ASSIGNMENT-7.5

HT.NO:2303A510G2

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 Buggy Code:

```
# Bug: Mutable default argument
def add_item(item, items=[]):
    items.append(item)
    return items
print("Original buggy function output:")
print(f"Call 1: {add_item(1)}")
print(f"Call 2: {add_item(2)}")
print(f"Expected for Call 2: [2] (but it's [1, 2])")
```

Output:

```
print(f"Expected for Call 2: [2] (but it's [1, 2])")

Original buggy function output:
Call 1: [1]
Call 2: [1, 2]
Expected for Call 2: [2] (but it's [1, 2])
```

Explanation of the Bug:

The add_item function uses a list [] as a default argument for items. In Python, default arguments are evaluated only once, when the function is defined, not every time the

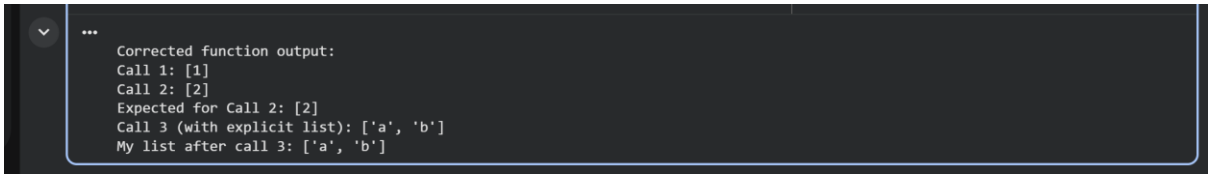
function is called. This means that all calls to add_item without explicitly providing an items list will share the *same* list object.

When add_item(1) is called, 1 is appended to the shared default list. When add_item(2) is called *again without a new list*, it appends 2 to the *same* list that already contains 1, resulting in [1, 2] instead of the expected [2] (if each call was meant to start with an empty list).

Corrected Code:

```
# Fix: Use None as default and initialize an empty list inside the function

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

print("\nCorrected function output:")

print(f"Call 1: {add_item_fixed(1)}")

print(f"Call 2: {add_item_fixed(2)}")

print(f"Expected for Call 2: [2]")

# Demonstrating with an explicit list

my_list = ['a']

print(f"Call 3 (with explicit list): {add_item_fixed('b', my_list)}")

print(f"My list after call 3: {my_list}")
```

Output:



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.

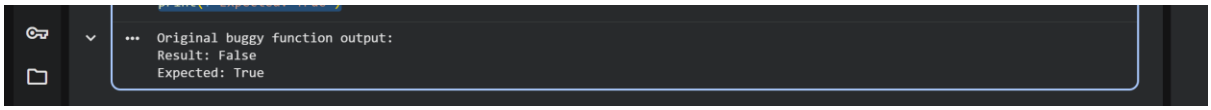
```
# Bug: Floating point precision issue

def check_sum():

return (0.1 + 0.2) == 0.3

print(check_sum())
```

Expected Output: Corrected function

Original Buggy Code:

```
# Bug: Floating point precision issue

def check_sum():

    return (0.1 + 0.2) == 0.3

print("Original buggy function output:")

print(f"Result: {check_sum()}")

print(f"Expected: True")
```

Output:



Explanation of the Bug:

Floating-point numbers (like 0.1, 0.2, 0.3) are represented in computers using binary fractions. Most decimal fractions, such as 0.1, cannot be represented exactly in binary. This leads to tiny precision errors.

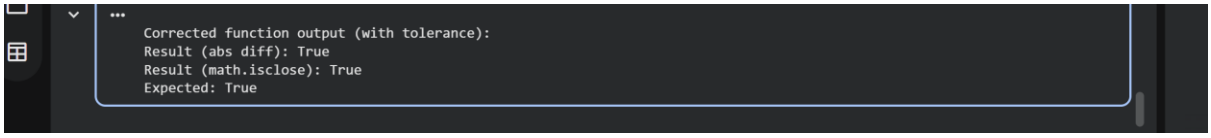
When 0.1 + 0.2 is computed, the result is not exactly 0.3, but a number very close to it (e.g., 0.30000000000000004). Directly comparing this slightly imprecise sum with the exactly represented 0.3 using == will return False, even though mathematically they should be equal.

Corrected Code:

```
# Fix: Use a tolerance for floating-point comparison

import math

def check_sum_fixed():

# Define a small tolerance (epsilon)
```

```
    tolerance = 1e-9  # A common choice for floating-point comparisons
    sum_val = 0.1 + 0.2
    expected_val = 0.3
    # Compare if the absolute difference is within the tolerance
    return abs(sum_val - expected_val) < tolerance
# Alternative using math.isclose() (Python 3.5+)
def check_sum_isclose():
    return math.isclose(0.1 + 0.2, 0.3)
print("\nCorrected function output (with tolerance):")
print(f"Result (abs diff): {check_sum_fixed()}")
print(f"Result (math.isclose): {check_sum_isclose()}")
print(f"Expected: True")
```

Output:



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.

```
# Bug: No base case
def countdown(n):
print(n)
return countdown(n-1)
countdown(5)
```

Expected Output : Correct recursion with stopping condition.

