AI ASSISTED CODING
ASSIGNMENT-7.3
ROLL NO.2303A510C8
B-27

Task 1: Fixing Syntax Errors

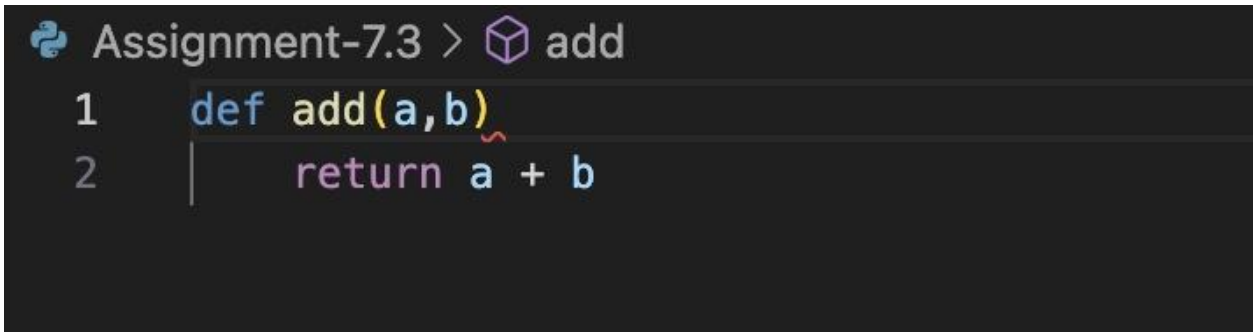
Scenario
You are reviewing a Python program where a basic function definition contains a syntax error.

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

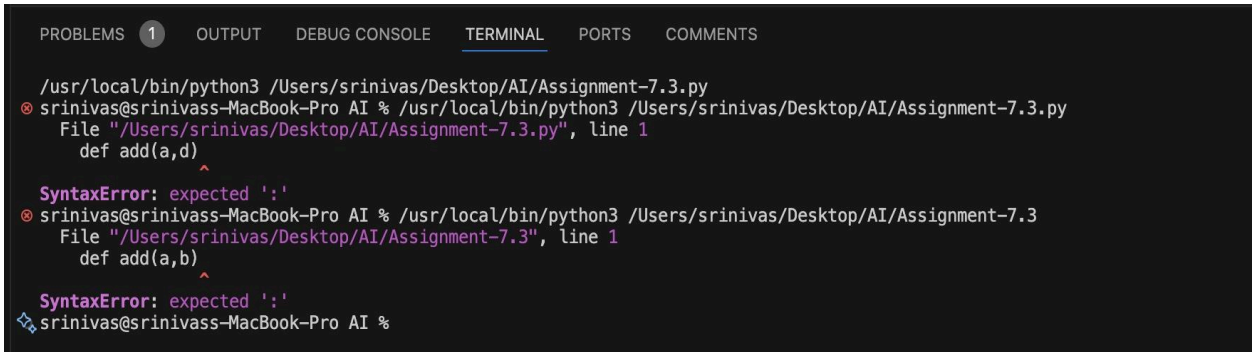
Requirements
- Provide a Python function add(a, b) with a **missing colon**
- Use an AI tool to detect the syntax error
- Allow AI to correct the function definition
- Observe how AI explains the syntax issue

Expected Output
- Corrected function with proper syntax
- Syntax error resolved successfully
- AI-generated explanation of the fix

Wrong CODE:



Output:



Explanation:

The error occurs because the colon (`:`) is missing at the end of the function definition. In Python, a colon is required after the function header to indicate the start of the function body.

Here's the corrected code:

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

Explanation:

1. **Function Definition**: The `def` keyword is used to define a function. It is followed by the function name (`add`), parentheses containing the parameters (`a, b`), and a colon (`:`).
2. **Colon**: The colon tells Python that the indented block of code that follows is the body of the function.
3. **Return Statement**: The `return` keyword specifies the value that the function will output when called.

