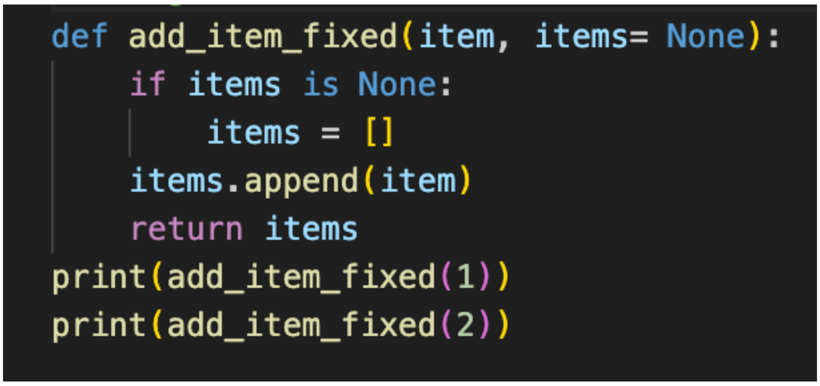
Ch.Sushanth
2303A51750
Batch-11

Task 1 (Mutable Default Argument – Function Bug)

Prompt used:

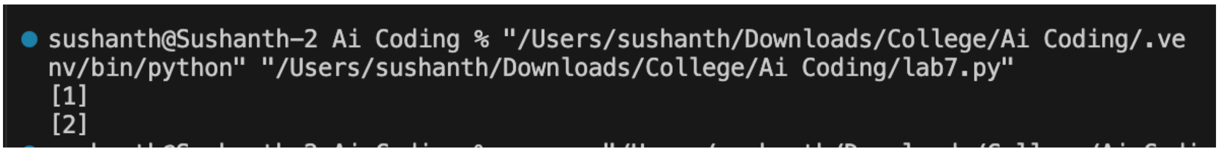
#analyze the given code above where a mutable default argument causes unexpected behaviour
# The given code defines a function `add_item` that takes an item and a list of items (with a default value of an empty list).

Code:



```python
def add_item_fixed(item, items= None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item_fixed(1))
print(add_item_fixed(2))
```

Output:



```
● sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.ve
  nv/bin/python" "/Users/sushanth/Downloads/College/Ai Coding/lab7.py"
  [1]
  [2]
```

Explanation:

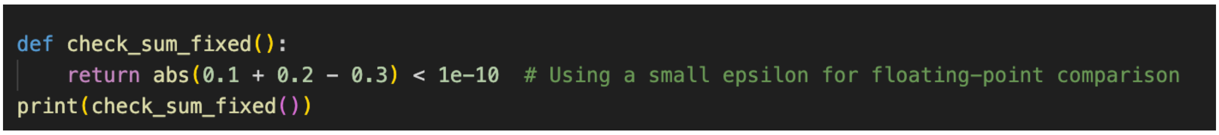
The common and recommended fix for mutable default arguments is to use None as the default value. Inside the function, check if items is None, and if it is, then initialize an empty list items = []. This ensures that a new, empty list is created each time the function is called without an explicit items argument, preventing the shared list issue.

Task 2 (Floating-Point Precision Error)

Prompt used:

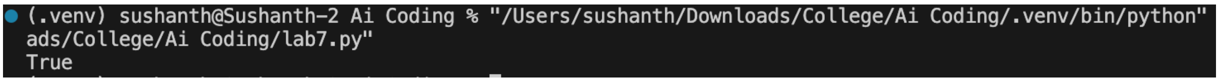
#analyze the given code where a floating-point comparison fails due to precision issues.
# The given code defines a function `check_sum` that checks if the sum of 0
#Analyze given code where floating-point comparison fails.

Code:



```
def check_sum_fixed():
    return abs(0.1 + 0.2 - 0.3) < 1e-10  # Using a small epsilon for floating-point comparison
print(check_sum_fixed())
```

Output:



```
(.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
ads/College/Ai Coding/lab7.py"
True
```

Explanation:

To correctly compare floating-point numbers, instead of checking for exact equality, we check if their absolute difference is less than a small tolerance value (often called epsilon). If the difference is smaller than this tolerance, the numbers are considered practically equal.

