

AI ASSISTED CODING

Name: P.Swaran Raj

Roll.No: 2303A52247

Batch: 37

Assignment- 7.3

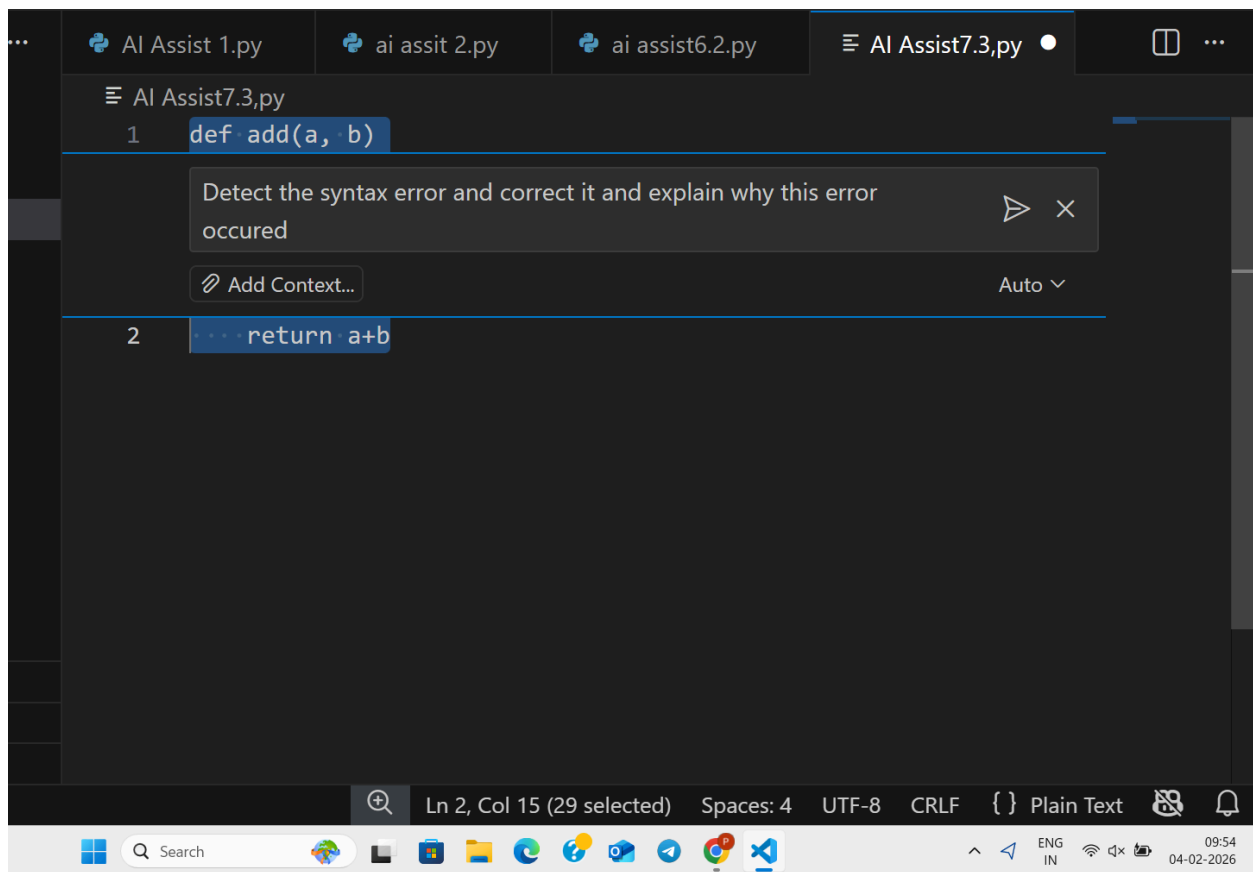
Task 1: Fixing Syntax Errors

Scenario:

You are reviewing a Python program where a basic function definition contains a syntax error.

