AI ASSISTANT CODING

ASSIGNMENT-7.5

Name: N. Paul Benjamin

Ht.no: 2303A51116

Batch: 03

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: Bug Mutable default argument

Code:

ass_7.5.py > add_item

```python
# 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))
```

ass_7.5.py > ...

```python
# 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
```

Output:

```
ai_coding/ass_7.5.py
[1]
[2]
```

Code Explanation: This function adds an item to a list.

items is set to None to avoid using a shared list as a default argument.

If items is None, a new empty list is created.

The item is then added to the list using append().

Each function call gets a fresh list, so results don't mix.

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: Floating point precision issue

Code :

```
# Bug: Floating point precision issue
def check_sum():
return (0.1 + 0.2) == 0.3
    return abs((0.1 + 0.2) - 0.3) < 1e-10  # Use a small tolerance for floating point comparison
print(check_sum())
```

```
# Bug: Floating point precision issue
def check_sum():
    return abs((0.1 + 0.2) - 0.3) < 1e-10  # Use a small tolerance for floating point comparison
print(check_sum())
```

Output:

```
True
```

Code Explanation:

This code checks floating-point precision safely.
In Python, 0.1 + 0.2 is not exactly equal to 0.3 due to rounding errors.
So the code subtracts 0.3 and takes the absolute value of the difference.
It compares this difference with a very small number (1e-10) called tolerance.
If the difference is smaller than the tolerance, it returns True.

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: No base case

Code:

```python
# Bug: No base case
def countdown(n):
    if n == 0:
        return
    print(n)
    countdown(n-1)
countdown(5)
```

Output:

```
5
4
3
2
1
```

Code Explanation: This program demonstrates recursion using a countdown function. The function countdown(n) takes a number as input and checks a base case where n == 0, at which point it stops executing further. If the base case is not met, it prints the current value of n and then calls itself with n-1. This process repeats, reducing the value of n each time. When the value finally reaches 0, the function returns and recursion ends. Calling countdown(5) therefore prints numbers from 5 down to 1.

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: Accessing non-existing key

Code:

ass_7.5.py > ...

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

ass_7.5.py > get_value

```python
# Bug: Accessing non-existing key
def get_value():
    data = {"a": 1, "b": 2}
    return data.get("c", "Key not found")
print(get_value())
```

Output:

```
Key not found
```

Code Explanation: This program defines a function get_value() that works with a dictionary called data containing keys "a" and "b". Inside the function, the get() method is used to safely access the key "c", which does not exist in the dictionary. Instead of raising an error, get() returns the default message "Key not found". The function then returns this message. When print(get_value()) is executed, it displays Key not found as the output.

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)

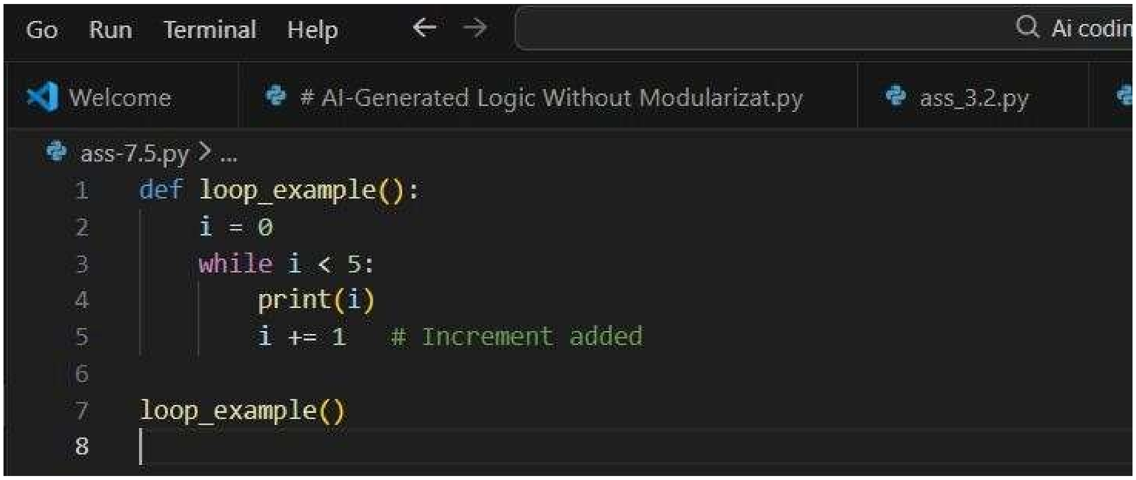
Expected Output: Corrected loop increments i.

Prompt: Infinite loop

Code:

ass_7.5.py > i

```
# Bug: Infinite loop
def loop_example():
i = 0
while i < 5:
print(i)
    i += 1
```



Output:

Code Explanation: This program defines a function loop_example() that demonstrates a while loop. The variable i is initialized to 0, and the loop runs as long as i is less than 5. Inside the loop, the current value of i is printed and then incremented by 1 to avoid an infinite loop. The function is called at the end, so it prints the numbers 0, 1, 2, 3, and 4 in order and then stops.

Task 6 (Unpacking Error – Wrong Variables)

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

fix it.

