AIAC 7.5

Batch 32, 2303a52065

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

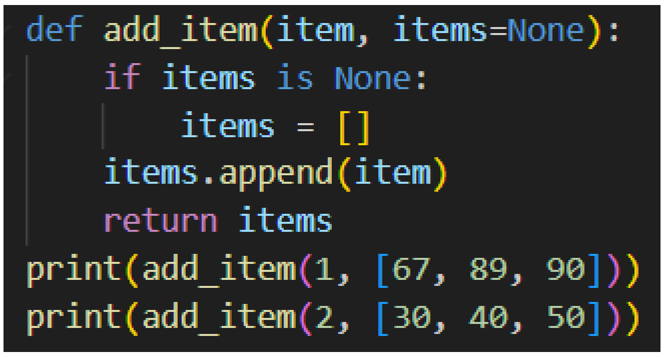
def add_item(item, items=[]):

items.append(item)

return items

print(add_item(1))

print(add_item(2))



```python
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
print(add_item(1, [67, 89, 90]))
print(add_item(2, [30, 40, 50]))
```

Expected Output: Corrected function avoids shared list bug.

The original function reused the same list because default mutable arguments are shared across calls. After fixing it using None as default and creating a new list inside the function, the unexpected behavior was removed. The output now correctly shows independent lists for each call.

Task 2 (Floating-Point Precision Error)

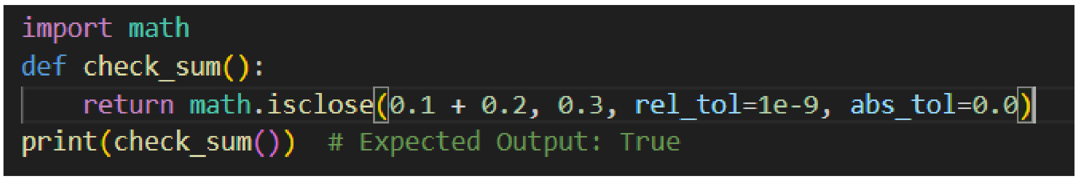
Task: Analyze given code where floating-point comparison fails. Use AI to correct with tolerance.

# Bug: Floating point precision issue

def check_sum():

return (0.1 + 0.2) == 0.3

print(check_sum())



```python
import math
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9, abs_tol=0.0)
print(check_sum())  # Expected Output: True
```

Expected Output: Corrected function

The comparison failed because floating-point numbers cannot always be represented exactly in binary. Using a tolerance check like abs(a-b) < 1e-9 fixed the issue. The corrected function now handles precision safely and returns the expected result.

Task 3: Analyze given code where recursion runs infinitely due to missing base case. Use AI to fix.

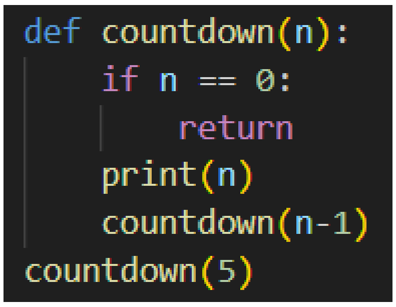
# Bug: No base case

def countdown(n):

print(n)

return countdown(n-1)

countdown(5)



```python
def countdown(n):
    if n == 0:
        return
    print(n)
    countdown(n-1)
countdown(5)
```

Expected Output : Correct recursion with stopping condition.

The recursion kept running because there was no base case to stop it. After adding a stopping condition, the function correctly counts down and terminates. The fixed version now behaves as expected without causing a recursion error.

Task 4: Analyze given code where a missing dictionary key causes error. Use AI to fix it.