Prompt Given for modification:

“#Detect the syntax error and correct it and explain”



```
1 def add(a, b):
2     return a+b
```

Error Explanation:

The code has a syntax error. In Python, function definitions must end with a colon (:) after the parameter list.

Output:

The corrected version should be:

```
def add(a, b):
    return a + b
```

Comments:

Ai fixed and explained the syntax error.

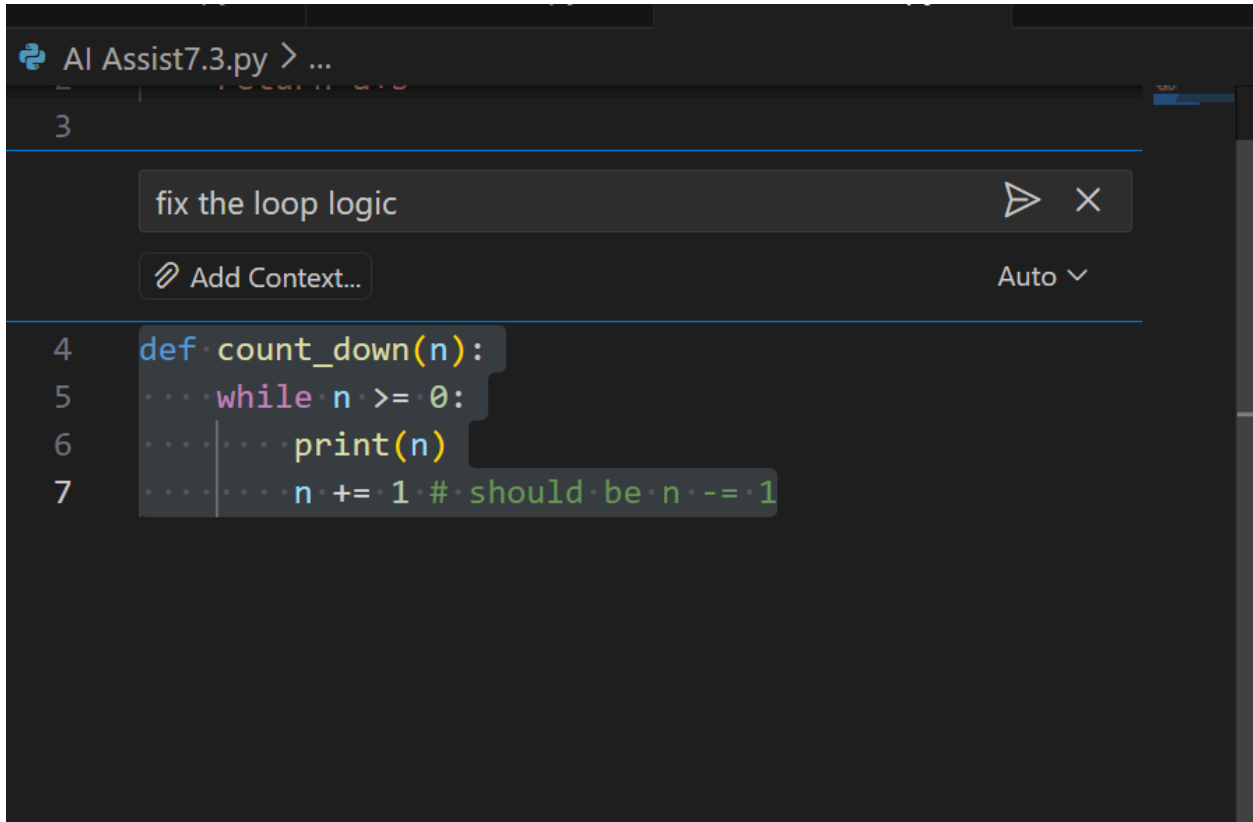
Task 2: Debugging Logic Errors in Loops

Scenario:

You are debugging a loop that runs infinitely due to a logical mistake.

