AI ASSISTED CODING ASSIGNMENT - 7.5

Name:B.SiriChandana
HTNo:2303A52004
Batch:31

Question:
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.

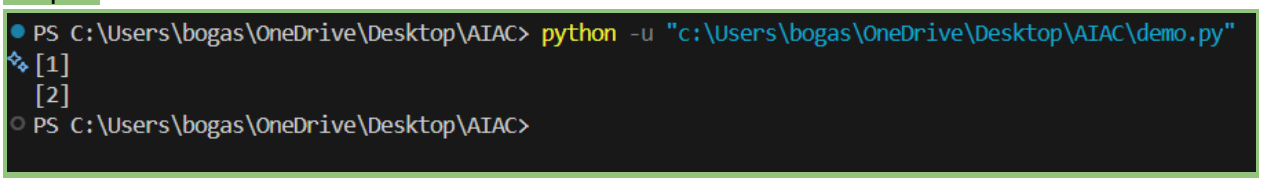
Prompt:

```
#fix the mutable default argument issue by using None as the default value and creating a new list inside the function if items is None.
```

Code:

```
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
if __name__ == "__main__":
    print(add_item(1))
    print(add_item(2))
```

Output:



Justification:
Mutable default arguments share the same object across function calls.
Using None creates a new list each time, avoiding unintended data sharing.

Question:
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
Prompt:

```
#fix the problem of floating point precision
```

Code:

```
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-9
if __name__ == "__main__":
    print(check_sum())
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\demo.py"
True
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

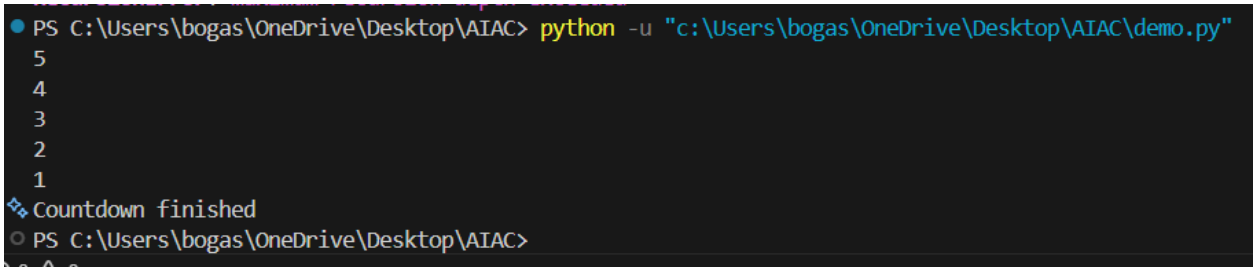
Justification:
Floating-point numbers have precision errors in binary representation.
Using a tolerance comparison avoids equality check failure.
Question:
Task: Analyze given code where recursion runs infinitely due to
missing base case. Use AI to fix.
# Bug: No base case
def countdown(n):
print(n)
return countdown(n-1)
countdown(5)
Expected Output : Correct recursion with stopping condition
Prompt:

```
#write the base case to fix issue of infinite recursion
```

Code:

```
def countdown(n):
    if n <= 0:
        print("Countdown finished")
        return
    print(n)
    return countdown(n-1)
if __name__ == "__main__":
    countdown(5)
```

Output:



Justification:

Recursion without a base case causes infinite calls and stack overflow.
Adding a stopping condition ensures proper termination.

Question:

Task 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 the problem of accessing non existing key in a
dictionary
```

Code:

```
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")
if __name__ == "__main__":
    print(get_value())
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Key not found
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Accessing a missing dictionary key raises a `KeyError`.

Using `.get()` safely handles missing keys with a default value.

Question:

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.

Prompt:

```
#fix the problem of infinite loop
```

Code:

```
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1
if __name__ == "__main__":
    loop_example()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\demo.py"
0
1
2
3
4
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The loop never ends because the counter was not incremented.
Adding `i += 1` ensures the loop terminates correctly.

Question:

Task 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 the problem of wrong unpacking
```

Code:

```
a, b, c = (1, 2, 3)
print(a, b, c)
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
1 2 3
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Tuple unpacking requires equal numbers of variables and values.
Matching them correctly prevents `ValueError`.

Question:

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.

Prompt:

```
#fix the problem of mixed indentation
```

Code:

```
def func():
    x = 5
    y = 10
    return x+y
if __name__ == "__main__":
    result = func()
    print("The result is:", result)
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
The result is: 15
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Mixed tabs and spaces cause `IndentationError` in Python.

Using consistent indentation fixes the issue.

Question:

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

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
4.0
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Incorrect module name causes `ModuleNotFoundError`.

Importing the correct `math` module resolves it.

Question:
Task 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:

```
def total(numbers):
    total_sum = 0
    for n in numbers:
        total_sum += n
    return total_sum
if __name__ == "__main__":
    print(total([1,2,3]))
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
6
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:
Return inside the loop stops execution after first iteration.
Moving return outside processes all elements properly.

Question:
Task 10 (Name Error – Undefined Variable)
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:

```
def calculate_area():
    length = 5
    width = 3
    return length * width
if __name__ == "__main__":
    print(calculate_area())
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\demo.py"
15
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Using undefined variables causes `NameError`.
Defining them as parameters makes the function valid and reusable.

Question:

Task 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:

```
def add_values():
```

```
    return 5 + int("10")
if __name__ == "__main__":
    print(add_values())
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
15
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

Python cannot add `int` and `str` directly.
Type conversion ensures both operands are compatible.

Question:

Task 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:

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

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\demo.py"
Numbers: [1, 2, 3]
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:
String and list cannot be concatenated directly.
Converting the list to string fixes the type mismatch.
Question:
Task 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:

```
def repeat_text():
    return "Hello" * int(2.5)
if __name__ == "__main__":
    print(repeat_text())
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
HelloHello
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:
Strings can only be multiplied by integers, not floats.
Converting float to int enables valid repetition.
Question:
Task 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:

```
def compute(a,value=0):
    return a * value + 10
if __name__ == "__main__":
    print(compute(5, 2))
    print(compute(5))
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
20
10
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

NoneType cannot be used in arithmetic operations.
Assigning a numeric default prevents the error.

Question:

Task 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 the problem of converting input string to integer
before performing addition
```

Code:

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

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Enter first number: 7
Enter second number: 2
9
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

`input()` always returns a string value.

Converting to `int` ensures correct numerical addition.