AI Assisted Coding

Lab Assignment 7.5

Name : V Sai Krishnachaithanya

Hall Ticket no : 2303A52154

Task 1 (Mutable Default Argument – Function Bug)

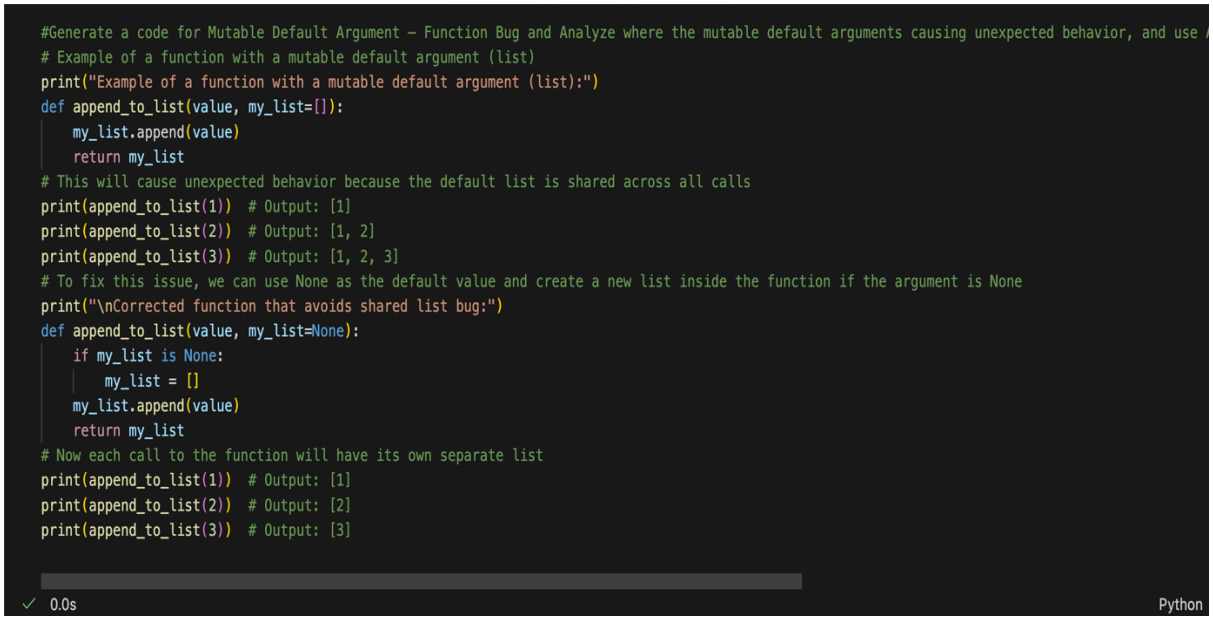
Task: Analyze given code where a mutable default argument causes unexpected behavior. Use AI to fix it.

```
# 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.

Prompt :

Generate a code for Mutable Default Argument – Function Bug and Analyze where the mutable default arguments causing unexpected behavior, and use AI to fix it And print the corrected function that avoids shared list bug



```
#Generate a code for Mutable Default Argument – Function Bug and Analyze where the mutable default arguments causing unexpected behavior, and use A
# Example of a function with a mutable default argument (list)
print("Example of a function with a mutable default argument (list):")
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list
# This will cause unexpected behavior because the default list is shared across all calls
print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [1, 2]
print(append_to_list(3))  # Output: [1, 2, 3]
# To fix this issue, we can use None as the default value and create a new list inside the function if the argument is None
print("\nCorrected function that avoids shared list bug:")
def append_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list
# Now each call to the function will have its own separate list
print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2]
print(append_to_list(3))  # Output: [3]

✓ 0.0s                                                                    Python
```

Output :

```
✓ 0.0s

Example of a function with a mutable default argument (list):
[1]
[1, 2]
[1, 2, 3]

Corrected function that avoids shared list bug:
[1]
[2]
[3]
```

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

Prompt :

Analyze given code where floating-point comparison fails. Use AI to correct with tolerance with clear comments and explanation

```python
# Bug: Floating point precision issue
def check_sum():
    return (0.1 + 0.2) == 0.3
print("Floating point precision issue:")
print(check_sum())


# To fix the floating point precision issue, we can use the math.isclose() function to compare the values with a tolerance
import math
def check_sum_fixed():
    return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9)
print("\nCorrected function that avoids floating point precision issue:")
print(check_sum_fixed())
```

✓ 0.0s Python

Output :

```
✓ 0.0s

Floating point precision issue:
False

Corrected function that avoids floating point precision issue:
True
```

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.

Prompt:

Analyze given code and Fix the code where recursion runs infinitely due to missing base case using AI

```
# Bug: No base case
def countdown(n):
    print(n)
    return countdown(n-1)
countdown(5)
# The above code will cause a RecursionError because there is no base case to stop the
# recursion. To fix this, we can add a base case that checks if n is less than or equal to 0, and if so, it will stop the recursion.
print("\nCorrected function that includes a base case to prevent infinite recursion:")
def countdown(n):
    if n <= 0:
        print("Countdown finished!")
        return
    print(n)
    return countdown(n-1)
countdown(5)
```

Output :

```
✓ 0.0s

Corrected function that includes a base case to prevent infinite recursion:
5
4
3
2
1
Countdown finished!
```

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.

Prompt :

Analyze given code where a missing dictionary key causes error.