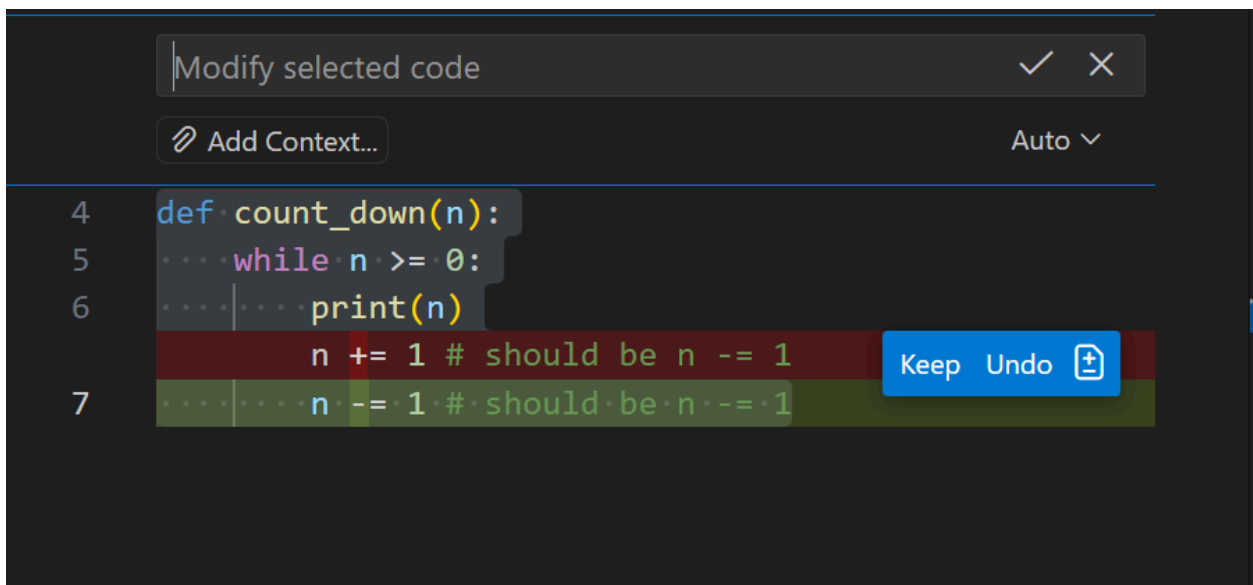
Prompt Given for modification:

“#Fix the loop logic.”