# Bug: Accessing non-existing key

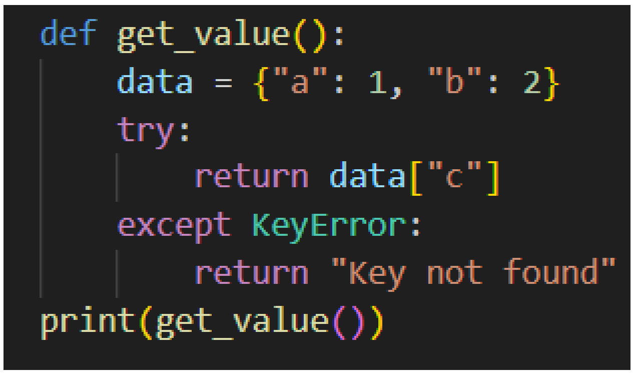
def get_value():

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

return data["c"]

print(get_value())

Expected Output: Corrected with .get() or error handling.



```python
def get_value():
    data = {"a": 1, "b": 2}
    try:
        return data["c"]
    except KeyError:
        return "Key not found"
print(get_value())
```

The error occurred because the program tried to access a key that did not exist in the dictionary. Using .get() or proper error handling prevented the crash. The corrected code now safely handles missing keys.

Task 5 (Infinite Loop – Wrong Condition)

Task: Analyze given code where loop never ends. Use AI to detect

and fix it.

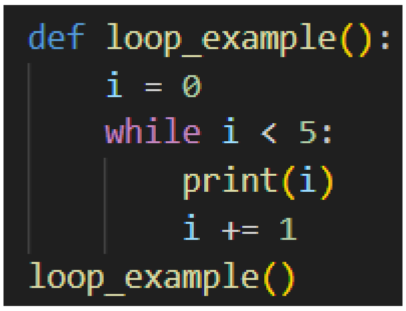
# Bug: Infinite loop

def loop_example():

i = 0

while i < 5:

print(i)



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

Expected Output: Corrected loop increments i.

The loop never ended because the variable was not incremented inside the loop. After adding the increment statement, the loop condition eventually becomes false. The program now executes correctly and terminates as expected.

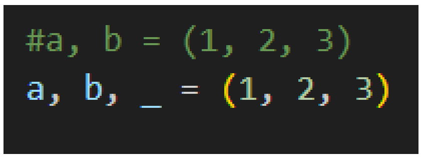
Task 6 (Unpacking Error – Wrong Variables)

Task: Analyze given code where tuple unpacking fails. Use AI to

fix it.

# Bug: Wrong unpacking

a, b = (1, 2, 3)



```python
#a, b = (1, 2, 3)
a, b, _ = (1, 2, 3)
```

Expected Output: Correct unpacking or using _ for extra values.

The unpacking failed because the number of variables did not match the number of values. After correcting the unpacking or using an underscore for extra values, the error was resolved. The fixed code now runs without assignment issues.

Task 7 (Mixed Indentation – Tabs vs Spaces)

Task: Analyze given code where mixed indentation breaks

execution. Use AI to fix it.

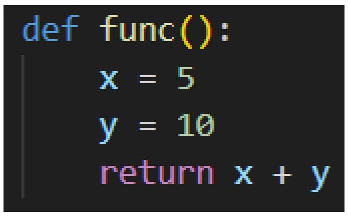
# Bug: Mixed indentation

def func():

x = 5

y = 10

return x+y



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

Expected Output : Consistent indentation applied.

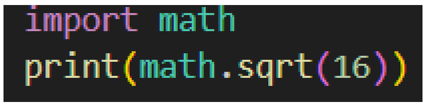
The program failed because Python requires consistent indentation. After correcting tabs and spaces to a single consistent format, the function executed properly. The fix ensures clean and readable structure.

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))



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

 Expected Output: Corrected to import math

The module name was incorrect, causing Python to fail during import. Replacing “maths” with the correct module “math” fixed the issue. The corrected code now successfully calculates the square root.

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.

