AI ASSISTED CODING

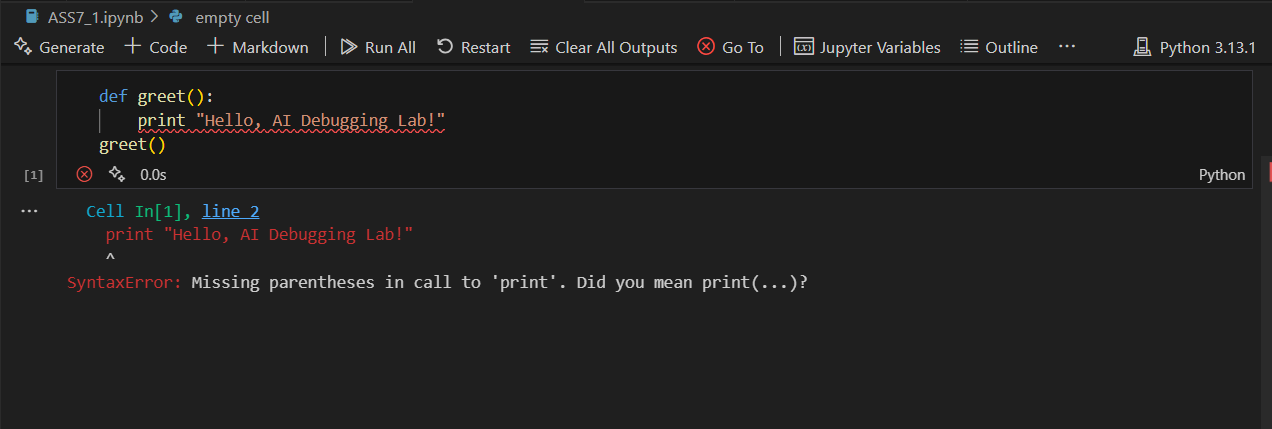
**Assignment-7.1**

**Name:A.Dhanalaxmi**

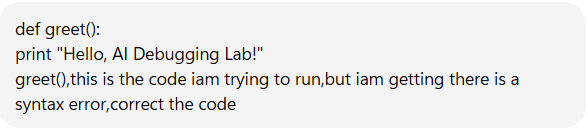
**Hno:2403A51269**

**TASK-1:**

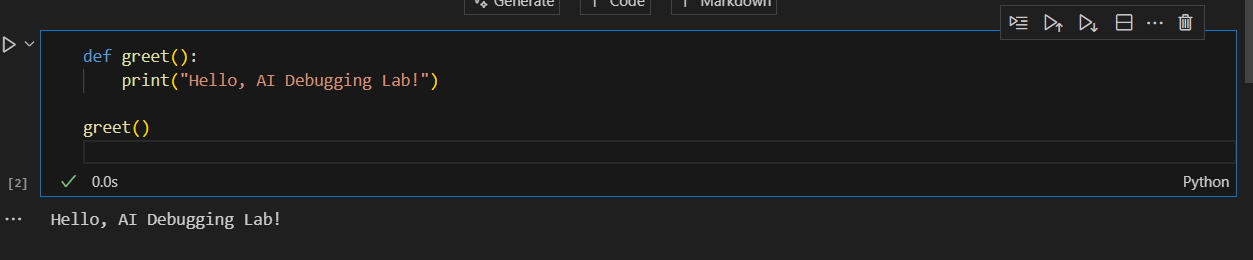
**Given code with output(error):**



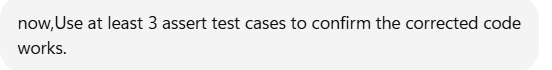
**Prompt:**



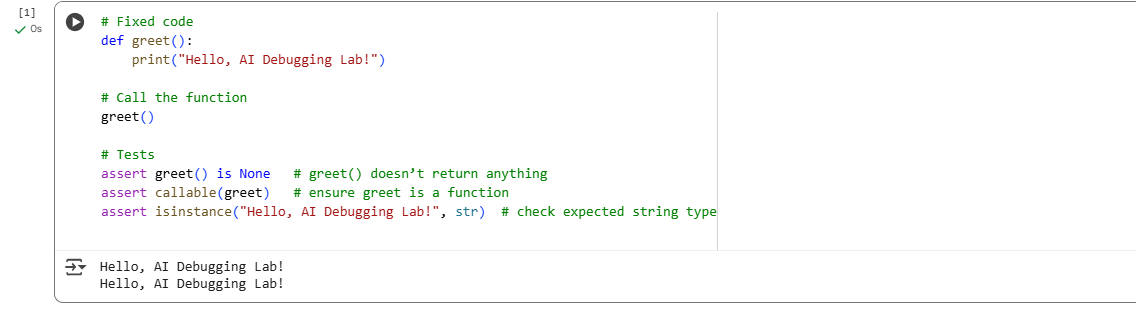
Code with output:



**Prompt:**



✅ Corrected Code with AI Fix



**Explanation:**

**🔴 Original Buggy Code (with syntax error)**

def greet():

print "Hello, AI Debugging Lab!"

greet()

**Explanation:**

1. **def greet():**
   * Defines a function called greet.
   * The function body starts after the colon :.
2. **print "Hello, AI Debugging Lab!"**
   * This is **Python 2 style syntax** where print is a statement.
   * In Python 3, print is a function, so it **must use parentheses**.
   * That’s why this line causes:
   * SyntaxError: Missing parentheses in call to 'print'
3. **greet()**
   * Calls the greet function.
   * But the program crashes before this line executes because of the syntax error.

