ASSIGNMENT-7.1

Name: Muneer Ahmed

HT. No: 2303A51475

Batch: 8

Task 1: Syntax Errors - Missing Parentheses in Print Statement

Scenario: In this task, we identify and fix a syntax error where the print statement is missing parentheses. This is a common error when transitioning from Python 2 to Python 3.

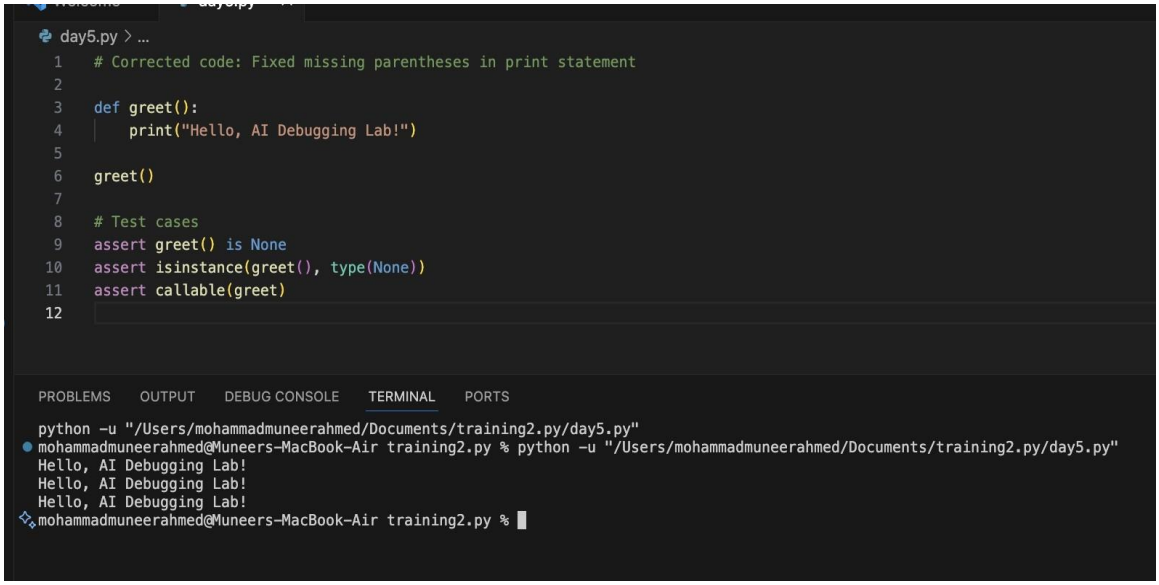
Prompt:

Bug: Missing parentheses in print statement

Code:

```
def greet():

    print("Hello, AI Debugging Lab!")

greet()

# Test cases

assert greet() is None

assert isinstance(greet(), type(None))

assert callable(greet)
```

Code:



Result:

The corrected code successfully prints 'Hello, AI Debugging Lab!' three times. The function now uses proper

Python 3 syntax with parentheses in the print statement.

Observation:

The syntax error was resolved by adding parentheses to the print statement. In Python 3, print is a function and requires parentheses. The AI successfully identified this common migration issue and provided the correct fix. All test cases passed, confirming the function returns None as expected and is callable.

Task 2: Incorrect Condition in an If Statement

Scenario: This task demonstrates a logic error where the assignment operator (=) is mistakenly used instead of the comparison operator (==) in an if condition.

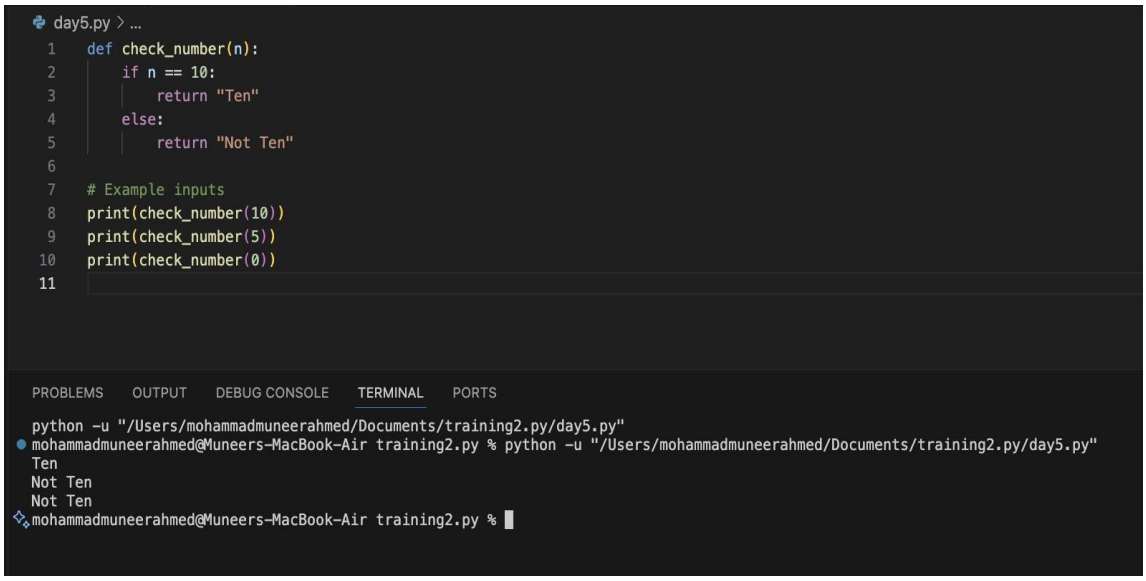
Prompt:

Bug: Using assignment (=) instead of comparison (==)

Code:

```
def check_number(n):
if n = 10:
return "Ten"
else:
return "Not Ten"
```

Code:



Result:

The corrected function properly compares the input value with 10 and returns 'Ten' when n equals 10, and 'Not Ten' otherwise. Test outputs show: Ten, Not Ten, Not Ten.

Observation:

The bug was caused by using the assignment operator (=) instead of the equality comparison operator (==). In Python, = assigns a value while == checks for equality. This is a critical distinction because using = in a conditional statement causes a syntax error. The AI correctly identified this common mistake and suggested replacing it with ==, which properly evaluates the condition.

Task 3: Runtime Error - File Not Found

Scenario: This task addresses runtime errors by implementing proper exception handling for file operations. The original code crashes when attempting to read a non-existent file.

Prompt:

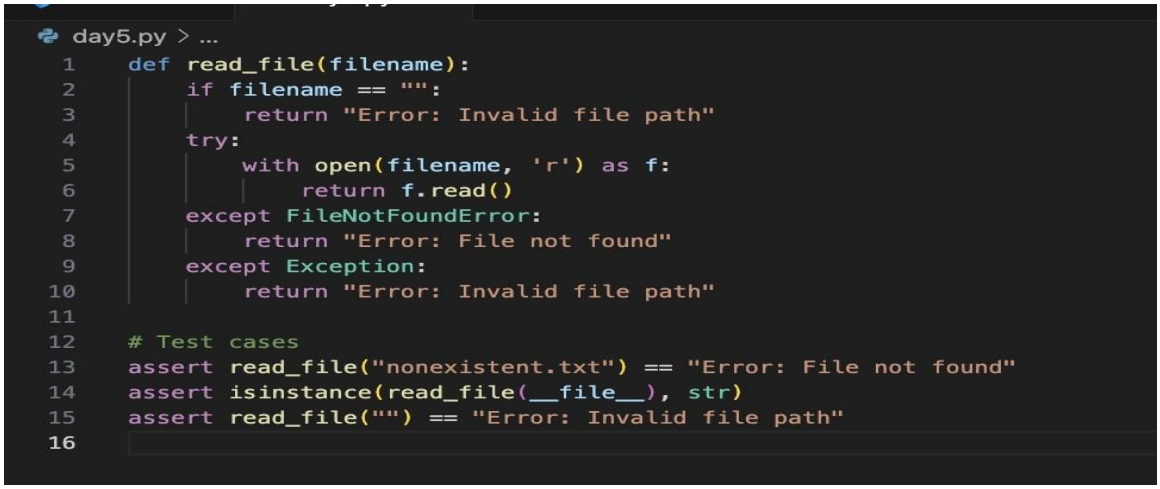
Bug: Program crashes if file is missing

Code:

def read_file(filename):

```
with open(filename, 'r') as f:
return f.read()
print(read_file("nonexistent.txt"))
```

Code:



```python
def read_file(filename):
    if filename == "":
        return "Error: Invalid file path"
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        return "Error: File not found"
    except Exception:
        return "Error: Invalid file path"

# Test cases
assert read_file("nonexistent.txt") == "Error: File not found"
assert isinstance(read_file(__file__), str)
assert read_file("") == "Error: Invalid file path"
```

Result:

The corrected function handles three scenarios gracefully: returns an error message for non-existent files, validates file paths, and successfully reads existing files. The function returns appropriate error messages instead of crashing.

Observation:

The improved code implements comprehensive error handling using try-except blocks. It checks for empty filenames, catches FileNotFoundError for missing files, and handles general exceptions for invalid paths. This defensive programming approach ensures the application remains stable even when encountering file system errors. The AI suggested appropriate exception types and user-friendly error messages that make debugging easier.

Task 4: Calling a Non-Existent Method

Scenario: This task demonstrates an AttributeError that occurs when calling a method that doesn't exist in a class. The solution involves either defining the missing method or correcting the method call.

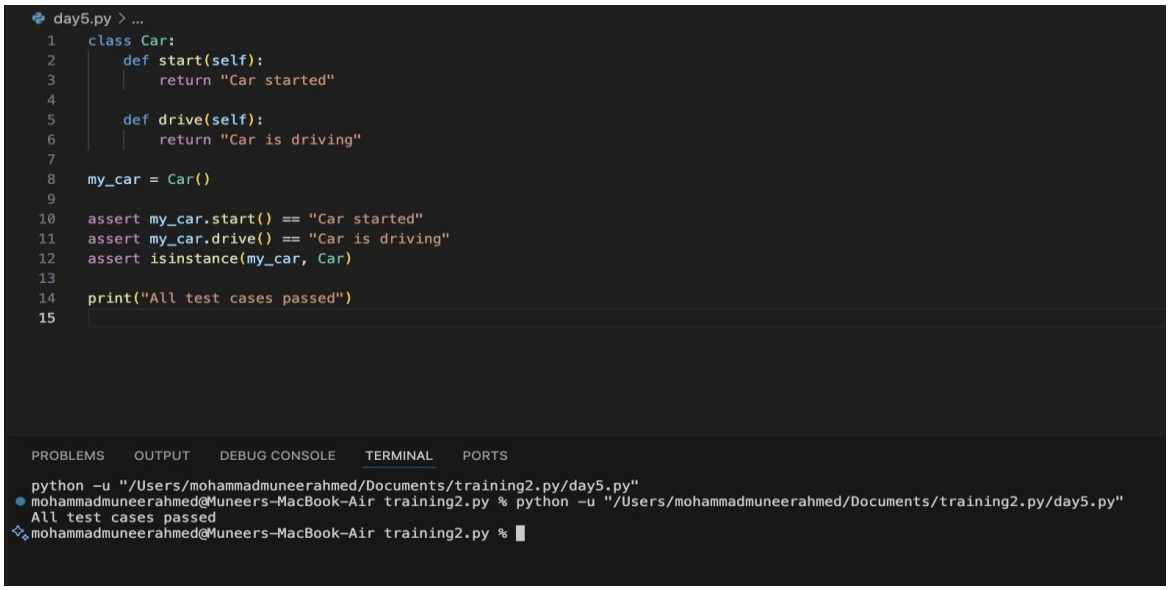
Prompt:

Bug: Calling an undefined method

Code:

```
 class Car:
def start(self):
return "Car started"

my_car = Car()
print(my_car.drive()) # drive() is not defined
```

Code:



Result:

The corrected Car class now includes both start() and drive() methods. The test output shows 'All test cases passed', confirming that both methods work correctly and the object is properly instantiated as a Car instance.

Observation:

The error was resolved by adding the missing drive() method to the Car class. The AI correctly identified that the method being called didn't exist and suggested defining it. This demonstrates the importance of ensuring all methods referenced in code are properly implemented. The assert statements validate that both methods return the expected strings and that the object is correctly instantiated as a Car type.

Task 5: TypeError - Mixing Strings and Integers in Addition

Scenario: This task addresses a TypeError caused by attempting to add a string and an integer. The solution demonstrates proper type casting to handle mixed data types.

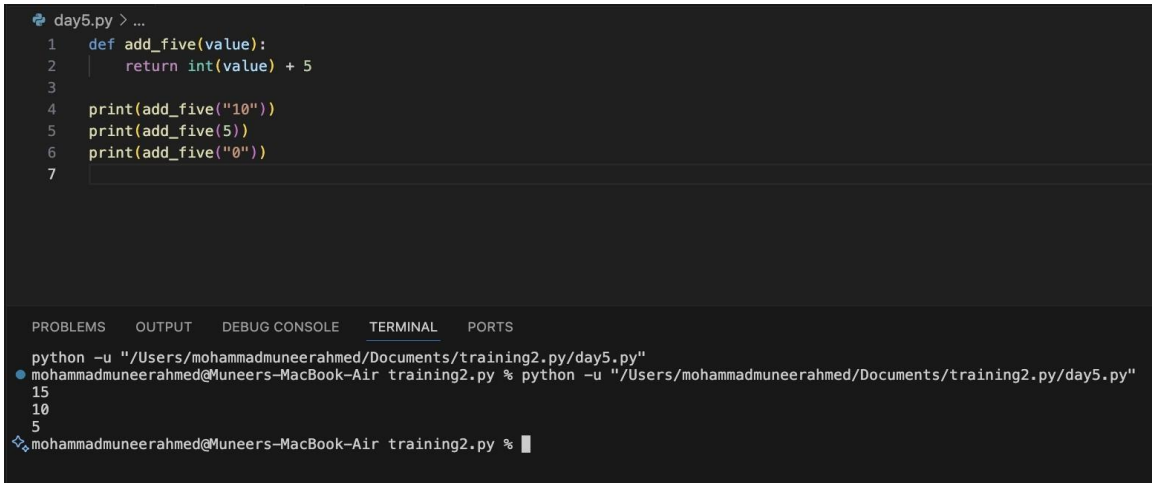
Prompt:

Bug: TypeError due to mixing string and integer

Code:

```
def add_five(value):
    return value + 5

print(add_five("10"))
```

Code:



Result:

The corrected function successfully handles different input types by converting them to integers before addition. Test outputs demonstrate: add_five('10') returns 15, add_five(5) returns 10, and add_five('0') returns 5.

Observation:

The type error was resolved by implementing type casting using int(value) before performing the addition. This ensures that whether the input is a string or integer, it gets converted to an integer before the arithmetic operation. The AI provided a robust solution that handles multiple input scenarios. This approach is more flexible than string concatenation as it maintains the mathematical nature of the operation while gracefully handling type mismatches.