```
# Bug: Accessing non-existing key
# def get_value():
#     data = {"a": 1, "b": 2}
#     return data["c"]
# print(get_value())
#correct the above code to handle the KeyError exception and return a default value instead of crashing the program
print("\nCorrected function that handles KeyError exception:")
def get_value():
    data = {"a": 1, "b": 2}
    try:
        return data["c"]
    except KeyError:
        return "Key not found"
print(get_value())
```

Output :

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.

Prompt:

Analyse the code using Corrected loop increments I where loop never ends

```
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)
#Analyse the code using Corrected loop increments I where loop never ends
print("\nCorrected function that includes loop increment to prevent infinite loop:")
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1  # Increment i to prevent infinite loop
loop_example()
```

Output :

```
✓ 0.0s

Corrected function that includes loop increment to prevent infinite loop:
0
1
2
3
4
```

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.

Prompt:

Analyze given code and Correct unpacking or using _ for extra values

```
# Bug: Wrong unpacking
#a, b = (1, 2, 3)
#Analyze given code and Correct unpacking or using _ for extra values
print("\nCorrected function that handles wrong unpacking:")
a, b, _ = (1, 2, 3)  # Using _
print(f"a: {a}, b: {b}")


✓ 0.0s
```

Output:

```
✓ 0.0s

Corrected function that handles wrong unpacking:
a: 1, b: 2
```

Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks execution. Use AI to fix it.

```
# Bug: Mixed indentation
def func():
    x = 5
       y = 10
    return x+y
```

Expected Output : Consistent indentation applied

Prompt:

Analyze given code and fix where mixed indentation breaks

```
# def func():
#     x = 5
#         y = 10
#     return x+y
#Analyze given code and fix where mixed indentation breaks
print("\nCorrected function that handles mixed indentation:")
def func():
    x = 5
    y = 10
    return x+y
func()
```

Output :

```
✓ 0.0s

Corrected function that handles mixed indentation:

15
```

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 given code and fix the libraries

```
# # Bug: Wrong import
# import maths
# print(maths.sqrt(16))
#Analyze given code and fix the libraries
print("\nCorrected function that handles wrong import:")
import math
print(math.sqrt(16))
```

Output :

```
✓ 0.0s

Corrected function that handles wrong import:
4.0
```

Task 9 (Unreachable Code – Return Inside Loop)

Task: Analyze given code where a return inside a loop prevents full iteration. Use AI to fix it.

```
# Bug: Early return inside loop
 def total(numbers):
     for n in numbers:
         return n
 print(total([1,2,3]))
```

Expected Output: Corrected code accumulates sum and returns after loop.

Prompt:

Analyze given code and fix the loop and return sum

```
# Bug: Early return inside loop
def total(numbers):
    for n in numbers:
        return n
print("Total sum of numbers with early return (incorrect):")
print(total([1,2,3]))
#Analyze given code and fix the loop and return sum
print("\nCorrected function that calculates the total sum of numbers:")
def total(numbers):
    total_sum = 0
    for n in numbers:
        total_sum += n
    return total_sum
print(total([1,2,3]))
```

✓ 0.0s

Output:

✓ 0.0s

```
Total sum of numbers with early return (incorrect):
1

Corrected function that calculates the total sum of numbers:
6
```

Task 10 (Name Error – Undefined Variable)

Task: Analyze given code where a variable is used before being defined. Let AI detect and fix the error.

# Bug: Using undefined variable
def calculate_area():
    return length * width
print(calculate_area())

Requirements:

- Run the code to observe the error.
- Ask AI to identify the missing variable definition.
- Fix the bug by defining length and width as parameters.
- Add 3 assert test cases for correctness.

Expected Output :

- Corrected code with parameters.
- AI explanation of the bug.

Successful execution of assertions.

Prompt : Analyze given code and define variables length and width before using them

```
# Bug: Using undefined variable
# def calculate_area():
#     return length * width
# print(calculate_area())
#Analyze given code and define variables length and width before using them
print("\nCorrected function that defines length and width variables:")
def calculate_area(length, width):
    return length * width
# Test cases
print("\nTest cases for calculate_area function:")
print(calculate_area(5,3))
print(calculate_area(10,2))
print(calculate_area(7,4))
```

Output:

```
Corrected function that defines length and width variables:

Test cases for calculate_area function:
15
20
28
```

Task 11 (Type Error – Mixing Data Types Incorrectly)

Task: Analyze given code where integers and strings are added incorrectly. Let AI detect and fix the error.

```
# Bug: Adding integer and string
def add_values():
    return 5 + "10"
print(add_values())
```

Requirements:

- Run the code to observe the error.
- AI should explain why int + str is invalid.
- Fix the code by type conversion (e.g., int("10") or str(5)).
- Verify with 3 assert cases.

Expected Output #6:

- Corrected code with type handling.
- AI explanation of the fix.

Successful test validation.

Prompt: Analyze given code and fix the type error by converting string to integer before addition.

```
# # Bug: Adding integer and string
# def add_values():
#     return 5 + "10"
# print(add_values())
#Analyze given code and fix the type error by converting string to integer before addition
print("\nCorrected function that handles type error by converting string to integer:")
def add_values():
    return 5 + int("10")
print(add_values())
```

Output :

```
✓ 0.0s

Corrected function that handles type error by converting string to integer:
15
```

Task 12 (Type Error – String + List Concatenation)

Task: Analyze code where a string is incorrectly added to a list.
# Bug: Adding string and list
def combine():
    return "Numbers: " + [1, 2, 3]
print(combine())
Requirements:

- Run the code to observe the error.
- Explain why str + list is invalid.
- Fix using conversion (str([1,2,3]) or " ".join()).
- Verify with 3 assert cases.

Expected Output:

- Corrected code
- Explanation
- Successful test validation

Prompt:

Analyze given code and fix the type error by converting list to string before concatenation.

```
# Bug: Adding string and list
# def combine():
#     return "Numbers: " + [1, 2, 3]
# print(combine())
#Analyze given code and fix the type error by converting list to string before concatenation
print("\nCorrected function that handles type error by converting list to string:")
def combine():
    return "Numbers: " + str([1, 2, 3])
print(combine())
```

Output :

```
✓ 0.0s

Corrected function that handles type error by converting list to string:
Numbers: [1, 2, 3]
```

Task 13 (Type Error – Multiplying String by Float)

Task: Detect and fix code where a string is multiplied by a float.
# Bug: Multiplying string by float
def repeat_text():
    return "Hello" * 2.5
print(repeat_text())
Requirements:

- Observe the error.
- Explain why float multiplication is invalid for strings.
- Fix by converting float to int.
- Add 3 assert test cases.

Prompt:

Analyze given code and fix the type error by converting float to integer before multiplication.

```
# Bug: Multiplying string by float
# def repeat_text():
#     return "Hello" * 2.5
# print(repeat_text())
#Analyze given code and fix the type error by converting float to integer before multiplication
print("\nCorrected function that handles type error by converting float to integer:")
def repeat_text():
    return "Hello" * int(2.5)
print(repeat_text())
```

Output:

```
✓ 0.0s

Corrected function that handles type error by converting float to integer:
HelloHello
```

Task 14 (Type Error – Adding None to Integer)

Task: Analyze code where None is added to an integer.

```
# Bug: Adding None and integer
def compute():
    value = None
    return value + 10
print(compute())
```

Requirements:

- Run and identify the error.
- Explain why NoneType cannot be added.
- Fix by assigning a default value.
- Validate using asserts.

Prompt :

Analyze given code and fix the type error by checking if value is None before addition.

```
# # Bug: Adding None and integer
# def compute():
#     value = None
#     return value + 10

# print(compute())
#Analyze given code and fix the type error by checking if value is None before addition
print("\nCorrected function that handles type error by checking for None value:")
def compute():
    value = None
    if value is None:
        return "Value is None, cannot perform addition"
    return value + 10
print(compute())
```

Output :

```
✓ 0.0s

Corrected function that handles type error by checking for None value:
Value is None, cannot perform addition
```

Task 15 (Type Error – Input Treated as String Instead of Number)

Task: Fix code where user input is not converted properly.

```
# Bug: Input remains string
def sum_two_numbers():
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    return a + b
print(sum_two_numbers())
```

Requirements:

- Explain why input is always string.
- Fix using int() conversion.
- Verify with assert test cases.

Prompt :

Analyze given code and fix the type error by converting input strings to integers before addition.

```
# # Bug: Input remains string
# def sum_two_numbers():
#     a = input("Enter first number: ")
#     b = input("Enter second number: ")
#     return a + b
# print(sum_two_numbers())
#Analyze given code and fix the type error by converting input strings to integers before addition
print("\nCorrected function that handles type error by converting input strings to integers:")
def sum_two_numbers():
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    return int(a) + int(b)
print(sum_two_numbers())
```

Output :

```
✓ 1.9s

Corrected function that handles type error by converting input strings to integers:
55
```