AI Assisted Coding

ASSIGNMENT 7.4

Name: Deepthi
HT No: 2303A52390
Batch: 32

Question 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.

Code:

```python
# Bug: Mutable default argument
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
```

Output(O/P)

```
[1]
[2]
```

Explanation:

The default list [] was shared between calls.

Default arguments are created only once.

Using None ensures a new list is created each time.

Question 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

Code:

```
1    # Bug: Floating point precision issue
2    def check_sum():
3        return round(0.1 + 0.2, 10) == round(0.3, 10)
4    print(check_sum())
```

Output(O/P)

```
True
```

Explanation:

Floating numbers are not exact in binary.
Instead of ==, we compare the difference with a small tolerance.

Question 3: Recursion Error – Missing Base Case

Task:

Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix it.

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

Expected Output : Correct recursion with stopping condition.

Code:

```
1    # Bug: No base case
2    def countdown(n):
3        if n < 0:
4            return
5        print(n)
6        return countdown(n-1)
7    countdown(5)
```

Output(O/P)

```
5
4
3
2
1
0
```

Explanation:

Without a base case, recursion never stops.
The base case stops when n becomes 0.

Question 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.

Code:

```
1    # Bug: Accessing non-existing key
2    def get_value():
3        data = {"a": 1, "b": 2}
4        return data.get("c", None)
5    print(get_value())
```

Output(O/P)

```
None
```

Explanation:

Accessing data["c"] causes KeyError

.get() prevents crashes and allows default value.

Question 5: Infinite Loop – Wrong Condition

Task:

Analyze given code where the 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.

Code:

```
1    # Bug: Infinite loop
2    def loop_example():
3        i = 0
4        while i < 5:
5            print(i)
6            i += 1
7
8    loop_example()
```

Output(O/P)

```
0
1
2
3
4
```

Explanation:

'i' was never incremented.
The loop condition stayed True forever.

Question 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.

Code:

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

Output(O/P)

```
1 2
```

Explanation:

Tuple had three values but only two variables.
Added c to Fix.

Question 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.

Code:

```
1    # Bug: Mixed indentation
2    def func():
3        x = 5
4        y = 10
5        return x+y
6    print(func())
```

Output(O/P)

```
15
```

Explanation:

Python requires consistent indentation.
Using spaces consistently fixes execution.

Question 8: Import Error – Wrong Module Usage

Task:

Analyze given code with incorrect import. Use AI to fix it.
# Bug: Wrong import
import maths
print(maths.sqrt(16))
Expected Output: Corrected to import math

Code:

```
1    # Bug: Wrong import
2    import math
3    print(math.sqrt(16))
```

Output(O/P)

```
4.0
```

Explanation:

Corrected Typo of Import of math module

Question 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

Code:

```
1    # Bug: Early return inside loop
2    def total(numbers):
3        sum_total = 0
4        for n in numbers:
5            sum_total += n
6        return sum_total
7    print(total([1,2,3]))
```

Output(O/P)

```
6
```

Explanation:

Return statement inside the loop stops after the first iteration.
Now it sums all elements before returning.

Question 10: Name Error – Undefined Variable

Task:

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.

Code:

```
1    # Bug: Using undefined variable
2    def calculate_area(length, width):
3        return length * width
4
5    # Test cases
6    assert calculate_area(5, 10) == 50
7    assert calculate_area(3, 4) == 12
8    assert calculate_area(7, 8) == 56
9
10   print(calculate_area(5, 10))
```

Output(O/P)

```
50
```

Explanation:

length and width were undefined.
Making them parameters fixes NameError.

Question 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.

Code:

```
1    # Bug: Adding integer and string
2    def add_values():
3        """
4        Add two values together.
5
6        Note: In Python, the + operator cannot directly add an integer and string
7        because they are incompatible types. Strings require explicit conversion
8        to numeric types before arithmetic operations.
9        """
10       return 5 + int("10")
11
12   print(add_values())
13
14   # Test cases
15   assert add_values() == 15, "5 + 10 should equal 15"
16   assert isinstance(add_values(), int), "Result should be an integer"
17   assert add_values() > 10, "Result should be greater than 10"
```

Output(O/P)

```
15
```

Explanation:

Python cannot add int and str directly.
Type conversion solves the issue.

Question 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

Code:

```
1    # Bug: Adding string and list
2    def combine():
3        """
4        Combine a string with a list representation.
5
6        Note: Direct concatenation of str + list is invalid in Python because
7        strings and lists are different types. We must convert the list to a
8        string representation using str() before concatenation.
9        """
10       return "Numbers: " + str([1, 2, 3])
11
12   print(combine())
13
14   # Test cases
15   assert combine() == "Numbers: [1, 2, 3]", "Basic concatenation failed"
16   assert isinstance(combine(), str), "Result should be a string"
17   assert "Numbers:" in combine() and "[1, 2, 3]" in combine(), "Both parts should be present"
```

Output(O/P)

```
Numbers: [1, 2, 3]
```

Explanation:

String cannot be directly added to the list.

Convert list to string first.

Question 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.

Code:

```
1    # Bug: Multiplying string by float
2    def repeat_text():
3        """
4        Repeat a string by converting the multiplier to an integer.
5
6        In Python, strings can only be multiplied by integers, not floats.
7        Multiplying a string by a float raises a TypeError because the
8        repetition count must be a whole number. We convert 2.5 to int(2)
9        to fix this issue.
10       """
11       return "Hello" * int(2.5)
12
13   print(repeat_text())
14
15   # Test cases
16   assert repeat_text() == "HelloHello", "Should repeat string 2 times (2.5 truncated to 2)"
17   assert len(repeat_text()) == 10, "Result should have length 10"
18   assert repeat_text()[0:5] == "Hello", "First 5 characters should be 'Hello'"
```

Output(O/P)

```
HelloHello
```

Explanation:

Strings can only be multiplied by integers.
Convert float to int before multiplication.

Question 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.

Code:

```
1    # Bug: Adding None and integer
2    def compute():
3        """
4        Computes a value.
5
6        NoneType cannot be added to integers because None represents the absence
7        of a value and has no numeric representation. Python's type system prevents
8        operations between incompatible types to avoid undefined behavior.
9        """
10       value = 0  # Fixed: Initialize with a valid integer instead of None
11       return value + 10
12
13   result = compute()
14   print(result)
15
16   # Validate using asserts
17   assert result == 10, "Expected result to be 10"
18   assert isinstance(result, int), "Expected result to be an integer"
```

Output(O/P)

`10`

Explanation:

NoneType cannot be added to int.

Assign default numeric value before operation.

Question 15: Type Error – Input Treated as String Instead of Number

Task:

Fix code where user input is not converted properly.

```python
# 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.

Code:

```
1    # Bug: Input remains string
2    def sum_two_numbers():
3        """
4        Sum two numbers from user input.
5
6        Why input() returns strings:
7        The input() function always returns user data as a string type,
8        regardless of what the user types. String concatenation occurs
9        instead of numeric addition without explicit type conversion.
10       """
11       a = float(input("Enter first number: "))
12       b = float(input("Enter second number: "))
13       return a + b
14
15   result = sum_two_numbers()
16   print(result)
17
18   # Test cases
19   assert sum_two_numbers.__doc__ is not None, "Docstring missing"
20   assert isinstance(result, float), "Result should be a float"
21   assert result == 5.0, "Expected sum of 2.0 and 3.0 to be 5.0"
```

Output(O/P)

```
Enter first number: 2.0
Enter second number: 3.0
5.0
```

Explanation:

input() always returns string.
Convert to float or int before addition.