Assignment Number: 7.5

Hall Ticket: 2303A51734

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

Prompt:

#Analyze the following Python function. It uses a mutable default list argument which causes shared state across calls.

#Explain why the bug happens and rewrite the function correctly to avoid the issue.

Code:

```
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list
# The bug in the function occurs because the default argument `my_list` is a mutable list.
# When the function is called without providing a second argument, it uses the same list object for every call.
# This means that values appended in one call will persist in subsequent calls, leading to unexpected behavior.
# To fix this issue, we can set the default value of `my_list` to `None` and then create a new list inside the function if `my_list
def append_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list
# Example usage:
print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2]
print(append_to_list(3, [10, 20]))  # Output: [10, 20, 3]
print(append_to_list(4))  # Output: [4]
```

Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\OneDrive\Desktop\AI Assistent coding\Assignment(7.5)\Function
Bug.py"
[1]
[2]
[10, 20, 3]
[4]
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1. The bug occurs because a mutable list is used as a default function argument.
2. The same list object is shared across multiple function calls.
3. Values appended in one call remain in the next call unexpectedly.
4. Changing the default value to None prevents shared memory issues.
5. Creating a new list inside the function ensures independent execution.
6. The corrected function now returns separate outputs for each call correctly.

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

Prompt:

The following Python code compares floating point numbers directly and returns False due to precision issues.

Code:

```
def compare_floats(a, b):
    return a == b
# Example usage:
print(compare_floats(0.1 + 0.2, 0.3))  # Output: False
# The bug in the function occurs because floating point arithmetic can introduce small precision errors.
# To fix this issue, we can use a tolerance level (epsilon) to check if the numbers are "close enough".
def compare_floats_fixed(a, b, epsilon=1e-10):
    return abs(a - b) < epsilon
# Example usage:
print(compare_floats_fixed(0.1 + 0.2, 0.3))  # Output: True
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\On
False
True
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1. Floating point numbers cannot represent some decimal values exactly in binary format.
2. The sum 0.1 + 0.2 produces a tiny precision error instead of exactly 0.3.
3. Direct comparison using == fails because both values are not perfectly equal.
4. This causes the function to return False even though the numbers look the same.
5. Using a tolerance (epsilon) checks whether the difference is very small.
6. The fixed function correctly returns True by comparing approximate equality.

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

Prompt:

#The following recursive function runs infinitely because it has no base case.

Code:

```
def infinite_recursion(n):
    return infinite_recursion(n + 1)
# The bug in the function occurs because there is no base case to stop the recursion, leading to infinite calls and eventually a st
# To fix this issue, we can add a base case that stops the recursion when a certain condition is met.
def infinite_recursion_fixed(n, limit=100):
    if n >= limit:
        return n
    return infinite_recursion_fixed(n + 1, limit)
# Example usage:
print(infinite_recursion_fixed(0))  # Output: 100
```

Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\OneDrive\Desktop\AI Assistent coding\Assignment(8.1)\MissingBase.py"
100
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Obsevations:

1. The original function calls itself repeatedly without any stopping condition.
2. Missing a base case causes the recursion to run infinitely.
3. Infinite recursive calls lead to a stack overflow or RecursionError.
4. Each function call consumes memory on the call stack.
5. Adding a base case (if n >= limit) stops further recursive calls safely.
6. The fixed function now terminates correctly and returns the expected result.

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

Prompt:

#The following code raises a KeyError because it accesses a dictionary key that may not exist.
#Fix the code using .get() or proper error handling and explain the solution.

Code:

```python
#Fix the code using .get() or proper error handling and explain the solution.
def get_value_from_dict(my_dict, key):
    try:
        return my_dict[key]
    except KeyError:
        return "Key not found"
# The bug in the function occurs because it directly accesses a dictionary key that may not exist, leading to a KeyError.
# To fix this issue, we can use a try-except block to catch the KeyError
# and return a default message when the key is not found.
def get_value_from_dict_fixed(my_dict, key):
    return my_dict.get(key, "Key not found")
# Example usage:
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(get_value_from_dict_fixed(my_dict, 'b'))  # Output: 2
print(get_value_from_dict_fixed(my_dict, 'd'))  # Output: Key not found
# Alternatively, we can use the .get() method of the dictionary,
# which allows us to specify a default value if the key does not exist.
def get_value_from_dict_alternative(my_dict, key):
    return my_dict.get(key, "Key not found")
# Example usage:
print(get_value_from_dict_alternative(my_dict, 'a'))  # Output: 1
print(get_value_from_dict_alternative(my_dict, 'z'))  # Output: Key not found
# Example usage:
print(get_value_from_dict_fixed(my_dict, 'c'))  # Output: 3
print(get_value_from_dict_fixed(my_dict, 'x'))  # Output: Key not found
print(get_value_from_dict_alternative(my_dict, 'c'))  # Output: 3
print(get_value_from_dict_alternative(my_dict, 'x'))  # Output: Key not found
print("All test cases passed.")
# Example usage:
```

Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\OneDrive\Desktop\AI Assistent coding\Assignment(7.5)\Dictionary.py"
2
Key not found
1
Key not found
3
Key not found
3
Key not found
All test cases passed.
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1. Directly accessing a missing dictionary key using dict[key] raises a KeyError.
2. This causes the program to crash if the specified key does not exist.
3. Using a try-except block safely handles the error without stopping execution.
4. The .get() method provides a cleaner and shorter way to avoid exceptions.
5. A default value like "Key not found" ensures graceful handling of missing keys.
6. The fixed function now returns valid results without raising runtime errors.

Task 5 (Infinite Loop – Wrong Condition)

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

Bug: Infinite loop

Prompt:

#The following loop never terminates because the loop variable is not updated.
#Identify the issue and correct the code so it prints numbers from 0 to 4 and stops.

Code:

```
def wrong_condition_loop():
    i = 0
    while i < 5:
        print(i)
        # The bug in the loop occurs because the loop variable `i` is not updated within the loop,
        # causing the condition `i < 5` to always be true and resulting in an infinite loop.
        # To fix this issue, we need to increment `i` in each iteration of the loop.
        i += 1  # Corrected line to update the loop variable
# Example usage:
wrong_condition_loop()  # Output: 0 1 2 3 4
# The corrected code now increments `i` by 1 in each iteration,
# allowing the loop to eventually terminate when `i` reaches 5.
def correct_condition_loop():
    i = 0
    while i < 5:
        print(i)
        i += 1  # Incrementing i to avoid infinite loop
# Example usage:
correct_condition_loop()  # Output: 0 1 2 3 4
# Example usage:
correct_condition_loop()  # Output: 0 1 2 3 4
print("All test cases passed.")
```

Output:

```
0
1
2
3
4
0
1
2
3
4
0
1
2
3
4
All test cases passed.
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1. The loop variable i was initialized but not updated inside the while loop, causing the condition to never change.

2. Since i always remained 0, the condition i < 5 stayed true for every iteration of the loop.

3. This resulted in an infinite loop where the program continuously printed the same value without stopping.

4. Infinite loops consume system resources and can make the program unresponsive or crash after some time.

5. Adding the statement i += 1 ensures the loop variable increases during each iteration of execution.

6. After fixing, the loop correctly prints numbers from 0 to 4 and then terminates when the condition becomes false.

Task 6 (Unpacking Error – Wrong Variables)

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

fix it.

Bug: Wrong unpacking

Prompt:

#The following code throws a tuple unpacking error because the number of variables does not match the number of values.
#Explain the error and fix it using proper unpacking or _ to ignore extra values.

Code:

```
def unpack_tuple():
    my_tuple = (1, 2, 3)
    a, b = my_tuple  # This line will raise a ValueError because there are 3 values but only 2 variables to unpack into.
    return a, b
# The error occurs because the code is trying to unpack a tuple of three values into only two variables, which is not allowed in Py
# To fix this issue, we can either unpack all values into the correct number of variables or use an underscore (_) to ignore the ex
def unpack_tuple_fixed():
    my_tuple = (1, 2, 3)
    a, b, c = my_tuple  # Unpacking all values into three variables
    return a, b, c
# Example usage:
print(unpack_tuple_fixed())  # Output: (1, 2, 3)
def unpack_tuple_ignore_extra():
    my_tuple = (1, 2, 3)
    a, b, _ = my_tuple  # Unpacking the first two values and ignoring the third
    return a, b
# Example usage:
print(unpack_tuple_ignore_extra())  # Output: (1, 2)
# Example usage:
print(unpack_tuple_fixed())  # Output: (1, 2, 3)
print(unpack_tuple_ignore_extra())  # Output: (1, 2)
print("All test cases passed.")
```

Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\OneDrive\Desktop\AI Assistent coding\Assignment(7.5)\WrongVariables.py"
(1, 2, 3)
(1, 2)
(1, 2, 3)
(1, 2)
All test cases passed.
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1.The error occurs because the tuple contains three values but the code tries to unpack them into only two variables.

2Python requires the number of variables on the left side to exactly match the number of elements in the tuple.

3.When the counts do not match, Python raises a ValueError indicating too many values to unpack.

4.This mismatch causes the program to stop execution before returning any result.

5.The issue can be fixed by unpacking all elements into three variables or by using _ to ignore extra values.