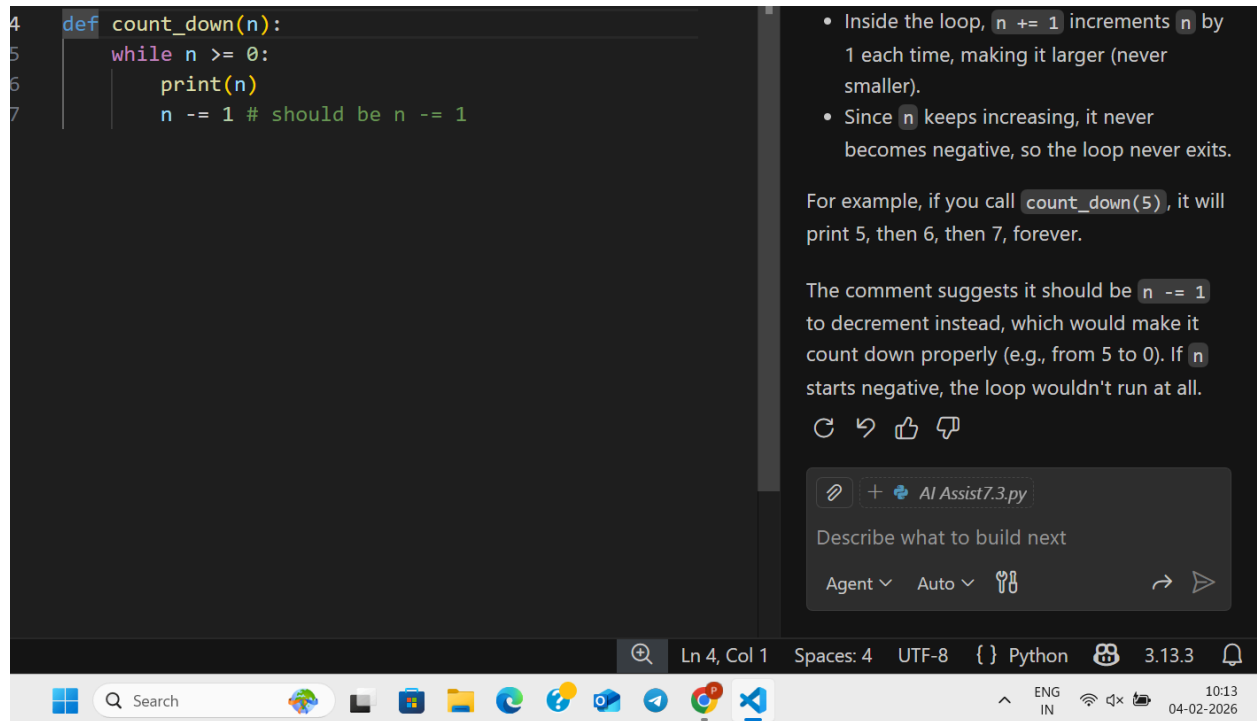
The screenshot shows a code editor with a file named 'AI Assist7.3.py'. The code defines a function `count_down(n)` with a `while` loop that runs as long as `n >= 0`. Inside the loop, it prints `n` and then increments `n` by 1. A comment in the code says `# should be n -= 1`. The AI Assist interface is open with the prompt 'fix the loop logic'.

```
3  
4 def count_down(n):  
5     while n >= 0:  
6         print(n)  
7         n += 1 # should be n -= 1
```



The screenshot shows the same code editor with the AI Assist interface open. The prompt is 'Modify selected code'. The code is highlighted, and the AI Assist interface shows the corrected code with a 'Keep' button.

```
4 def count_down(n):  
5     while n >= 0:  
6         print(n)  
7         n -= 1 # should be n -= 1
```



Error Explanation:

this code will go into an infinite loop if `n` starts as a non-negative integer (0 or greater).

- The while loop condition is `n >= 0`, so it runs as long as `n` is 0 or positive.
- Inside the loop, `n += 1` increments `n` by 1 each time, making it larger (never smaller).
- Since `n` keeps increasing, it never becomes negative, so the loop never exits.

Output:

it should be `n -= 1` to decrement instead, which would make it count down properly (e.g., from 5 to 0). If `n` starts negative, the loop wouldn't run at all

```
def count_down(n):  
    while n >= 0:  
        print(n)  
        n -= 1
```

Comments:

AI fixed the loop logic, instead of incrementing it went for decrementing that made count down properly and if negative number starts the loop will end.

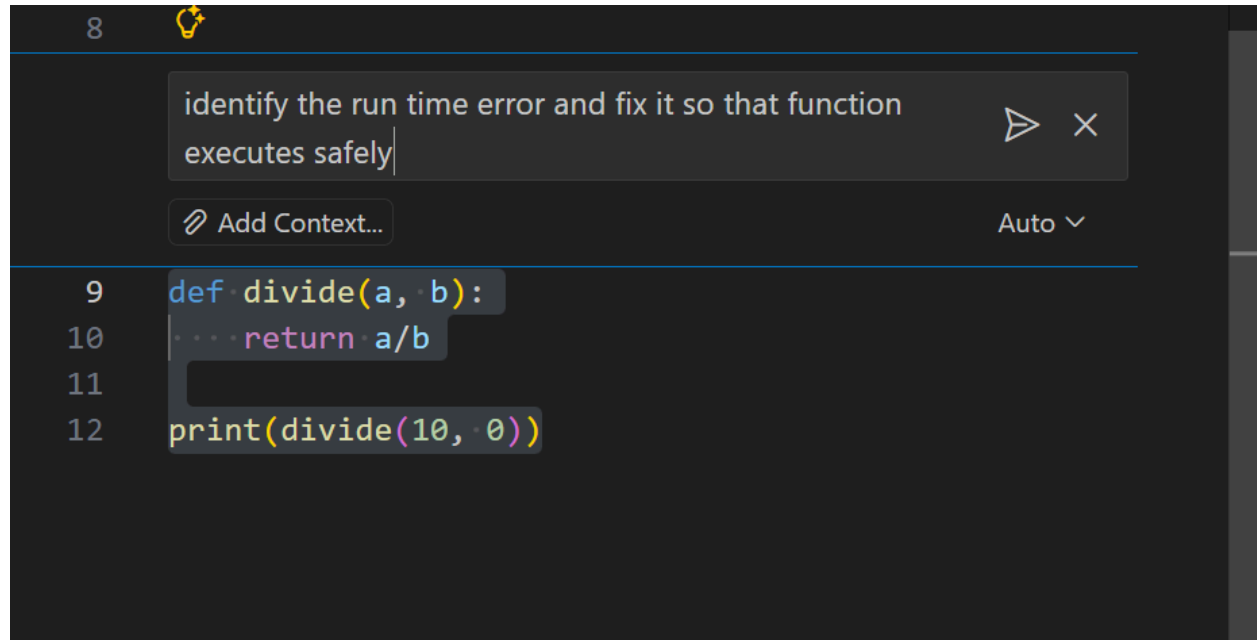
Task 3: Handling Runtime Errors (Division by Zero)

Scenario:

A Python function crashes during execution due to a division by zero error.

Prompt Given for modification:

“#identify the run time error and fix it so that function executes safely.”



The screenshot shows a code editor with a dark theme. At the top, there is a toolbar with a magnifying glass icon and a dropdown menu. The dropdown menu is open, showing the text "identify the run time error and fix it so that function executes safely" with a search icon and a close icon. Below the dropdown menu, there is a button labeled "Add Context..." and a dropdown menu labeled "Auto". The code editor displays the following Python code:

```
8  
9 def divide(a, b):  
10     return a/b  
11  
12 print(divide(10, 0))
```

```
Modify selected code ✓ ✕
Add Context... Auto ▾

9 def divide(a, b):
10     if b == 0:
11         return "Error: Cannot divide by zero"
12     return a/b
13
14 print(divide(10, 0))
```

```
Error: Cannot divide by zero
None
```

```
9 def divide(a, b):
10     if b == 0:
11         print("Error: Cannot divide by zero")
12         return None
13     return a/b
14
15 print(divide(10, 2))
```

5.0

Error Explanation:

The code will raise a `ZeroDivisionError` at runtime when you try to divide by zero. In Python, division by zero is mathematically undefined and not allowed, so the interpreter throws an exception instead of returning a result.

Output:

When you run `print(divide(10, 0))`, you'll see output like:

ZeroDivisionError: division by zero

To handle this safely, you could add error checking in the function:

```
def divide(a, b):  
    if b == 0:  
        return "Cannot divide by zero"  
    return a / b
```

o/p>Error: Cannot divide by zero

None

```
def divide(a, b):  
    if b == 0:  
        print("Error: Cannot divide by zero")  
        return None  
    return a/b  
  
print(divide(10, 2))
```

o/p:5.0

Comments:

AI fixed the error and explained it , provided why it can't divide by zero and when we add 2 instead of 0 it gave the output 5.0 it handled the runtime error and executed safely

Task 4: Debugging Class Definition Errors

Scenario:

You are given a faulty Python class where the constructor is incorrectly defined

Prompt Given for modification:

“#identify the issue in the __init__ constructor and correct the class definition.”

identify the issue in the `__init__` constructor and correct the class definition

Add Context... Auto

```
17 class Rectangle:
18     def __init__(self, length, width):
19         self.length = length
20         self.width = width
```

Modify selected code

Add Context... Keep (Ctrl+Enter)

```
17 class Rectangle:
18     def __init__(self, length, width):
19         self.length = length
20         self.width = width
21
22     def __str__(self):
23         return f"Rectangle(length={self.length}, width={self.width})"
```

```
17 class Rectangle:
18     def __init__(self, length, width):
19         self.length = length
20         self.width = width
21
22     def __str__(self):
23         return f"Rectangle(length={self.length}, width={self.width})"
```

Error Explanation:

The provided code defines a `Rectangle` class with an `__init__` method that correctly initializes `length` and `width` attributes. The AI detected that **self**

was missing in the constructor. It corrected the class definition by adding **self** to properly reference object attributes.

Output:

However, the class is incomplete for practical use. A typical Rectangle class would include methods to calculate properties like area or perimeter. For example, you could add:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width})"
```

Comments:

AI fixed the runtime error, `__init__` constructor and class definition and it explained the error.

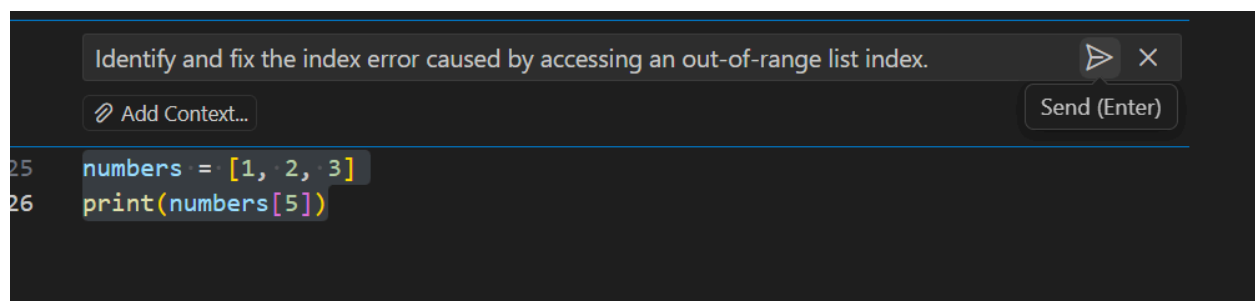
Task 5: Resolving Index Errors in Lists

Scenario:

A program crashes when accessing an invalid index in a list.

Prompt Given for modification:

"#Identify and fix the index error caused by accessing an out-of-range list index."



The screenshot shows a code editor interface. At the top, there is a prompt bar with the text "Identify and fix the index error caused by accessing an out-of-range list index." and a "Send (Enter)" button. Below the prompt bar, there is a code snippet with line numbers 25 and 26. The code is as follows:

```
25 numbers = [1, 2, 3]
26 print(numbers[5])
```

```
25 numbers = [1, 2, 3]
26 print(numbers[5])
    print(numbers[2])
```

```
25 numbers = [1, 2, 3]
26 print(numbers[2])
```

Error Explanation:

Lists in Python are zero-indexed (start at 0), so a list with 3 items goes from index 0 to 2. Asking for index 5 is like asking for a 6th item that isn't there. The AI identified the index error and suggested bounds checking. Safe access was implemented using length checks or exception handling.

Output:

The program accesses list elements safely without crashing.

```
numbers = [1, 2, 3]
print(numbers[2])
```

o/p:3

Comments:

AI provides effective solutions for common runtime errors. Validating list indices improves code reliability.

Conclusion:

This lab demonstrates how AI tools assist in systematically identifying and fixing syntax, logic, and runtime errors in Python programs.