AI AC Assignment 7.4

Name : P. Thrishank

Hall-Tiket No. 2303A52013

Task 1: Mutable Default Argument – Function Bug

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

Buggy Code:

```
# Bug: Mutable default argument
def add_item(item, items=[]):
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
```

Corrected Version:

```
# Fixed: Use None as default and create new list
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1))
print(add_item(2))
```

Justification:

The original code uses a mutable default argument (an empty list), which is created once when the function is defined and shared across all function calls. This causes items to persist between calls, leading to unexpected behavior. The fix uses None as the default and creates a new list inside the function if no argument is provided, ensuring each call gets its own list.

Task 2: Floating-Point Precision Error

Question: Analyze given code where floating-point comparison fails. Use AI to correct with tolerance.

Buggy Code:

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

Corrected Version:

```
# Fixed: Use tolerance-based comparison
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9
print(check_sum())
```

Justification:

Floating-point arithmetic in computers is not always exact due to how numbers are represented in binary. Direct equality checks can fail even when values appear equal. The fix uses a tolerance-based comparison by checking if the absolute difference is less than a very small number (epsilon), which accounts for floating-point precision errors.

Task 3: Recursion Error – Missing Base Case

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

Buggy Code:

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

Corrected Version:

```
# Fixed: Added base case
def countdown(n):
    if n <= 0:
        return
    print(n)
    return countdown(n-1)
countdown(5)
```

Justification:

Recursive functions must have a base case to stop recursion, otherwise they run infinitely and cause a stack overflow. The fix adds a condition to check if n is less than or equal to 0, at which point the function returns without making another recursive call. This ensures the recursion terminates properly.

Task 4: Dictionary Key Error

Question: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

Buggy Code:

```
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    return data["c"]
print(get_value())
```

Corrected Version:

```
# Fixed: Use get() method with default value
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", None)
print(get_value())
```

Justification:

Accessing a dictionary key that doesn't exist using square brackets raises a KeyError. The fix uses the get() method, which returns None (or a specified default value) if the key doesn't exist, preventing the error. Alternatively, you could check if the key exists using "c" in data before accessing it.

Task 5: Infinite Loop – Wrong Condition

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

Buggy Code:

```
# Bug: Infinite loop
def loop_example():
    i = 0
```

```
    while i < 5:
        print(i)
```

Corrected Version:

```
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1
loop_example()
```

Justification:

The loop never ends because the counter variable i is never incremented, so it remains 0 and the condition i < 5 is always true. The fix adds i += 1 inside the loop to increment the counter after each iteration, ensuring the loop eventually terminates when i reaches 5.

Task 6: Unpacking Error – Wrong Variables

Question: Analyze given code where tuple unpacking fails. Use AI to fix it.

Buggy Code:

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

Corrected Version:

```
a, b, c = (1, 2, 3)
# Or if you only need first two values:
# a, b, c = (1, 2, 3)
```

Justification:

Tuple unpacking requires the number of variables on the left side to match the number of elements in the tuple. The original code tries to unpack 3 elements into 2 variables, causing a ValueError. The fix either uses 3 variables to capture all values, or uses an underscore (_) to ignore the unwanted third value.

Task 7: Mixed Indentation – Tabs vs Spaces

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

Buggy Code:

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

Corrected Version:

```
def func():
    x = 5
    y = 10
    return x+y
```

Justification:

Python does not allow mixing tabs and spaces for indentation in the same block, as it causes an IndentationError or TabError. The buggy code has inconsistent indentation where some lines use spaces and others use tabs. The fix ensures all lines use the same indentation method (4 spaces is Python's standard convention).

Task 8: Import Error – Wrong Module Usage

Question: Analyze given code with incorrect import. Use AI to fix.

Buggy Code:

```
# Bug: Wrong import
import maths
print(maths.sqrt(16))
```

Corrected Version:

```
import math
print(math.sqrt(16))
```

Justification:

The correct module name in Python's standard library is "math", not "maths". Trying to import "maths" results in a ModuleNotFoundError because no such module exists. The fix simply corrects the module name to "math", which provides mathematical functions including sqrt().

Task 9: Unreachable Code – Return Inside Loop

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

Buggy Code:

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

Corrected Version:

```
def total(numbers):
    sum_total = 0
    for n in numbers:
        sum_total += n
    return sum_total
print(total([1,2,3]))
```

Justification:

The return statement inside the loop causes the function to exit immediately after the first iteration, returning only the first element instead of processing all elements. The fix creates a variable to accumulate the sum, adds each number to it inside the loop, and returns the total only after the loop completes.

Task 10: Name Error – Undefined Variable

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

Buggy Code:

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

Corrected Version:

```
def calculate_area(length, width):
    return length * width
```

```
print(calculate_area(20, 10))
```

Justification:

The variables length and width are used in the function but never defined, causing a NameError. The fix adds them as function parameters so they can be passed when calling the function. The function call is also updated to provide the required arguments (20 and 10).

Task 11: Type Error – Mixing Data Types Incorrectly

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

Buggy Code:

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

Corrected Version:

```
def add_values():
    return 5 + int("10")
print(add_values())
```

Justification:

Python doesn't allow direct addition of different data types like integers and strings, resulting in a TypeError. The fix converts the string "10" to an integer using int() before performing the addition. Alternatively, if string concatenation was intended, you could convert the integer to a string using str(5).

Task 12: Type Error – String + List Concatenation

Question: Analyze code where a string is incorrectly added to a list.

Buggy Code:

```
def combine():
    return "Numbers: " + [1, 2, 3]
print(combine())
```

Corrected Version:

```
def combine():
    return "Numbers: " + str([1, 2, 3])
print(combine())
```

Justification:

You cannot concatenate a string with a list directly in Python, which causes a TypeError. The fix converts the list to a string representation using str() before concatenation. Alternatively, you could use string formatting like f"Numbers: {[1, 2, 3]}" for a cleaner approach.

Task 13: Type Error – Multiplying String by Float

Question: Detect and fix code where a string is multiplied by a float.

Buggy Code:

```
# Bug: Multiplying string by float
def repeat_text():
    return "Hello" * 2.5
print(repeat_text())
```

Corrected Version:

```
def repeat_text():
    return "Hello" * int(2.5)
print(repeat_text())
```

Justification:

In Python, strings can only be multiplied by integers, not floats, which causes a TypeError. The fix converts the float 2.5 to an integer using int(), which truncates it to 2. This allows the string repetition operation to work correctly, repeating "Hello" 2 times.

Task 14: Type Error – Adding None to Integer

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

Buggy Code:

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

Corrected Version:

```
def compute():
    value = 0
    return value + 10
print(compute())
```

Justification:

None is a special type in Python that represents the absence of a value and cannot be used in arithmetic operations with numbers, causing a TypeError. The fix initializes value with 0 instead of None, making it a valid integer that can be added to 10. If None is necessary, you should check for it before performing arithmetic.

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

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

Buggy Code:

```
# 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())
```

Corrected Version:

```
def sum_two_numbers():
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    return a + b
print(sum_two_numbers())
```

Justification:

Just put int() infront of input() for mathematical operation addition, as input() by default declares only string values.