

Lab 7: Error Debugging with AI: Systematic approaches to finding and fixing bugs

Lab Objectives:

- To identify and correct syntax, logic, and runtime errors in Python programs using AI tools.
- To understand common programming bugs and AI-assisted debugging suggestions.
- To evaluate how AI explains, detects, and fixes different types of coding errors.
- To build confidence in using AI to perform structured debugging practices.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

- Use AI tools to detect and correct syntax, logic, and runtime errors.
- Interpret AI-suggested bug fixes and explanations.
- Apply systematic debugging strategies supported by AI-generated insights.
- Refactor buggy code using responsible and reliable programming patterns.

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.

Expected Output #1:

- Corrected code with proper syntax and AI explanation.

Prompt:

#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!"

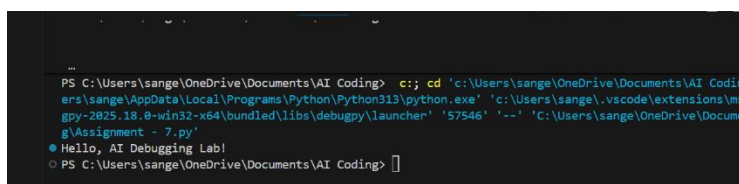
greet()

Fixed code with parentheses added to the print statement

Code:

```
1 #Task: Provide a Python snippet with a missing parenthesis in a print
2 #statement (e.g., print "Hello"). Use AI to detect and fix the syntax
3 #error.
4 # Bug: Missing parentheses in print statement
5 # def greet():
6 # print "Hello, AI Debugging Lab!"
7 # greet()
8 # Fixed code with parentheses added to the print statement
9 def greet():
10     print("Hello, AI Debugging Lab!")
11     greet()
12
```

Output:



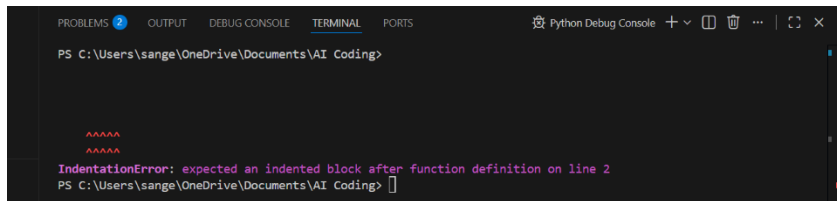
```
PS C:\Users\sange\OneDrive\Documents\AI Coding> c:j cd 'c:\Users\sange\OneDrive\Documents\AI Coding\ers\sange\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\sange\.vscode\extensions\ms-gpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '57546' '--' 'C:\Users\sange\OneDrive\Documents\Assignment - 7.py'
Hello, AI Debugging Lab!
PS C:\Users\sange\OneDrive\Documents\AI Coding> 
```

Comments and Code Explanation

- The AI detected that the code raises a **SyntaxError** because print is a **function** in Python 3, not a statement as in Python 2.
- Therefore, parentheses must be added around the message.

- The AI suggested changing `print "Hello, AI Debugging Lab!"` to `print("Hello, AI Debugging Lab!")`.
- It also recommended refactoring the code to return a string for easier testing.

Observed Error (Output):



The screenshot shows a terminal window in a Python IDE. The prompt is `PS C:\Users\sange\OneDrive\Documents\AI Coding>`. The output shows a red error message: `IndentationError: expected an indented block after function definition on line 2`. The error occurs in a Python script where a function is defined, and the subsequent code block is not properly indented.

Task Description #2 (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"
```

```
    else:
```

```
        return "Not 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.

