ASSIGNMENT-7.5

HT.NO:2303A51731

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 **Original**

Corrected Code:

C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING\a2.py

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

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/python.exe" "c:/Users/sanja/OneDrive/D
esktop/PYTHON TRAINING/lab7.py"
[1]
[2]
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING>
```

Explanation of the Fix:

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)

Task: Analyze given code where floating-point comparison fails.

Use AI to correct with tolerance.

Corrected Code:

C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING\a2.py

```
#Bug: Floating point precision issue
#Analyze given code where floating-point comparison fails.Use AI to correct with tolerance.
def check_sum():
    return abs((0.1+0.2) - 0.3) < 1e-9
print(check_sum())

```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/python.exe" "c:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/lab7.py"
True
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING>
```

Explanation of the Fix:

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)

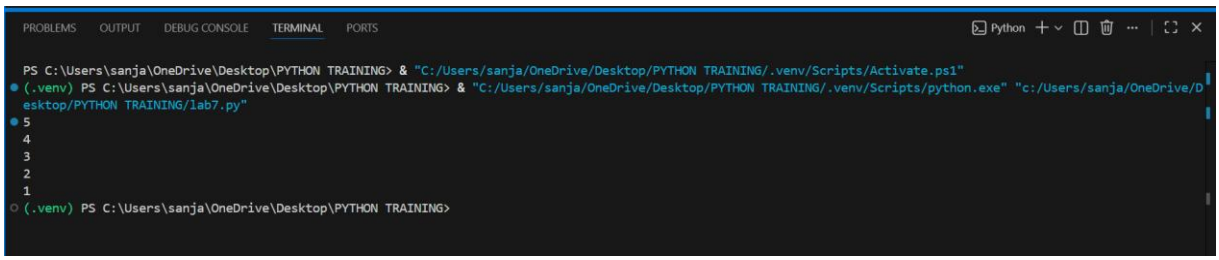
Task: Analyze given code where recursion runs infinitely due to missing

base case. Use AI to fix.

Corrected Code:

```python
#Task: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.
# Bug: No base case
def countdown(n):
    if n == 0:
        return
    print(n)
    return countdown(n - 1)
countdown(5)
```

Output:



Explanation of the Fix:

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)

Task: Analyze given code where a missing dictionary key causes

error. Use AI to fix it.

Corrected Code:

```python
#Task: Analyze given code where a missing dictionary key causeserror. Use AI to fix it.
# Bug: Accessing non-existing key
def get_value():
    data={"a":1,"b":2}
    return data["c"]
try:
    print(get_value())
except KeyError:
    print("Key not found")

```

Output:

```
PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/python.exe" "c:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/lab7.py"
Key not found
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING>
```

Explanation of the Fix:

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)

Task: Analyze given code where loop never ends. Use AI to detect

and fix it.

Corrected Code:

Fix: Increment the loop counter within the loop

```
C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING\a2.py
#Task 5: Analyze given code where loop never ends. Use AI to detect and fix it.
#bug:Infinite loop
def loop_example():
    i=0
    while i<5:
        print(i)
        i += 1

```

Output:

Explanation of the Fix:

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)

Corrected Code:

```
lab7.py > ...
#task 6: Analyze given code where tuple unpacking fails.Use AI to fix it.
#Bug: Wrong Unpacking
#FIX THE CODE BELOW
a,b,c=(1,2,3)
print(a)
print(b,c)import random, time, cProfile, pstats, io
```

Output:

Explanation of the Fix:

```
PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING> & "C:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/.venv/Scripts/python.exe" "c:/Users/sanja/OneDrive/Desktop/PYTHON TRAINING/lab7.py"
1
2 3
(.venv) PS C:\Users\sanja\OneDrive\Desktop\PYTHON TRAINING>
```

There are several ways to fix an unpacking error, depending on your intent:

1. **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.
2. **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.
3. **Use extended unpacking (* operator)**: For more flexible unpacking, especially with sequences of unknown length or when you want to capture multiple remaining items, 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.

Task7:

Correceted Code:

Fix: Ensure consistent indentation (use only spaces or only tabs, preferably spaces)

a2.py lab7.py

lab7.py > fun

```python
#task7: Analyze given code where mixed indentation breaks execution .Use AI to fix it.
#bug:Mixed identation
#fix the code below:
def fun():
    x=5
    y=10
    return x+y
print(fun())

```

Output:

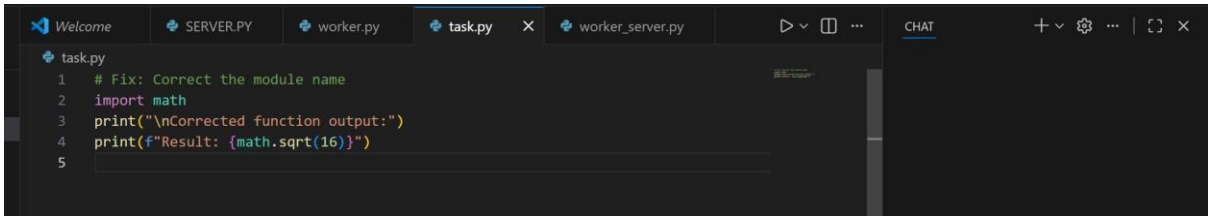
Explanation of the Fix:

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)

Task: Analyze given code with incorrect import. Use AI to fix.

Corrected Code:



Output:

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.