2303A52300

Batch-42

LAB_ASSIGNMENT-7.3

Task 1: Fixing Syntax Errors

Scenario

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

```
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

Prompt:

def add(a,b) return a+b
detect the syntax error in the above code, correct the function definition and explain the syntax issue and give correct output

Code:

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

result = add(2, 3)
print(result)
```

Explanation:

First code → error, function not complete

Second code → correct, runs, prints output

Task 2: Debugging Logic Errors in Loops

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

Prompt:

fix the error in the code by Providing a loop with an increment or decrement error, identify the cause of infinite iteration and fix the loop logic
• Analyze the corrected loop behavior and explain the logic error

Code:

```
#below code have infinite loop fix the error in the code by Providing a loop
with an increment or decrement error, identify the cause of infinite iteration
and fix the loop logic
#Analyze the corrected loop behavior and explain the logic error

"""def count_down(n):
    while n >= 0:
        print(n)
        n += 1  # Should be n -= 1"""
def count_down(n):
    while n >= 0:
        print(n)
        n -= 1  # Decrement n to avoid infinite loop
count_down(5)
# The original code had an infinite loop because the variable 'n' was being
incremented (n += 1) inside the loop, which meant that 'n' would always be
greater than or equal to 0, causing the loop to never terminate. By changing
the increment to a decrement (n -= 1), 'n' will eventually reach -1, at which
point the loop condition (n >= 0) will no longer be true, allowing the loop to
exit correctly.
```

Explaination:

The original loop was infinite because n was increased (n += 1) while the condition required n to become smaller to stop. Since n always stayed greater than or equal to 0, the while condition never became false. This caused the loop to run forever, printing numbers endlessly.Changing n += 1 to n -= 1 makes n decrease on each iteration. When n reaches -1, the condition n >= 0 fails and the loop stops correctly.

Task 3: Handling Runtime Errors (Division by Zero)

Scenario

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

```
# 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

Prompt:

Debug the following code.add try-except blocks to handle division by zero error.Provide a function that performs division without validation.give error handlin approach and identify the runtime error

Code:

```
#Debug the following code
#add try-except blocks to handle division by zero error
#Provide a function that performs division without validation
#give error handlin approach and identify the runtime error
# Debug the following code give explaination for the error
def divide(a, b):
    return a / b
try:
    print(divide(10, 0))
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
# Explanation:
# The original code attempts to divide a number by zero, which raises a
ZeroDivisionError at runtime.
# By adding a try-except block, we can catch this specific error and handle it
gracefully, preventing the program from crashing.
```

Explaination:

The runtime error occurs because dividing any number by zero is not allowed in Python.This causes a ZeroDivisionError when divide(10, 0) is executed.The try block runs the division code that may cause an error.The except ZeroDivisionError block catches the error instead of crashing the program.This approach handles the error gracefully by displaying a clear error message.

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
```

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

Prompt:

Provide a class definition with missing self-parameter. identify the issue in the __init__() method.correct the class definition.Understand why self is required and explain it

Code:

```
# Provide a class definition with missing self-parameter
# identify the issue in the __init__() method
#correct the class definition
#Understand why self is required and explain it

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

```
        self.width = width

    def area(self):
        return self.length * self.width
# Example usage
rect = Rectangle(5, 10)
print("Area of rectangle:", rect.area())
# Explanation:
# In Python, instance methods (including __init__) must have 'self' as their
first parameter. This parameter refers to the instance of the class itself,
allowing access to its attributes and methods. Without 'self', the method
would not be able to modify or access the instance's state.
```

Explanation:

The issue was that the __init__() method was missing the self parameter.Without self, the class cannot store values in instance variables.Adding self allows length and width to be saved as object attributes.The corrected class properly initializes each Rectangle object.self is required because it refers to the current object and lets methods access its data.

Task 5: Resolving Index Errors in Lists

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

• AI suggestion using length checks or exception handling

Prompt:

Provide code that accesses an out-of-range list index.identify the Index Error.suggest safe access methods.Apply bounds checking or exception handling.explain the changes made

Code:

```
#Provide code that accesses an out-of-range list index
# identify the Index Error
#suggest safe access methods
#Apply bounds checking or exception handling
#explain the changes made
#numbers = [1, 2, 3]import numbers print(numbers[5])
numbers = [1, 2, 3]
try:
    print(numbers[5])  # Attempt to access an out-of-range index
except IndexError:
    print("IndexError: The index you are trying to access is out of range.")
# Changes made:
# Wrapped the list access in a try-except block to catch IndexError.
# This prevents the program from crashing and provides a user-friendly message
instead.
```

Explaination:

The list only has three items, but the code tries to access a sixth one.Python gets confused because that position doesn’t exist and raises an IndexError.The try block is used to attempt the risky operation.When the error happens, the except block steps in instead of crashing the program.This way, the program stays safe and shows a helpful message instead of stopping.