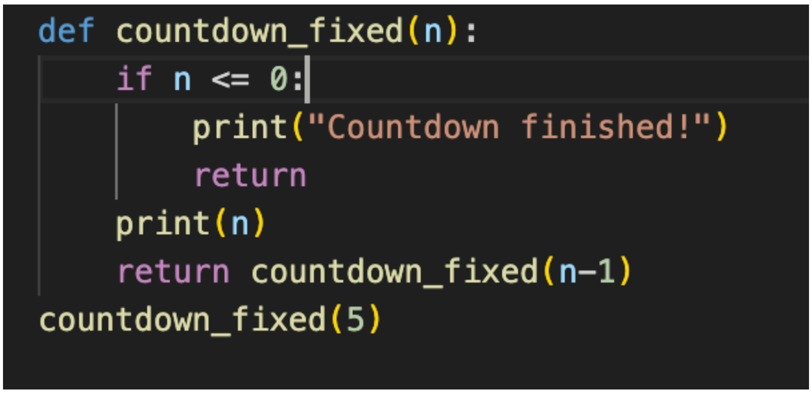
Python's math module also provides math.isclose(), which is a convenient and robust way to perform such comparisons, taking into account both relative and absolute tolerances.

Task 3 (Recursion Error – Missing Base Case)

Prompt used:

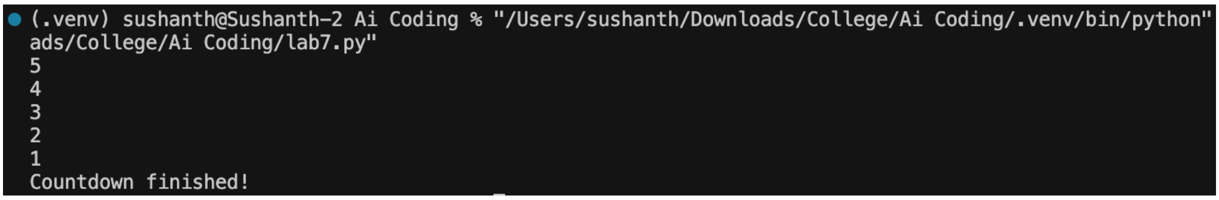
#analyze the given code where infinite recursion occurs due to lack of a base case.
# The given code defines a recursive function `countdown` that prints the number `n` and then calls itself with `n-1`.

Code:



```
def countdown_fixed(n):
    if n <= 0:
        print("Countdown finished!")
        return
    print(n)
    return countdown_fixed(n-1)
countdown_fixed(5)
```

Output:



```
● (.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
  ads/College/Ai Coding/lab7.py"
  5
  4
  3
  2
  1
  Countdown finished!
```

Explanation

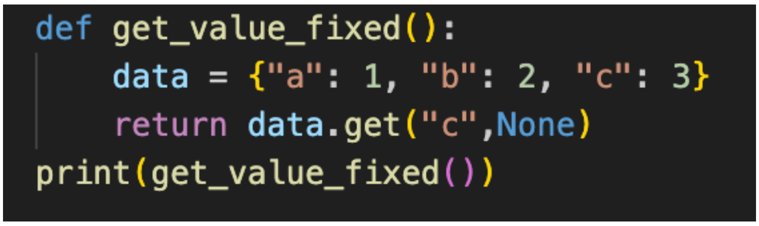
The fix involves adding an if n <= 0: condition at the beginning of the countdown_fixed function. This is our base case. When n becomes 0 or less, the function prints "Countdown finished!" and then returns, effectively stopping the chain of recursive calls. This prevents the RecursionError and ensures the function behaves as intended.

Task 4 (Dictionary Key Error)

Prompt used:

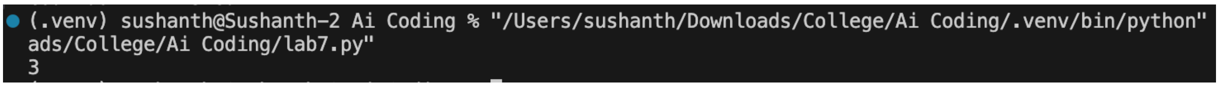
#analyze the code given where a KeyError may occur due to accessing a non-existent key in a dictionary.
# The given code defines a function `get_value` that attempts to access the value associated with the key "a" in a dictionary.

Code:



```
def get_value_fixed():
    data = {"a": 1, "b": 2, "c": 3}
    return data.get("c",None)
print(get_value_fixed())
```

Output:



```
(.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
ads/College/Ai Coding/lab7.py"
3
```

Explanation:

There are two common ways to handle missing dictionary keys gracefully:

1. **Using the .get() method**: Instead of dictionary[key], you can use dictionary.get(key).

