

LAB ASSIGNMNET 7.5

Subject : AI Assistant Coding

Hall Ticket No : 2303A52465

Name : M.Avinash

Batch : 50

Date : 6/2/26

Task-1 :

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

```
# RollNo : 2303A52465
# Buggy function with mutable default argument
def add_to_list(item, items=[]):
    items.append(item)
    return items

# Demonstrate unexpected behavior
print("Buggy calls:")
print(add_to_list(1)) # expected [1]
print(add_to_list(2)) # unexpected [1, 2] - list is shared across calls
print(add_to_list(3, [])) # explicit new list works -> [3]

# Analysis (concise)
print("\nAnalysis: The default list 'items' is created once at function definition time and reused across calls, causing state to leak between calls.")

# AI-fixed function avoiding shared-list bug
def add_to_list_fixed(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

# Demonstrate correct behavior
print("\nFixed calls:")
print(add_to_list_fixed(1)) # [1]
print(add_to_list_fixed(2)) # [2] - no leakage
print(add_to_list_fixed(3, [])) # [3] - still supports explicit lists

# Analysis of fixed function
print("\nAnalysis of fixed function: By using 'None' as the default value and creating a new list inside the function, we ensure that each call to 'add_to_list_fixed' gets its own separate list, preventing unintended side effects.")
```

Output :

```
Bug calls:
[1]
[1, 2]
[3]

Analysis: The default list 'items' is created once at function definition time and reused across calls, causing state to leak between calls.

Fixed calls:
[1]
[2]
[3]

Analysis of fixed function: By using 'None' as the default value and creating a new list inside the function, we ensure that each call to 'add_to_list_fixed' gets its own separate list, preventing unintended side effects.
```

Justification : This task explains a common Python pitfall involving mutable default arguments, which can lead to unexpected behavior. It demonstrates the issue with a buggy function, provides an analysis of the problem, and then shows how to fix it using a more appropriate default value. This is crucial for writing robust and bug-free code in Python.

Task-2 :

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

```
# RollNo : 2303A52465
# Bug: Floating point precision issue
print("\nOriginal function:")
def check_sum():
    return (0.1 + 0.2) == 0.3
print(check_sum())
Analysis="The expression (0.1 + 0.2) does not exactly equal 0.3 due to the way floating-point numbers are represented in binary, leading to a precision issue."
print("\nAnalysis:", Analysis)
print("\nfixed function:")
# Analyze given code where floating-point comparison fails. Use AI to correct with tolerance with clear comments and explanation
def check_sum_fixed():
    # Use a tolerance to account for floating-point precision issues
    return abs((0.1 + 0.2) - 0.3) < 1e-9
print(check_sum_fixed())
analysis_for_fixed = """By using a small tolerance (1e-9), we can check if the difference between (0.1 + 0.2) and 0.3 is negligible,
effectively bypassing the precision issue and confirming that they are "close enough" to be considered equal."""
print("\nAnalysis of fixed function:", analysis_for_fixed)
```

Output :

```
Original function:
False

Analysis: The expression (0.1 + 0.2) does not exactly equal 0.3 due to the way floating-point numbers are represented in binary, leading to a precision issue.

Fixed function:
True

Analysis of fixed function: By using a small tolerance (1e-9), we can check if the difference between (0.1 + 0.2) and 0.3 is negligible,
effectively bypassing the precision issue and confirming that they are "close enough" to be considered equal.
```

Justification : This task highlights a common and subtle bug in Python related to mutable default arguments, which can lead to unexpected behavior. It demonstrates how to analyze and fix such bugs using AI, improving code reliability and maintainability. Understanding and addressing this issue is crucial for writing robust Python code, especially in functions that may be called multiple times or in different contexts.

Task-3 :

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)
```

```
# Analyze given code and Fix the code where recursion runs infinitely due to missing base case using AI.
def countdown_fixed(n):
    if n <= 0: # Base case to stop recursion
        print("Liftoff!")
    else:
        print(n)
        return countdown_fixed(n-1)
countdown_fixed(5)

print("\nAnalysis of fixed function: By adding a base case that checks if 'n' is less than or equal to 0, we prevent infinite recursion and allow the function to terminate properly.")
```

Output :

```
5
4
3
2
1
Liftoff!
Analysis of fixed function: By adding a base case that checks if 'n' is less than or equal to 0, we prevent infinite recursion and allow the function to terminate properly.
```

Justification : This task explains a common pitfall in Python where mutable default arguments can lead to unexpected behavior due to shared state across function calls. It also addresses a fundamental issue with floating-point arithmetic, demonstrating how to handle precision issues effectively. Finally, it reinforces the importance of base cases in recursive functions to prevent infinite recursion, which is crucial for writing robust code.

Task-4 :

Prompt : Analyze given code and Fix the code where a missing dictionary key causes error using AI

```
# Roll no: 2303A52465
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]
print(get_value())

# Analyze given code and Fix the code where a missing dictionary key causes error| using AI
→ def get_value_fixed():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found") # Return a default value if key doesn't exist
print(get_value_fixed())
```

Output :

```
Original function:

Analysis: Accessing a non-existing key 'c' in the dictionary raises a KeyError.

