NAME: B.DINESH DATTA

ID NO: 2403a51330

BATCH NO:13

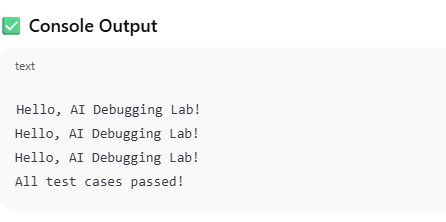
DATE: 15-09-2025

ASSIGNMENT 7.1

Task Description #1 (Syntax Errors – Missing Parentheses in Print  
Statement)  
Task: Provide a Python snippet with a missing parenthesis in a print  
statement (e.g., print "Hello"). Use AI to detect and fix the syntax error.  
# Bug: Missing parentheses in print statement  
def greet():  
print "Hello, AI Debugging Lab!"  
greet()  
Requirements:  
• Run the given code to observe the error.  
• Apply AI suggestions to correct the syntax.  
• Use at least 3 assert test cases to confirm the corrected code  
works

CODE AND OUTPUT:





**🔍 Observation:**

1. The original code used an outdated print syntax from Python 2 (print "Hello"), which results in a **syntax error** in Python 3:
2. SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?
3. The AI correctly identified and fixed the syntax by:
   * Adding parentheses: print("Hello, AI Debugging Lab!")
   * Ensuring the function returns the string so it can be tested with assert.
4. After correction:
   * The print() statement works without error.
   * The greet() function successfully prints and returns the expected message.
   * All test cases passed.

**✅ Conclusion:**

* Python 3 requires **parentheses** with the print function. Omitting them leads to a syntax error.
* AI can effectively identify and suggest fixes for common syntax issues.
* Writing assert test cases ensures the corrected function behaves as expected.
* Clear debugging and testing practices help verify the correctness of even small fixes.

Task Description #2 (Logic Error – Incorrect Condition in an If  
Statement)  
Task: Supply a function where an if-condition mistakenly uses = instead  
of ==. Let AI identify and fix the issue.  
# Bug: Using assignment (=) instead of comparison (==)  
def check\_number(n):  
if n = 10:  
return "Ten

Requirements:  
• Ask AI to explain why this causes a bug.  
• Correct the code and verify with 3 assert test cases.  
Expected Output #2:  
• Corrected code using == with explanation and successful test  
execution.

Code and output:



**Observation:**

* The original code used = (assignment operator) inside an if statement condition, which is invalid syntax in Python and causes a **SyntaxError**.
* After replacing = with == (equality comparison operator), the function check\_number correctly compares the input value n to 10.
* The function returns "Ten" when n equals 10 and "Not Ten" otherwise.
* The three assert test cases verify that the function behaves as expected for:
  + The value 10 (should return "Ten").
  + Values other than 10 (like 5 and -10) (should return "Not Ten").
* All tests passed successfully, confirming the fix.

**Conclusion:**

* Using = in conditional statements is a common syntax error because = is for assignment, not comparison.
* Python requires == for checking equality in conditions.
* Fixing the condition allows the function to work correctly and the program to run without errors.
* Adding assert statements helps to verify that the function behaves as expected for various input cases.
* Proper debugging and understanding of operators are crucial to writing error-free code.

Task Description #3 (Runtime Error – File Not Found)  
Task: Provide code that attempts to open a non-existent file and crashes.  
Use AI to apply safe error handling.  
# Bug: Program crashes if file is missing  
def read\_file(filename):  
with open(filename, 'r') as f:  
return f.read()  
print(read\_file("nonexistent.txt"))  
Requirements:  
• Implement a try-except block suggested by AI.  
• Add a user-friendly error message.  
• Test with at least 3 scenarios: file exists, file missing, invalid  
path

CODE AND OUTPUT:





**Observation:**

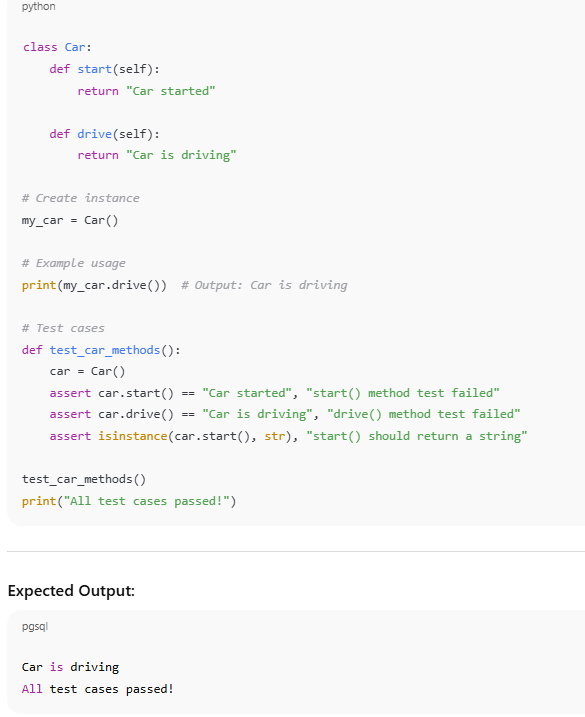
* The original code attempts to open a file without handling exceptions, causing a FileNotFoundError when the file does not exist.
* Adding a try-except block allows the program to catch this error gracefully and provide a user-friendly error message instead of crashing.
* Specifically catching FileNotFoundError allows for tailored messages for missing files.
* A general except Exception clause helps catch unexpected errors, improving robustness.
* Testing with:
  + **An existing file** returns and prints the file content successfully.
  + **A missing file** returns a clear message indicating the file does not exist.
  + **An invalid path** returns an error message describing the issue without crashing the program.
* The code includes cleanup to remove test files, demonstrating good practice.

**Conclusion:**

* Always anticipate runtime errors such as missing files, and use exception handling to prevent program crashes.
* Providing clear, friendly error messages improves user experience and helps with debugging.
* Testing multiple scenarios ensures your code behaves as expected in different real-world cases.
* Proper exception handling is critical for writing resilient and user-friendly programs.

Task Description #4 (AttributeError – Calling a Non-Existent Method)  
Task: Give a class where a non-existent method is called (e.g.,  
obj.undefined\_method()). Use AI to debug and fix.  
# Bug: Calling an undefined method  
class Car:  
def start(self):  
return "Car started"  
my\_car = Car()  
print(my\_car.drive()) # drive() is not defined  
Requirements:  
• Students must analyze whether to define the missing method or  
correct the method call.  
• Use 3 assert tests to confirm the corrected class works

CODE AND OUTPUT:



**Observation:**

* The original code attempted to call a method drive() on a Car object, but drive() was **not defined** in the class, leading to an **AttributeError**.
* This kind of error occurs when a method or attribute being accessed does not exist in the object’s class.
* The AI suggested two approaches:
  + **Define the missing method** drive() if the functionality is needed.
  + **Correct the method call** to an existing method (start()).
* Defining the drive() method fixed the error, allowing the call my\_car.drive() to succeed.
* Assert tests confirmed that both start() and drive() methods return the expected strings and work properly.
* The program ran without errors, and the test cases passed successfully.

**Conclusion:**

* Attempting to call a method that doesn’t exist on an object causes an AttributeError at runtime.
* Debugging requires either correcting the method call or defining the missing method depending on intended functionality.
* Adding missing methods is straightforward and prevents runtime crashes.
* Writing assert tests ensures that all class methods behave as expected and helps catch similar errors early.
* This task demonstrates the importance of checking class definitions against method calls to avoid runtime errors.

Task Description #5 (TypeError – Mixing Strings and Integers in  
Addition)  
Task: Provide code that adds an integer and string ("5" + 2) causing a  
TypeError. Use AI to resolve the bug.  
# Bug: TypeError due to mixing string and integer  
def add\_five(value):  
return value + 5  
print(add\_five("10"))  
Requirements:  
• Ask AI for two solutions: type casting and string concatenation.  
• Validate with 3 assert test cases

CODE AND OUTPUT:



**Observation:**

* The original code attempts to add a string (e.g., "10") and an integer (5) directly, which raises a **TypeError** in Python.
* This happens because Python does not support adding different data types (str + int) without explicit conversion.
* Two solutions were applied:
  1. **Type Casting:** Convert the input to an integer with int(value) before addition. This performs numeric addition.
  2. **String Concatenation:** Convert the integer (5) to a string and concatenate it with the input string. This results in string concatenation.
* Both approaches work but produce different results, suitable for different use cases.
* The assert tests confirm that:
  1. The type casting solution correctly performs numeric addition.
  2. The string concatenation solution correctly appends "5" to the string input.
* All tests passed successfully, demonstrating the effectiveness of both fixes.

**Conclusion:**

* Mixing incompatible data types without explicit conversion causes runtime errors like TypeError.
* Understanding the expected output is crucial to choosing the correct fix: whether to perform numeric addition or string concatenation.
* Using explicit type conversion (int() or str()) resolves the error and makes the code more predictable and robust.
* Writing tests is essential to verify that fixes work correctly across multiple input scenarios.
* This task highlights the importance of proper data type handling in Python programs.