If key exists, it returns its corresponding value. If key does not exist, it returns None by default, or a specified default value if provided
(e.g., dictionary.get(key, 'default_value')). This avoids raising a KeyError.

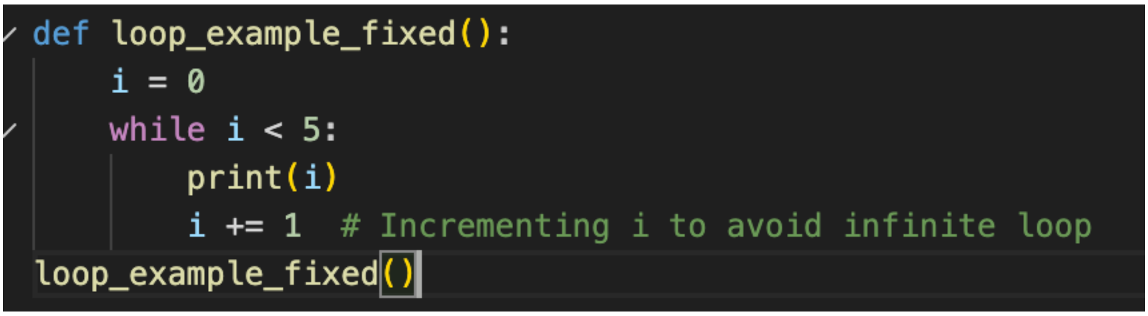
2. **Using a try-except block**: You can wrap the dictionary access dictionary[key] within a try block. If a KeyError occurs, it will be caught by the except KeyError block, where you can define how to handle the error (e.g., return a default value, log the error, or raise a different exception).

Task 5 (Infinite Loop – Wrong Condition)

Prompt used:

#analyze the given code and detect the error and fix it
# The given code defines a function `loop_example` that initializes a variable `i` to
# 0 and then enters a while loop that continues as long as `i` is less than 5. However, the variable `i` is never incremented within the loop, resulting in an infinite loop.

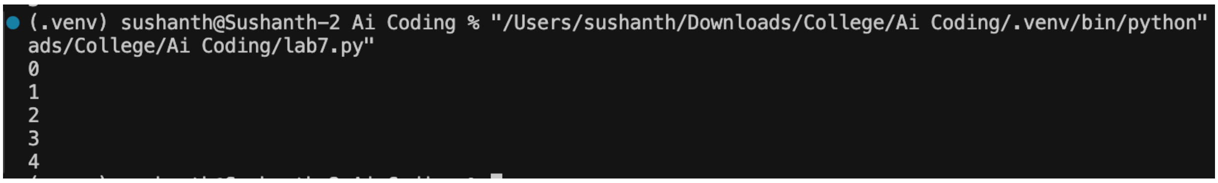
Code:



```python
def loop_example_fixed():
    i = 0
    while i < 5:
        print(i)
        i += 1  # Incrementing i to avoid infinite loop
loop_example_fixed()
```

#

Output:



```
(.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
ads/College/Ai Coding/lab7.py"
0
1
2
3
4
```

Explanation:

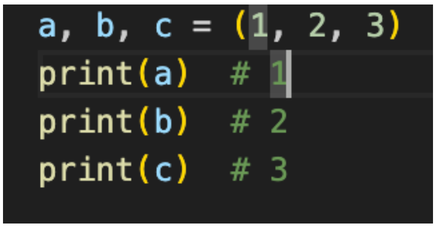
The fix involves adding i += 1 inside the while loop. This statement increments the value of i in each iteration. With i increasing, it will eventually reach 5 (or greater), causing the loop condition i < 5 to become false, and the loop will terminate as intended. This ensures that the loop executes a finite number of times.

Task 6 (Unpacking Error – Wrong Variables)

Prompt used:

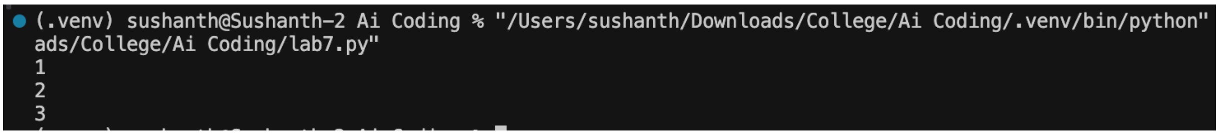
#analyze the given code where a ValueError occurs due to unpacking more values than expected.
# The given code attempts to unpack a tuple with three values into two variables `a` and
# `b`, which results in a ValueError because there are more values in the tuple than variables to unpack into.
# To fix this, we can either reduce the number of values in the tuple or increase.

