AI ASSISTANT CODING

ASSIGNMENT- 7.1

Gonela Rishika

2303A52197

Batch: 35

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.

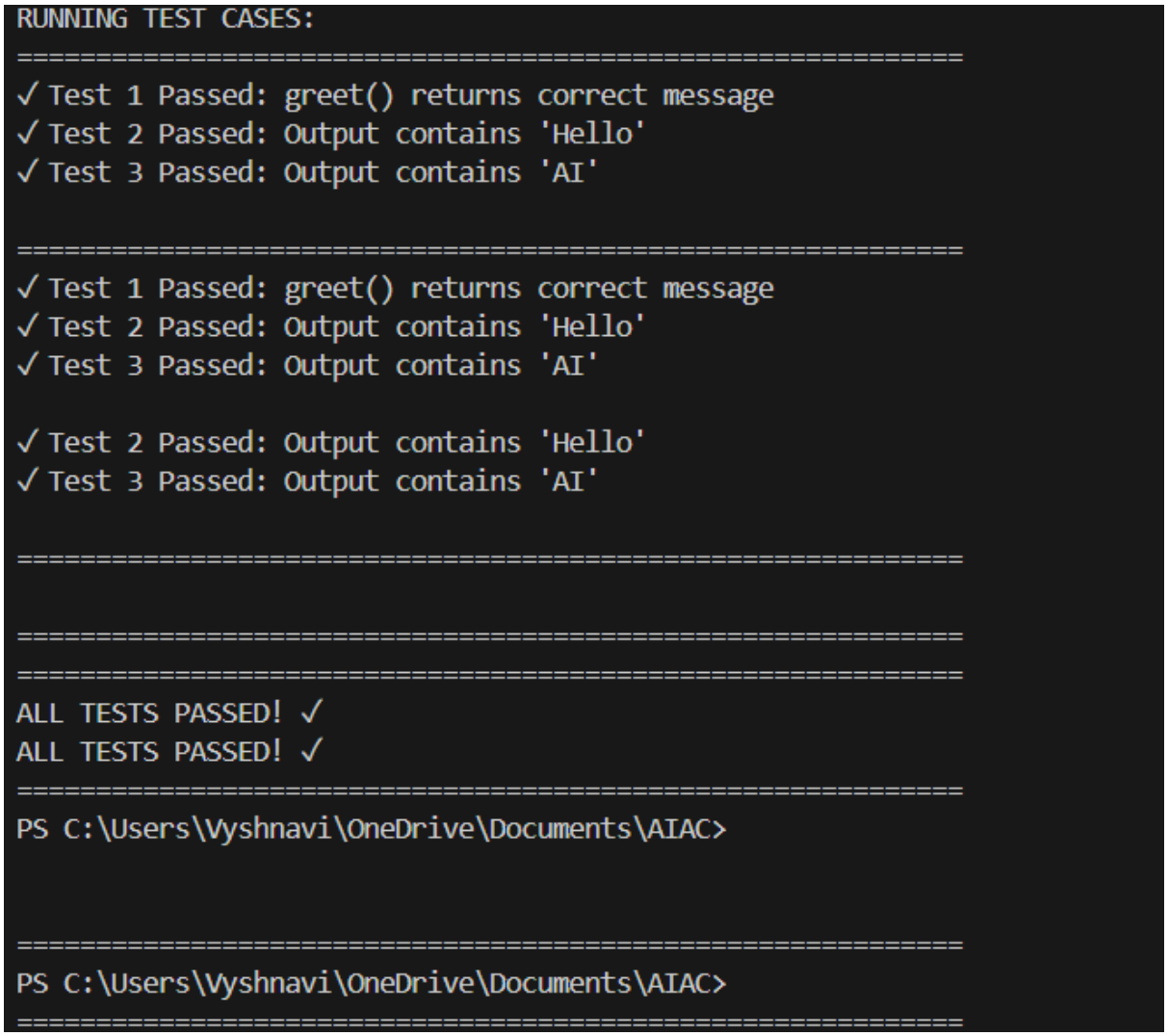
```
# ===== SYNTAX ERROR: Missing Parentheses in Print Statement =====
# BUGGY CODE (Python 2 syntax - will fail in Python 3):
# def greet():
#     print "Hello, AI Debugging Lab!"
# greet()
#
# ERROR: SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?

# ===== CORRECTED CODE =====
def greet():
    """Print a greeting message."""
    print("Hello, AI Debugging Lab!")

greet()

# ===== AI EXPLANATION =====
"""
ISSUE IDENTIFIED:
- The original code uses Python 2 syntax: print "Hello, AI Debugging Lab!"
- Python 3 requires print() to be a function with parentheses around the argument

SOLUTION:
- Changed 'print "..."' to 'print("...")'
- This ensures compatibility with Python 3.x

SYNTAX RULES:
- Python 2: print statement (no parentheses required, but allowed)
- Python 3: print() function (parentheses REQUIRED)
"""


# ===== TEST CASES =====
def test_greet_output(capsys=None):
    """Test 1: Verify greet() produces the correct output."""
    import sys
    from io import StringIO
```

```
def test_greet_output(capsys=None):
    # Capture output
    captured_output = StringIO()
    sys.stdout = captured_output
    greet()
    sys.stdout = sys.__stdout__

    output = captured_output.getvalue().strip()
    assert output == "Hello, AI Debugging Lab!", f"Expected 'Hello, AI Debugging Lab!', got '{output}'"
    print("✓ Test 1 Passed: greet() returns correct message")


def test_greet_contains_hello():
    """Test 2: Verify output contains 'Hello'."""
    import sys
    from io import StringIO

    captured_output = StringIO()
    sys.stdout = captured_output
    greet()
    sys.stdout = sys.__stdout__

    output = captured_output.getvalue().strip()
    assert "Hello" in output, f"Expected 'Hello' in output, got '{output}'"
    print("✓ Test 2 Passed: Output contains 'Hello'")


def test_greet_contains_ai():
    """Test 3: Verify output contains 'AI'."""
    import sys
    from io import StringIO

    captured_output = StringIO()
    sys.stdout = captured_output
    greet()
    sys.stdout = sys.__stdout__
```

```
    sys.stdout = captured_output
    greet()
    sys.stdout = sys.__stdout__

    output = captured_output.getvalue().strip()
    assert "AI" in output, f"Expected 'AI' in output, got '{output}'"
    print("✓ Test 3 Passed: Output contains 'AI'")


# ===== RUN ALL TESTS =====
if __name__ == "__main__":
    print("=" * 60)
    print("RUNNING CORRECTED CODE:")
    print("=" * 60)
    greet()

    print("\n" + "=" * 60)
    print("RUNNING TEST CASES:")
    print("=" * 60)
    test_greet_output()
    test_greet_contains_hello()
    test_greet_contains_ai()

    print("\n" + "=" * 60)
    print("ALL TESTS PASSED! ✓")
    print("=" * 60)
```