Fixed function:
Key not found

Analysis of fixed function: By using the get() method with a default value, we avoid KeyError and provide a graceful fallback when the key does not exist.
```

Task-5 :

Prompt : Analyze given code and Fix the code where loop never ends due to missing increment using AI

```
# Roll-No : 2303A52465
print("\nOriginal function:")
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)
print(loop_example())
print("\nAnalysis:", "The loop runs infinitely because 'i' is never incremented, so the condition 'i < 5' is always true.")
# Analyze given code and Fix the code where loop never ends due to missing increment using AI
print("\nFixed function:")
def loop_example_fixed():
    i = 0
    while i < 5:
        print(i)
        i += 1 # Increment 'i' to eventually terminate the loop
    return "Loop ended"
print(loop_example_fixed())
print("\nAnalysis of fixed function: By adding 'i += 1' inside the loop, we ensure that 'i' eventually reaches 5, allowing the loop to terminate as intended.")
```

Output :

```
Original function:

Analysis: The loop runs infinitely because 'i' is never incremented, so the condition 'i < 5' is always true.

Fixed function:
0
1
2
3
4
Loop ended

Analysis of fixed function: By adding 'i += 1' inside the loop, we ensure that 'i' eventually reaches 5, allowing the loop to terminate as intended.
```

Task-6 :

Prompt : Analyze given code and Fix the code where tuple unpacking fails due to mismatched number of values using AI

```
# Roll-No : 2303A52465
print("\nOriginal function:")
# Bug: Wrong unpacking
a, b = (1, 2, 3)
print(a, b)
print("\nAnalysis:", "The unpacking fails because there are more values (3) than variables (2) to assign them to, causing a ValueError.")
# Analyze given code and Fix the code where tuple unpacking fails due to mismatched number of values using AI
print("\nFixed function:")
a, b, c = (1, 2, 3) # Match number of variables to values
print(a, b, c)
print("\nAnalysis:", "By matching the number of variables (3) to the number of values (3), we ensure successful unpacking without errors.")
```

Output :

```
Original function:

Analysis: The unpacking fails because there are more values (3) than variables (2) to assign them to, causing a ValueError.

Fixed function:
1 2 3

Analysis: By matching the number of variables (3) to the number of values (3), we ensure successful unpacking without errors.
```

Task-7 :

Prompt : Analyze given code and Fix the code where mixed indentation causes error using AI

```
# Roll=No : 2303A52465
print("\nOriginal function:")
# Bug: Mixed indentation
def func():
    x = 5
    ~~~~~y = 10
    return x+y
print(func())
print("\nAnalysis:", "The function has mixed indentation (spaces and tabs), leading to an IndentationError.")
# Analyze given code and Fix the code where mixed indentation causes error using AI
print("\nFixed function:")
def func_fixed():
    x = 5
    y = 10
    return x+y
print(func_fixed())
print("\nAnalysis of fixed function: By ensuring consistent indentation (using only spaces), we eliminate the IndentationError and allow the function to execute correctly.")
```

Output :

```
Original function:
Analysis: The function has mixed indentation (spaces and tabs), leading to an IndentationError.

Fixed function:
15

Analysis of fixed function: By ensuring consistent indentation (using only spaces), we eliminate the IndentationError and allow the function to execute correctly
```

Task-8 :

Prompt : Analyze given code and Fix the code where wrong module is imported using AI

```
# Roll=No : 2303A52465
print("\nOriginal function:")
# Bug: Wrong import
import ~maths
print(maths.sqrt(16))
print("\nAnalysis:", "The module 'maths' does not exist; the correct module name is 'math', leading to a ModuleNotFoundError.")
# Analyze given code and Fix the code where wrong module is imported using AI
print("\nFixed function:")
import math
print(math.sqrt(16))
print("\nAnalysis of fixed function: By importing the correct module 'math', we can successfully use its functions without errors.")
```

Output :

```
Original function:
Analysis: The module 'maths' does not exist; the correct module name is 'math', leading to a ModuleNotFoundError.

