AI Assisted Coding

Lab Assignment 7.5

Name : AVULA SHIVA SPANDANA

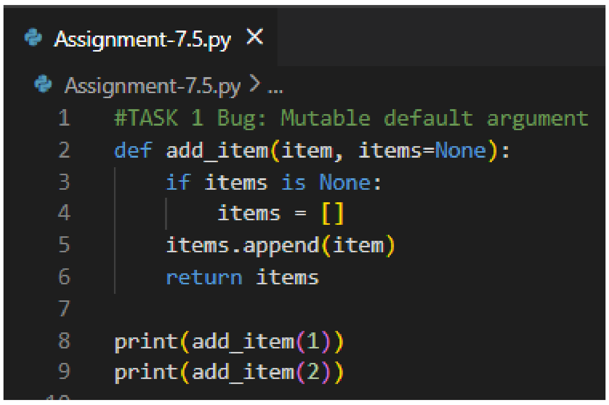
Hall Ticket no : 2303A51320

Batch No : 19

Task -1:

Prompt:

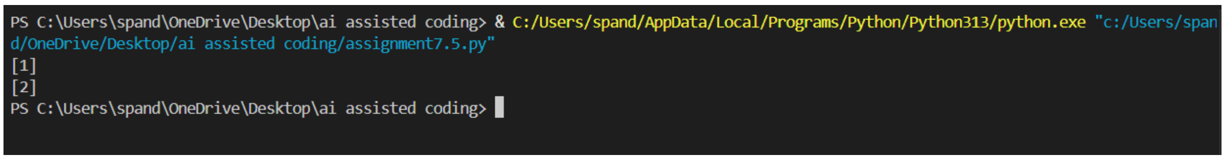
# Bug: Mutable default argument
def add_item(item, items=[]):
items.append(item)
return items
print(add_item(1))
print(add_item(2))



```
Assignment-7.5.py ×
Assignment-7.5.py > ...
#TASK 1 Bug: Mutable default argument
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

print(add_item(1))
print(add_item(2))
```

OUTPUT:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/spand/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
[1]
[2]
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> 
```

Justification:

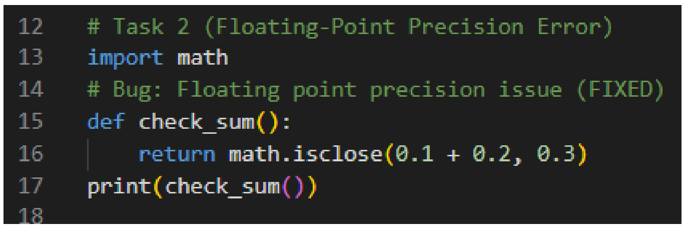
Using a mutable object (like a list) as a default argument causes the same list to be shared across function calls, leading to unexpected results. By using None as the default value and creating a new list inside the function, each call gets a fresh list. This prevents data leakage between calls and ensures correct, predictable behavior.

Task 2:

Prompt:

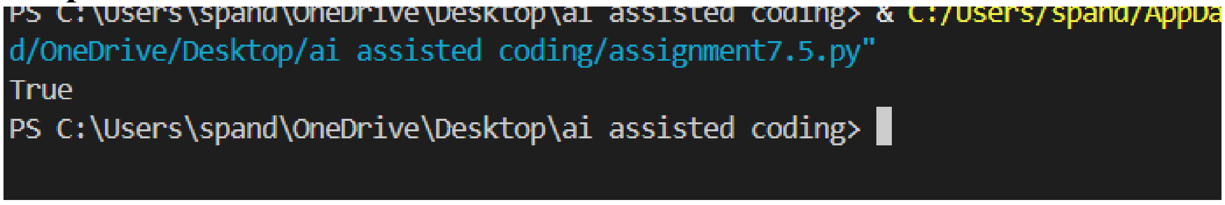
# Bug: Floating point precision issue
def check_sum():
return (0.1 + 0.2) == 0.3

print(check_sum())