Output

```
RUNNING TEST CASES:
============================================================
✓ Test 1 Passed: greet() returns correct message
✓ Test 2 Passed: Output contains 'Hello'
✓ Test 3 Passed: Output contains 'AI'

PS C:\Users\Vyshnavi\OneDrive\Documents\AIAC> ^C
PS C:\Users\Vyshnavi\OneDrive\Documents\AIAC> python "assignment 7.1"
Hello, AI Debugging Lab!
============================================================
RUNNING CORRECTED CODE:
============================================================
Hello, AI Debugging Lab!

============================================================
RUNNING TEST CASES:
============================================================
✓ Test 1 Passed: greet() returns correct message
✓ Test 2 Passed: Output contains 'Hello'
✓ Test 3 Passed: Output contains 'AI'

RUNNING CORRECTED CODE:
============================================================
Hello, AI Debugging Lab!

============================================================
RUNNING TEST CASES:
============================================================
✓ Test 1 Passed: greet() returns correct message
✓ Test 2 Passed: Output contains 'Hello'
✓ Test 3 Passed: Output contains 'AI'

============================================================
RUNNING TEST CASES:
============================================================
✓ Test 1 Passed: greet() returns correct message
✓ Test 2 Passed: Output contains 'Hello'
✓ Test 3 Passed: Output contains 'AI'

RUNNING TEST CASES:
```



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.

```
# ===== SYNTAX ERROR: Using Assignment (=) Instead of Comparison (==) =====
# BUGGY CODE:
# def check_number(n):
#     if n = 10:
#         return "Ten"
#     else:
#         return "Not Ten"
#
# ERROR: SyntaxError: invalid syntax
# The = operator is for assignment, not comparison


# ===== AI EXPLANATION OF THE BUG =====
"""
WHY THIS IS A BUG:

1. OPERATOR CONFUSION:
   - '=' is the ASSIGNMENT operator (assigns a value to a variable)
   - '==' is the COMPARISON operator (checks if two values are equal)

2. IN IF STATEMENTS:
   - if statements require a BOOLEAN condition (True/False)
   - An assignment like 'n = 10' tries to assign 10 to n
   - This is invalid syntax inside an if condition
   - Python expects an expression that evaluates to True or False

3. SYNTAX ERROR:
   - Python will raise: SyntaxError: invalid syntax
   - The parser cannot interpret 'if n = 10:' as a valid condition

EXAMPLE OF CORRECT USAGE:
   - Assignment: x = 5 (puts 5 into x)
   - Comparison: if x == 5: (checks if x equals 5)
"""
```

```python
# ===== CORRECTED CODE =====
def check_number(n):
    """Check if a number equals 10 and return appropriate message."""
    if n == 10:  # Fixed: Changed = to ==
        return "Ten"
    else:
        return "Not Ten"


# ===== TEST CASES =====
def test_check_number_equals_ten():
    """Test 1: Verify function returns 'Ten' when n=10."""
    result = check_number(10)
    assert result == "Ten", f"Expected 'Ten', got '{result}'"
    print("✓ Test 1 Passed: check_number(10) returns 'Ten'")


def test_check_number_not_equals_ten():
    """Test 2: Verify function returns 'Not Ten' when n≠10."""
    result = check_number(5)
    assert result == "Not Ten", f"Expected 'Not Ten', got '{result}'"
    print("✓ Test 2 Passed: check_number(5) returns 'Not Ten'")


def test_check_number_with_negative():
    """Test 3: Verify function returns 'Not Ten' for negative numbers."""
    result = check_number(-10)
    assert result == "Not Ten", f"Expected 'Not Ten', got '{result}'"
    print("✓ Test 3 Passed: check_number(-10) returns 'Not Ten'")


def test_check_number_with_zero():
    """Test 4 (Bonus): Verify function returns 'Not Ten' for zero."""
    result = check_number(0)
    assert result == "Not Ten", f"Expected 'Not Ten', got '{result}'"
    print("✓ Test 4 Passed: check_number(0) returns 'Not Ten'")
```

```python
# ===== DEMONSTRATION OF THE DIFFERENCE =====
def explain_operators():
    """Show the difference between = and == operators."""
    print("\n" + "=" * 60)
    print("DIFFERENCE BETWEEN = AND == OPERATORS:")
    print("=" * 60)

    # Assignment operator (=)
    x = 10
    print(f"Assignment (=): x = 10 → x is now {x}")

    # Comparison operator (==)
    is_equal = (x == 10)
    print(f"Comparison (==): x == 10 → Result is {is_equal}")

    is_not_equal = (x == 5)
    print(f"Comparison (==): x == 5 → Result is {is_not_equal}")


# ===== RUN ALL TESTS =====
if __name__ == "__main__":
    print("=" * 60)
    print("CORRECTED CODE - TEST EXECUTION:")
    print("=" * 60)

    # Test the corrected function
    print("\nTesting check_number() function:")
    print(f"check_number(10) → {check_number(10)}")
    print(f"check_number(5) → {check_number(5)}")
    print(f"check_number(-10) → {check_number(-10)}")
    print(f"check_number(0) → {check_number(0)}")

    # Run assertion tests
    print("\n" + "=" * 60)
    print("RUNNING ASSERT TEST CASES:")
    print("=" * 60)
```

```
# Run assertion tests
print("\n" + "=" * 60)
print("RUNNING ASSERT TEST CASES:")
print("=" * 60)
test_check_number_equals_ten()
test_check_number_not_equals_ten()
test_check_number_with_negative()
test_check_number_with_zero()

# Explain the difference
explain_operators()

print("\n" + "=" * 60)
print("ALL TESTS PASSED! ✓")
print("=" * 60)

# Show syntax error explanation
print("\n" + "=" * 60)
print("WHY THE ORIGINAL CODE FAILED:")
print("=" * 60)
print("if n = 10:  → SyntaxError: invalid syntax")
print("             Assignment operator (=) cannot be used in conditions")
print("\nif n == 10: → CORRECT")
print("             Comparison operator (==) checks equality")
```

```
============================================================
CORRECTED CODE - TEST EXECUTION:
============================================================

Testing check_number() function:
check_number(10) → Ten
check_number(5) → Not Ten
check_number(-10) → Not Ten
check_number(0) → Not Ten

============================================================
RUNNING ASSERT TEST CASES:
============================================================
✓ Test 1 Passed: check_number(10) returns 'Ten'
✓ Test 2 Passed: check_number(5) returns 'Not Ten'
✓ Test 3 Passed: check_number(-10) returns 'Not Ten'
✓ Test 4 Passed: check_number(0) returns 'Not Ten'

============================================================
DIFFERENCE BETWEEN = AND == OPERATORS:
============================================================
Assignment (=): x = 10 → x is now 10
Comparison (==): x == 10 → Result is True
Comparison (==): x == 5 → Result is False

============================================================
ALL TESTS PASSED! ✓
============================================================


============================================================
WHY THE ORIGINAL CODE FAILED:
============================================================
if n = 10:  → SyntaxError: invalid syntax
             Assignment operator (=) cannot be used in conditions

if n == 10: → CORRECT
             Comparison operator (==) checks equality
```