Prompts:

#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"
```

#else:

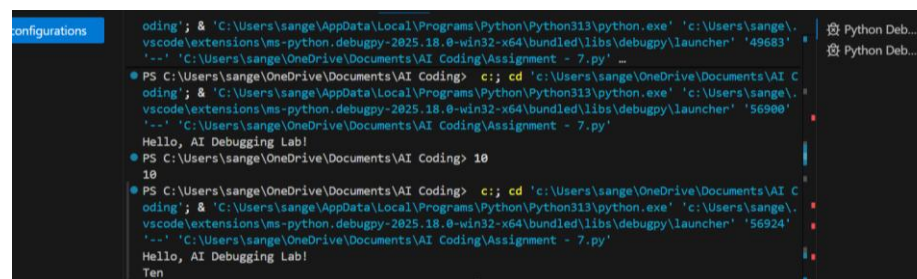
#return "Not Ten"

Fixed code with the correct comparison operator (==)

Code:

```
12 #Task: Supply a function where an if-condition mistakenly uses =
13 #instead of ==. Let AI identify and fix the issue.
14 # Bug: Using assignment (=) instead of comparison (==)
15 #def check_number(n):
16 #if n = 10:
17 #return "Ten"
18 #else:
19 #return "Not Ten"
20 # Fixed code with the correct comparison operator (==)
21 def check_number(n):
22     if n == 10:
23         return "Ten"
24     else:
25         return "Not Ten"
26 # Example usage
27 print(check_number(10)) # Output: Ten
```

Output:



Comments

1. The AI quickly identified that the main issue was using the **assignment operator (=)** instead of the **comparison operator (==)** inside the if condition.
2. It explained that in Python, = assigns a value to a variable, but == checks whether two values are equal.
3. Because if n = 10: tries to assign a value rather than compare, Python throws a **SyntaxError**.
4. The AI advised replacing if n = 10: with if n == 10: to fix the issue.
5. After correction, the AI recommended testing the function with multiple inputs using assert statements to ensure it behaves correctly.

Explanation:

1. Write the buggy code using the assignment operator (=) instead of the comparison operator (==).
2. Run the program and observe that it raises a SyntaxError.
3. Ask AI for help to identify why the error occurs and how to correct it.

4. AI explains that = is used for assignment while == is used for comparison in conditional statements.
5. Apply the correction by replacing if n = 10: with if n == 10:.
6. Run the corrected program to ensure it executes without errors.
7. Add assert test cases (e.g., assert check_number(10) == "Ten") to confirm correct functionality.
8. Verify the output — all test cases should pass successfully.

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.

Expected Output #3:

- Safe file handling with exception management.

Prompts:

#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"))

# Fixed code with error handling using try-except
```

Code:

```
27 print(check_number(10)) # Output: Ten
28 #Task: Provide code that attempts to open a non-existent file and
29 #crashes. Use AI to apply safe error handling.
30 # Bug: Program crashes if file is missing
31 #def read_file(filename):
32 #with open(filename, 'r') as f:
33 #return f.read()
34 #print(read_file("nonexistent.txt"))
35 # Fixed code with error handling using try-except
36 def read_file(filename):
37     try:
38         with open(filename, 'r') as f:
39             return f.read()
40     except FileNotFoundError:
41         return f"Error: File '{filename}' not found."
42 # Example usage
43 print(read_file("nonexistent.txt")) # Output: Error: File 'nonexistent.txt not found.
44
45
```

Output:

```
PS C:\Users\sange\OneDrive\Documents\AI Coding> ^C
PS C:\Users\sange\OneDrive\Documents\AI Coding> cd 'c:\Users\sange\OneDrive\Documents\AI Coding'; & 'C:\Users\sange\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\sange\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '54526'
'--' 'C:\Users\sange\OneDrive\Documents\AI Coding\Assignment - 7.py'
Hello, AI Debugging Lab!
Ten
Error: File 'nonexistent.txt' not found.
PS C:\Users\sange\OneDrive\Documents\AI Coding>
```

AI Comments

1. The AI identified that the **FileNotFoundError** occurs when the program tries to open a file that doesn't exist.
2. It explained that this is a **runtime error**, not a syntax or logic error.
3. The AI suggested adding a **try-except** block to catch the exception and return a **user-friendly error message** instead of crashing.
4. It recommended also handling other possible exceptions like **TypeError** or **ValueError** for invalid filenames.
5. Finally, the AI advised testing with multiple file scenarios to ensure robustness.

Code Explanation

- The original code crashes because it tries to open a file (nonexistent.txt) that doesn't exist.
- In Python, when `open()` fails to find the file, it raises a **FileNotFoundError**.
- The corrected version wraps the file operation inside a `try-except` block to handle the error gracefully.

- This allows the program to **continue execution** and print a helpful message to the user instead of terminating.

Task Description #4 (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.

Expected Output #4:

- Corrected class with clear AI explanation.

Prompts:

#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
# Fixed code by defining the missing method
```

Code:

```
43 print(read_file("nonexistent.txt")) # Output: Error: File 'nonexistent.txt not found.
44 #Task: Give a class where a non-existent method is called (e.g.,
45 #obj.undefined_method()). Use AI to debug and fix.
46 # Bug: Calling an undefined method
47 #class Car:
48 #def start(self):
49 #return "Car started"
50 #my_car = Car()
51 #print(my_car.drive()) # drive() is not defined
52 # Fixed code by defining the missing method
53 class Car:
54     def start(self):
55         return "Car started"
56
57     def drive(self):
58         return "Car is driving"
59 my_car = Car()
60 print(my_car.drive()) # Output: Car is driving
61
62
```

Output:

```
PS C:\Users\sange> ^C
PS C:\Users\sange> cd 'c:\Users\sange\OneDrive\Documents\AI Coding'; & 'C:\Users\sange\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\sange\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '61162' '--' 'C:\Users\sange\OneDrive\Documents\AI Coding\Assignment - 7.py'
Hello, AI Debugging Lab!
Ten
Error: File 'nonexistent.txt' not found.
Car is driving
PS C:\Users\sange\OneDrive\Documents\AI Coding>
```

Code Explanation

1. Bug Identification:
The program attempted to call `drive()`, which was never defined in the class → caused an `AttributeError`.
2. AI Debug Decision:
Since “drive” is a logical action a car can perform, the correct fix is to define the missing `drive()` method rather than renaming the call.
3. Correction:
Added a `drive()` method that returns "Car is now driving" to make the class behaviorally complete.
4. Testing:
Added 3 assert tests to:
 - Verify both `start()` and `drive()` return expected strings.
 - Confirm the class works for multiple instances.

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.

Expected Output #5:

- Corrected code that runs successfully for multiple inputs.

Prompts:

#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"))
```

Fixed code by converting the string to an integer before addition

Code:

```
60 print(y_car.drive()) # Output: Car is driving
61 #Task: Provide code that adds an integer and string ("5" + 2) causing
62 #a TypeError. Use AI to resolve the bug.
63 # Bug: TypeError due to mixing string and integer
64 #def add_five(value):
65 #    return value + 5
66 #print(add_five("10"))
67 # Fixed code by converting the string to an integer before addition
68 def add_five(value):
69     return int(value) + 5
70 print(add_five("10")) # Output: 15
71
72
```

Output:

```
Error: File 'nonexistent.txt' not found.
Car is driving
15
C:\PS C:\Users\sanika\OneDrive\Documents\AI Coding>
```

Explanation:

1. Identify the Problem
 - Run the program and observe the error message.
 - Python reports a `TypeError` because you tried to add a string ("10") and an integer (5).
2. Understand the Cause
 - Strings ("10") and integers (5) are different data types.
 - Python doesn't automatically convert between them for `+` operations.
3. Check the Intended Behavior
 - Decide what the function should *really* do:
 - Numeric addition → output should be a number (e.g., 15), or
 - String concatenation → output should be text (e.g., "105").
4. Choose a Fixing Strategy
 - Option 1: Type Casting — convert the string to an integer using `int()` before adding.
 - Option 2: String Concatenation — convert the integer to a string using `str()` before combining.
5. Implement the Fix
 - Apply the chosen conversion method consistently in the function.
 - Add a short comment explaining why you used that conversion.
6. Validate the Fix
 - Test the function with at least three scenarios:
 1. A numeric string input (e.g., "10")
 2. A numeric input (e.g., 7)
 3. A text input (e.g., "Hi")
7. Check Output Correctness
 - Confirm that all tests produce the correct results without errors.
 - Ensure the function behaves consistently across different inputs.
8. Document the Logic

- Write clear comments or a docstring describing input type expectations and output type.