```python
# Task 2 (Floating-Point Precision Error)
import math
# Bug: Floating point precision issue (FIXED)
def check_sum():
    return math.isclose(0.1 + 0.2, 0.3)
print(check_sum())
```

Output:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppDa
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
True
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

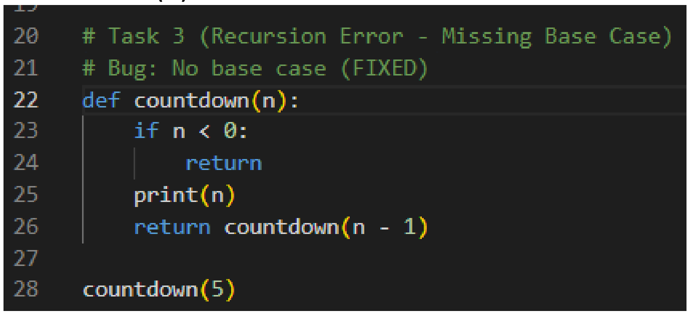
Justification:

Floating-point numbers cannot always be represented exactly in binary, so direct equality comparison may fail. Using math.isclose() compares values within a small tolerance, giving reliable and correct results.

Task 3:

Prompt:

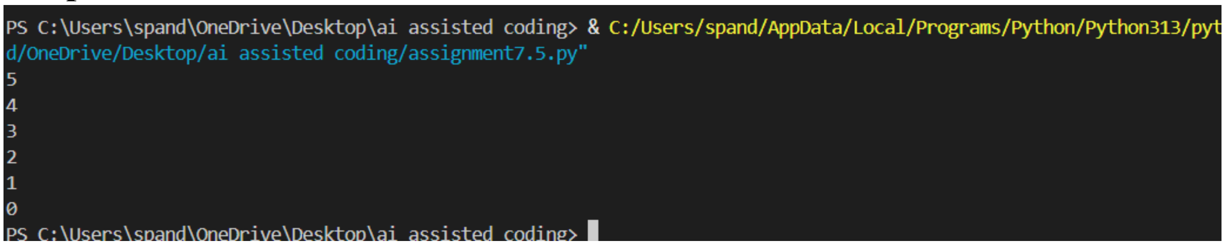
Bug: No base case
def countdown(n):
print(n)
return countdown(n-1)
countdown(5)



```python
# Task 3 (Recursion Error - Missing Base Case)
# Bug: No base case (FIXED)
def countdown(n):
    if n < 0:
        return
    print(n)
    return countdown(n - 1)

countdown(5)
```

Output:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local/Programs/Python/Python313/pyt
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
5
4
3
2
1
0
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

Justification:

Without a base case, recursion continues indefinitely and causes a stack overflow error. Adding a stopping condition (base case) ensures the function terminates safely after reaching the required limit.

Task -4:

Prompt:

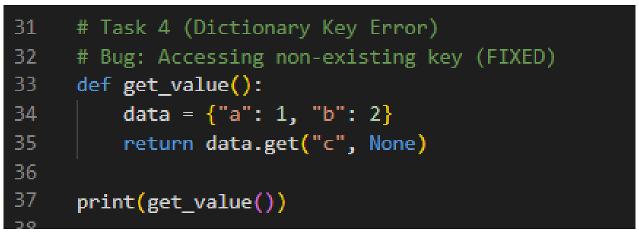
Bug: Accessing non-existing key

def get_value():

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

return data["c"]

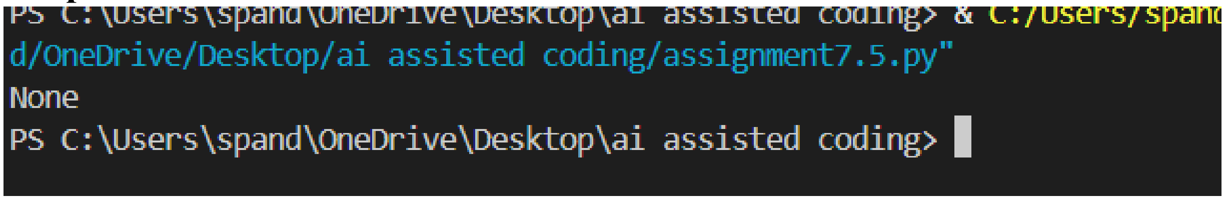
print(get_value())