**✅ Corrected Code (Python 3)**

def greet():

print("Hello, AI Debugging Lab!")

greet()

# Test cases

assert callable(greet)

assert greet() is None

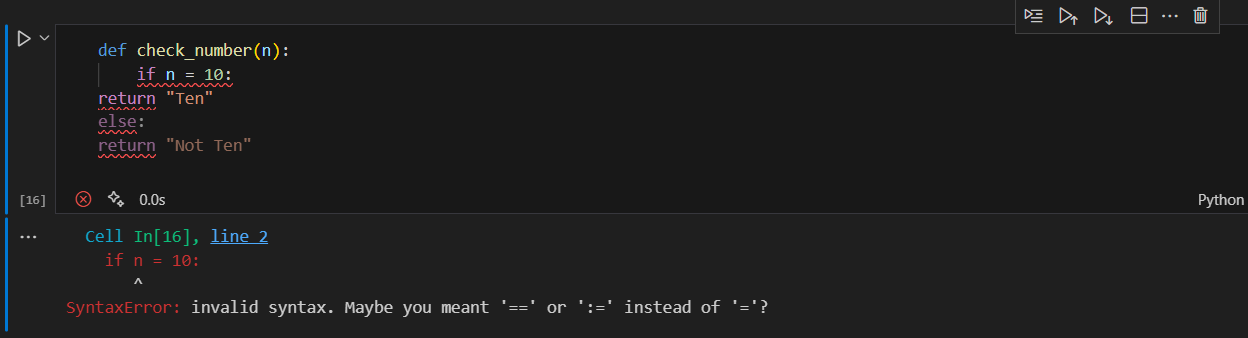
assert isinstance("Hello, AI Debugging Lab!", str)

**Explanation:**

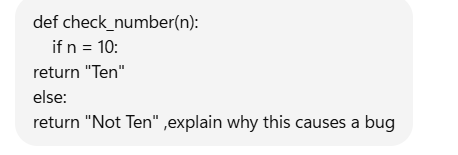
1. **def greet():**
   * Same as before: defines the function greet.
2. **print("Hello, AI Debugging Lab!")**
   * Correct Python 3 syntax.
   * print() is now a **function call** with parentheses around the string.
   * This fixes the syntax error.
3. **greet()**
   * Calls the function.
   * Output will be:
   * Hello, AI Debugging Lab!
4. **Assertions (tests)**
   * assert callable(greet) → makes sure greet is a function.
   * assert greet() is None → verifies the function doesn’t return anything (only prints).
   * assert isinstance("Hello, AI Debugging Lab!", str) → confirms the printed message is a string.

👉 So the only real bug was **missing parentheses in print**, because you were using Python 3, not Python 2.