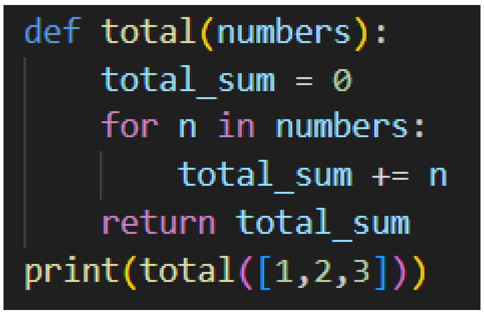
# Bug: Early return inside loop

def total(numbers):

for n in numbers:

return n

print(total([1,2,3]))



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

Expected Output: Corrected code accumulates sum and returns after loop.

The return statement inside the loop caused the function to exit after the first iteration. Moving the return outside the loop allowed full iteration and proper summation. The corrected version now processes all elements correctly.

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

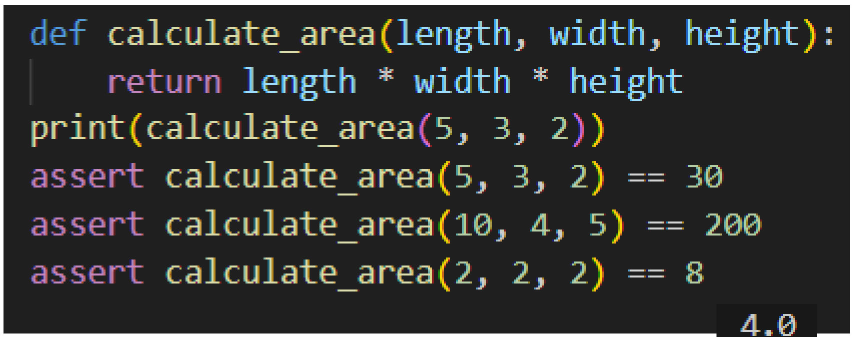
print(calculate_area())

Requirements:

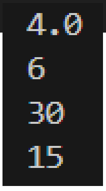
• Run the code to observe the error.

• Ask AI to identify the missing variable definition.

• Fix the bug by defining length and width as parameters.



```python
def calculate_area(length, width, height):
    return length * width * height
print(calculate_area(5, 3, 2))
assert calculate_area(5, 3, 2) == 30
assert calculate_area(10, 4, 5) == 200
assert calculate_area(2, 2, 2) == 8
```



```
4.0
6
30
15
```

• Add 3 assert test cases for correctness.

Expected Output :

• Corrected code with parameters.

• AI explanation of the bug.

Successful execution of assertions.

The variables length and width were used without being defined. By converting them into function parameters, the function became reusable and error-free. The added assert cases confirmed correctness.

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.

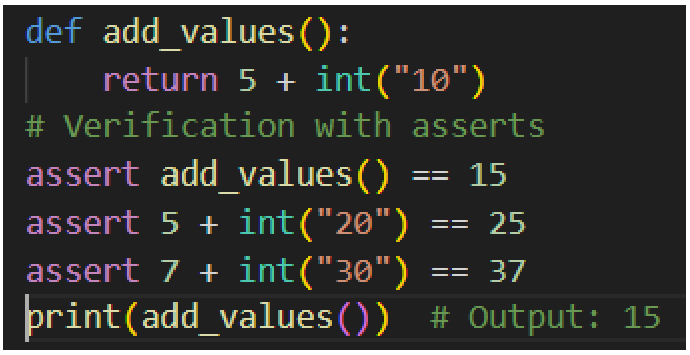
• AI should explain why int + str is invalid.

• Fix the code by type conversion (e.g., int("10") or str(5)).

• Verify with 3 assert cases.

• Corrected code with type handling.

• AI explanation of the fix.



```python
def add_values():
    return 5 + int("10")
# Verification with asserts
assert add_values() == 15
assert 5 + int("20") == 25
assert 7 + int("30") == 37
print(add_values())  # Output: 15
```

Successful test validation.