```python
# Task 4 (Dictionary Key Error)
# Bug: Accessing non-existing key (FIXED)
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", None)

print(get_value())
```

Output:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
None
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

Justification:

Accessing a key that does not exist in a dictionary raises a KeyError. Using dict.get() safely handles missing keys by returning None (or a default value), preventing runtime errors and improving program robustness.

Task – 5:

Prompt:

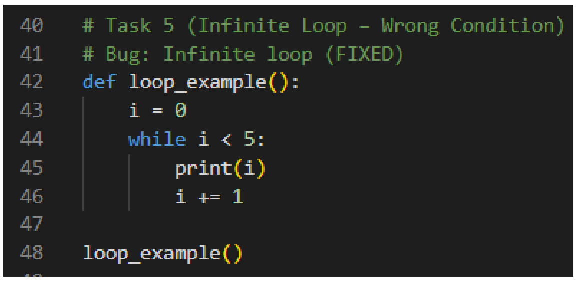
Bug: Infinite loop

def loop_example():

i = 0

while i < 5:

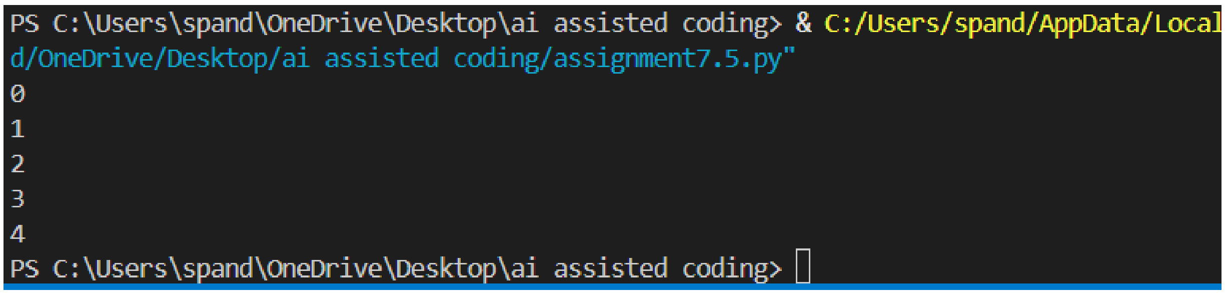
print(i)



```python
# Task 5 (Infinite Loop - Wrong Condition)
# Bug: Infinite loop (FIXED)
def loop_example():
    i = 0
    while i < 5:
        print(i)
        i += 1

loop_example()
```

Output:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
0
1
2
3
4
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

Justification:

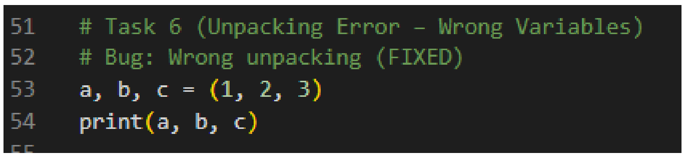
The loop became infinite because the loop variable was never updated. By incrementing i inside the while loop, the condition eventually becomes false, allowing the loop to terminate correctly.

Task - 6:

Prompt:

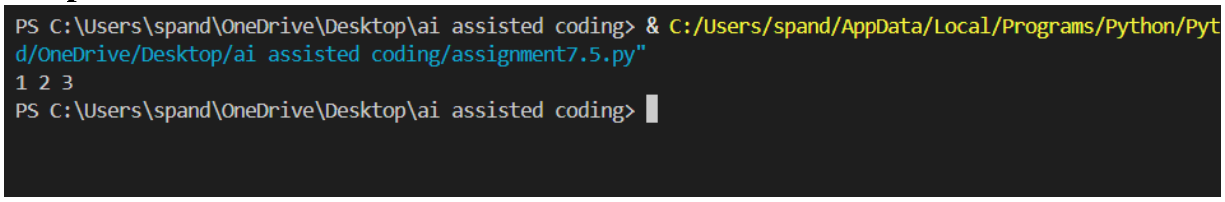
Bug: Wrong unpacking

a, b = (1, 2, 3)



```
# Task 6 (Unpacking Error – Wrong Variables)
# Bug: Wrong unpacking (FIXED)
a, b, c = (1, 2, 3)
print(a, b, c)
```

Output:



```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local/Programs/Python/Pyt
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
1 2 3
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

Justification:

Tuple unpacking requires the number of variables to match the number of values. By providing three variables for the three elements in the tuple, the unpacking succeeds without raising a ValueError.