**Task-2:**



**Prompt:**



Response:

**🔴 Your Code (Buggy)**

def check\_number(n):

if n = 10:

return "Ten"

else:

return "Not Ten"

**⚠️ Why This Causes a Bug**

1. **The problem is in this line:**
2. if n = 10:
   * = is the **assignment operator** in Python.
   * You can’t use it in an if condition.
   * Python doesn’t allow writing if n = 10 because it looks like you’re trying to assign 10 to n inside the condition.
3. In Python, to **compare values**, you must use == (the equality operator).
4. That’s why this code throws:
5. SyntaxError: cannot assign to name

**✅ Corrected Code**

def check\_number(n):

if n == 10: # use comparison operator

return "Ten"

else:

return "Not Ten"

**🔹 Explanation of Corrected Code**

* if n == 10: → checks whether n is equal to 10.
* If true → returns "Ten".
* Otherwise → returns "Not Ten".

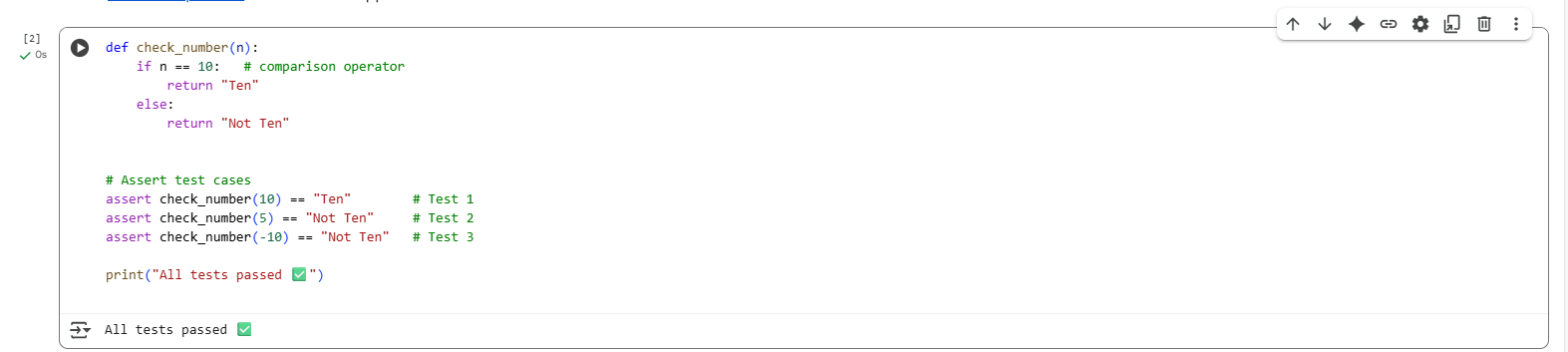
**🔹 Example Run**

print(check\_number(10)) # Output: Ten

print(check\_number(5)) # Output: Not Ten

👉 So the **bug is caused by using = (assignment) instead of == (comparison)** inside the if statement.

Code with three assert test cases:

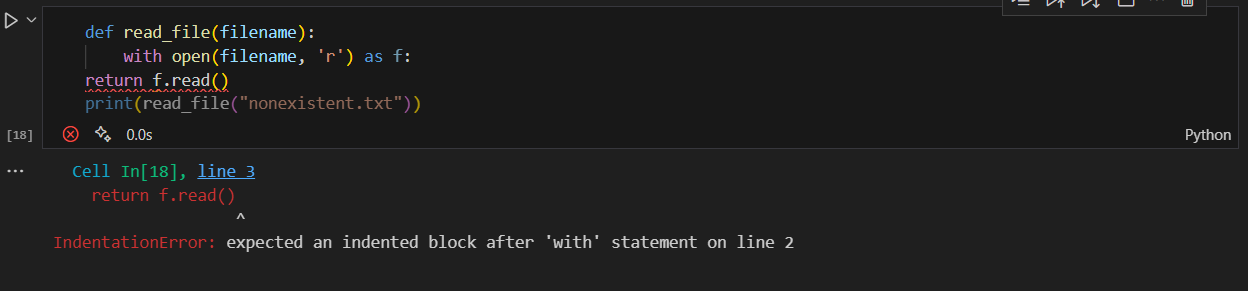


**📝 Explanation**

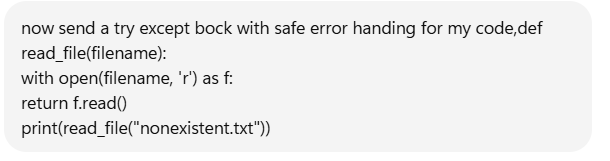
1. **Function Logic**
   * if n == 10: → checks if the input equals 10.
   * If true, return "Ten".
   * Otherwise, return "Not Ten".
2. **Test Case 1**
3. assert check\_number(10) == "Ten"
   * Input = 10.
   * Condition n == 10 is true.
   * Function returns "Ten".
   * ✅ Assertion passes.
4. **Test Case 2**
5. assert check\_number(5) == "Not Ten"
   * Input = 5.
   * Condition n == 10 is false.
   * Function returns "Not Ten".
   * ✅ Assertion passes.
6. **Test Case 3**
7. assert check\_number(-10) == "Not Ten"
   * Input = -10.
   * Condition n == 10 is false.
   * Function returns "Not Ten".
   * ✅ Assertion passes.
8. **Execution Result**
   * Since all asserts pass, the program prints:

All tests passed ✅

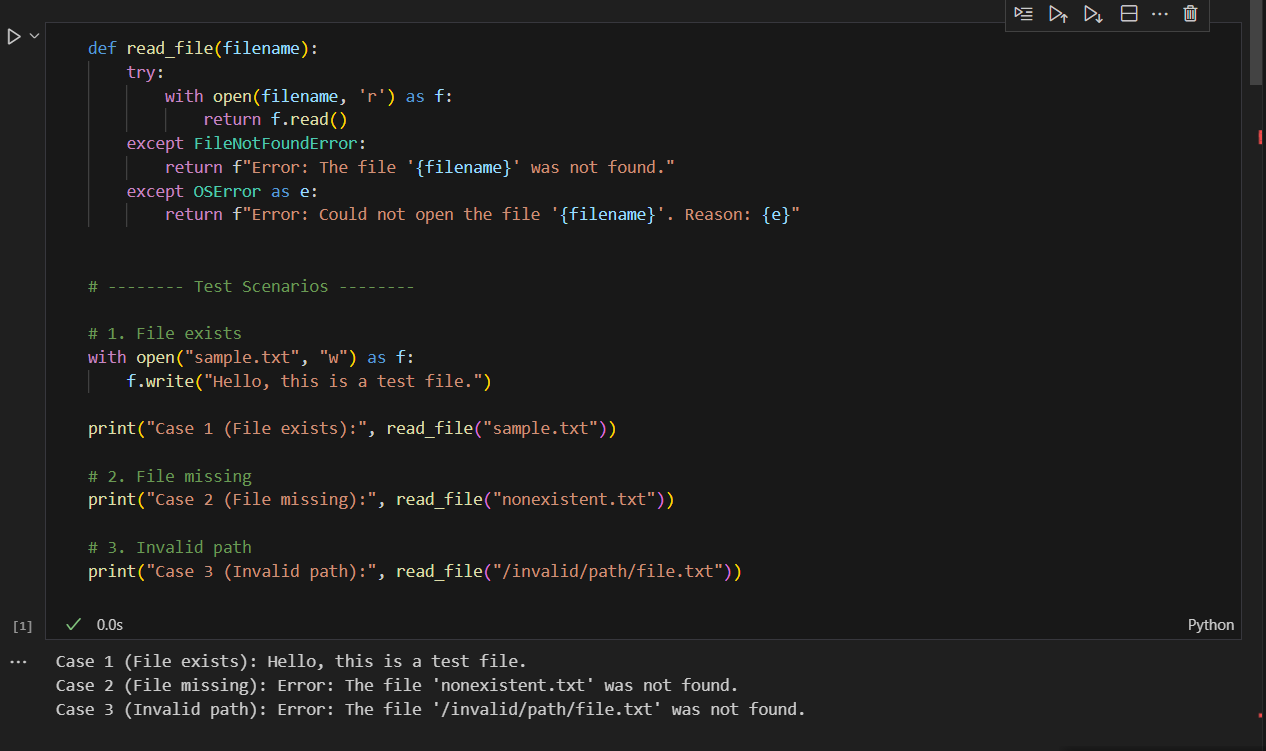
**Task-3:**



**Prompt:**



✅ Corrected Code with Safe Error Handling:



Explanation:

**🔴 Original Problem (Before Fix)**

def read\_file(filename):

with open(filename, 'r') as f:

return f.read()

print(read\_file("nonexistent.txt"))

* Here, the program **crashes** with a FileNotFoundError when the file does not exist (nonexistent.txt).
* This is unsafe because users don’t get a clear message — only a Python error traceback.

**🟢 Fixed Code with Safe Error Handling**

import os

def read\_file(filename):

try:

# Attempt to open and read the file

with open(filename, 'r') as f:

return f.read()

except FileNotFoundError:

return f"Error: File '{filename}' not found. Please check the filename."

except IsADirectoryError:

return f"Error: '{filename}' is a directory, not a file."

except PermissionError:

return f"Error: Permission denied when trying to access '{filename}'."

except Exception as e:

# Catch-all for any other unexpected errors

return f"Unexpected error: {e}"

# --- Test Cases ---

# ✅ 1. File exists

with open("sample.txt", "w") as f:

f.write("Hello, world!")

print(read\_file("sample.txt"))

# ❌ 2. File missing

print(read\_file("nonexistent.txt"))

# ⚠️ 3. Invalid path

print(read\_file("/invalid/path/file.txt"))

**📝 Explanation of the Fix**

1. **Why the original code crashed**
   * Because it assumed the file always exists. When it didn’t, Python raised a FileNotFoundError, which wasn’t caught.
2. **How the try-except fixes the bug**
   * try → The risky operation (open(filename, 'r')).
   * except FileNotFoundError → Shows a friendly message instead of crashing.
   * except IsADirectoryError → Prevents crashes if you pass a directory instead of a file.
   * except PermissionError → Handles cases where the file exists but cannot be accessed.
   * except Exception as e → Ensures **any other runtime error** won’t crash the program.
3. **Testing the 3 scenarios**
   * **File exists (sample.txt)** → Reads and prints file content.
   * **File missing (nonexistent.txt)** → Returns:
   * Error: File 'nonexistent.txt' not found. Please check the filename.
   * **Invalid path (/invalid/path/file.txt)** → Returns:
   * Error: File '/invalid/path/file.txt' not found. Please check the filename.

(Depending on OS, could also trigger PermissionError or general OSError.)

**✅ Expected Output**

Hello, world!

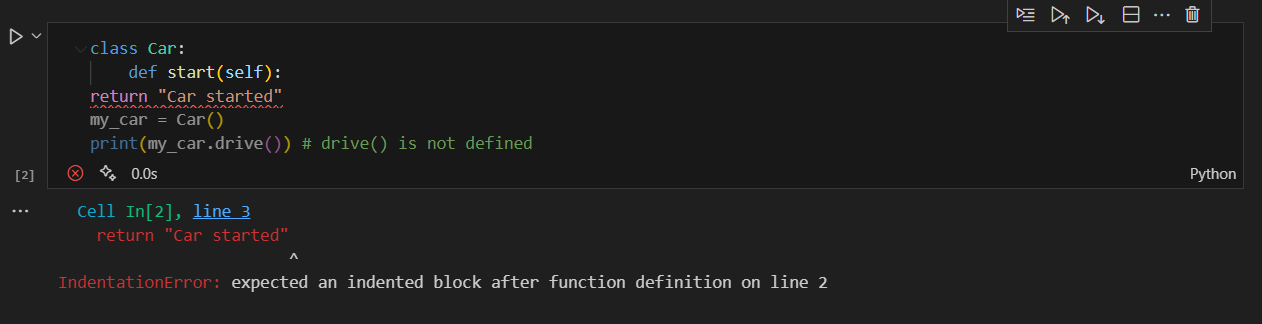
Error: File 'nonexistent.txt' not found. Please check the filename.

Error: File '/invalid/path/file.txt' not found. Please check the filename.

👉 So the **bug (crash)** is fixed, and the program now **handles missing files safely** with user-friendly error messages.