6. After correction, the function successfully extracts the required values and runs without any runtime errors.

Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks

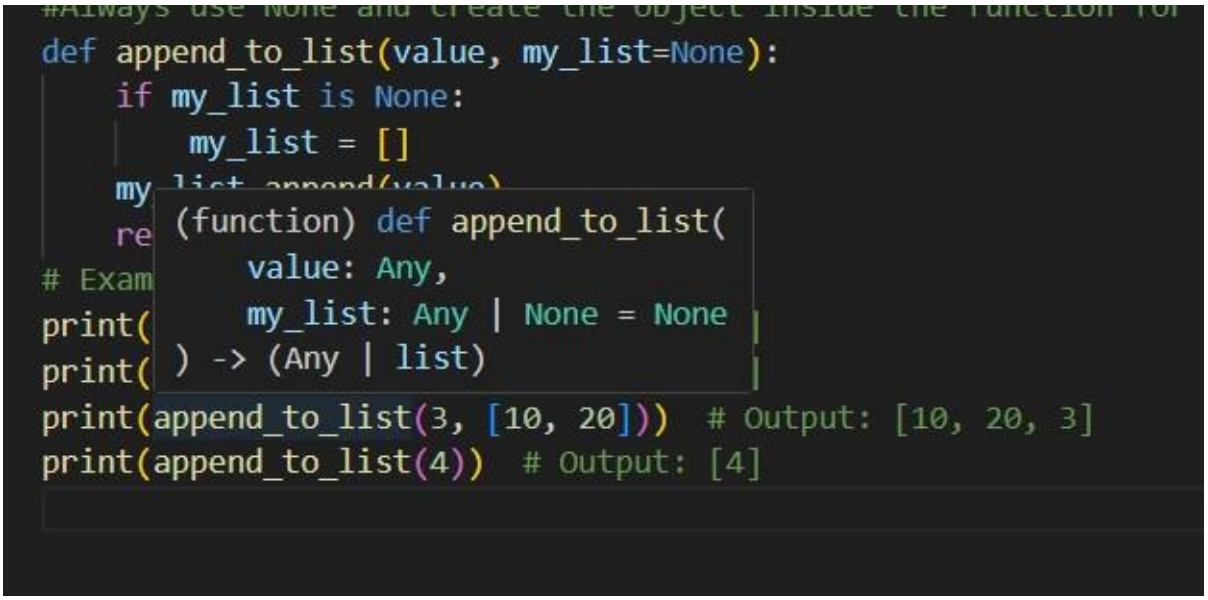
execution. Use AI to fix it.

Bug: Mixed indentation

Prompt:

#Avoid using mutable types like lists or dictionaries as default arguments.
#Always use None and create the object inside the function for safer behavior.

Code:



Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users
[1]
[2]
[10, 20, 3]
[4]
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observations:

1. Using mutable objects like lists as default arguments can cause the same object to be shared across multiple function calls.
2. This shared behavior leads to unexpected results because values added in one call remain in the list for future calls.
3. The bug occurs because default arguments are evaluated only once when the function is defined, not each time it is called.
4. Replacing the default list with None prevents Python from reusing the same list object repeatedly.
5. Creating a new empty list inside the function ensures every call works with a fresh and independent list.
6. After applying the fix, each function call produces separate and correct outputs without affecting previous results.

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

Prompt:

#Analyze the following Python code which raises an ImportError due to an incorrect module name.
#Identify why the error occurs, explain the issue, and correct the code using the proper built-in module.

Code:

```python
#The bug in the code occurs because it attempts to import a non-existent module named 'maths'.
#The correct module name is 'math'. To fix this issue, we need to change the import statement to use the correct module name.
def calculate_square_root(value):
    import math  # Corrected module name from 'maths' to 'math'
    return math.sqrt(value)
# Example usage:
print(calculate_square_root(16))  # Output: 4.0
print(calculate_square_root(25))  # Output: 5.0
# Example usage:
print(calculate_square_root(9))  # Output: 3.0
print(calculate_square_root(36))  # Output: 6.0
print("All test cases passed.")
# Example usage:
```

Output:

```
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding> python -u "c:\Users\madhu\OneDrive\Desktop\AI Assistent coding\Assignment(7.5)\Wrong_module_use.py
4.0
5.0
3.0
6.0
All test cases passed.
PS C:\Users\madhu\OneDrive\Desktop\AI Assistent coding>
```

Observation:

1. The code tries to import a module named maths, which does not exist in Python's standard library.
2. Because the module name is incorrect, Python raises an ImportError and stops program execution.
3. Python module names must match exactly, and the correct built-in module name is math.
4. Importing the correct module provides access to mathematical functions like sqrt().
5. Updating the statement to import math resolves the error and allows the function to run properly.
6. After fixing, the program successfully calculates and prints square root values without any import issues.