Additional Suggestion:

To make the function more robust, you could add type hints to indicate the expected input and output types:

```
def add(a: int, b: int) -> int:
    return a + b
```

This indicates that `a` and `b` are expected to be integers, and the function will return an integer. While Python won't enforce these types, they serve as documentation and can help tools like linters or IDEs catch potential errors.

Fixed Code:

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

# Example usage
result = add(3, 5)
print("The sum is:", result)

```

Output:

```
SyntaxError: expected ':'
srinivas@srinivass-MacBook-Pro AI % /usr/local/bin/python3 /Users/srinivas/Desktop/AI/Assignment-7.3
The sum is: 8
srinivas@srinivass-MacBook-Pro AI %
```

Justification:

This task demonstrates how AI tools can quickly identify and resolve basic syntax errors in Python programs. Syntax errors, such as missing colons in function definitions, prevent code execution entirely. Using AI for detection helps reduce manual debugging time and provides clear explanations of language rules. This improves understanding of Python syntax and reinforces correct coding practices.

Task 2: Debugging Logic Errors in Loops

File preview

Scenario

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

```python
def count_down(n):
    while n >= 0:
        print(n)
        n += 1  # Should be n -= 1
```

Requirements

- Provide a loop with an **increment or decrement error**
- Use AI to identify the cause of infinite iteration
- Let AI fix the loop logic
- Analyze the corrected loop behavior

Expected Output

- Infinite loop issue resolved
- Correct increment/decrement logic applied
- AI explanation of the logic error

Wrong Code:

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

Output:

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipython-input-2219605999.py in <cell line: 0>()
      4     n += 1
      5
----> 6 count_down(5)

 2 frames
/usr/local/lib/python3.12/dist-packages/ipykernel/iostream.py in _is_master_process(self)
    436
    437     def _is_master_process(self):
--> 438         return os.getpid() == self._master_pid
    439
    440     def set_parent(self, parent):

KeyboardInterrupt:
```

Explanation:

```
<> def count_down(n):
```

The code defines a function called `count_down` that takes one argument, `n`. It then enters a `while` loop that continues as long as `n` is greater than or equal to 0. Inside the loop, it prints the current value of `n` and then increments `n` by 1. The function is called with `count_down(5)`. This code creates an infinite loop because `n` starts at 5 and is always incremented, so the condition `n >= 0` will always be true.

Fixed code:

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

count_down(5)
```

Output:

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

Justification:

This task focuses on identifying logical errors that cause unintended infinite loops. Such errors are harder to detect than syntax errors because the program runs but behaves incorrectly. AI-assisted debugging helps analyze loop conditions and variable updates, ensuring proper termination. This task highlights the importance of correct logic flow and demonstrates how AI can explain and correct algorithmic mistakes.

```
# Debug the following code
def divide(a, b):
    return a / b

print(divide(10, 0))
```

Requirements

- Provide a function that performs division without validation
- Use AI to identify the runtime error
- Let AI add try-except blocks for safe execution
- Review AI's error-handling approach

Expected Output

- Function executes safely without crashing
- Division by zero handled using try-except
- Clear AI-generated explanation of runtime error handling

Wrong code:

```
def divide(a, b):
 return a / b
print(divide(10,0))
```

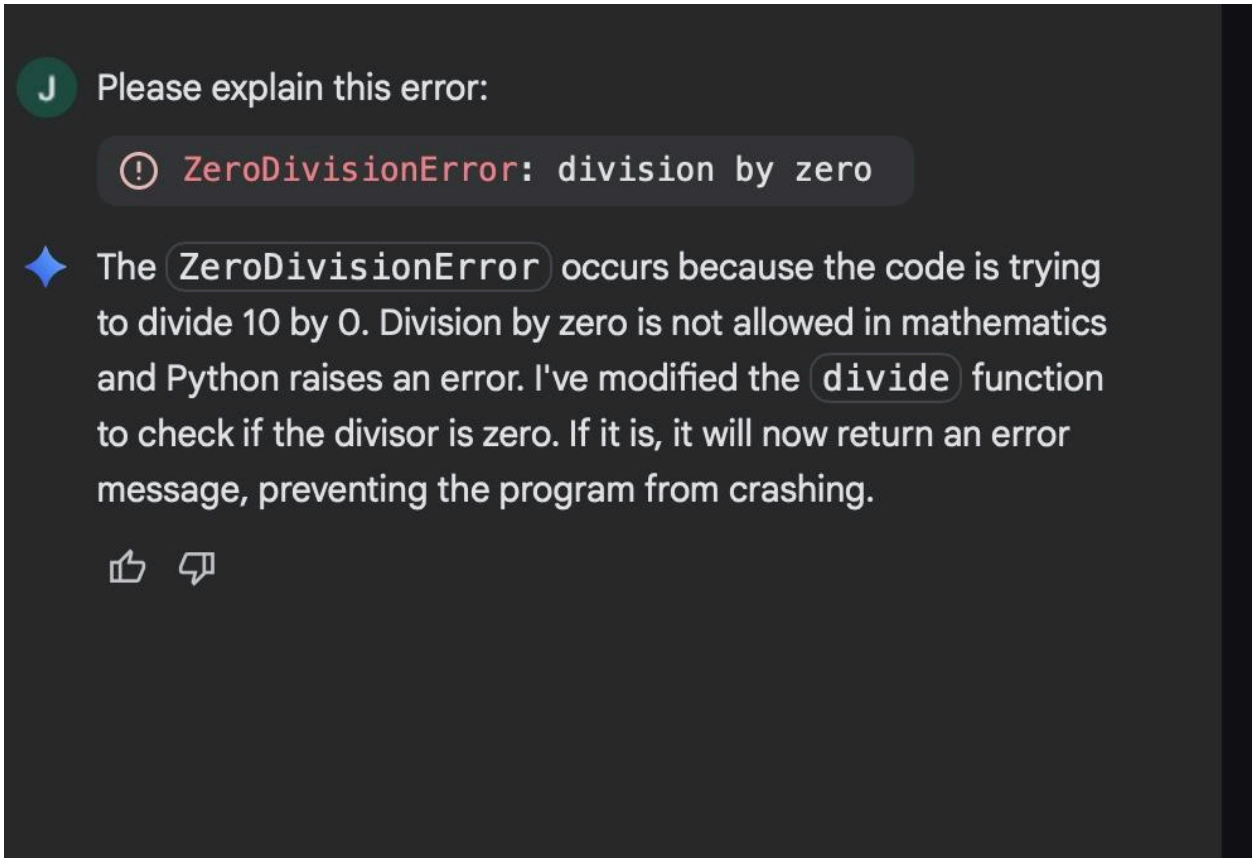
output:

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-3300205478.py in <cell line: 0>()
      1 def divide(a, b):
      2   return a / b
----> 3 print(divide(10,0))

/tmp/ipython-input-3300205478.py in divide(a, b)
      1 def divide(a, b):
----> 2   return a / b
      3 print(divide(10,0))

ZeroDivisionError: division by zero
```

Explanation:



Prompt:
Fix the code

Fixed code:

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

Output:

```
Error: Cannot divide by zero
```

Justification:

Runtime errors occur during program execution and can cause unexpected crashes. This task illustrates how AI detects runtime exceptions such as division by zero and suggests structured error handling using try-except blocks. Implementing exception handling improves program robustness and reliability. AI explanations help understand defensive programming techniques and safe execution strategies.

Task 4: Debugging Class Definition Errors

Scenario

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

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

File preview

Requirements

- Provide a class definition with **missing self-parameter**
- Use AI to identify the issue in the __init__() method
- Allow AI to correct the class definition
- Understand why self is required

Expected Output

- Corrected __init__() method
- Proper use of self in class definition
- AI explanation of object-oriented error

Wrong code:

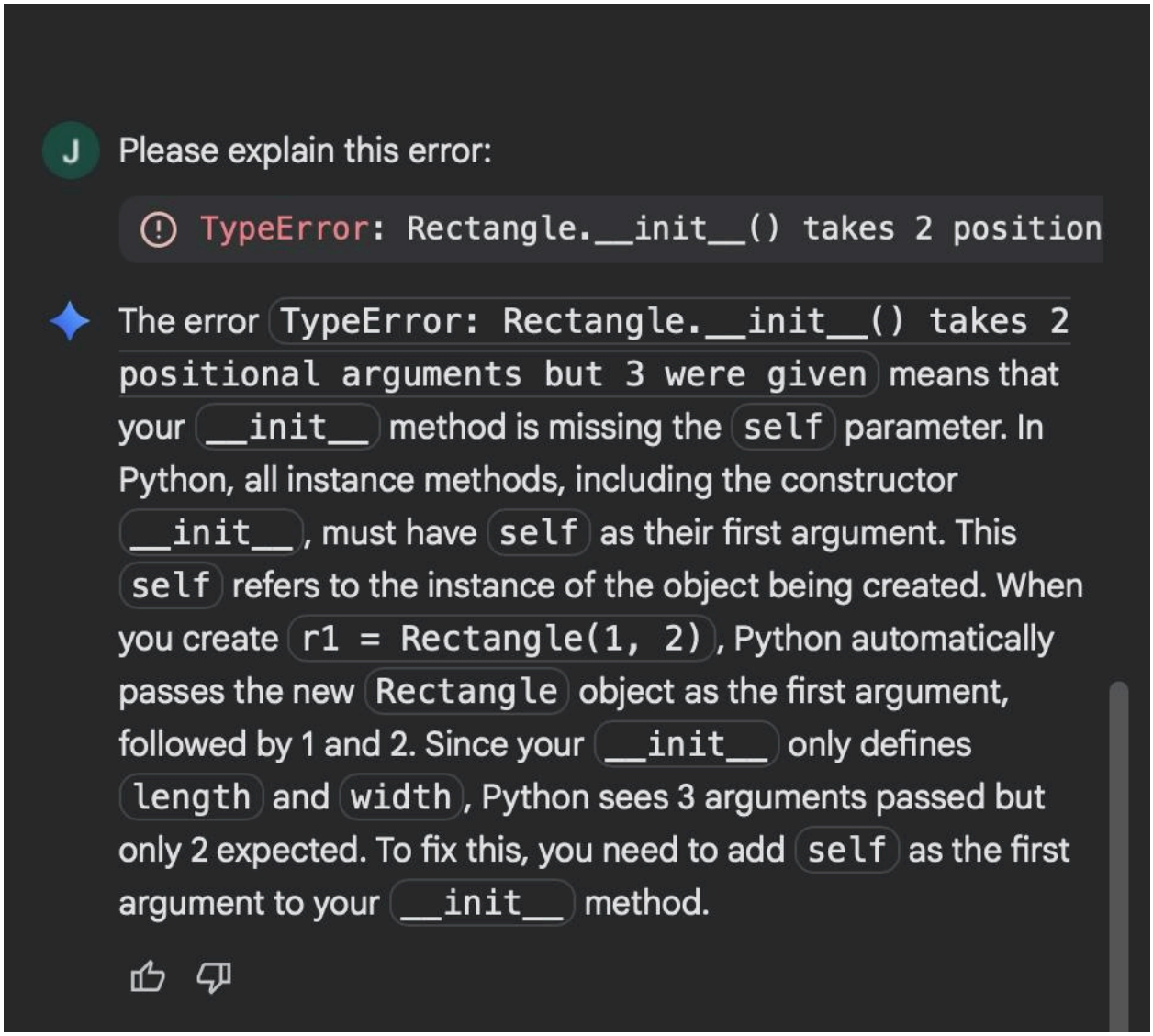
```
class Rectangle():
 def __init__(length, width):
   self.length = length
   self.width = width
r1 = Rectangle(1, 2)
print(r1.length)
print(r1.width)
```

Output:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-166674501.py in <cell line: 0>()
      3     self.length = length
      4     self.width = width
----> 5 r1 = Rectangle(1, 2)
      6 print(r1.length)
```

```
      7 print(r1.width)
```

```
TypeError: Rectangle.__init__() takes 2 positional arguments but 3 were given
```



Fixed code:

```
class Rectangle():
 def __init__(self, length, width):
   self.length = length
   self.width = width
r1 = Rectangle(1, 2)
print(r1.length)
print(r1.width)
```

Output:

```
1
2
```

Justification:

This task addresses common object-oriented programming mistakes, specifically incorrect constructor definitions. The absence of the self parameter leads to runtime errors and improper object initialization. AI-assisted debugging identifies this issue and explains the role of self in accessing instance variables. This task strengthens understanding of Python's class structure and object behavior.

Task 5: Resolving Index Errors in Lists

File preview

Scenario

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

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

Requirements

- Provide code that accesses an **out-of-range list index**
- Use AI to identify the Index Error
- Let AI suggest safe access methods
- Apply bounds checking or exception handling

Expected Output

- Index error resolved
- Safe list access logic implemented

Wrong code:

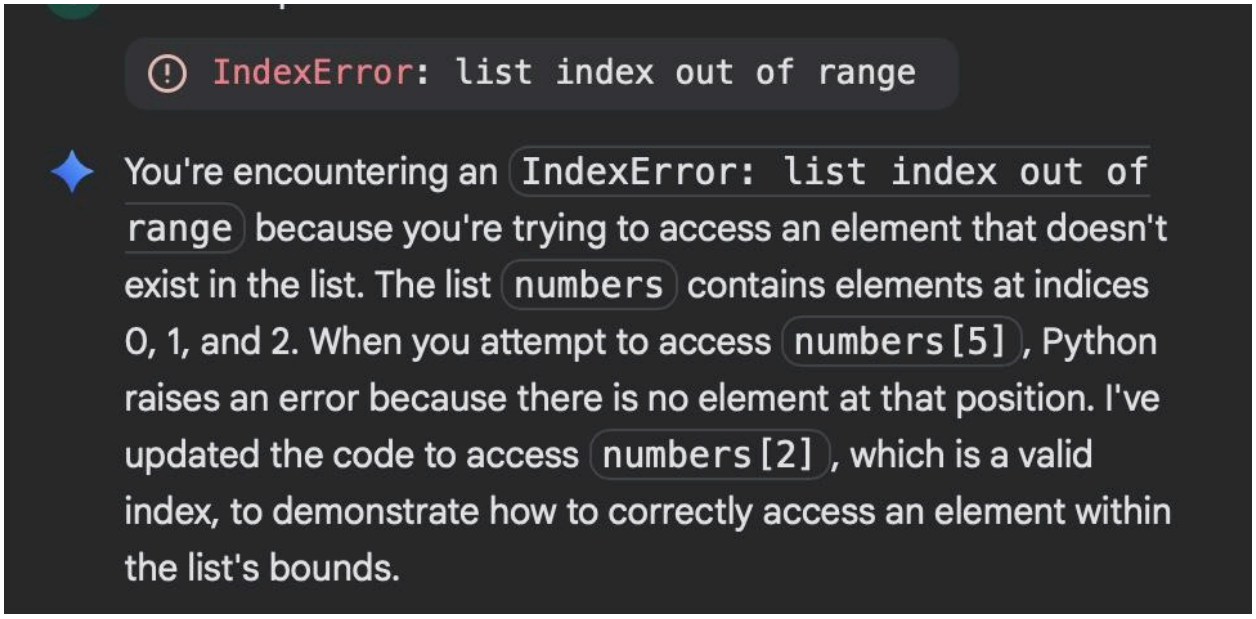
```
numbers = [1, 2, 3]
print(numbers[5])
```

Output:

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipython-input-1537628302.py in <cell line: 0>()
      1 numbers = [1, 2, 3]
----> 2 print(numbers[5])

IndexError: list index out of range
```

Explanation:



Fixed Code:

```
numbers = [1, 2, 3]
try:
 print(numbers[5])
except:
 print("Error: Index out of range")
```

Output:

```
Error: Index out of range
```

Justification:

Index errors occur when attempting to access elements outside the valid range of a list. This task demonstrates how AI identifies such errors and recommends safe access techniques like bounds checking or exception handling. Proper list handling prevents program crashes and ensures data safety. The AI explanations promote writing more reliable and error-resistant code.