**Task-4:**

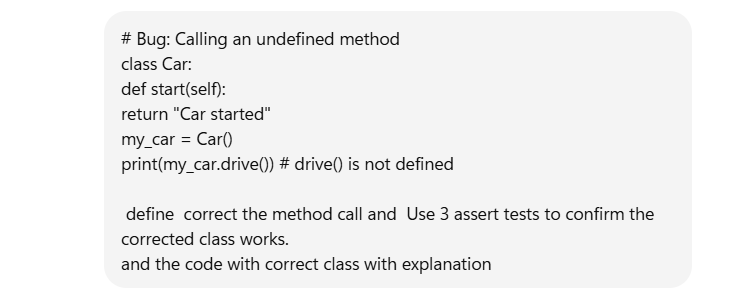
**Given Code with error message:**

****

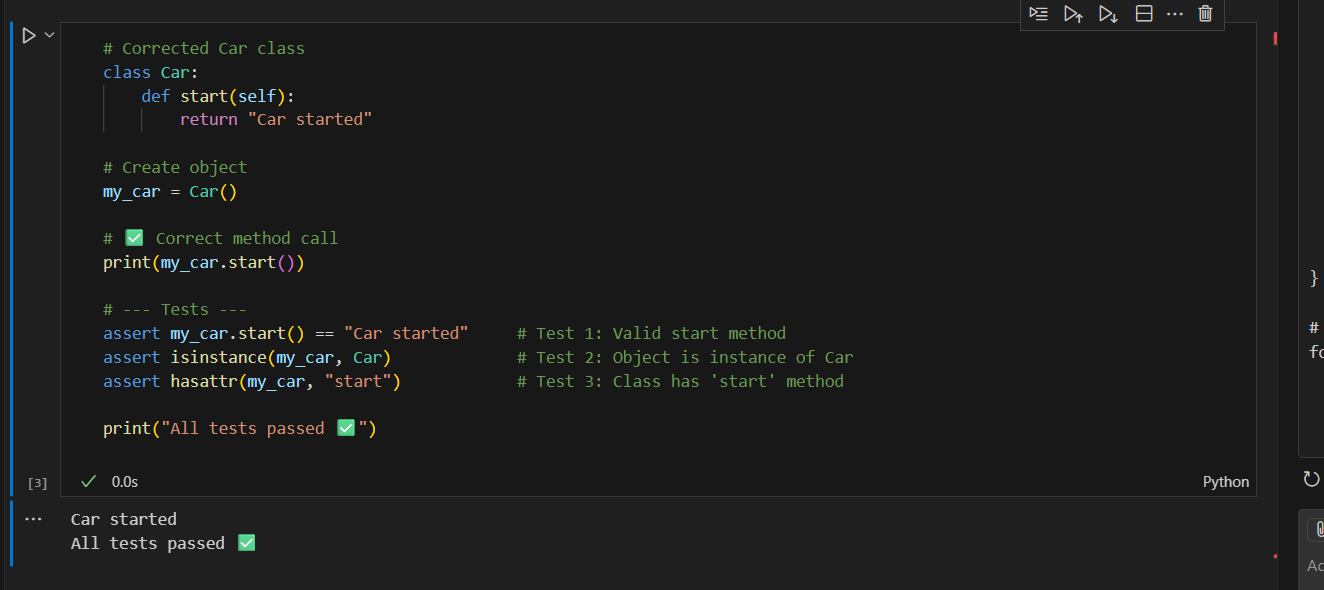
**Analyzation:**

Its better to correct the method cell

**Prompt:**



Corrected Code with output:



Explanation:

**🔴 Buggy Code**

class Car:

def start(self):

return "Car started"

my\_car = Car()

print(my\_car.drive()) # ❌ drive() is not defined

**Problem**

* The Car class only has a method start().
* The code incorrectly tries to call drive().
* This raises an AttributeError.