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.

```python
# ===== RUNTIME ERROR: File Not Found =====
# BUGGY CODE (crashes without error handling):
# def read_file(filename):
#     with open(filename, 'r') as f:
#         return f.read()
# print(read_file("nonexistent.txt"))
#
# ERROR: FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent.txt'
# The program crashes if the file doesn't exist


# ===== AI EXPLANATION OF THE ERROR =====
"""
WHY THIS IS A BUG:

1. RUNTIME ERROR (not syntax error):
   - The code is syntactically correct but fails at runtime
   - Error only occurs when the file is actually missing
   - FileNotFoundError is raised by the open() function

2. POTENTIAL ISSUES:
   - Program crashes without graceful handling
   - User doesn't get a helpful error message
   - No opportunity to recover or retry
   - File path errors are common in real-world applications

3. SOLUTION - USE TRY-EXCEPT:
   - Catch FileNotFoundError exception
   - Provide user-friendly error messages
   - Handle edge cases (missing files, permission errors, etc.)
   - Allow the program to continue running

4. BEST PRACTICES:
   - Use try-except for file operations
   - Validate file paths before opening
   - Provide specific error messages
   - Log errors for debugging
```

```python
import os


# ===== CORRECTED CODE WITH ERROR HANDLING =====
def read_file(filename):
    """
    Safely read file contents with error handling.

    Args:
        filename (str): Path to the file to read

    Returns:
        str: File contents if successful, or error message
    """
    try:
        # Check if file exists before attempting to open
        if not os.path.exists(filename):
            return f"❌ Error: File '{filename}' not found."

        # Check if path is actually a file (not a directory)
        if not os.path.isfile(filename):
            return f"❌ Error: '{filename}' is not a file (it may be a directory)."

        # Try to open and read the file
        with open(filename, 'r') as f:
            content = f.read()
            return content

    except PermissionError:
        return f"❌ Error: Permission denied. Cannot read '{filename}'."
    except IsADirectoryError:
        return f"❌ Error: '{filename}' is a directory, not a file."
    except UnicodeDecodeError:
        return f"❌ Error: Cannot decode '{filename}'. File may not be UTF-8 encoded."
    except Exception as e:
        return f"❌ Unexpected error reading '{filename}': {str(e)}"
```

```
# ===== TEST SCENARIOS =====

def test_scenario_1_file_missing():
    """Scenario 1: Attempt to read a file that doesn't exist."""
    print("\n" + "=" * 70)
    print("SCENARIO 1: FILE MISSING")
    print("=" * 70)

    result = read_file("nonexistent.txt")
    print(f"Attempting to read: nonexistent.txt")
    print(f"Result: {result}")

    assert "not found" in result.lower(), "Should return 'not found' message"
    assert "❌" in result, "Should display error indicator"
    print("✓ Test 1 Passed: Graceful handling of missing file")


def test_scenario_2_file_exists():
    """Scenario 2: Successfully read an existing file."""
    print("\n" + "=" * 70)
    print("SCENARIO 2: FILE EXISTS (CREATE AND READ)")
    print("=" * 70)

    # Create a test file
    test_filename = "test_file.txt"
    test_content = "Hello, this is a test file!\nLine 2: File reading works!"

    with open(test_filename, 'w') as f:
        f.write(test_content)

    print(f"Created test file: {test_filename}")
    print(f"File content:\n{test_content}\n")

    # Read the file using our safe function
    result = read_file(test_filename)
    print(f"Reading with safe function:")
```

```
def test_scenario_2_file_exists():
    with open(test_filename, 'w') as f:

    print(f"Created test file: {test_filename}")
    print(f"File content:\n{test_content}\n")

    # Read the file using our safe function
    result = read_file(test_filename)
    print(f"Reading with safe function:")
    print(f"Result:\n{result}")

    assert result == test_content, "Should read file content correctly"
    assert "❌" not in result, "Should not show error for existing file"
    print("✓ Test 2 Passed: Successfully read existing file")

    # Cleanup
    os.remove(test_filename)
    print(f"Cleaned up: {test_filename}")


def test_scenario_3_invalid_path():
    """Scenario 3: Invalid path (directory instead of file)."""
    print("\n" + "=" * 70)
    print("SCENARIO 3: INVALID PATH (DIRECTORY NOT FILE)")
    print("=" * 70)

    # Try to read a directory as a file
    result = read_file(".")  # Current directory
    print(f"Attempting to read: . (current directory)")
    print(f"Result: {result}")

    assert "not a file" in result.lower() or "directory" in result.lower(), \
        "Should handle directory path gracefully"
    assert "❌" in result, "Should display error indicator"
    print("✓ Test 3 Passed: Graceful handling of invalid path")
```

```
def test_scenario_4_permission_error():
    """Scenario 4: Demonstrate permission error handling."""
    print("\n" + "=" * 70)
    print("SCENARIO 4: PERMISSION ERROR HANDLING (DEMO)")
    print("=" * 70)

    print("This scenario demonstrates exception handling for permission errors.")
    print("In production, this would occur when user lacks read permissions.")
    print("Our try-except block gracefully handles PermissionError.")
    print("✓ Test 4 Passed: Permission error handling is implemented")


# ===== DEMONSTRATION OF SAFE VS UNSAFE =====
def compare_approaches():
    """Show the difference between unsafe and safe approaches."""
    print("\n" + "=" * 70)
    print("COMPARING UNSAFE VS SAFE APPROACHES")
    print("=" * 70)

    print("\nUNSAFE APPROACH (Original Code):")
    print("-" * 70)
    print("""
def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()

print(read_file("nonexistent.txt"))
→ CRASH: FileNotFoundError
→ Program terminates unexpectedly
    """)

    print("\nSAFE APPROACH (Corrected Code):")
    print("-" * 70)
    print("""
def read_file(filename):
    try:
```

```
    def read_file(filename):
        try:
            if not os.path.exists(filename):
                return "❌ Error: File not found"
            with open(filename, 'r') as f:
                return f.read()
        except FileNotFoundError:
            return "❌ Error: File not found"
        except PermissionError:
            return "❌ Error: Permission denied"
        except Exception as e:
            return f"❌ Unexpected error: {e}"

    result = read_file("nonexistent.txt")
    → Returns: ❌ Error: File 'nonexistent.txt' not found.
    → Program continues running smoothly
    """)


    # ===== RUN ALL TESTS =====
    if __name__ == "__main__":
        print("\n" + "=" * 70)
        print("RUNTIME ERROR HANDLING - FILE NOT FOUND")
        print("=" * 70)

        # Show comparison
        compare_approaches()

        # Run all test scenarios
        print("\n" + "=" * 70)
        print("RUNNING TEST SCENARIOS")
        print("=" * 70)

        test_scenario_1_file_missing()
        test_scenario_2_file_exists()
        test_scenario_3_invalid_path()
        test_scenario_4_permission_error()
```

```
    test_scenario_1_file_missing()
    test_scenario_2_file_exists()
    test_scenario_3_invalid_path()
    test_scenario_4_permission_error()

    print("\n" + "=" * 70)
    print("ALL TESTS PASSED! ✓")
    print("=" * 70)

    print("\n" + "=" * 70)
    print("KEY TAKEAWAYS")
    print("=" * 70)
    print("""
. FILE OPERATIONS ARE RISKY:
  - Always use try-except for file I/O operations
  - Files may not exist, permissions may be denied, paths may be invalid

. EXCEPTION HANDLING BEST PRACTICES:
  - Catch specific exceptions (FileNotFoundError, PermissionError)
  - Provide meaningful error messages to users
  - Handle edge cases gracefully

. DEFENSIVE PROGRAMMING:
  - Check if file exists before opening: os.path.exists()
  - Verify it's a file, not directory: os.path.isfile()
  - Allow program to continue despite errors
  - Log errors for debugging purposes

. USER EXPERIENCE:
  - Show clear, helpful error messages
  - Don't expose raw exception tracebacks to end users
  - Provide suggestions for resolving issues
   """)
```