Original Buggy Code:

```
# Bug: No base case
def countdown(n):
    print(n)
    return countdown(n-1)

print("Original buggy function output (will raise RecursionError):")
try:
    countdown(5)
except RecursionError as e:
    print(f"\nError: {e}")
    print("This is expected because the function lacks a base case.")
```

-684

-685

-686

-687

-688

-689

-690

-691

-692

-693

-694

-695

-696

-697

-698

-699

-700

-701

-702

-703

-704

-705

-706

-707

Explanation of the Bug:

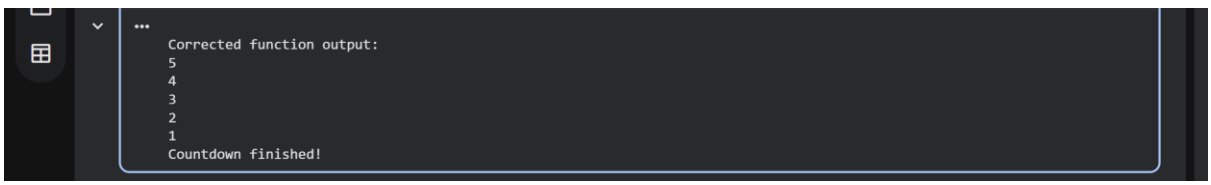
Recursion is a programming technique where a function calls itself to solve a problem. Every recursive function *must* have a **base case**, which is a condition that stops the recursion. Without a base case, the function will call itself infinitely, leading to an ever-growing stack of function calls.

In the countdown function, there's no condition to stop the countdown(n-1) call.
When n reaches 0, it continues to call countdown(-1), countdown(-2), and so on. Eventually, Python's call stack limit is exceeded, resulting in a RecursionError.

Corrected Code:

```
# Fix: Add a base case to stop the recursion

def countdown_fixed(n):

    if n <= 0:  # Base case: stop when n is 0 or less

        print("Countdown finished!")

        return

    print(n)

    countdown_fixed(n-1) # Recursive call


print("\nCorrected function output:")

countdown_fixed(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.

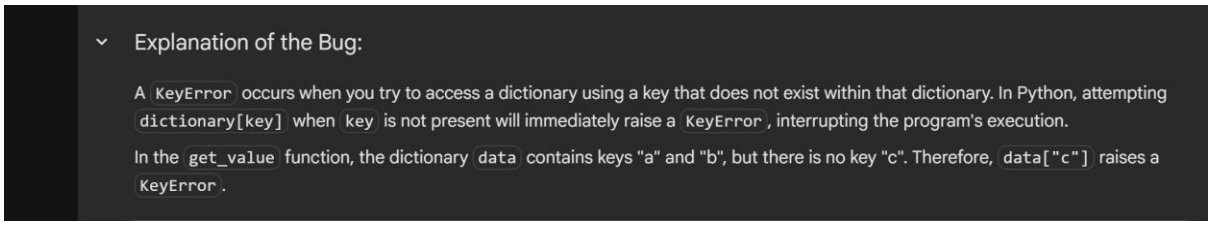
```
# Bug: Accessing non-existing key

def get_value():

data = {"a": 1, "b": 2}

return data["c"]

print(get_value())
```

Expected Output: Corrected with .get() or error handling.

Original Buggy Code:

```
# Bug: Accessing non-existing key

def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]

print("Original buggy function output (will raise KeyError):")

try:
    print(get_value())

except KeyError as e:
    print(f"\nError: {e}")
    print("This is expected because the key 'c' does not exist in the dictionary.")
```

Output:



Explanation of the Bug:

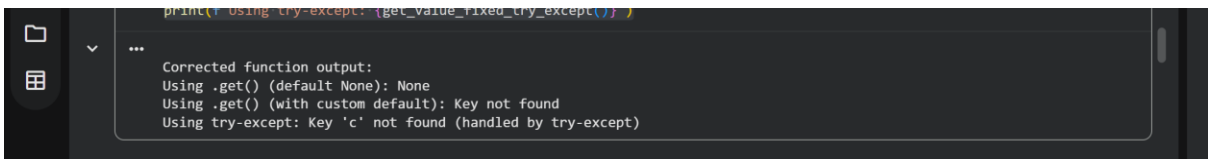
A KeyError occurs when you try to access a dictionary using a key that does not exist within that dictionary. In Python, attempting dictionary[key] when key is not present will immediately raise a KeyError, interrupting the program's execution.

In the get_value function, the dictionary data contains keys "a" and "b", but there is no key "c". Therefore, data["c"] raises a KeyError.

Corrected Code:

```
# Fix: Use .get() method or error handling
def get_value_fixed_get():
    data = {"a": 1, "b": 2}
    # Using .get() method with a default value (None if not found)
    return data.get("c")
def get_value_fixed_default():
    data = {"a": 1, "b": 2}
    # Using .get() method with a specified default value
    return data.get("c", "Key not found")
def get_value_fixed_try_except():
    data = {"a": 1, "b": 2}
    try:
        return data["c"]
    except KeyError:
        return "Key 'c' not found (handled by try-except)"
print("\nCorrected function output:")
print(f"Using .get() (default None): {get_value_fixed_get()}")
print(f"Using .get() (with custom default): {get_value_fixed_default()}")
print(f"Using try-except: {get_value_fixed_try_except()}")
```

Output:



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.

Bug: Infinite loop

def loop_example():

i = 0

while i < 5:

print(i)

Expected Output: Corrected loop increments i.

Original Buggy Code:

```
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)


print("Original buggy function output (will run infinitely if not stopped):")
# Calling the function to demonstrate the infinite loop.
# You might need to manually stop the execution of this cell if it runs too long.
loop_example()
```

Output:

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

Explanation of the Bug:

An infinite loop occurs when the condition that controls the loop never becomes false. In this while loop, the variable i is initialized to 0, and the loop continues as long as i < 5. However, there is no code inside the loop that modifies the value of i. As a result, i remains 0 indefinitely, and the condition i < 5 is always true, causing the loop to print 0 repeatedly without ever terminating.

Corrected Code:

```
# Fix: Increment the loop counter within the loop

def loop_example_fixed():

    i = 0
```

```
    while i < 5:
        print(i)
        i += 1  # Increment i so the loop condition eventually becomes false

print("\nCorrected function output:")
loop_example_fixed()
```

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)

Task: Analyze given code where tuple unpacking fails. Use AI to

fix it.

```
# Bug: Wrong unpacking
a, b = (1, 2, 3)
```

Expected Output: Correct unpacking or using _ for extra values.

Original Buggy Code:

```
# Bug: Wrong unpacking
try:
    a, b = (1, 2, 3)
except ValueError as e:
    print(f"Error: {e}")
    print("This is expected because the number of variables does not match the number of values.")
```

Explanation of the Bug:

In Python, sequence unpacking allows you to assign elements of an iterable (like a tuple or list) to multiple variables simultaneously. A ValueError occurs during unpacking when the number of variables on the left-hand side of the assignment does not match the number of elements in the sequence on the right-hand side.

In this buggy code, we attempt to unpack a tuple (1, 2, 3) (which has three elements) into only two variables, a and b. This mismatch causes Python to raise a ValueError, indicating that there are too many values to unpack.

Corrected Code:

Fix: Match the number of variables to the number of values, or use extended unpacking

Option 1: Match the number of variables

x, y, z = (1, 2, 3)

print(f"Correct unpacking (match variables): x={x}, y={y}, z={z}")

Option 2: Use underscore '_' for unwanted values (if you only need a subset)

a, b, _ = (1, 2, 3) # Unpacks 1 into a, 2 into b, and 3 into the discard variable _

print(f"Correct unpacking (discarding with _): a={a}, b={b}")

Option 3: Use extended unpacking (Python 3+) for flexible assignments

If you only need the first two and want to catch the rest

first, second, *rest = (1, 2, 3, 4, 5)

print(f"Correct unpacking (extended unpacking, *rest): first={first}, second={second}, rest={rest}")

If you only need the first two and explicitly discard the rest

value1, value2, *_ = (10, 20, 30, 40)

print(f"Correct unpacking (extended unpacking, *_): value1={value1}, value2={value2}")

Explanation of the Fix:

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

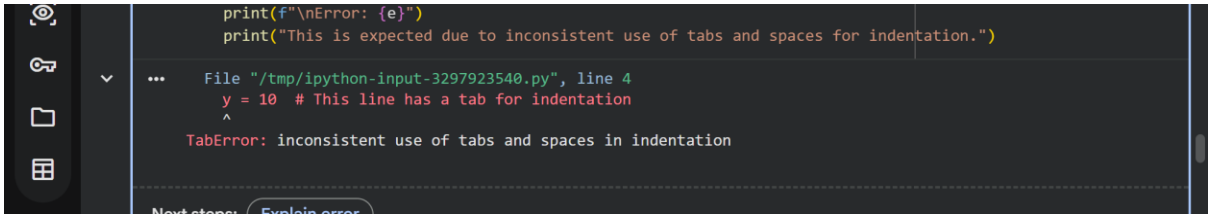
1. **Match the number of variables**: The most straightforward fix is to ensure that the number of variables on the left-hand side exactly matches the number of elements in the sequence being unpacked. If the sequence has three elements, you need three variables.

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.

Original Buggy Code:

```
# Bug: Mixed indentation
def func():
    x = 5
  y = 10  # This line has a tab for indentation
    return x+y
print("Original buggy function output (will raise IndentationError):")
try:
    func()
except IndentationError as e:
    print(f"\nError: {e}")
    print("This is expected due to inconsistent use of tabs and spaces for indentation.")
```

Output:



Explanation of the Bug:

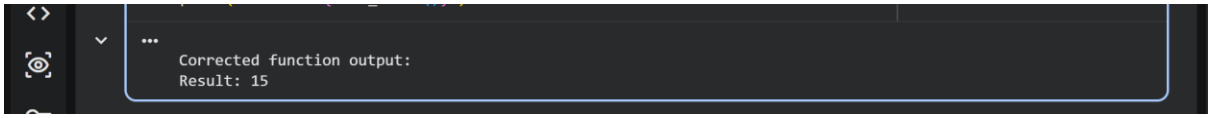
Python uses indentation to define code blocks. Mixing tabs and spaces for indentation within the same file or even within the same code block is a common source of IndentationError. While Python 3 largely warns against and often prohibits such mixing, older Python 2 versions might interpret them differently, leading to subtle logic errors. Even in Python 3, if a tab and spaces are used to define what should be the same indentation level, Python raises an IndentationError because it cannot consistently determine the block structure.

In the func function, the line x = 5 is indented with spaces, while the line y = 10 is intended to be at the same level but is indented with a tab. This inconsistency causes Python to raise an IndentationError.

Correceted Code:

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

def func_fixed():

    x = 5

    y = 10  # Corrected to use spaces consistently

    return x+y

print("\nCorrected function output:")

print(f"Result: {func_fixed()}")
```

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.

```
# Bug: Wrong import

import maths

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

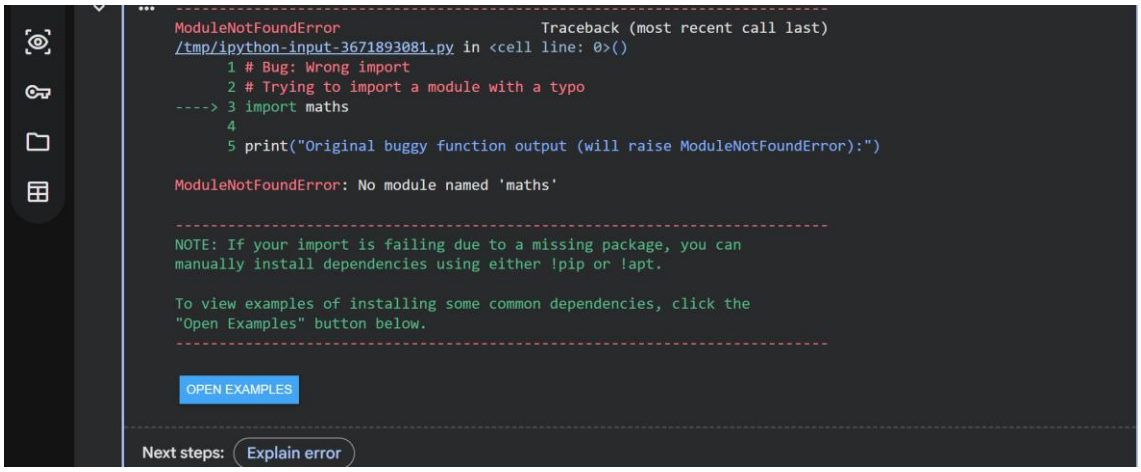
Expected Output: Corrected to import math

Original Buggy Code:

```
# Bug: Wrong import

# Trying to import a module with a typo

import maths


print("Original buggy function output (will raise ModuleNotFoundError):")
```

```
try:
    print(maths.sqrt(16))
except ModuleNotFoundError as e:
    print(f"\nError: {e}")
    print("This is expected because the module 'maths' does not exist (typo for 'math').")
```

Output:



Explanation of the Bug:

A ModuleNotFoundError occurs when Python cannot find the module you are trying to import. This can happen due to several reasons, including a typo in the module name, the module not being installed, or issues with Python's path.

In this specific buggy code, the error is a simple typo: maths instead of the correct module name math. When Python attempts to import maths, it cannot locate a module with that name and thus raises a ModuleNotFoundError.

Corrected Code:

```
# Fix: Correct the module name
import math
print("\nCorrected function output:")
print(f"Result: {math.sqrt(16)}")
```

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.