Ai Assisted Coding

Week:7.5

Ht.No:2303A52350

Batch:45

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)

Prompt:

def add_item(item, items=[]):

items.append(item)

return items

print(add_item(1))

print(add_item(2) in this code there is a bug and it's a mutable default argument. Correct it.

Code:

```
def add_item(item, items=[]):
    items.append(item)
    return items
print(add_item("apple"))  # Output: ['apple']
```

print(add_item("banana")) # Output: ['apple', 'banana']

print(add_item("orange", [])) # Output: ['orange']

print(add_item("grape")) # Output: ['apple', 'banana', 'grape']

Output:

```
PS C:\Users\mouni\OneDrive\Desktop\Ai Ass labs>  & 'c:\Users\mouni\AppData\Local\Programs\Py
e' 'c:\Users\mouni\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\deb
--' 'C:\Users\mouni\OneDrive\Desktop\Ai Ass labs\Ai Ass.7.5.py'
['apple']
['apple', 'banana']
['orange']
['apple', 'banana', 'grape']
PS C:\Users\mouni\OneDrive\Desktop\Ai Ass labs>
```

Explanation:

This code shows how a bug can be fixed in one line.

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:

Analyze given code where floating-point comparison fails.correct with tolerance. Bug: Floating point precision issue

def check_sum():

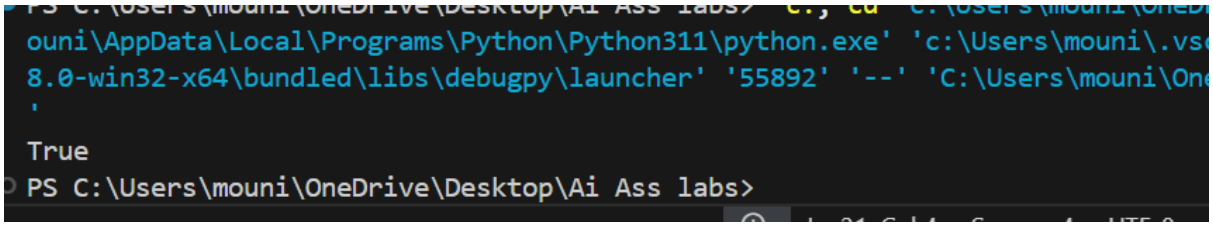
return (0.1 + 0.2) == 0.3

print(check_sum()) correct the code with tolerance level and precision recall.

Code:

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

Output:



Explanation:

The original function fails due to floating-point precision issues. The corrected function uses a tolerance level (1e-9) to compare the sum, ensuring that minor precision errors do not affect the outcome.

Task 3 (Recursion Error – Missing Base Case)

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)
```

Prompt:

Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.Bug: No base case

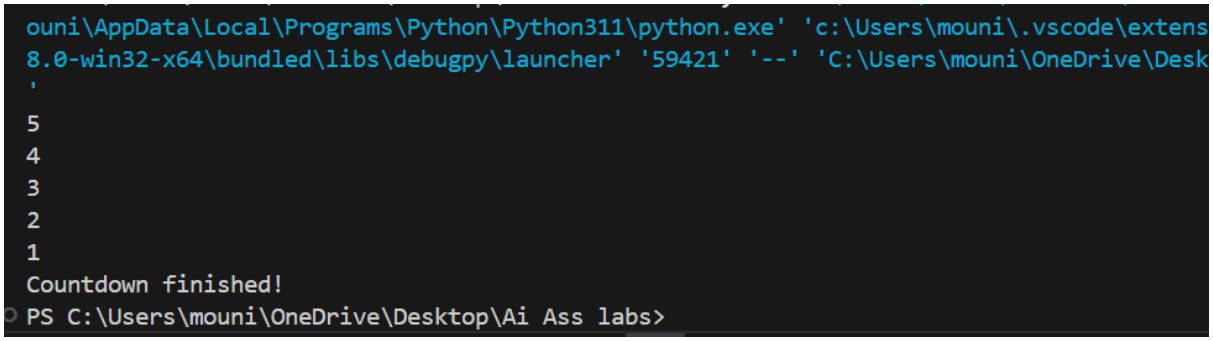
```
def countdown(n):
print(n)
return countdown(n-1)
```

countdown(5)

Code:

```
def countdown(n):
    if n <= 0:  # Base case to stop recursion
        print("Countdown finished!")
        return
    print(n)
    return countdown(n - 1)
countdown(5)
```

Output:



Explanation:

The original code lacked a base case, causing it to call itself indefinitely. The corrected version includes a base case that checks if n is less than or equal to 0, at which point it prints a message and stops the recursion.

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())

Prompt:

Generate a code for accessing non existing key,to fix a bug

Code:

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

Output:

```
ouni\AppData\Local\Programs\Python\Python311\python.exe' 'c:\Users\n
8.0-win32-x64\bundled\libs\debugpy\launcher' '54429' '--' 'C:\Users\
'
Key not found
```

Explanation: The corrected function uses the get() method with a default value to avoid KeyError.

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)
```

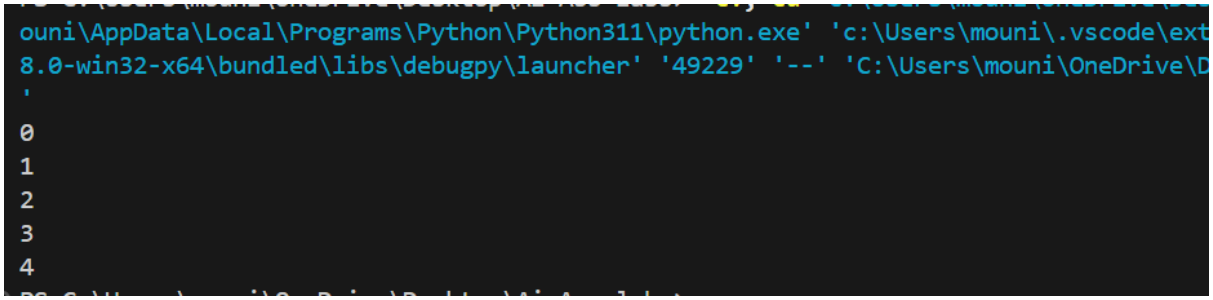
Prompt:

Generate a python code where the loop never ends and to fix the bug.

Code:

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

Output:



Explanation:

The original code had an infinite loop because the variable 'i' was never incremented. By adding 'i += 1', we ensure that the loop will eventually terminate when 'i' reaches 5.

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)
```

Prompt:

Generate a python code to fix a bug where tuple unpacking fails.

Code:

```
def unpack_example():
```

```
a, b, _ = (1, 2, 3)  # Using _ to ignore extra value
print("a:", a)
print("b:", b)
```

.

Output:

```
ouni\AppData\Local\Programs\Python\Python311\python.exe' 'c:\Users\mouni\.v
8.0-win32-x64\bundled\libs\debugpy\launcher' '51235' '--' 'C:\Users\mouni\O
'
a: 1
b: 2
```

Explanation:

In the original code, there were three values in the tuple but only two variables to unpack into, which caused an error. By adding a third variable (using _), we can ignore the extra value and successfully unpack the first two values into a and b.

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
```

Prompt:

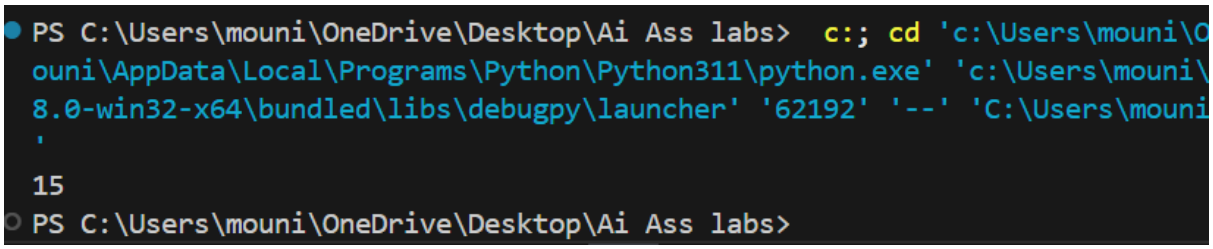
Generate a python code to fox the bug where mixed indentation breaks the execution.

Code:

```
def func():
```

```
    x = 5
    y = 10
    return x + y
print(func())
```

Output:



Explanation:

In this code the print statement was not present.so without print statement we cant expext the output.