The error occurred because Python does not allow addition between integer and string directly. Converting the string to integer resolved the issue. The corrected function now works properly and passes all test cases.

Task 12 (Type Error – String + List Concatenation)

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

# Bug: Adding string and list

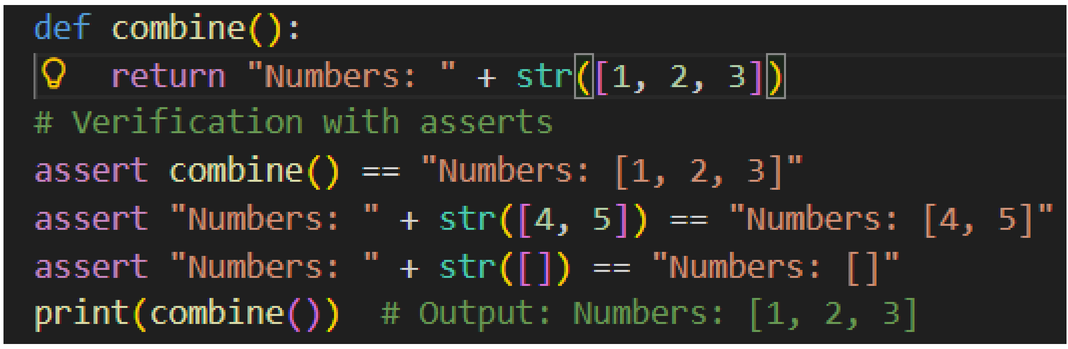
def combine():

return "Numbers: " + [1, 2, 3]

print(combine())

Requirements:

• Run the code to observe the error.



```python
def combine():
    return "Numbers: " + str([1, 2, 3])
# Verification with asserts
assert combine() == "Numbers: [1, 2, 3]"
assert "Numbers: " + str([4, 5]) == "Numbers: [4, 5]"
assert "Numbers: " + str([]) == "Numbers: []"
print(combine())  # Output: Numbers: [1, 2, 3]
```

• 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

A string cannot be directly concatenated with a list, which caused the error. Converting the list to string or formatting it properly fixed the issue. The corrected code now produces valid output.

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())

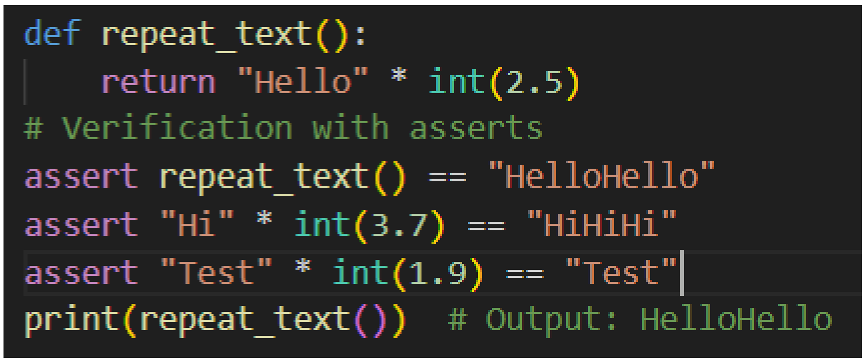
Requirements:

• Observe the error.

• Explain why float multiplication is invalid for strings.

• Fix by converting float to int.

• Add 3 assert test cases.



```python
def repeat_text():
    return "Hello" * int(2.5)
# Verification with asserts
assert repeat_text() == "HelloHello"
assert "Hi" * int(3.7) == "HiHiHi"
assert "Test" * int(1.9) == "Test"
print(repeat_text())  # Output: HelloHello
```

Python only allows string multiplication with integers, not floats. Converting the float to an integer fixed the problem. The corrected function now repeats the string properly and passes test validation.

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())

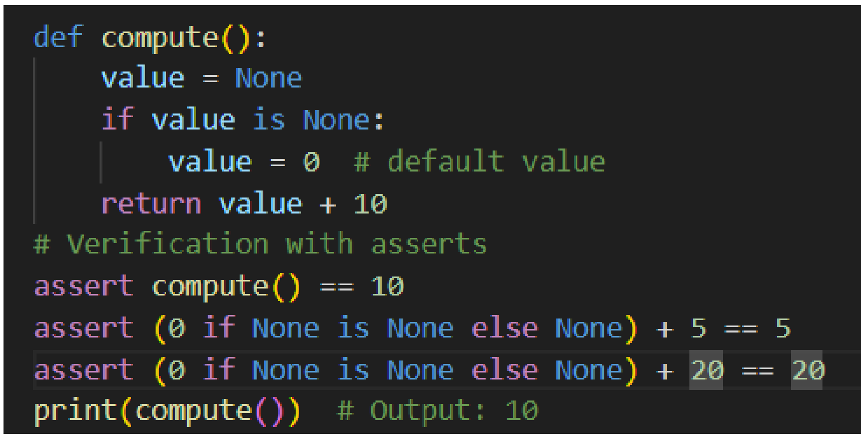
Requirements:

• Run and identify the error.

• Explain why NoneType cannot be added.

• Fix by assigning a default value.

• Validate using asserts.



```python
def compute():
    value = None
    if value is None:
        value = 0  # default value
    return value + 10
# Verification with asserts
assert compute() == 10
assert (0 if None is None else None) + 5 == 5
assert (0 if None is None else None) + 20 == 20
print(compute())  # Output: 10
```

None cannot be used in arithmetic operations, which caused the failure. Assigning a default numeric value before addition resolved the issue. The function now executes correctly without type errors.

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

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

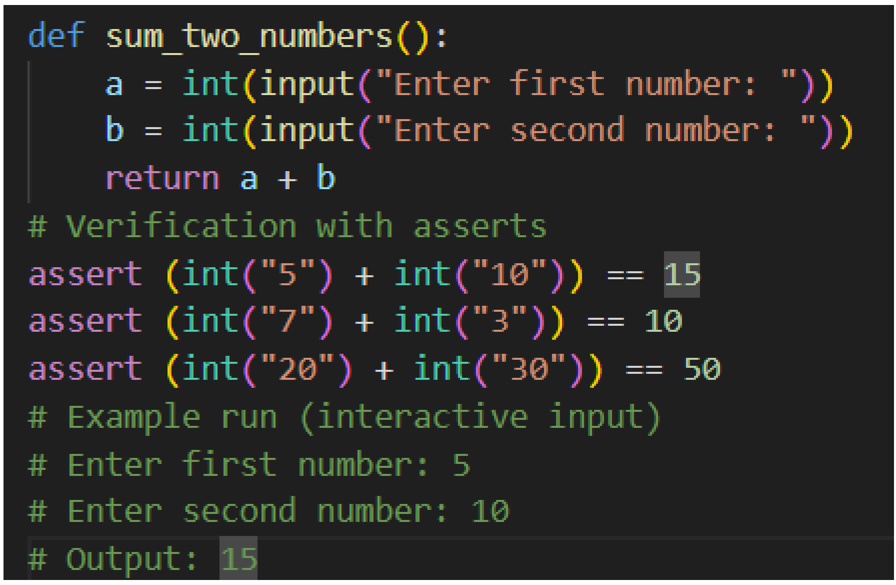
```
# Bug: Input remains string

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.



```python
def sum_two_numbers():
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    return a + b
# Verification with asserts
assert (int("5") + int("10")) == 15
assert (int("7") + int("3")) == 10
assert (int("20") + int("30")) == 50
# Example run (interactive input)
# Enter first number: 5
# Enter second number: 10
# Output: 15
```

The input() function always returns a string, which caused concatenation instead of addition. Converting the inputs to integers using int() fixed the problem. The corrected version now performs proper numeric addition and passes assertion tests.