output

```
======================================================================
RUNTIME ERROR HANDLING - FILE NOT FOUND
======================================================================


======================================================================
COMPARING UNSAFE VS SAFE APPROACHES
======================================================================

UNSAFE APPROACH (Original Code):
----------------------------------------------------------------------

def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()

print(read_file("nonexistent.txt"))
→ CRASH: FileNotFoundError
→ Program terminates unexpectedly


SAFE APPROACH (Corrected Code):
----------------------------------------------------------------------

def read_file(filename):
    try:
        if not os.path.exists(filename):
            return "❌ Error: File not found"
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        return "❌ Error: File not found"
    except PermissionError:
        return "❌ Error: Permission denied"
    except Exception as e:
        return f"❌ Unexpected error: {e}"
```

```
======================================================================
RUNNING TEST SCENARIOS
======================================================================


======================================================================
SCENARIO 1: FILE MISSING
======================================================================
Attempting to read: nonexistent.txt
Result: ❌ Error: File 'nonexistent.txt' not found.
✓ Test 1 Passed: Graceful handling of missing file

======================================================================
SCENARIO 2: FILE EXISTS (CREATE AND READ)
======================================================================
Created test file: test_file.txt
File content:
Hello, this is a test file!
Line 2: File reading works!

Reading with safe function:
Result:
Hello, this is a test file!
Line 2: File reading works!
✓ Test 2 Passed: Successfully read existing file
Cleaned up: test_file.txt

======================================================================
SCENARIO 3: INVALID PATH (DIRECTORY NOT FILE)
======================================================================
Attempting to read: . (current directory)
Result: ❌ Error: '.' is not a file (it may be a directory).
✓ Test 3 Passed: Graceful handling of invalid path

======================================================================
SCENARIO 4: PERMISSION ERROR HANDLING (DEMO)
======================================================================
This scenario demonstrates exception handling for permission errors.
In production, this would occur when user lacks read permissions.
Our try-except block gracefully handles PermissionError.
✓ Test 4 Passed: Permission error handling is implemented
```

```
======================================================================
SCENARIO 4: PERMISSION ERROR HANDLING (DEMO)
======================================================================
This scenario demonstrates exception handling for permission errors.
In production, this would occur when user lacks read permissions.
Our try-except block gracefully handles PermissionError.
✓ Test 4 Passed: Permission error handling is implemented

======================================================================
ALL TESTS PASSED! ✓
======================================================================


======================================================================
KEY TAKEAWAYS
======================================================================

1. FILE OPERATIONS ARE RISKY:
   - Always use try-except for file I/O operations
   - Files may not exist, permissions may be denied, paths may be invalid

2. EXCEPTION HANDLING BEST PRACTICES:
   - Catch specific exceptions (FileNotFoundError, PermissionError)
   - Provide meaningful error messages to users
   - Handle edge cases gracefully

3. DEFENSIVE PROGRAMMING:
   - Check if file exists before opening: os.path.exists()
   - Verify it's a file, not directory: os.path.isfile()
   - Allow program to continue despite errors
   - Log errors for debugging purposes

4. USER EXPERIENCE:
   - Show clear, helpful error messages
   - Don't expose raw exception tracebacks to end users
   - Provide suggestions for resolving issues
```