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))
```

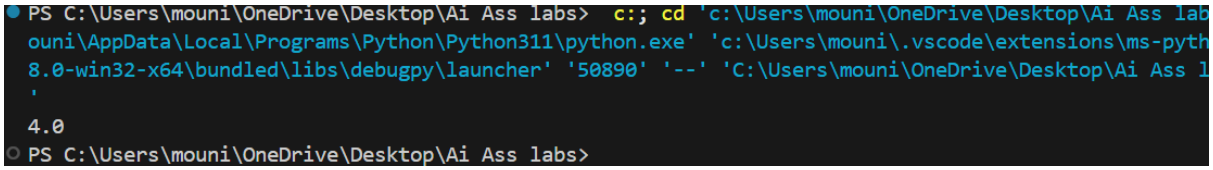
Prompt:

Generate a python code to fix the bug for a given code with correct import.

Code:

```
import math
def calculate_square_root(num):
    return math.sqrt(num)
print(calculate_square_root(16))
```

Output:



Explanation:

The original code had a typo in the module name ("maths" instead of "math"). The corrected code imports the correct module and defines a function to calculate the square root, which is then called with the argument 16.

Task:9.(Unreachable Code – Return Inside Loop) 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]))
```

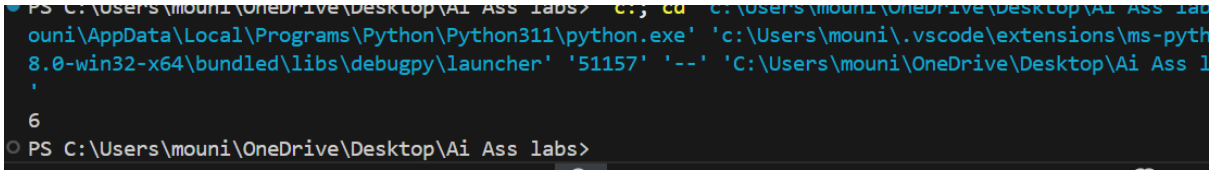
Prompt:

Generate a python code from unreachable code to reachable inside loop and prevents full iteration to fix a given bug.

Code:

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

Output:



Explanation:

The original code had a return statement inside the loop, which caused it to return after the first iteration, making the rest of the numbers unreachable. The corrected code initializes a total_sum variable and accumulates the sum of all numbers in the loop, returning the final total after the loop completes.

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())
```

Prompt:

Generate a python code where a variable is used before being defined and fix the bug by defining length and width as parameters.Add 3 assert test cases for correctness.

Code:

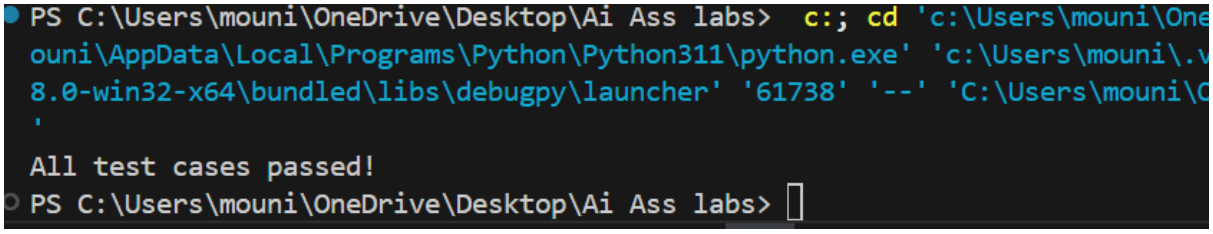
```
def calculate_area(length, width):
    return length * width
# Test cases
assert calculate_area(5, 10) == 50
```

assert calculate_area(7, 3) == 21

assert calculate_area(0, 5) == 0

print("All test cases passed!")

Output:



Explanation:

The original code had a bug where the variables 'length' and 'width' were used before being defined, which would lead to a NameError. The corrected code defines 'length' and 'width' as parameters of the function 'calculate_area', allowing it to compute the area correctly. Additionally, three assert statements are included to test the correctness of the function with different inputs.

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.

PROMPT:

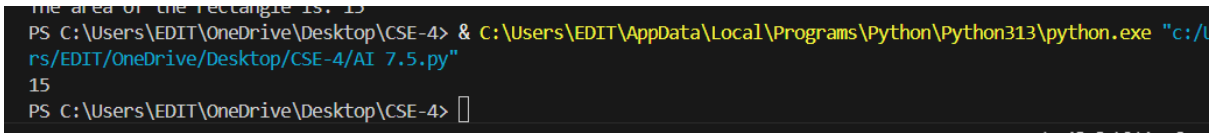
```
def add_values():
    return 5 + "10"
print(add_values())
```

Fix the bug:adding integer and string and ai explanation of the fix.

Code:

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

Output:



#AI explanation of the fix.
The original code attempted to add an integer (5) and a string ("10"), which is not allowed in Python and results in a TypeError. The fixed code converts the string "10" to an integer using the int() function before performing the addition, allowing the operation to succeed and return the correct result of 15.

EXPLANATION:

To the given code we should fix the bug: Adding integer and string.and ai explanation of the fixed code

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 validatation

Prompt:

Bug: Adding string and list

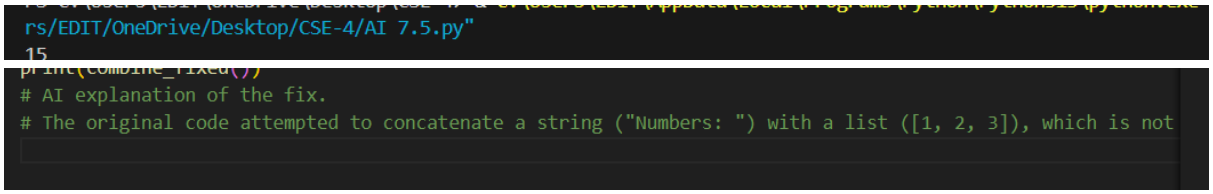
```
def combine():
   return "Numbers: " + [1, 2, 3]
print(combine())
```

Fix the bug in the code and explanation of the fix.

CODE:

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

OUTPUT:



EXPLANATION:

Fix the bug in the code and explanation of the fix.add string and list in the code.

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.

PROMPT:

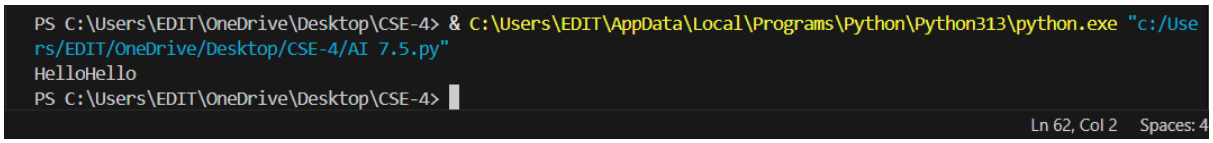
```
def repeat_text():
    return "Hello" * 2.5
```

print(repeat_text()) Fix the bugs

CODE:

```
def repeat_text_fixed():
    return "Hello" * 2  # Multiplying by an integer
print(repeat_text_fixed())
```

OUTPUT:



AI explanation of the fix.

The original code attempted to multiply a string ("Hello") by a float (2.5), which is not allowed in Python and results in a TypeError. The fixed code changes the multiplication factor to an integer (2), allowing the operation to succeed and return the correct result of "HelloHello".

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.

PROMPT:

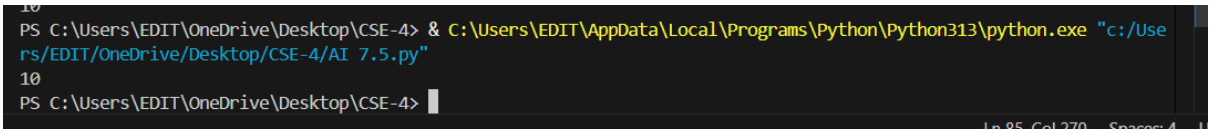
```
def compute():
    value = None
    return value + 10

print(compute()) Fix the bugs.
```

CODE:

```
def compute_fixed():
```

```
    value = 0  # Initialize value to a number
    return value + 10
print(compute_fixed())
```

OUTPUT:



AI explanation of the fix.

The original code attempted to add 10 to a variable (value) that was initialized to None, which is not allowed in Python and results in a TypeError. The fixed code initializes value to 0, allowing the addition operation to succeed and return the correct result of 10.

EXPLANATION:

I n the given code fix the bugs and explained.

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:

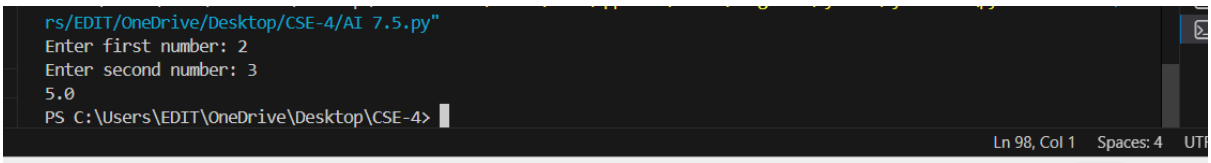
```
def sum_two_numbers():
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    return a + b

print(sum_two_numbers())
```

Fix the bugs.

CODE:

```
def sum_two_numbers_fixed():
    a = float(input("Enter first number: "))  # Convert input to float
    b = float(input("Enter second number: "))  # Convert input to
float
    return a + b
print(sum_two_numbers_fixed())
```

OUTPUT:



EXPLANATION:

Give two numbers and input and return the sum of two numbers and fix the bug.