

AI Assistant Coding **Assignment-7.3**

Name of Student : G.SriSahith
Enrollment No. : 2303A52243
Batch : 37

Lab 7: Error Debugging with AI

Course: AI Assisted Coding (23CS002PC304)

Assignment: 7.3

Theme: Systematic approaches to finding and fixing bugs

Task 1: Fixing Syntax Errors

Description

The goal of this task is to identify and resolve a syntax error in a simple Python function. The provided code defines an addition function but is missing a specific syntactic element required by Python's grammar.

Prompt

"I have written a Python function to add two numbers, but I am getting a syntax error. Can you identify what is missing and fix the code?"

Code

Faulty Code:

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

Corrected Code:

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

# Testing the function
result = add(5, 3)
print(f"The sum is: {result}")
```

Output

```
s/AI Assistant Coding/Assign-7.py"
● The sum is: 8
```

Explanation

Think of the colon (:) in Python like a gate opening to a path. Without it, Python doesn't know you are about to list the instructions for the function. It's a small punctuation mark, but it's essential syntax—like putting a period at the end of a sentence. By adding the colon, we told Python, "Okay, the name of the function is done; here comes the code for what it actually does."

Task 2: Debugging Logic Errors in Loops

Description

This task involves debugging a logic error in a while loop intended to count down. The original code creates an infinite loop because the counter is incremented instead of decremented, preventing the termination condition from ever being met.

Prompt

"My countdown function is running forever instead of stopping at 0. Please look at the loop logic and fix the infinite loop issue."

Code

Faulty Code:

```
def count_down(n): while n
    >= 0: print(n) n += 1 #
    Logic error here
```

Corrected Code:

```
def count_down(n):
    while n >= 0:
        print(n)
        n -= 1 # Corrected to decrement

# Testing the function
count_down(3)
```

Output

```
s/AI Assistant Coding/Assign-7.py"
```

```
3
2
1
0
```

Explanation

Imagine you are on the 3rd floor of a building and want to go down to the ground floor (0). In the original code, instead of going down, the loop was accidentally pressing the "Up" button (`n += 1`). This meant you went to floors 4, 5, 6, and so on, moving further away from 0. Since you never hit 0, the loop ran forever. Changing the `+` to a `-` fixed the direction so we actually reach the destination and stop.

Task 3: Handling Runtime Errors (Division by Zero)

Description

This task focuses on handling runtime errors that crash the program, specifically the "Division by Zero" error. The objective is to make the function robust using exception handling so the program can handle bad input gracefully.

Prompt

"I have a division function that crashes when the second number is zero. Can you modify this code to use try-except blocks to handle the ZeroDivisionError gracefully?"

Code

Faulty Code:

```
def divide(a, b):
    return a / b

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

Corrected Code:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Error: Cannot divide by zero."

# Testing the function
print(divide(10, 2))
print(divide(10, 0))
```

Output

```
s/AI Assistant Coding/Assign-7.py"
5.0
Error: Cannot divide by zero.
```

Explanation

Dividing by zero is a universal "no-no" in math—it's undefined and causes computer programs to crash instantly. The fix here is like adding a safety net. We wrapped the code in a try...except block. We told Python, "Go ahead and *try* to do this division, but if you catch a zero error, don't crash! Just show this friendly error message instead." This keeps the program running smoothly even when bad input happens.

Task 4: Debugging Class Definition Errors

Description

This task involves fixing an Object-Oriented Programming (OOP) error. The provided class constructor is missing the `self` parameter, which is required in Python to refer to the specific instance of the object being created.

Prompt

"I am getting an error in my Python class constructor. It seems I defined `_init_` incorrectly.
Can you check why the variables aren't being assigned and fix the class definition?"

Code

Faulty Code:

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

Corrected Code:

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

# Testing the class
rect = Rectangle(10, 5)
print(f"Rectangle: Length={rect.length}, Width={rect.width}")
```

Output

```
s/AI Assistant Coding/Assign-7.py"
> Rectangle: Length=10, Width=5
```

Explanation

The `self` keyword is one of the trickier parts of Python classes. Think of it like a nametag or an ID card. When you create a new `Rectangle` object, Python needs to know *whose* length and width we are talking about. Are we setting the size for Rectangle A or Rectangle B? By adding `self`, we tell the code, "Use the length and width for *this specific* object I'm building right now." Without it, the variables are just floating around with nowhere to stick.

Task 5: Resolving Index Errors in Lists

Description

The final task addresses a common runtime error: `IndexError`. This occurs when trying to access a list element at a position that does not exist. The goal is to implement safe access methods using bounds checking.

Prompt

"I have a list with 3 numbers, but my code crashes when I try to print an index that doesn't exist. Can you suggest a way to access list elements safely using an if-condition or exception handling?"

Code

Faulty Code:

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

Corrected Code:

```
numbers = [1, 2, 3]
index_to_access = 5

# Method: Bounds Checking
if 0 <= index_to_access < len(numbers):
    print(numbers[index_to_access])
else:
    print(f"Index {index_to_access} is out of range.")
```

Output

```
s/AI Assistant Coding/Assign-7.py"
Index 5 is out of range.
```

Explanation

Lists in Python are numbered starting at 0. So, a list with 3 items ([1, 2, 3]) has slots 0, 1, and 2. Trying to grab slot 5 is like trying to open drawer #5 in a cabinet that only has 3 drawers—it doesn't exist! Python throws a fit (an error) if you try. The fix uses a simple check: "Does this drawer exist?" If the index is within the valid range (less than the total count), we open it. If not, we print a warning instead of crashing.