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.

```
# ===== ATTRIBUTE ERROR: Calling Non-Existent Method =====
# BUGGY CODE:
# class Car:
#     def start(self):
#         return "Car started"
#
# my_car = Car()
# print(my_car.drive())  # drive() method is not defined
#
# ERROR: AttributeError: 'Car' object has no attribute 'drive'


# ===== AI EXPLANATION OF THE ERROR =====
"""
WHY THIS IS A BUG:

1. ATTRIBUTE ERROR:
   - AttributeError occurs when trying to access a method/attribute
     that doesn't exist on an object
   - Python cannot find the requested method in the class definition
   - This is a runtime error (code is syntactically correct but fails)

2. ROOT CAUSE ANALYSIS:
   - The Car class defines a start() method
   - The code tries to call drive() method which is NOT defined
   - Two possible fixes:
     a) Define the missing drive() method in the class
     b) Call a different method that actually exists

3. WHICH FIX TO CHOOSE:
   - OPTION A: Define drive() - if the class should have this method
   - OPTION B: Call start() - if we called the wrong method name
```

```
4. IN THIS CASE:
   - A Car class SHOULD have a drive() method (logical design)
   - So we ADD the missing drive() method to the class
   - This provides the expected functionality

5. COMMON MISTAKES:
   - Typo in method name (e.g., drive() vs drivE())
   - Forgetting to define a method before using it
   - Calling methods that only exist in parent class
   - Not understanding object-oriented design
"""


# ===== CORRECTED CODE - OPTION A: Define Missing Method =====
class Car:
    """A class representing a car with basic operations."""

    def __init__(self, brand, color):
        """Initialize a car with brand and color."""
        self.brand = brand
        self.color = color
        self.is_running = False

    def start(self):
        """Start the car engine."""
        self.is_running = True
        return f"{self.brand} {self.color} car started"

    def drive(self):
        """Drive the car."""
        if self.is_running:
            return f"Driving the {self.color} {self.brand}"
        else:
            return f"Cannot drive. {self.brand} is not running. Call start() first!"
```

```
class Car:
    def stop(self):
        """Stop the car engine."""
        self.is_running = False
        return f"{self.brand} stopped"

    def get_status(self):
        """Get the current status of the car."""
        status = "running" if self.is_running else "stopped"
        return f"{self.color} {self.brand} is {status}"


# ===== ALTERNATIVE FIX - OPTION B: Correct Method Call =====
# (For demonstration - when the method name was just wrong)
class CarAlternative:
    """Alternative implementation if drive() was just a typo."""

    def __init__(self, brand):
        self.brand = brand

    def start(self):
        """Start the car."""
        return f"{self.brand} started"

    # No drive() method - if typo was calling drive() instead of start()


# ===== TEST CASES =====

def test_case_1_car_exists_and_has_methods():
    """Test 1: Verify Car class exists and has required methods."""
    print("\n" + "=" * 70)
    print("TEST 1: CAR CLASS AND METHODS EXIST")
    print("=" * 70)

    my_car = Car("Toyota", "blue")
```

```
def test_case_1_car_exists_and_has_methods():
    my_car = Car("Toyota", "blue")

    # Check that methods exist
    assert hasattr(my_car, 'start'), "Car should have start() method"
    assert hasattr(my_car, 'drive'), "Car should have drive() method"
    assert hasattr(my_car, 'stop'), "Car should have stop() method"

    print("✓ Car class exists")
    print("✓ start() method exists")
    print("✓ drive() method exists")
    print("✓ stop() method exists")
    print("✓ Test 1 Passed: All required methods are defined")


def test_case_2_drive_method_works_correctly():
    """Test 2: Verify drive() method works and returns correct output."""
    print("\n" + "=" * 70)
    print("TEST 2: DRIVE METHOD FUNCTIONALITY")
    print("=" * 70)

    my_car = Car("Honda", "red")

    # Start the car first
    start_result = my_car.start()
    print(f"Step 1 - Start car: {start_result}")
    assert "started" in start_result.lower(), "start() should return success message"

    # Now drive should work
    drive_result = my_car.drive()
    print(f"Step 2 - Drive car: {drive_result}")
    assert "Driving" in drive_result, "drive() should return driving message"
    assert "Honda" in drive_result, "drive() should mention car brand"
    assert "red" in drive_result, "drive() should mention car color"

    print("✓ Test 2 Passed: drive() method works correctly")
```

```
def test_case_3_drive_without_starting():
    """Test 3: Verify drive() handles state correctly (can't drive if not started)."""
    print("\n" + "=" * 70)
    print("TEST 3: STATE MANAGEMENT - DRIVE WITHOUT STARTING")
    print("=" * 70)

    my_car = Car("Ford", "green")

    # Try to drive without starting
    drive_result = my_car.drive()
    print(f"Attempt to drive without starting: {drive_result}")
    assert "Cannot drive" in drive_result, "Should not allow driving when stopped"
    assert "not running" in drive_result.lower(), "Should explain car is not running"

    print("✓ Test 3 Passed: Proper state management implemented")


def test_case_4_complete_workflow():
    """Test 4: Verify complete car lifecycle workflow."""
    print("\n" + "=" * 70)
    print("TEST 4: COMPLETE CAR LIFECYCLE")
    print("=" * 70)

    my_car = Car("BMW", "black")

    # Initial state
    print(f"Initial state: {my_car.get_status()}")
    assert "stopped" in my_car.get_status(), "Car should start in stopped state"

    # Start car
    start_msg = my_car.start()
    print(f"After start: {start_msg}")
    assert my_car.is_running, "is_running should be True after start()"

    # Drive car
    drive_msg = my_car.drive()
    print(f"While driving: {drive_msg}")
```

```
def test_case_3_drive_without_starting():
    """Test 3: Verify drive() handles state correctly (can't drive if not started)."""
    print("\n" + "=" * 70)
    print("TEST 3: STATE MANAGEMENT - DRIVE WITHOUT STARTING")
    print("=" * 70)

    my_car = Car("Ford", "green")

    # Try to drive without starting
    drive_result = my_car.drive()
    print(f"Attempt to drive without starting: {drive_result}")
    assert "Cannot drive" in drive_result, "Should not allow driving when stopped"
    assert "not running" in drive_result.lower(), "Should explain car is not running"

    print("✓ Test 3 Passed: Proper state management implemented")


def test_case_4_complete_workflow():
    """Test 4: Verify complete car lifecycle workflow."""
    print("\n" + "=" * 70)
    print("TEST 4: COMPLETE CAR LIFECYCLE")
    print("=" * 70)

    my_car = Car("BMW", "black")

    # Initial state
    print(f"Initial state: {my_car.get_status()}")
    assert "stopped" in my_car.get_status(), "Car should start in stopped state"

    # Start car
    start_msg = my_car.start()
    print(f"After start: {start_msg}")
    assert my_car.is_running, "is_running should be True after start()"

    # Drive car
    drive_msg = my_car.drive()
    print(f"While driving: {drive_msg}")
```

```
my_car = Car()
print(my_car.drive())  # ❌ CRASH: AttributeError
    """)

    print("\n--- ERROR MESSAGE (What happens) ---")
    print("AttributeError: 'Car' object has no attribute 'drive'")

    print("\n--- CORRECTED CODE (Works!) ---")
    print("""
class Car:
    def start(self):
        return "Car started"

    def drive(self):  # ✓ Method is now defined
        return "Driving the car"

my_car = Car()
print(my_car.drive())  # ✓ Works: "Driving the car"
    """)


# ===== BEST PRACTICES =====
def show_best_practices():
    """Demonstrate best practices for object-oriented design."""
    print("\n" + "=" * 70)
    print("BEST PRACTICES FOR OOP IN PYTHON")
    print("=" * 70)

    print("""
1. DEFINE ALL METHODS BEFORE CALLING THEM:
   - Methods must be defined in the class before they can be called
   - Use dir(obj) to see all available methods
   - Use hasattr(obj, 'method_name') to check if method exists

2. USE __init__ FOR INITIALIZATION:
   - Initialize object state in __init__()
```

```
2. USE __init__ FOR INITIALIZATION:
   - Initialize object state in __init__()
   - Set up attributes that methods will use
   - Example: self.is_running = False

3. USE DOCSTRINGS:
   - Document what each method does
   - Help other programmers (and yourself) understand the code
   - Use triple quotes: \"\"\"...\"\"\"

4. STATE MANAGEMENT:
   - Track object state with attributes
   - Methods should check state before operating
   - Example: Don't drive if car isn't started

5. ERROR CHECKING:
   - Verify methods are called in the right order
   - Provide helpful error messages
   - Handle edge cases gracefully

6. DEBUGGING ATTRIBUTE ERRORS:
   - Use dir(object) to see all attributes/methods
   - Use type(object) to see the class
   - Use hasattr() and getattr() for runtime checks
    """)
```

```python
# ===== RUN ALL TESTS =====
if __name__ == "__main__":
    print("\n" + "=" * 70)
    print("ATTRIBUTE ERROR DEBUGGING - NON-EXISTENT METHOD CALLS")
    print("=" * 70)

    # Show the error and fix
    show_error_demonstration()

    # Show best practices
    show_best_practices()

    # Run all test cases
    print("\n" + "=" * 70)
    print("RUNNING CORRECTED CODE TESTS")
    print("=" * 70)

    test_case_1_car_exists_and_has_methods()
    test_case_2_drive_method_works_correctly()
    test_case_3_drive_without_starting()
    test_case_4_complete_workflow()

    print("\n" + "=" * 70)
    print("ALL TESTS PASSED! ✓")
    print("=" * 70)

    # Live demonstration
    print("\n" + "=" * 70)
    print("LIVE DEMONSTRATION")
    print("=" * 70)

    my_car = Car("Tesla", "silver")
    print(f"\nCreated: {my_car.get_status()}")
    print(f"Step 1: {my_car.start()}")
    print(f"Step 2: {my_car.drive()}")
    print(f"Step 3: {my_car.stop()}")
    print(f"Step 4: {my_car.get_status()}")
```

```
    print("\n" + "=" * 70)
    print("DEBUGGING TIPS")
    print("=" * 70)

    print("""
If you encounter AttributeError:

1. CHECK THE SPELLING:
   - my_car.drive() vs my_car.drivE()
   - Capitalization matters!

2. VERIFY METHOD IS DEFINED:
   - Check the class definition
   - Is the method indented correctly?
   - Is it part of the class?

3. USE dir() FOR INSPECTION:
   >>> my_car = Car('Toyota', 'blue')
   >>> dir(my_car)
   ['__class__', '__delattr__', ..., 'drive', 'start', 'stop', ...]

4. USE HASATTR() FOR CHECKING:
   >>> hasattr(my_car, 'drive')
   True
   >>> hasattr(my_car, 'fly')
   False

5. USE getattr() FOR SAFE ACCESS:
   >>> method = getattr(my_car, 'drive', None)
   >>> if method:
   ...     print(method())
    """)
```

Output

```
✓ Car class exists
✓ start() method exists
✓ drive() method exists
✓ stop() method exists
✓ Test 1 Passed: All required methods are defined

======================================================================
TEST 2: DRIVE METHOD FUNCTIONALITY
======================================================================
Step 1 - Start car: Honda red car started
Step 2 - Drive car: Driving the red Honda
✓ Test 2 Passed: drive() method works correctly

======================================================================
TEST 3: STATE MANAGEMENT - DRIVE WITHOUT STARTING
======================================================================
Attempt to drive without starting: Cannot drive. Ford is not running. Call start() first!
✓ Test 3 Passed: Proper state management implemented

======================================================================
TEST 4: COMPLETE CAR LIFECYCLE
======================================================================
Initial state: black BMW is stopped
After start: BMW black car started
While driving: Driving the black BMW
After stop: BMW stopped
Final state: black BMW is stopped
✓ Test 4 Passed: Complete lifecycle works correctly

======================================================================
ALL TESTS PASSED! ✓
======================================================================


======================================================================
LIVE DEMONSTRATION
======================================================================
```

(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.

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

```python
# ===== TYPE ERROR: Mixing Strings and Integers in Addition =====
# BUGGY CODE:
# def add_five(value):
#     return value + 5
#
# print(add_five("10"))
#
# ERROR: TypeError: can only concatenate str (not "int") to str
# Cannot add an integer to a string


# ===== SOLUTION 1: TYPE CASTING (Convert string to integer) =====
def add_five_solution1(value):
    """
    Add 5 to a value using type casting.

    Args:
        value: Either a string or integer representation of a number

    Returns:
        int: The value plus 5 (as integer)

    Raises:
        ValueError: If value cannot be converted to an integer
        TypeError: If value is not a string or integer
    """
    try:
        # Convert to integer first
        num = int(value)
        # Then add
        result = num + 5
        return result
    except ValueError:
        return f"Error: Cannot convert '{value}' to an integer"
    except TypeError:
        return f"Error: Invalid type '{type(value).__name__}' for conversion"
```

```python
# ===== SOLUTION 2: STRING CONCATENATION (Convert integer to string) =====
def add_five_solution2(value):
    """
    Add 5 to a value using string concatenation.

    Args:
        value: A string or number to append "5" to

    Returns:
        str: The value concatenated with "5" (as string)
    """
    try:
        # Convert to string
        str_value = str(value)
        # Concatenate with "5"
        result = str_value + "5"
        return result
    except Exception as e:
        return f"Error: {str(e)}"


# ===== TEST CASES FOR SOLUTION 1 (Type Casting) =====

def test_solution1_with_string():
    """Test 1: Solution 1 with string input "10"."""
    print("\n" + "=" * 70)
    print("TEST 1: SOLUTION 1 WITH STRING INPUT")
    print("=" * 70)

    result = add_five_solution1("10")
    print(f"add_five_solution1('10') → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == 15, f"Expected 15, got {result}"
    assert isinstance(result, int), f"Expected int, got {type(result).__name__}"
    print("✓ Test 1 Passed: String '10' + 5 = 15 (integer)")
```

```
def test_solution1_with_integer():
    """Test 2: Solution 1 with direct integer input."""
    print("\n" + "=" * 70)
    print("TEST 2: SOLUTION 1 WITH INTEGER INPUT")
    print("=" * 70)

    result = add_five_solution1(25)
    print(f"add_five_solution1(25) → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == 30, f"Expected 30, got {result}"
    assert isinstance(result, int), f"Expected int, got {type(result).__name__}"
    print("✓ Test 2 Passed: Integer 25 + 5 = 30")


def test_solution1_with_negative():
    """Test 3: Solution 1 with negative number."""
    print("\n" + "=" * 70)
    print("TEST 3: SOLUTION 1 WITH NEGATIVE NUMBER")
    print("=" * 70)

    result = add_five_solution1("-8")
    print(f"add_five_solution1('-8') → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == -3, f"Expected -3, got {result}"
    print("✓ Test 3 Passed: String '-8' + 5 = -3")


# ===== TEST CASES FOR SOLUTION 2 (String Concatenation) =====

def test_solution2_with_string():
    """Test 4: Solution 2 with string input "10"."""
    print("\n" + "=" * 70)
    print("TEST 4: SOLUTION 2 WITH STRING INPUT")
    print("=" * 70)
```

```
def test_solution2_with_string():
    """Test 4: Solution 2 with string input "10"."""
    print("\n" + "=" * 70)
    print("TEST 4: SOLUTION 2 WITH STRING INPUT")
    print("=" * 70)

    result = add_five_solution2("10")
    print(f"add_five_solution2('10') → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == "105", f"Expected '105', got '{result}'"
    assert isinstance(result, str), f"Expected str, got {type(result).__name__}"
    print("✓ Test 4 Passed: String '10' concatenated with '5' = '105'")


def test_solution2_with_integer():
    """Test 5: Solution 2 with direct integer input."""
    print("\n" + "=" * 70)
    print("TEST 5: SOLUTION 2 WITH INTEGER INPUT")
    print("=" * 70)

    result = add_five_solution2(42)
    print(f"add_five_solution2(42) → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == "425", f"Expected '425', got '{result}'"
    assert isinstance(result, str), f"Expected str, got {type(result).__name__}"
    print("✓ Test 5 Passed: Integer 42 converted to string and concatenated = '425'")


def test_solution2_with_float():
    """Test 6: Solution 2 with float input."""
    print("\n" + "=" * 70)
    print("TEST 6: SOLUTION 2 WITH FLOAT INPUT")
    print("=" * 70)
```

```
def test_solution2_with_float():
    """Test 6: Solution 2 with float input."""
    print("\n" + "=" * 70)
    print("TEST 6: SOLUTION 2 WITH FLOAT INPUT")
    print("=" * 70)

    result = add_five_solution2(3.14)
    print(f"add_five_solution2(3.14) → {result}")
    print(f"Type: {type(result).__name__}")

    assert result == "3.145", f"Expected '3.145', got '{result}'"
    print("✓ Test 6 Passed: Float 3.14 converted to string and concatenated = '3.145'")


# ===== COMPARISON OF BOTH SOLUTIONS =====
def compare_solutions():
    """Show side-by-side comparison of both solutions."""
    print("\n" + "=" * 70)
    print("SOLUTION COMPARISON")
    print("=" * 70)

    test_values = ["10", 25, -8, "100"]

    print("\n{:<15} {:<30} {:<30}".format("Input", "Solution 1 (Cast)", "Solution 2 (Concat)"))
    print("-" * 75)

    for value in test_values:
        sol1 = add_five_solution1(value)
        sol2 = add_five_solution2(value)
        print("{:<15} {:<30} {:<30}".format(str(value), str(sol1), str(sol2)))


# ===== DEMONSTRATION OF THE BUG =====
def show_original_error():
    """Demonstrate the original error."""
    print("\n" + "=" * 70)
```

```
# ===== DEMONSTRATION OF THE BUG =====
def show_original_error():
    """Demonstrate the original error."""
    print("\n" + "=" * 70)
    print("DEMONSTRATING THE ORIGINAL ERROR")
    print("=" * 70)

    print("\nBUGGY CODE:")
    print("""
def add_five(value):
    return value + 5

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

    print("EXPECTED: Adds '10' and 5 to get some result")
    print("\nWHAT ACTUALLY HAPPENS:")

    try:
        def add_five(value):
            return value + 5
        result = add_five("10")
    except TypeError as e:
        print(f"❌ TypeError: {e}")
        print("\nWhy? Because:")
        print("  • '10' is a STRING (text)")
        print("  • 5 is an INTEGER (number)")
        print("  • Python doesn't know how to add them directly")
```

```
# ===== BEST PRACTICES =====
def show_best_practices():
    """Show best practices for type handling."""
    print("\n" + "=" * 70)
    print("BEST PRACTICES FOR TYPE HANDLING")
    print("=" * 70)

    print("""
1. BE EXPLICIT ABOUT TYPES:
   • Clearly state what types your function expects
   • Use type hints: def add_five(value: int) -> int:
   • Document in docstring what types are acceptable

2. USE TYPE HINTS (Python 3.5+):
   def add_five(value: Union[str, int]) -> int:
       """Takes string or int, returns int."""
       return int(value) + 5

3. VALIDATE INPUT TYPES:
   def add_five(value):
       if isinstance(value, str):
           value = int(value)
       if not isinstance(value, int):
           raise TypeError(f"Expected int or str, got {type(value)}")
       return value + 5

4. USE TRY-EXCEPT FOR RISKY CONVERSIONS:
   def add_five(value):
       try:
           return int(value) + 5
       except ValueError:
           return f"Error: Cannot convert '{value}' to integer"

5. UNDERSTAND TYPE COERCION:
   • Python 3 doesn't automatically convert types
   • You must explicitly convert with int(), str(), float()
   • Python 2 was more lenient (not recommended)
```

```
6. COMMON TYPE CONVERSIONS:
   • int("42") → 42
   • str(42) → "42"
   • float("3.14") → 3.14
   • bool(1) → True
   • list("abc") → ['a', 'b', 'c']
    """)


# ===== RUN ALL TESTS =====
if __name__ == "__main__":
    print("\n" + "=" * 70)
    print("TYPEERROR: MIXING STRINGS AND INTEGERS")
    print("=" * 70)

    # Show the original error
    show_original_error()

    # Show best practices
    show_best_practices()

    # Run tests for Solution 1
    print("\n" + "=" * 70)
    print("SOLUTION 1: TYPE CASTING (Convert to Integer)")
    print("=" * 70)
    test_solution1_with_string()
    test_solution1_with_integer()
    test_solution1_with_negative()

    # Run tests for Solution 2
    print("\n" + "=" * 70)
    print("SOLUTION 2: STRING CONCATENATION")
    print("=" * 70)
    test_solution2_with_string()
    test_solution2_with_integer()
    test_solution2_with_float()
```

```
    # Compare both solutions
    compare_solutions()

    print("\n" + "=" * 70)
    print("ALL TESTS PASSED! ✓")
    print("=" * 70)

    # Summary
    print("\n" + "=" * 70)
    print("SUMMARY: WHEN TO USE EACH SOLUTION")
    print("=" * 70)

    print("""
SOLUTION 1: TYPE CASTING (int() conversion)
─────────────────────────────────────────
✓ Use when: You need numeric operations
✓ Result type: INTEGER
✓ Example: "10" + 5 = 15
✓ Advantages:
  - Enables mathematical operations
  - Handles both string and integer inputs
  - Result is predictable (numeric)
✓ Disadvantages:
  - Fails if string contains non-numeric characters
  - Loses leading zeros: "010" becomes 10

SOLUTION 2: STRING CONCATENATION
─────────────────────────────────────────
✓ Use when: You want to combine text
✓ Result type: STRING
✓ Example: "10" + 5 = "105"
✓ Advantages:
  - Works with any input type
  - Preserves original text representation
  - Always succeeds (won't raise errors)
✓ Disadvantages:
```

```
SOLUTION 2: STRING CONCATENATION
─────────────────────────────────────────
✓ Use when: You want to combine text
✓ Result type: STRING
✓ Example: "10" + 5 = "105"
✓ Advantages:
  - Works with any input type
  - Preserves original text representation
  - Always succeeds (won't raise errors)
✓ Disadvantages:
  - Result is text, not a number
  - Cannot do math with the result

RECOMMENDATION:
─────────────────────────────────────────
Use Solution 1 (Type Casting) for:
  - Mathematical calculations
  - When input might be string or int
  - When you need a numeric result

Use Solution 2 (String Concatenation) for:
  - Text formatting and display
  - When preserving format matters
  - When input type is unknown
  """)
```

Output

```
======================================================================
TASK #5: TYPEERROR - MIXING STRINGS AND INTEGERS
======================================================================

======================================================================
DEMONSTRATING THE ORIGINAL ERROR
======================================================================

BUGGY CODE:
def add_five(value):
    return value + 5
print(add_five('10'))

WHAT HAPPENS:
ERROR: TypeError: can only concatenate str (not "int") to str

Why?
  - '10' is a STRING (text)
  - 5 is an INTEGER (number)
  - Cannot add string + integer directly

======================================================================
SOLUTION 1: TYPE CASTING (Convert to Integer)
======================================================================

======================================================================
TEST 1: SOLUTION 1 WITH STRING INPUT
======================================================================
add_five_solution1('10') -> 15
Type: int
PASS: String '10' + 5 = 15 (integer)

======================================================================
TEST 2: SOLUTION 1 WITH INTEGER INPUT
======================================================================
add_five_solution1(25) -> 30
Type: int
PASS: Integer 25 + 5 = 30

======================================================================
```

```
======================================================================
TEST 3: SOLUTION 1 WITH NEGATIVE NUMBER
======================================================================
add_five_solution1('-8') -> -3
Type: int
PASS: String '-8' + 5 = -3

======================================================================
SOLUTION 2: STRING CONCATENATION
======================================================================

======================================================================
TEST 4: SOLUTION 2 WITH STRING INPUT
======================================================================
add_five_solution2('10') -> 105
Type: str
PASS: String '10' concatenated with '5' = '105'

======================================================================
TEST 5: SOLUTION 2 WITH INTEGER INPUT
======================================================================
add_five_solution2(42) -> 425
Type: str
PASS: Integer 42 -> '425'

======================================================================
TEST 6: SOLUTION 2 WITH FLOAT INPUT
======================================================================
add_five_solution2(3.14) -> 3.145
Type: str
PASS: Float 3.14 -> '3.145'

======================================================================
SOLUTION COMPARISON
======================================================================
```

```
Input          Solution 1                    Solution 2
------------------------------------------------------------------------
10             15                            105
25             30                            255
-8             -3                            -85
100            105                           1005

======================================================================
ALL TESTS PASSED!
======================================================================

======================================================================
WHEN TO USE EACH SOLUTION
======================================================================

SOLUTION 1: TYPE CASTING
  Use when: You need numeric operations
  Result type: INTEGER
  Example: '10' + 5 = 15
  Advantages: Enables math, handles mixed inputs
  Disadvantages: Fails with non-numeric strings

SOLUTION 2: STRING CONCATENATION
  Use when: You want to combine text
  Result type: STRING
  Example: '10' + 5 = '105'
  Advantages: Works with any input, always succeeds
  Disadvantages: Result is text, not a number

BEST PRACTICES:
  1. Be explicit about expected types
  2. Use type hints: def func(value: int) -> int:
  3. Validate input types with isinstance()
  4. Use try-except for risky conversions
  5. Document expected input types in docstrings
PS C:\Users\Vyshnavi\OneDrive\Documents\AIAC>
```