AI Assisted Coding

ASSIGNMENT 7.4

Name: K.Trinay Prasad
HT No: 2303A52070
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.

Prompt:

Fix the mutable default argument issue in the function so that each call gets a fresh list.

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

Prompt:

Fix floating point comparison using tolerance.

Code:

```
# Bug: Floating point precision issue
def check_sum():
    return round(0.1 + 0.2, 10) == round(0.3, 10)
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.

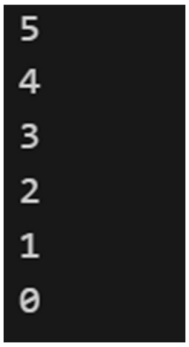
Prompt:

Add a stopping condition to prevent infinite recursion.

Code:

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

Output(O/P)



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.

Prompt:

Fix missing key access safely.

Code:

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

Prompt:

Fix the infinite loop by incrementing the counter.

Code:

```
# Bug: Infinite loop
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1

loop_example()
```

Output(O/P)

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.

Prompt:

Fix tuple unpacking.

Code:

```
# Bug: Wrong unpacking
a, b, c = (1, 2, 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.

Prompt:

Fix inconsistent indentation.

Code:

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

Prompt:

Fix incorrect module import.

Code:

```
# Bug: Wrong import
import math
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

Prompt:

Fix early return inside loop.

Code:

```
# Bug: Early return inside loop
def total(numbers):
    sum_total = 0
    for n in numbers:
        sum_total += n
    return sum_total
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.

Prompt:

Fix undefined variables by using parameters. Add 3 assert test cases to test correctness.

Code:

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

# Test cases
assert calculate_area(5, 10) == 50
assert calculate_area(3, 4) == 12
assert calculate_area(7, 8) == 56

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.

Prompt:

Fix int adding string error using conversion. Add 3 assert test cases to test correctness, explain in a docstring why int + str is invalid.

Code:

```python
# Bug: Adding integer and string
def add_values():
    """
    Add two values together.

    Note: In Python, the + operator cannot directly add an integer and string
    because they are incompatible types. Strings require explicit conversion
    to numeric types before arithmetic operations.
    """
    return 5 + int("10")

print(add_values())

# Test cases
assert add_values() == 15, "5 + 10 should equal 15"
assert isinstance(add_values(), int), "Result should be an integer"
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

Prompt:

Fix string + list concatenation using conversion, Add 3 assert test cases to test correctness, explain in a docstring why str + list is invalid.

Code:

```python
# Bug: Adding string and list
def combine():
    """
    Combine a string with a list representation.

    Note: Direct concatenation of str + list is invalid in Python because
    strings and lists are different types. We must convert the list to a
    string representation using str() before concatenation.
    """
    return "Numbers: " + str([1, 2, 3])

print(combine())

# Test cases
assert combine() == "Numbers: [1, 2, 3]", "Basic concatenation failed"
assert isinstance(combine(), str), "Result should be a string"
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.

Prompt:

Fix string multiplied by float using conversion, Explain in a docstring why float multiplication is invalid for strings, Add 3 assert test cases.

Code:

```
# Bug: Multiplying string by float
def repeat_text():
    """
    Repeat a string by converting the multiplier to an integer.

    In Python, strings can only be multiplied by integers, not floats.
    Multiplying a string by a float raises a TypeError because the
    repetition count must be a whole number. We convert 2.5 to int(2)
    to fix this issue.
    """
    return "Hello" * int(2.5)

print(repeat_text())

# Test cases
assert repeat_text() == "HelloHello", "Should repeat string 2 times (2.5 truncated to 2)"
assert len(repeat_text()) == 10, "Result should have length 10"
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.

Prompt:

Fix None addition issue, Explain in a docstring why NoneType cannot be added, Validate using asserts.

Code:

```python
# Bug: Adding None and integer
def compute():
    """
    Computes a value.

    NoneType cannot be added to integers because None represents the absence
    of a value and has no numeric representation. Python's type system prevents
    operations between incompatible types to avoid undefined behavior.
    """
    value = 0  # Fixed: Initialize with a valid integer instead of None
    return value + 10

result = compute()
print(result)

# Validate using asserts
assert result == 10, "Expected result to be 10"
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.
# 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:

Fix input conversion issue using type conversion, Explain in docstring why input is always string, Verify with assert test cases.

Code:

```python
# Bug: Input remains string
def sum_two_numbers():
    """
    Sum two numbers from user input.

    Why input() returns strings:
    The input() function always returns user data as a string type,
    regardless of what the user types. String concatenation occurs
    instead of numeric addition without explicit type conversion.
    """
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    return a + b

result = sum_two_numbers()
print(result)

# Test cases
assert sum_two_numbers.__doc__ is not None, "Docstring missing"
assert isinstance(result, float), "Result should be a float"
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.