Code:



```
a, b, c = (1, 2, 3)
print(a)  # 1
print(b)  # 2
print(c)  # 3
```

Output:



```
● (.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
  ads/College/Ai Coding/lab7.py"
  1
  2
  3
```

Explanation:

Match the number of variables: The most straightforward fix is to ensure that the number of variables on the left-hand side exactly matches the number of elements in the sequence being unpacked. If the sequence has three elements, you need three variables.

Use _ for unwanted values: If you only care about a subset of the values in the sequence, you can use the underscore _ as a placeholder variable for the elements you want to ignore. This is a convention in Python to indicate a variable whose value is not going to be used.

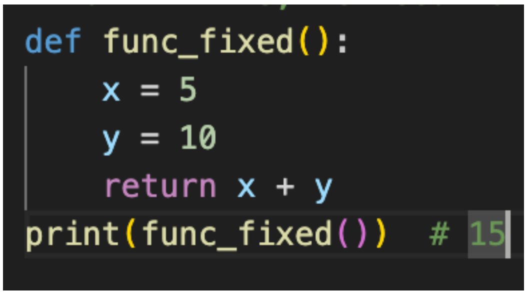
Use extended unpacking (* operator): For more flexible unpacking, especially with sequences of unknown length or when you want to capture multiple remainingitems, Python 3+ allows the use of the * operator (e.g., *rest). This will collect all remaining items into a list. You can also use *_ to discard multiple remaining items explicitly.

Task 7 (Mixed indentation- tabs vs spaces)

Prompt used:

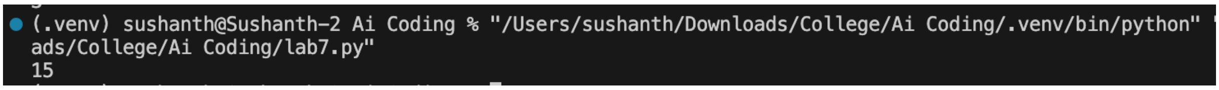
#analyze the given code where an IndentationError occurs due to inconsistent indentation.
# The given code defines a function `func` that initializes two variables `x` and `
# To fix this, we need to ensure that both lines are indented at the same level

Code:



```
def func_fixed():
    x = 5
    y = 10
    return x + y
print(func_fixed())  # 15
```

Output:



```
● (.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python" '
  ads/College/Ai Coding/lab7.py"
  15
```

Explanation:

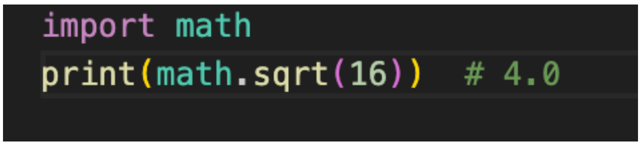
The fix involves ensuring consistent indentation throughout the code. The Python community standard (PEP 8) recommends using 4 spaces per indentation level. By replacing the tab with spaces (or vice-versa, as long as it's consistent), the IndentationError is resolved, and the code runs as expected.

Task 8 (Import Error – Wrong Module Usage)

Prompt used:

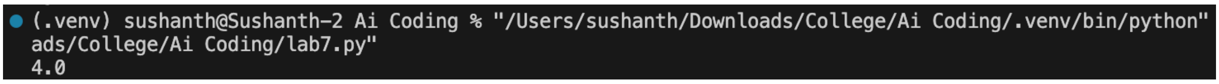
#analyze the code and fix the error wrong import statement and now fix it
# The given code attempts to import a module named `maths`, which does not exist in the Python standard library. The correct module name is `math`.

Code:



```
import math
print(math.sqrt(16))  # 4.0
```

Output:



```
● (.venv) sushanth@Sushanth-2 Ai Coding % "/Users/sushanth/Downloads/College/Ai Coding/.venv/bin/python"
  ads/College/Ai Coding/lab7.py"
  4.0
```

Explanation of the Fix:

The fix is straightforward: correct the typo in the import statement from import maths to import math. The math module is a standard Python library that provides mathematical functions, including sqrt for square root. Once the correct module is imported, its functions can be called without error.