Bug: Wrong unpacking

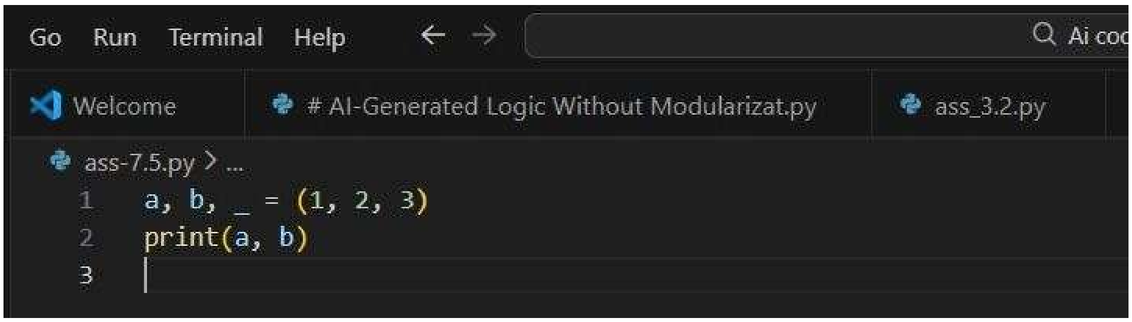
a, b = (1, 2, 3)

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

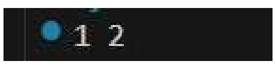
Prompt:

Code:

Bug: Wrong unpacking
a, b = (1, 2, 3)
Expected Output: Correct unpacking or using _ for extra values.



Output:



Code Explanation: This program demonstrates tuple unpacking in Python. The tuple (1, 2, 3) is unpacked into variables where a gets the value 1, b gets the value 2, and the underscore _ is used to ignore the third value 3. The underscore is a common convention in Python for values that are not needed. Finally, print(a, b) displays the values of a and b, which are 1 and 2.

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

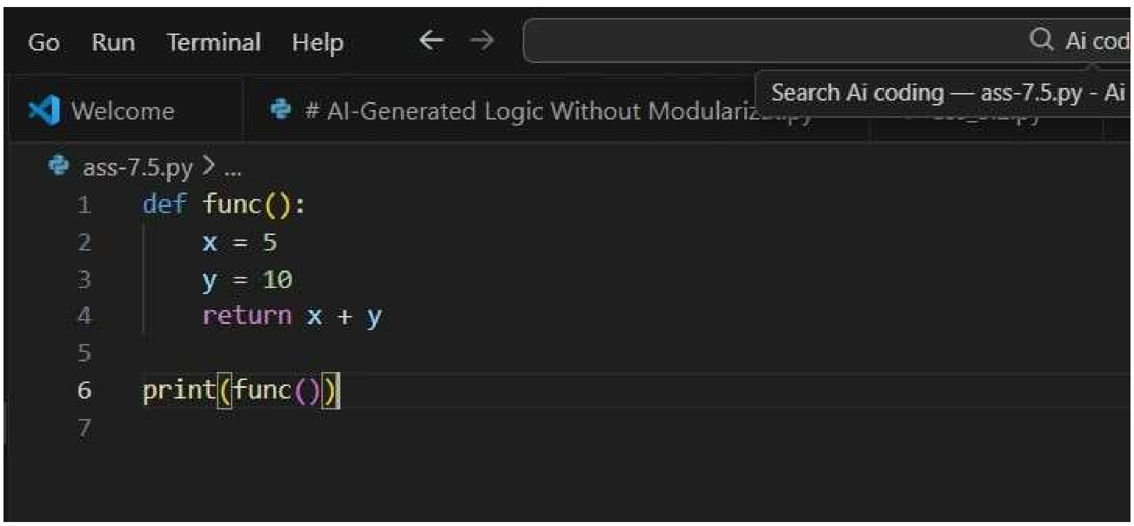
Expected Output : Consistent indentation applied.

Prompt : Mixed indentation

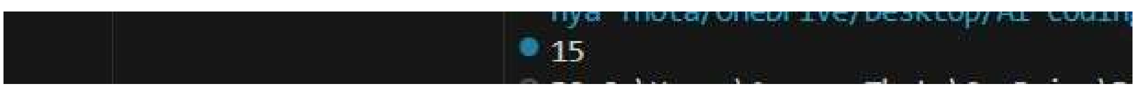
Code:

```
# Bug: Mixed indentation
def func():
    x = 5
        y = 10
    return x+y
```

Expected Output : Consistent indentation applied.



Output:



Code explanation: This Python code defines a function named func() that performs a simple calculation.

Inside the function, two variables x and y are assigned the values 5 and 10.

The function returns the sum of these two variables using return x + y.

Outside the function, func() is called inside the print() statement.

As a result, the output displayed on the screen is 15.

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:

Bug: Wrong import

import maths

print(maths.sqrt(16))

Expected Output: Corrected to import math

Code:

Go Run Terminal Help

Welcome # AI-Generated Logic Without Modulariz... Search AI

ass-7.5.py

```python
import math

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

Output:

```
4.0
```

Code explanation:

This Python code first imports the built-in math module, which provides mathematical functions.

The math.sqrt() function is used to calculate the square root of a number.

Here, the value 16 is passed as an argument to math.sqrt().

The function computes the square root of 16, which is 4.

The print() statement displays the result 4.0 on the screen.