Task - 7:

Prompt:

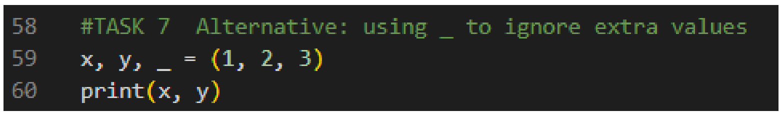
# Bug: Mixed indentation

def func():

x = 5

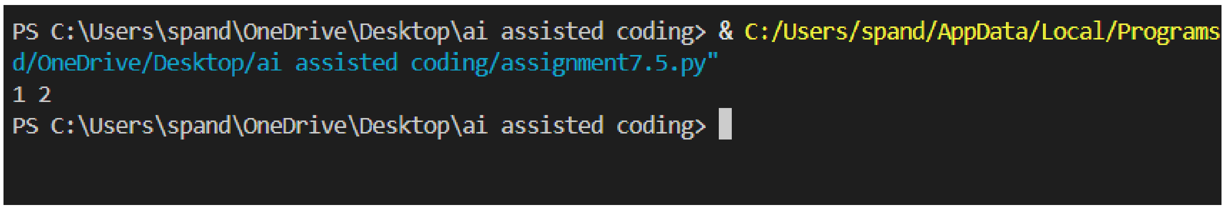
y = 10

return x+y



```
#TASK 7  Alternative: using _ to ignore extra values
x, y, _ = (1, 2, 3)
print(x, y)
```

Output:


```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local/Programs
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
1 2
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

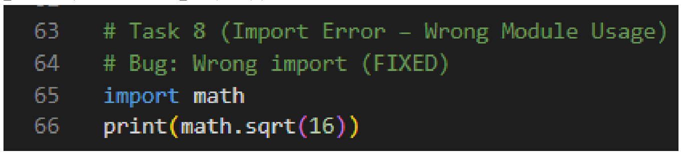
Justification:

When a tuple has more values than needed, Python allows using _ as a throwaway variable. This safely ignores extra elements, prevents unpacking errors, and keeps the code clean and readable.

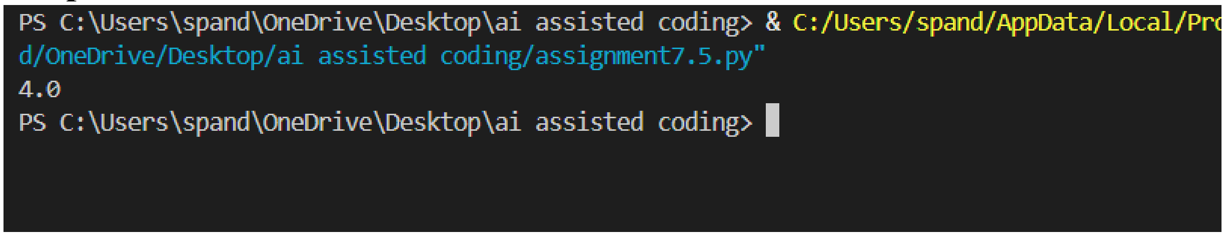
Task - 8:

Prompt:

Bug: Wrong import
import maths
print(maths.sqrt(16))


```python
# Task 8 (Import Error – Wrong Module Usage)
# Bug: Wrong import (FIXED)
import math
print(math.sqrt(16))
```

Output:


```
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding> & C:/Users/spand/AppData/Local/Pro
d/OneDrive/Desktop/ai assisted coding/assignment7.5.py"
4.0
PS C:\Users\spand\OneDrive\Desktop\ai assisted coding>
```

Justification:

The error occurred because maths is not a valid Python standard library module. Importing the correct math module provides access to sqrt(), ensuring the program runs successfully and returns the correct result.