2303A51449
Batch 03

LAB_7.5:

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.

CODE:

```
# Corrected function: Avoids mutable default argument bug

def add_item_fixed(item, items=None):

    if items is None:

        items = []

    items.append(item)

    return items

print("\nCorrected function:")

print(f"First call: {add_item_fixed(1)}")

print(f"Second call: {add_item_fixed(2)}")

print(f"Third call: {add_item_fixed(3)}")

# You can also pass in your own list

my_list = ['a', 'b']

print(f"Fourth call with existing list: {add_item_fixed('c', my_list)}")

print(f"Original list after call: {my_list}")
```

OUTPUT:

```
Corrected function:
```

First call: [1]

Second call: [2]

Third call: [3]

Fourth call with existing list: ['a', 'b', 'c']

Original list after call: ['a', 'b', 'c']

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

Code:
```
import math

def check_sum():
    # Using math.isclose handles the precision tolerance automatically
    return math.isclose(0.1 + 0.2, 0.3)

print(f"Corrected result: {check_sum()}")
print(f"Actual value of 0.1 + 0.2: {0.1 + 0.2}")
```

Output:
```
Corrected result: True
Actual value of 0.1 + 0.2: 0.30000000000000004
```

Task 3 (Recursion Error – Missing Base Case)

CODE:

```
# Corrected function: Includes a base case to stop recursion

def countdown_fixed(n):

    if n <= 0:  # Base case: stop when n is zero or negative

        print("Countdown finished!")
        return
    print(n)
    return countdown_fixed(n-1)


print("\nExecuting corrected countdown:")
countdown_fixed(5)
```

OUTPUT:

```
Executing corrected countdown:
5
4
3
2
1
Countdown finished!
```

Task 4 (Dictionary Key Error)

CODE:

```
# Corrected function 1: Using dict.get() method
def get_value_get(key, default_value=None):
    data = {"a": 1, "b": 2}
    # .get() returns the value for key if key is in the dictionary, else default_value
    return data.get(key, default_value)


print("\nCorrected function using .get():")
print(f"Value for 'a': {get_value_get('a')}")

print(f"Value for 'c' (default None): {get_value_get('c')}")

print(f"Value for 'd' (default 0): {get_value_get('d', 0)}")
```

OUTPUT:

```
Corrected function using .get():

Value for 'a': 1

Value for 'c' (default None): None

Value for 'd' (default 0): 0
```

CODE:

```
# Corrected function 2: Using a try-except block for error handling

def get_value_try_except(key):

    data = {"a": 1, "b": 2}

    try:

        return data[key]

    except KeyError:

        print(f"Key '{key}' not found in dictionary. Returning a default/handling error.")

        return "Key not found"



print("\nCorrected function using try-except:")

print(f"Value for 'b': {get_value_try_except('b')}")

print(f"Value for 'e': {get_value_try_except('e')}")
```

OUTPUT:

```
Corrected function using try-except:

Value for 'b': 2
```

Key 'e' not found in dictionary. Returning a default/handling error.

Value for 'e': Key not found

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.

CODE:

```
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1  # Increment i to avoid infinite loop

print("Executing corrected loop:")
loop_example()
```

OUTPUT:

```
Executing corrected loop:
0
1
2
3
4
```

```
Task 6 (Unpacking Error - Wrong Variables)

CODE:

# Corrected function 1: Unpack into the correct number of variables

print("\nCorrected function 1: Unpack into correct number of
variables")

x, y, z = (1, 2, 3)

print(f"x: {x}, y: {y}, z: {z}")
```

OUTPUT:

```
Corrected function 1: Unpack into correct number of variables

x: 1, y: 2, z: 3
```

CODE:

```
# Corrected function 2: Using the * operator to catch extra values

print("\nCorrected function 2: Using * to catch extra values")

p, q, *rest = (10, 20, 30, 40, 50)

print(f"p: {p}, q: {q}, rest: {rest}")



# Or, if you only care about the first few and want to ignore the rest

r, s, *_ = (100, 200, 300, 400)

print(f"r: {r}, s: {s}")



# You can also use _ for a single unwanted value

i, j, _, k = (1, 2, 3, 4)

print(f"i: {i}, j: {j}, k: {k}")
```

OUTPUT:

```
Corrected function 2: Using * to catch extra values

p: 10, q: 20, rest: [30, 40, 50]

r: 100, s: 200

i: 1, j: 2, k: 4
```

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

CODE:

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

result = func()
print(f"The result is: {result}")
```

OUTPUT:

```
The result is: 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

CODE:

```
import math


# Corrected the module name from maths to math

result = math.sqrt(16)

print(f"The square root of 16 is: {result}")
```

OUTPUT:

```
The square root of 16 is: 4.0
```