Fixed function:
4.0

Analysis of fixed function: By importing the correct module 'math', we can successfully use its functions without errors.
```

Task-9 :

Prompt : Analyze given code and Fix the code where early return causes function to exit prematurely using AI

```
# Roll-No : 2303A52465
print("\nOriginal function:")
# Bug: Early return inside loop
def total(numbers):
    for n in numbers:
        return n
print(total([1,2,3]))
print("\nAnalysis: "The function returns after processing the first number due to the early return statement, resulting in an incorrect total.")
# Analyze given code and Fix the code where early return causes function to exit prematurely using AI
print("\nfixed function:")
def total_fixed(numbers):
    total_sum = 0
    for n in numbers:
        total_sum += n # Accumulate the sum instead of returning early
    return total_sum
print(total_fixed([1,2,3]))
print("\nAnalysis of fixed function: By accumulating the sum in a variable and returning it after the loop, we ensure that all numbers are processed and the correct total is returned.")
```

Output :

```
Original function:
1

Analysis: The function returns after processing the first number due to the early return statement, resulting in an incorrect total.

Fixed function:
6

Analysis of fixed function: By accumulating the sum in a variable and returning it after the loop, we ensure that all numbers are processed and the correct total is returned.
```

Task-10 :

Prompt : Analyze given code and Fix the code where a variable is used before being defined causing error using AI

```
# Roll-No : 2303A52465
print("\nOriginal function:")
# Bug: Using undefined variable
def calculate_area():
    return length * width
print(calculate_area())
print("\nAnalysis: "The variables 'length' and 'width' are not defined within the function, leading to a NameError.")

# Analyze given code and Fix the code where a variable is used before being defined causing error using AI
print("\nfixed function:")
def calculate_area_fixed(length, width):
    return length * width
print(calculate_area_fixed(5, 10))
print("\nAnalysis of fixed function: By defining 'length' and 'width' as parameters of the function, we ensure that they are provided when the function is called, allowing it to execute without error")
```

Output :

```
Original function:

Analysis: The variables 'length' and 'width' are not defined within the function, leading to a NameError.

Fixed function:
50

Analysis of fixed function: By defining 'length' and 'width' as parameters of the function, we ensure that they are provided when the function is called, allowing it to execute without errors.
```

Task-11 :

Prompt : Analyze given code and Fix the code where integers and strings are added incorrectly causes error using AI and verify with 3 test cases

```
# Roll=No : 2303A52465
print("\nOriginal function:")
# Bug: Adding integer and string
def add_values():
    return 5 + "10"
print("\nAnalysis:", "Adding an integer (5) and a string ('10') raises a TypeError because they are incompatible types for addition.")

# Analyze given code and Fix the code where integers and strings are added incorrectly causes error using AI and verify with 3 test cases
print("\nfixed function:")
def add_values_fixed(a, b):
    return str(a) + str(b) # Convert both to strings before concatenation
print(add_values_fixed(5, "10")) # "510"
print(add_values_fixed(3, 7)) # "37"
print(add_values_fixed("Hello, ", "world!")) # "Hello, world!"
print("\nAnalysis of fixed function: By converting both inputs to strings before concatenation, we can successfully combine them without type errors, allowing for flexible input types.")
```

Output :

```
Original function:
Analysis: Adding an integer (5) and a string ('10') raises a TypeError because they are incompatible types for addition.

Fixed function:
510
37
Hello, world!

Analysis of fixed function: By converting both inputs to strings before concatenation, we can successfully combine them without type errors, allowing for flexible input types.
```

Task-12 :

Prompt : Analyze given code and Fix the code where string and list are added incorrectly causes error using AI and verify with 3 test cases

```
# Roll=No : 2303A52465
print("\nOriginal function:")
# Bug: Adding string and list
def combine():
    return "Numbers: " + [1, 2, 3]
print([combine()])
print("\nAnalysis:", "Adding a string and a list raises a TypeError because they are incompatible types for addition.")

# Analyze given code and Fix the code where string and list are added incorrectly causes error using AI and verify with 3 test cases
print("\nFixed function:")
def combine_fixed():
    return "Numbers: " + str([1, 2, 3]) # Convert list to string before concatenation
print(combine_fixed()) # "Numbers: [1, 2, 3]"
print("Numbers: " + str([4, 5, 6])) # "Numbers: [4, 5, 6]"
print("Numbers: " + str([])) # "Numbers: []"
print("\nAnalysis of fixed function: By converting the list to a string before concatenation, we can successfully combine it with the string without type errors, allowing for flexible input t
```

Output :

```
Original function:
Analysis: Adding a string and a list raises a TypeError because they are incompatible types for addition.