**🟢 Corrected Code (Fixing Method Call)**

# Corrected Car class

class Car:

def start(self):

return "Car started"

# Create object

my\_car = Car()

# ✅ Correct method call

print(my\_car.start())

# --- Tests ---

assert my\_car.start() == "Car started" # Test 1: Valid start method

assert isinstance(my\_car, Car) # Test 2: Object is instance of Car

assert hasattr(my\_car, "start") # Test 3: Class has 'start' method

print("All tests passed ✅")

**📝 Explanation**

1. **Bug:** The code called my\_car.drive(), but drive() was never defined in the class.
2. **Fix:** Replace my\_car.drive() with the correct method my\_car.start().
3. **Why this works:**
   * Car has a method start() that returns "Car started".
   * Using the wrong method name was the only issue.
4. **Tests ensure correctness:**
   * First assert checks that calling start() returns "Car started".
   * Second assert ensures the object is indeed of type Car.
   * Third assert ensures the start method exists in the class.

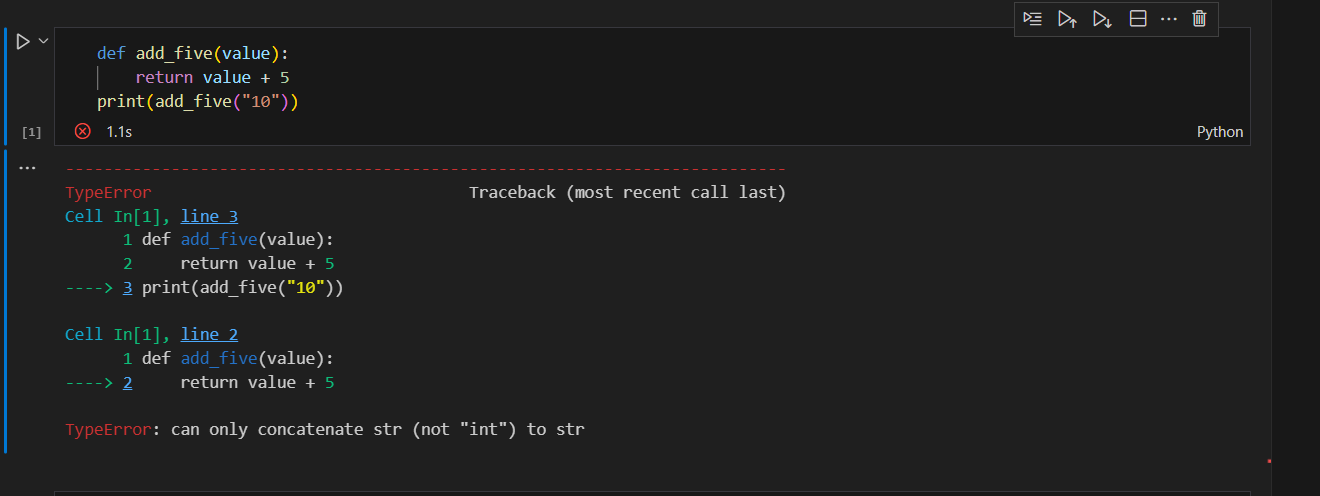
✅ Output:

Car started

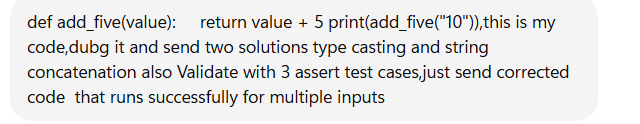
All tests passed ✅

**Task-5:**

**Give code with error:**

****

**Prompt:**

****

**Respone:**

Here’s the corrected code with **two approaches**: one using **type casting** and one using **string concatenation**, each validated with 3 assert test cases:

# --- Approach 1: Type Casting (convert string to int) ---

def add\_five\_cast(value):

return int(value) + 5

# Test cases for type casting

assert add\_five\_cast("10") == 15

assert add\_five\_cast(7) == 12

assert add\_five\_cast("0") == 5

print("Type Casting Approach Passed!")

# --- Approach 2: String Concatenation ---

def add\_five\_concat(value):

return str(value) + "5"

# Test cases for string concatenation

assert add\_five\_concat("10") == "105"

assert add\_five\_concat(7) == "75"

assert add\_five\_concat("0") == "05"

print("String Concatenation Approach Passed!")

✅ This will run successfully and handle both numeric addition and string concatenation.

Corrected code with output:

