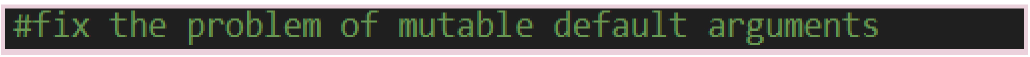
AI ASSISTED CODEING ASSIGNMENT 7.4

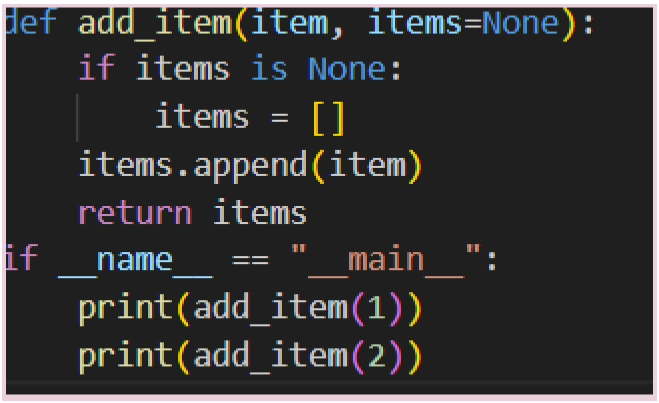
Name: B Vrindha
HT No.:2303A52003
Batch-31

Question:1
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:



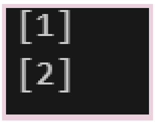
```
#fix the problem of mutable default arguments
```

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:



```
[1]
[2]
```

Justification:

- The list was getting reused every time the function was called.
- Because of that, previous values were not getting cleared.
- This created unexpected results in the output.
- Changing the default to None makes sure a new list is created each time.

Question:2

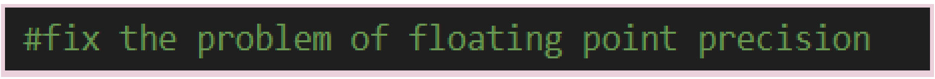
Task 2 (Floating-Point Precision Error)

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())
```

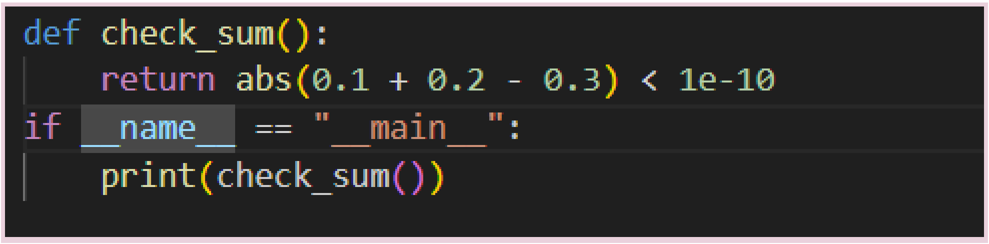
Expected Output: Corrected function

Prompt:



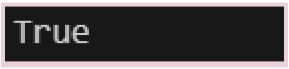
```
#fix the problem of floating point precision
```

Code:



```
def check_sum():
    return abs(0.1 + 0.2 - 0.3) < 1e-10
if __name__ == "__main__":
    print(check_sum())
```

Output:



```
True
```

Justification:

- Computers store decimal numbers slightly differently in memory.
- So sometimes 0.1 + 0.2 is not exactly equal to 0.3.
- Direct comparison using == may fail because of tiny differences.
- Using a small tolerance value solves this issue safely.

Question:3

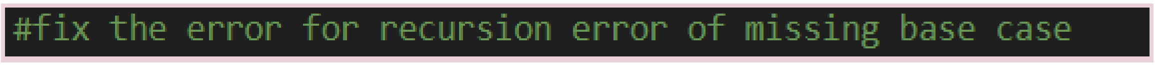
Task 3 (Recursion Error – Missing Base Case)

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)
```

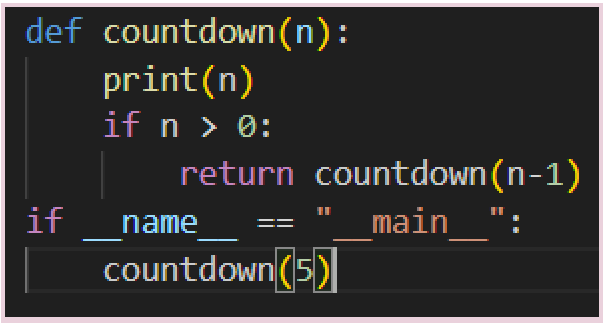
Expected Output : Correct recursion with stopping condition.

Prompt:



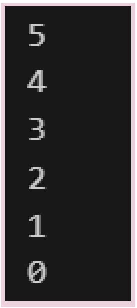
```
#fix the error for recursion error of missing base case
```

Code:



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

Output:



```
5
4
3
2
1
0
```

Justification:

- A recursive function must know when to stop.
- Without a stopping condition, it keeps calling itself forever.
- This causes a recursion error.
- Adding a base case stops the function at the right time.

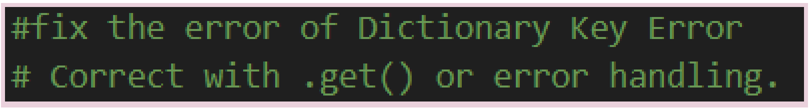
Question:4

Task 4 (Dictionary Key Error)

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

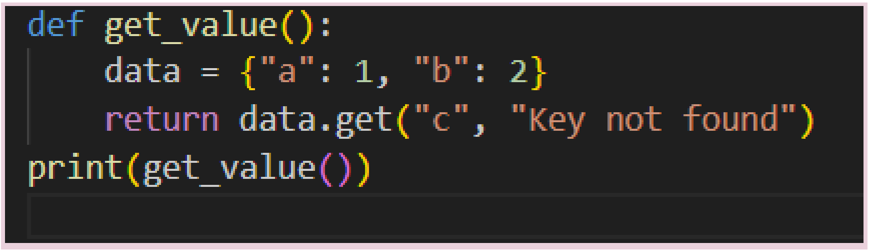
# Bug: Accessing non-existing key
def get_value():
data = {"a": 1, "b": 2}
return data["c"]
print(get_value())
Expected Output: Corrected with .get() or error handling.

Prompt:



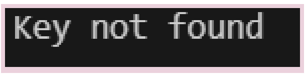
```python
#fix the error of Dictionary Key Error
# Correct with .get() or error handling.
```

Code:



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

Output:



```
Key not found
```

Justification:

- The code tried to access a key that does not exist.
- Python throws an error when a key is missing.
- This can crash the program.
- Using `.get()` prevents the error and makes the code safer.

Question:5

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():

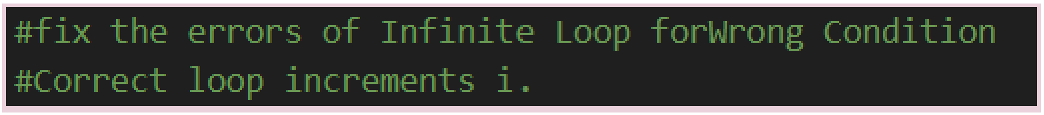
i = 0

while i < 5:

print(i)

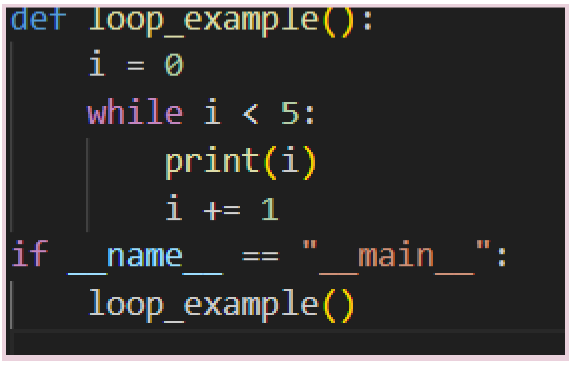
Expected Output: Corrected loop increments i.

Prompt:



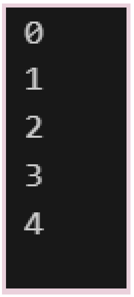
```
#fix the errors of Infinite Loop forWrong Condition
#Correct loop increments i.
```

Code:



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

Output:



```
0
1
2
3
4
```

Justification:

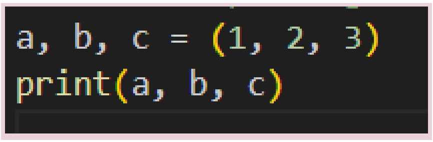
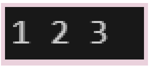
- The loop variable was never increasing.
- Because of that, the condition never became false.
- So the loop kept running forever.
- Incrementing the variable fixes the issue.

Question:6

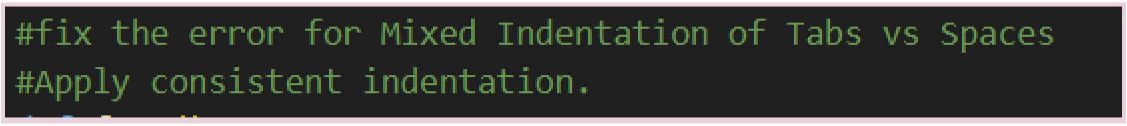
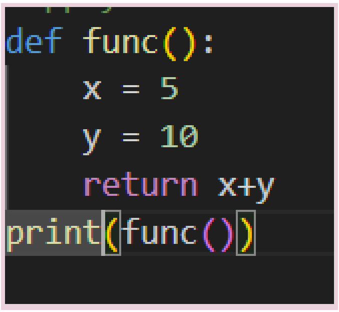
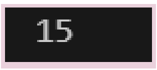
Task 6 (Unpacking Error – Wrong Variables)

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

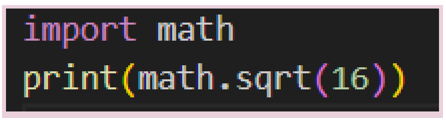
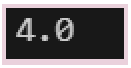
# Bug: Wrong unpacking

a, b = (1, 2, 3)
Expected Output: Correct unpacking or using _ for extra values.
Prompt:
Did Manually
Code:

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

```
1 2 3
```
Justification:

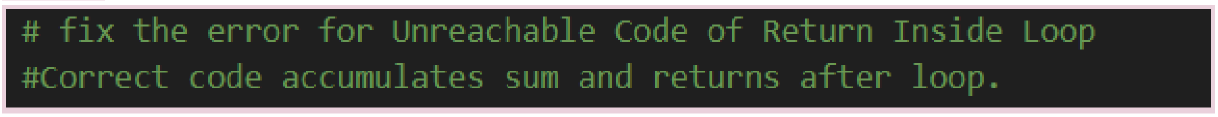
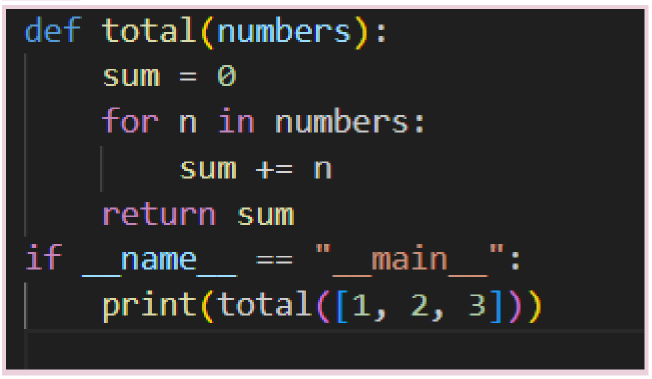
- The number of variables and values did not match.
- Python expects equal numbers during unpacking.
- Extra values caused an error.
- Adjusting the variables or using _ solves the problem.

Question:7
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 error for Mixed Indentation of Tabs vs Spaces
#Apply consistent indentation.
```
Code:

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

```
15
```
Justification:

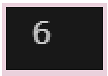
- Python depends on proper indentation.
- Mixing tabs and spaces confuses the interpreter.
- This results in an indentation error.
- Using consistent spacing fixes the issue.

Question:8
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
Prompt:
Corrected Manually
Code:

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

```
4.0
```
Justification:

- The module name was written incorrectly.
- Python could not find a module called "maths".
- Because of that, the program failed.
- Correcting it to "math" solves the problem.

Question:9
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.
Prompt:

```python
# fix the error for Unreachable Code of Return Inside Loop
#Correct code accumulates sum and returns after loop.
```
Code:

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

Output:



```
6
```

Justification:

- The function returned a value in the first iteration itself.
- So it never processed the remaining elements.
- That's why the total was not calculated properly.
- Moving the return statement outside the loop fixes it.

Question:10

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 :

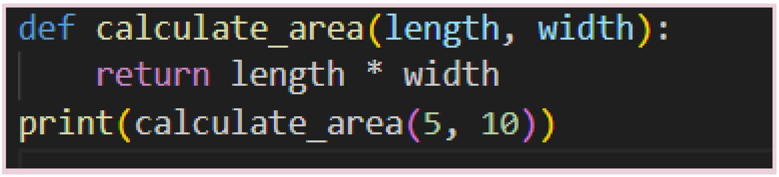
• Corrected code with parameters.
• AI explanation of the bug.

Successful execution of assertions.

Prompt:

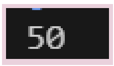
Manually Corrected

Code:



```
def calculate_area(length, width):
    return length * width
print(calculate_area(5, 10))
```

Output:



```
50
```

Justification:

- The variables were used without defining them.
- Python cannot calculate with unknown values.
- This caused a NameError.
- Defining them as parameters makes the function correct and flexible.

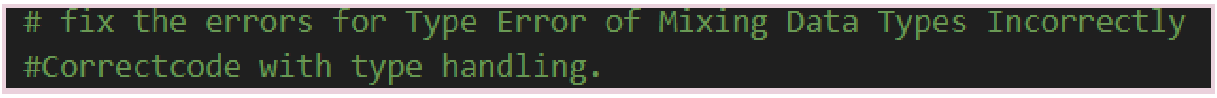
Question:11

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.

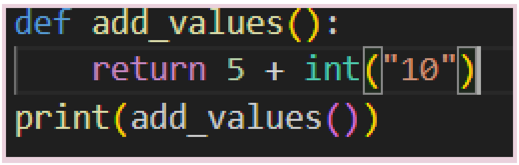
# Bug: Adding integer and string
def add_values():

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:



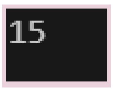
```
# fix the errors for Type Error of Mixing Data Types Incorrectly
#Correctcode with type handling.
```

Code:



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

Output:

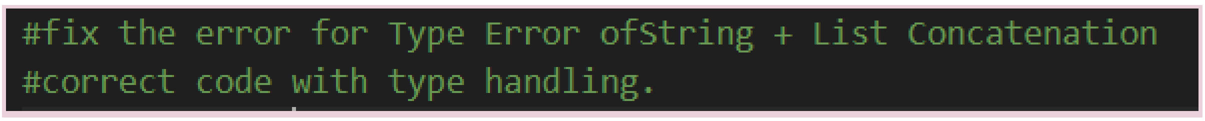


```
15
```

Justification:

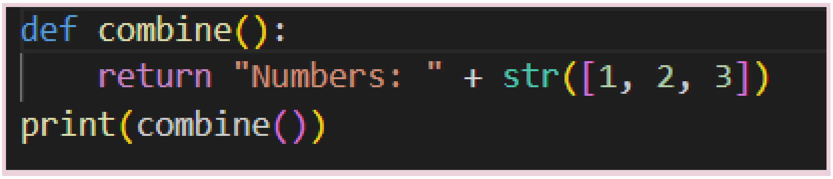
- Integer and string are different data types.
- Python does not allow adding them directly.
- This caused a TypeError.
- Converting them into the same type solves the issue.

Question:12
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
Prompt:



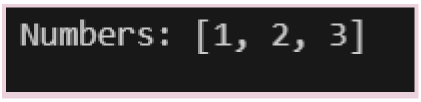
```
#fix the error for Type Error ofString + List Concatenation
#correct code with type handling.
```

Code:



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

Output:



```
Numbers: [1, 2, 3]
```

Justification:

- A string and list cannot be added together.
- They are completely different data types.
- Python raises a TypeError in this case.
- Converting the list to string makes it work.

Question:13

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.

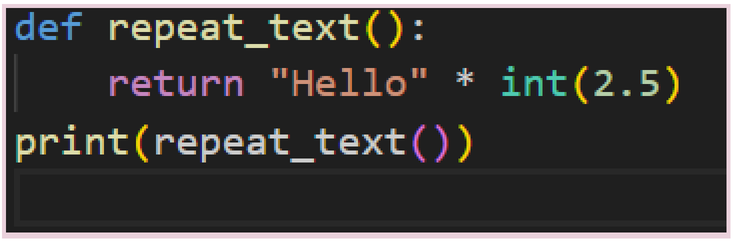
• Fix by converting float to int.

• Add 3 assert test cases.

Prompt:

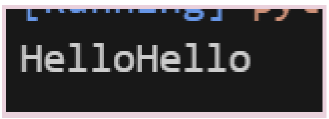
Manually solved

Code:



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

Output:



```
HelloHello
```

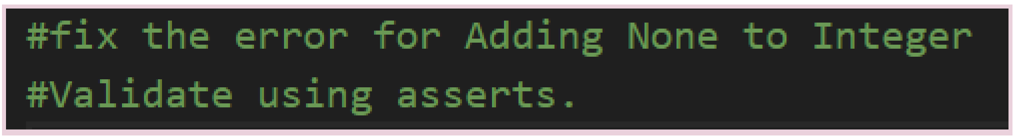
Justification:

- Strings can only be multiplied by integers.
- Multiplying by a float is not allowed.
- This caused a TypeError.
- Converting the float into an integer fixes it.

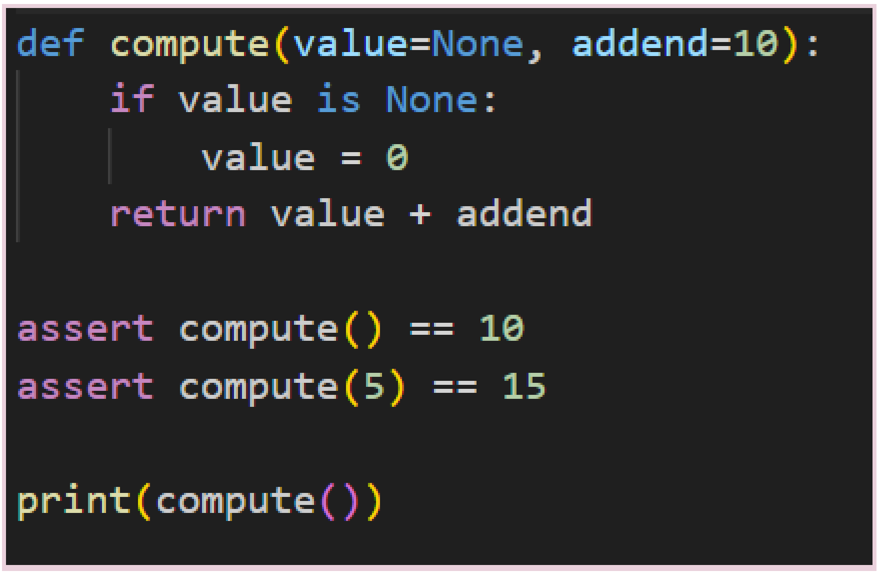
Question:14

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:


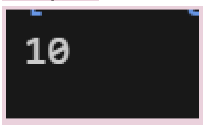
```
#fix the error for Adding None to Integer
#Validate using asserts.
```

Code:


```python
def compute(value=None, addend=10):
    if value is None:
        value = 0
    return value + addend

assert compute() == 10
assert compute(5) == 15

print(compute())
```

Output:


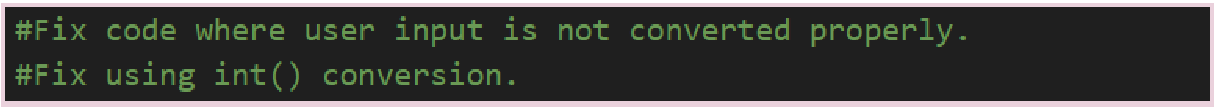
```
10
```

Justification:

- None means no value is assigned.
- Arithmetic operations need numeric values.
- Adding None to a number is invalid.
- Assigning a default number solves the issue.

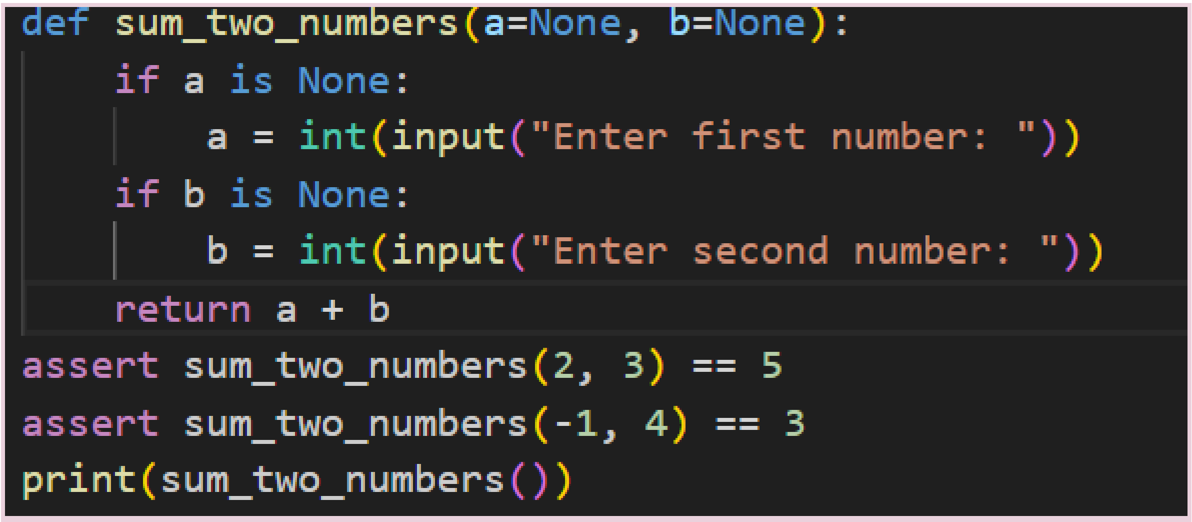
Question:15
Task 15 (Type Error – Input Treated as String Instead of Number)
Task: Fix code where user input is not converted properly.
# Bug: Input remains string
def sum_two_numbers():
a = input("Enter first number: ")
b = input("Enter second number: ")

return a + b
print(sum_two_numbers())
Requirements:
• Explain why input is always string.
• Fix using int() conversion.
• Verify with assert test cases.
Prompt:



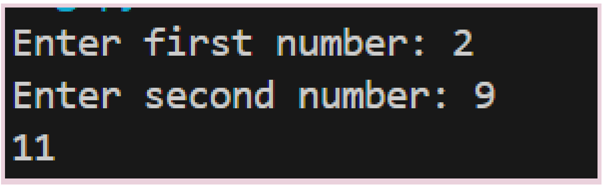
```
#Fix code where user input is not converted properly.
#Fix using int() conversion.
```

Code:



```python
def sum_two_numbers(a=None, b=None):
    if a is None:
        a = int(input("Enter first number: "))
    if b is None:
        b = int(input("Enter second number: "))
    return a + b
assert sum_two_numbers(2, 3) == 5
assert sum_two_numbers(-1, 4) == 3
print(sum_two_numbers())
```

Output:



```
Enter first number: 2
Enter second number: 9
11
```

Justification:

- The input function always returns a string.
- So adding two inputs joins them instead of adding numbers.
- This gives wrong results.
- Converting them to integers fixes the problem.