Fixed function:
Numbers: [1, 2, 3]
Numbers: [4, 5, 6]
Numbers: []

Analysis of fixed function: By converting the list to a string before concatenation, we can successfully combine it with the string without type errors, allowing for flexible input types.
```

Task-13 :

Prompt : Analyze given code and Fix the code where string is multiplied by a float causing error using AI and verify with 3 test cases

```
# RollNo : 2303A52465
print("\nOriginal function:")

# Bug: Multiplying string by float
def repeat_text():
    return "Hello" * 2.5
print([repeat_text()])
print("\nAnalysis:", "Multiplying a string by a float raises a TypeError because the repetition operator expects an integer.")

# Analyze given code and Fix the code where string is multiplied by a float causing error using AI and verify with 3 test cases
print("\nFixed function:")
def repeat_text_fixed(times):
    if isinstance(times, float):
        times = int(times) # Convert float to integer for repetition
    return "Hello" * times
print(repeat_text_fixed(2.5)) # "HelloHello"
print(repeat_text_fixed(3)) # "HelloHelloHello"
print(repeat_text_fixed(0)) # ""
print("\nAnalysis of fixed function: By checking if the input is a float and converting it to an integer, we can allow for flexible input while ensuring that the string repetition works correctly without type errors.")
```

Output :

```
Original function:

Analysis: Multiplying a string by a float raises a TypeError because the repetition operator expects an integer.

Fixed function:
HelloHello
HelloHelloHello

Analysis of fixed function: By checking if the input is a float and converting it to an integer, we can allow for flexible input while ensuring that the string repetition works correctly without type errors.
```

Task-14 :

Prompt : Analyze given code and Fix the code where None is added to an integer causing error using AI and verify with 3 test cases

```
# RollNo : 2303A57465
print("\nOriginal function:")
# Bug: Adding None and integer
def compute():
    value = None
    return value + 10
print(compute())
print("\nAnalysis:", "Adding 'None' and an integer raises a TypeError because 'None' is not a valid operand for addition.")

# Analyze given code and Fix the code where None is added to an integer causing error using AI and verify with 3 test cases
print("\nFixed function:")
def compute_fixed(value):
    if value is None:
        value = 0 # Default to 0 if None is provided
    return value + 10
print(compute_fixed(None)) # 10
print(compute_fixed(5)) # 15
print(compute_fixed(-3)) # 7
print("\nAnalysis of fixed function: By checking if the input value is 'None' and assigning it a default value (0), we can ensure that the function operates correctly without raising errors, while still allowing for flexible input.")
```

Output :

```
Original function:

Analysis: Adding 'None' and an integer raises a TypeError because 'None' is not a valid operand for addition.

Fixed function:
10
15
7

Analysis of fixed function: By checking if the input value is 'None' and assigning it a default value (0), we can ensure that the function operates correctly without raising errors, while still allowing for flexible input.
```

Task-15 :

Prompt : Analyze given code and Fix the code where integers are added as strings causing error using AI and verify with 3 test cases

```
# RollNo : 2303AS2465
print("\nOriginal function:")
# 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())
print("\nAnalysis:", "The input function returns a string, so adding 'a' and 'b' concatenates them instead of performing numeric addition.")

# Analyze given code and Fix the code where integers are added as strings causing error using AI and verify with 3 test cases
print("\nFixed function:")
def sum_two_numbers_fixed():
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    try:
        return float(a) + float(b) # Convert inputs to floats for numeric addition
    except ValueError:
        return "Invalid input. Please enter numeric values."
print(sum_two_numbers_fixed()) # Test with user input
print(sum_two_numbers_fixed()) # Test with user input
print(sum_two_numbers_fixed()) # Test with user input
print("\nAnalysis of fixed function: By converting the input strings to floats, we can perform numeric addition as intended. The try-except block also handles cases where the input cannot be converted to a number")
```

Output :

```
Original function:
50100

Analysis: The input function returns a string, so adding 'a' and 'b' concatenates them instead of performing numeric addition.

Fixed function:
150.0
450.0
1054.0

Analysis of fixed function: By converting the input strings to floats, we can perform numeric addition as intended. The try-except block also handles cases where the input cannot be converted to a number
```