AI Assisted Coding Assignment- 1

> 2303A51543 BT: 29 VELDI.HRUTHIKA 09.01.2026

▼ **Task 1: AI-Generated Logic Without Modularization (String Reversal Without Functions)**

❖ Scenario

You are developing a basic text-processing utility for a messaging application.

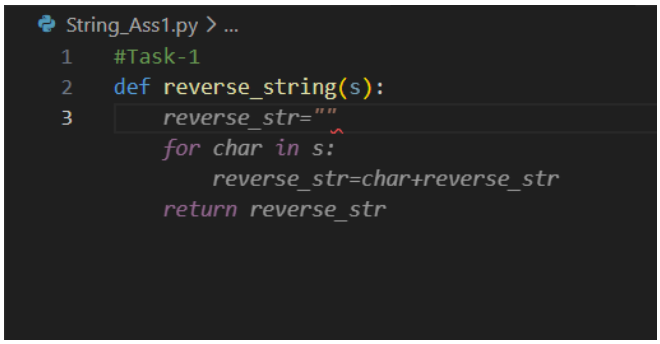
❖ Task Description

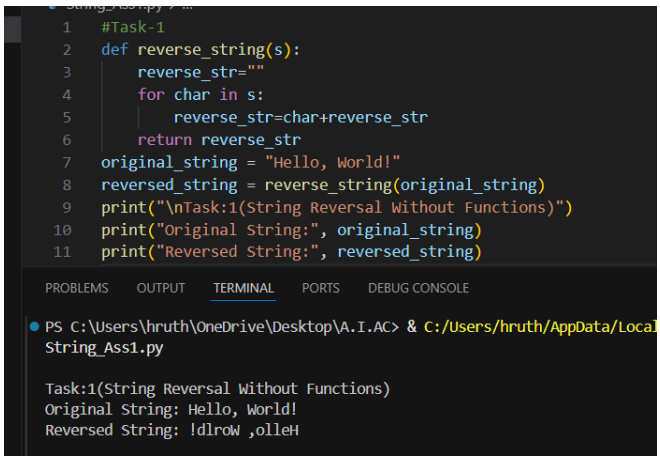
Use GitHub Copilot to generate a Python program that:

➢ Reverses a given string
➢ Accepts user input
➢ Implements the logic directly in the main code
➢ Does not use any user-defined functions

❖ Expected Output

➢ Correct reversed string
➢ Screenshots showing Copilot-generated code suggestions
➢ Sample inputs and outputs






```
#Task-1
def reverse_string(s):
```

```
    reverse_str=""
    for char in s:
        reverse_str=char+reverse_str
    return reverse_str
original_string = "Hello, World!"
reversed_string = reverse_string(original_string)
print("\nTask:1(String Reversal Without Functions)")
print("Original String:", original_string)
print("Reversed String:", reversed_string)
```

- **Function Definition**: A function called `reverse_string(s)` is created to reverse the string.
- **Empty String**: Inside the function, an empty string `reverse_str` is initialized to hold the reversed string.
- **Loop**: The code loops through each character in the original string `s`.
- **Reversing Process**: In each loop, the current character is added at the beginning of `reverse_str` to reverse the string.
- **Return**: Once all characters are processed, the reversed string is returned.
- **Output**: The original and reversed strings are printed out.

▼ **Task 2: Efficiency & Logic Optimization (Readability Improvement)**

❖ Scenario

The code will be reviewed by other developers.

❖ Task Description

Examine the Copilot-generated code from Task 1 and improve it by:

➢ Removing unnecessary variables

➢ Simplifying loop or indexing logic

➢ Improving readability

➢ Use Copilot prompts like:

▪ “Simplify this string reversal code”

▪ “Improve readability and efficiency”

Hint:

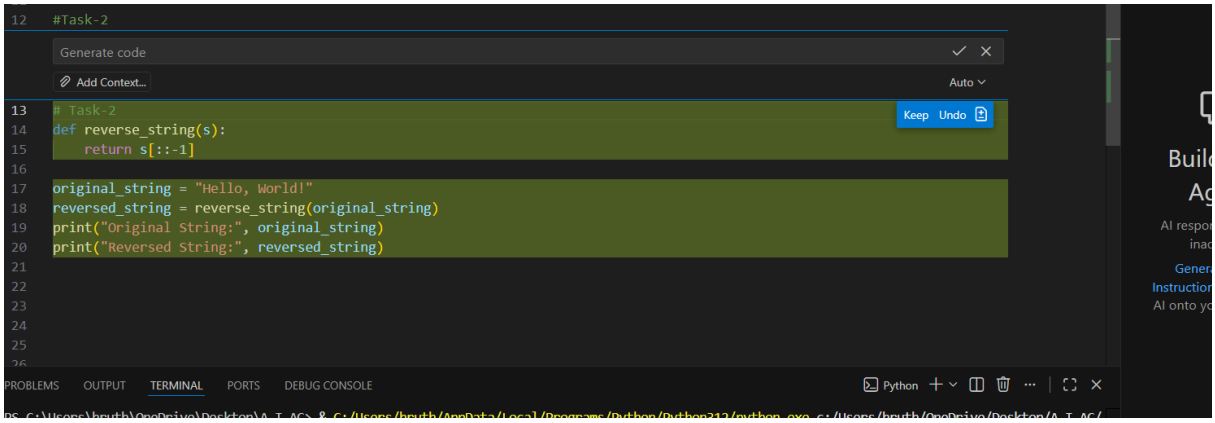
Prompt Copilot with phrases like

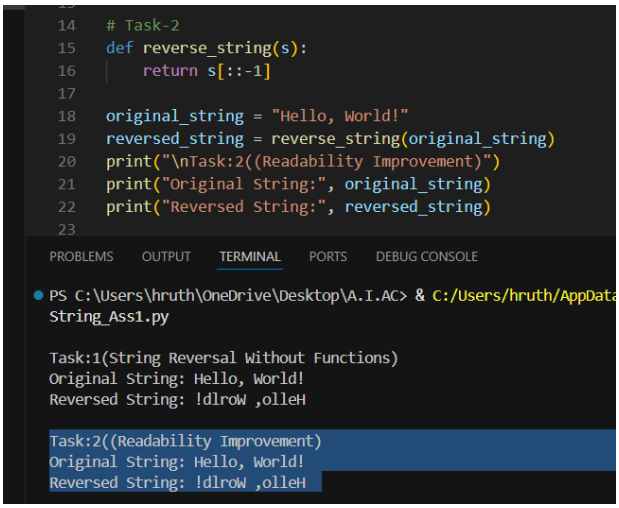
“optimize this code”, “simplify logic”, or “make it more readable”

❖ Expected Output

➢ Original and optimized code versions

➢ Explanation of how the improvements reduce time complexity






```
# Task-2
def reverse_string(s):
    return s[::-1]

original_string = "Hello, World!"
reversed_string = reverse_string(original_string)
print("\nTask:2((Readability Improvement)")
print("Original String:", original_string)
print("Reversed String:", reversed_string)
```

- **Function Definition**: A function called `reverse_string(s)` is created to reverse the string.
- **Slicing**: Inside the function, the string `s` is reversed using Python's slicing feature (`s[::-1]`).
  - `s[::-1]` means take the entire string but with a step of `1`, which reverses the string.
- **Return**: The reversed string is returned.
- **Output**: The original and reversed strings are printed out.

▼ **Task 3: Modular Design Using AI Assistance (String Reversal Using Functions)**

❖ Scenario

The string reversal logic is needed in multiple parts of an application.

❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

➢ Uses a user-defined function to reverse a string

➢ Returns the reversed string

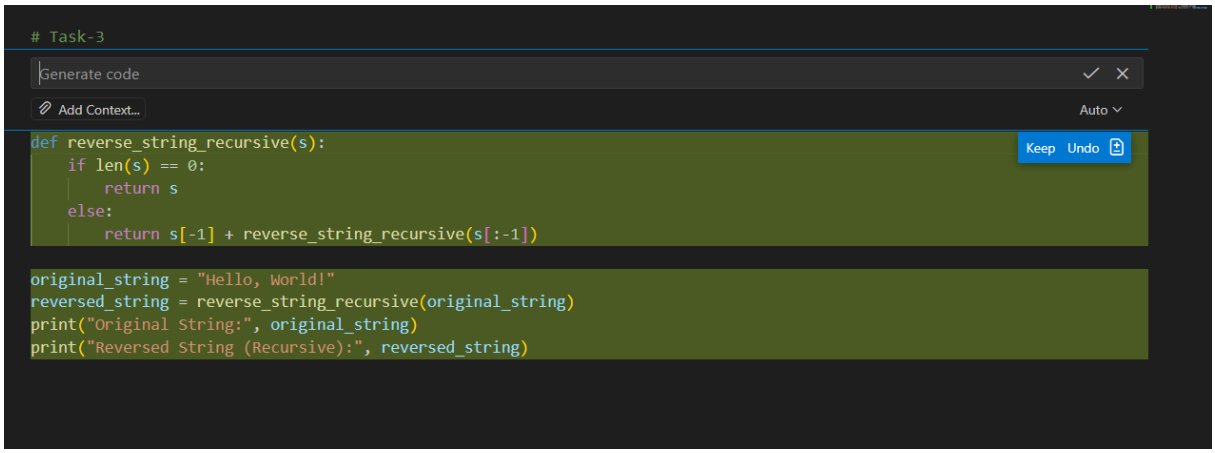
➢ Includes meaningful comments (AI-assisted)

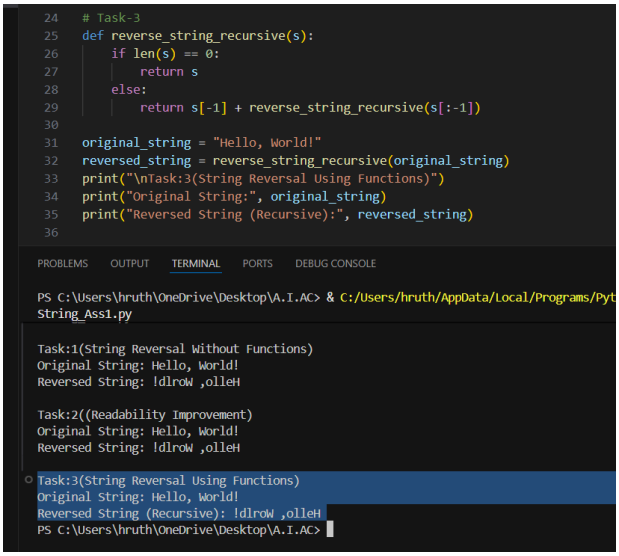
❖ Expected Output

➢ Correct function-based implementation

➢ Screenshots documenting Copilot's function generation

➢ Sample test cases and outputs






```
# Task-3
def reverse_string_recursive(s):
    if len(s) == 0:
        return s
    else:
        return s[-1] + reverse_string_recursive(s[:-1])

original_string = "Hello, World!"
reversed_string =
reverse_string_recursive(original_string)
print("\nTask:3(String Reversal Using Functions)")
print("Original String:", original_string)
print("Reversed String (Recursive):",
reversed_string)
```

- **Function Definition**: A recursive function called `reverse_string_recursive(s)` is created to reverse the string.
- **Base Case**: If the string `s` is empty (length 0), it simply returns the empty string (this stops the recursion).
- **Recursive Case**: If the string is not empty:

- It takes the last character (`s[-1]`) and adds it to the result of calling the function again with the rest of the string (`s[:-1]`).

- **Return**: This process keeps repeating (recursion) until the string is reversed completely.
- **Output**: The original and reversed strings are printed out.

▼ **Task 4: Comparative Analysis – Procedural vs Modular Approach (With vs Without Functions)**

❖ Scenario

You are asked to justify design choices during a code review.

❖ Task Description

Compare the Copilot-generated programs:

➢ Without functions (Task 1)

➢ With functions (Task 3)

Analyze them based on:

➢ Code clarity

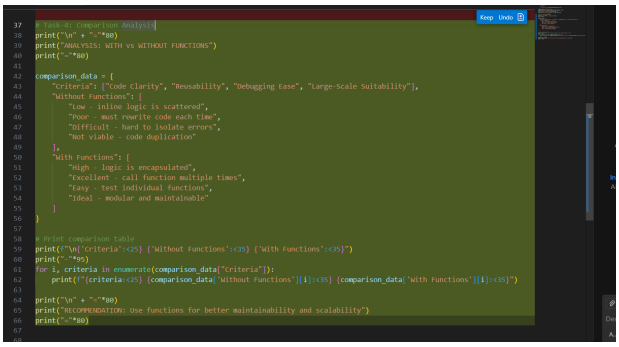
➢ Reusability

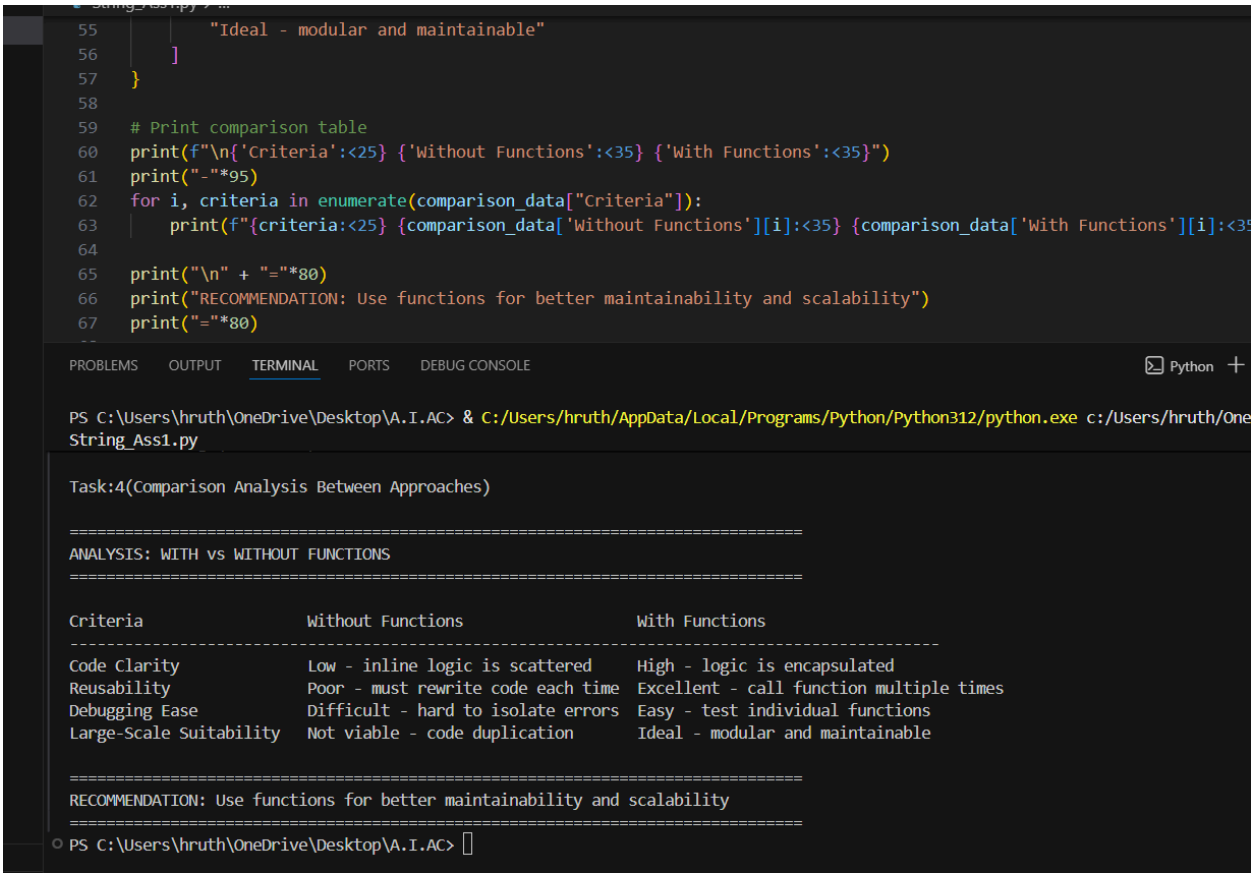
➢ Debugging ease

➢ Suitability for large-scale applications

❖ Expected Output

Comparison table or short analytical report






```
# Task-4: Comparison Analysis
print("\nTask:4(Comparison Analysis Between Approaches)")
print("\n" + "="*80)
print("ANALYSIS: WITH vs WITHOUT FUNCTIONS")
print("="*80)

comparison_data = {
    "Criteria": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability"],
    "Without Functions": [
        "Low - inline logic is scattered",
        "Poor - must rewrite code each time",
        "Difficult - hard to isolate errors",
        "Not viable - code duplication"
    ],
    "With Functions": [
        "High - logic is encapsulated",
        "Excellent - call function multiple times",
        "Easy - test individual functions",
        "Ideal - modular and maintainable"
    ]
}

# Print comparison table
```

```
print(f"\n{'Criteria':<25} {'Without Functions':<35} {'With Functions':<35}")
print("-"*95)
for i, criteria in enumerate(comparison_data["Criteria"]):
    print(f"{criteria:<25} {comparison_data['Without Functions'][i]:<35} {comparison_data['With Functions'][i]:<35}")

print("\n" + "="*80)
print("RECOMMENDATION: Use functions for better maintainability and scalability")
print("="*80)
```

- **Comparison Criteria**:
  - **Code Clarity**: Functions make code clearer, while without functions it's scattered.
  - **Reusability**: Functions can be reused, but without them you need to rewrite the code.
  - **Debugging Ease**: Functions are easier to debug, while without them it's tough.
  - **Large-Scale Suitability**: Functions are ideal for large-scale projects, whereas without them, it leads to code duplication.
- **Conclusion**: Use functions for better clarity, reusability, easier debugging, and scalability.

▼ Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to String Reversal)

❖ Scenario

Your mentor wants to evaluate how AI handles alternative logic paths.

❖ Task Description

Prompt GitHub Copilot to generate:

➢ A loop-based string reversal approach

➢ A built-in / slicing-based string reversal approach

❖ Expected Output

➢ Two correct implementations

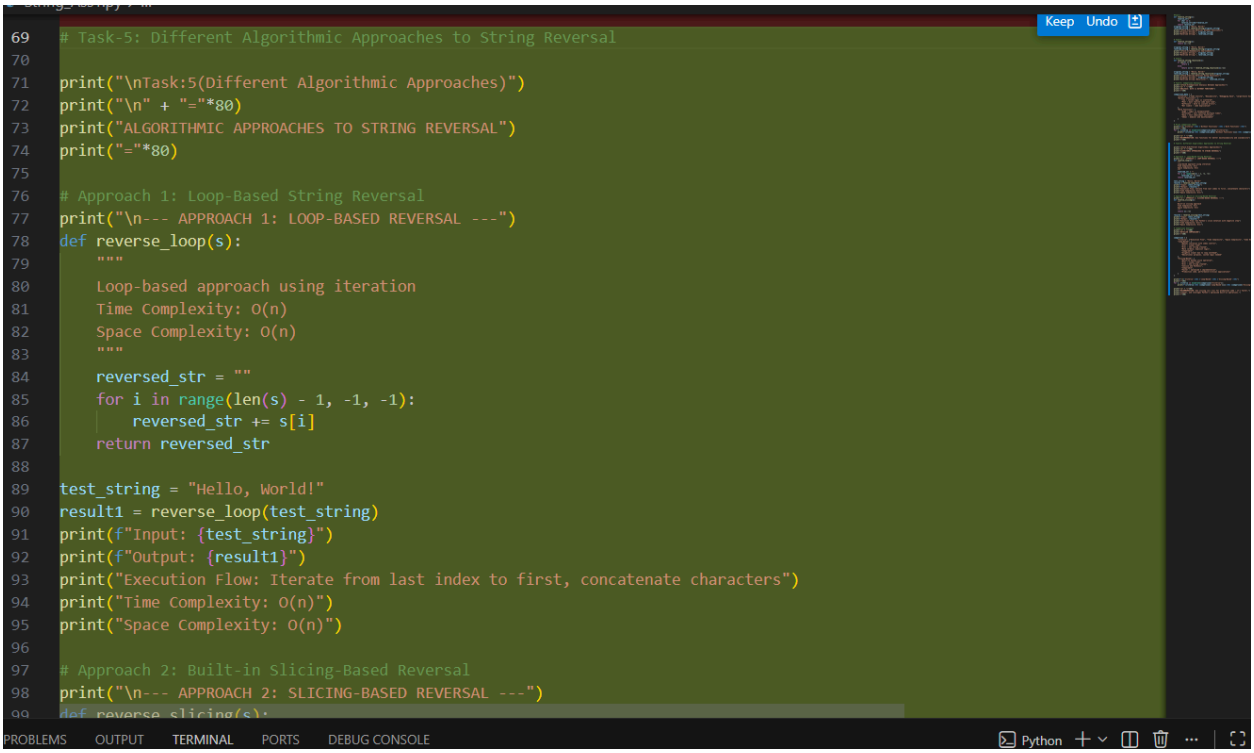
➢ Comparison discussing:

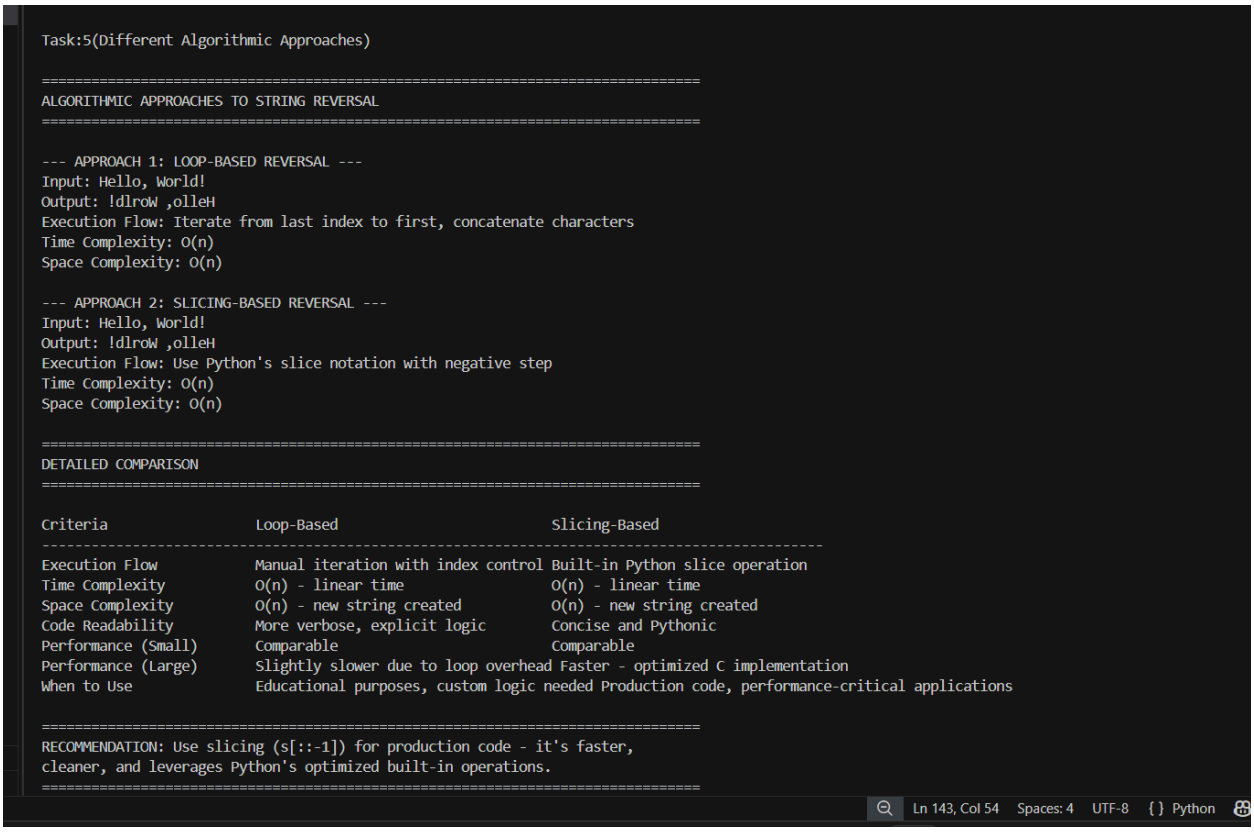
▪ Execution flow

▪ Time complexity

▪ Performance for large inputs

▪ When each approach is appropriate





Task-5: Different Algorithmic Approaches to String Reversal

```
print("\nTask:5(Different Algorithmic Approaches)")
print("\n" + "="*80)
print("ALGORITHMIC APPROACHES TO STRING REVERSAL")
print("="*80)

# Approach 1: Loop-Based String Reversal
print("\n--- APPROACH 1: LOOP-BASED REVERSAL ---")
def reverse_loop(s):
    """
    Loop-based approach using iteration
    Time Complexity: O(n)
    Space Complexity: O(n)
    """
    reversed_str = ""
    for i in range(len(s) - 1, -1, -1):
        reversed_str += s[i]
    return reversed_str

test_string = "Hello, World!"
result1 = reverse_loop(test_string)
print(f"Input: {test_string}")
print(f"Output: {result1}")
print("Execution Flow: Iterate from last index to first, concatenate characters")
print("Time Complexity: O(n)")
print("Space Complexity: O(n)")

# Approach 2: Built-in Slicing-Based Reversal
print("\n--- APPROACH 2: SLICING-BASED REVERSAL ---")
def reverse_slicing(s):
    """
    Built-in slicing approach
    Time Complexity: O(n)
    Space Complexity: O(n)
    """
    return s[::-1]

result2 = reverse_slicing(test_string)
print(f"Input: {test_string}")
print(f"Output: {result2}")
print("Execution Flow: Use Python's slice notation with negative step")
print("Time Complexity: O(n)")
print("Space Complexity: O(n)")

# Comparison Analysis
print("\n" + "="*80)
print("DETAILED COMPARISON")
print("="*80)

comparison = {
    "Criteria": ["Execution Flow", "Time Complexity", "Space Complexity", "Code Readability", "Performance (Smal
l)", "Performance (Large)", "When to Use"],
```

```
    "Loop-Based": [
        "Manual iteration with index control",
        "O(n) - linear time",
        "O(n) - new string created",
        "More verbose, explicit logic",
        "Comparable",
        "Slightly slower due to loop overhead",
        "Educational purposes, custom logic needed"
    ],
    "Slicing-Based": [
        "Built-in Python slice operation",
        "O(n) - linear time",
        "O(n) - new string created",
        "Concise and Pythonic",
        "Comparable",
        "Faster - optimized C implementation",
        "Production code, performance-critical applications"
    ]
}

print(f"\n{'Criteria':<25} {'Loop-Based':<35} {'Slicing-Based':<35}")
print("-"*95)
for i, criteria in enumerate(comparison["Criteria"]):
    print(f"{criteria:<25} {comparison['Loop-Based'][i]:<35} {comparison['Slicing-Based'][i]:<35}")

print("\n" + "="*80)
print("RECOMMENDATION: Use slicing (s[::-1]) for production code - it's faster,")
print("cleaner, and leverages Python's optimized built-in operations.")
print("="*80)
```

1. **Approach 1 (Loop-Based Reversal)**:
    - **Execution**: Iterates from the last character to the first, building the reversed string.
    - **Complexity**: Both time and space complexity are O(n) (linear).
    - **When to Use**: Best for educational purposes or custom logic.
2. **Approach 2 (Slicing-Based Reversal)**:
    - **Execution**: Uses Python's slicing feature (`[::-1]`) to reverse the string.
    - **Complexity**: Time and space complexity are O(n) (linear).
    - **When to Use**: Ideal for production code, performance-critical applications.
3. **Comparison**:
    - **Execution Flow**: Loop-based is manual, while slicing is more concise.
    - **Time & Space Complexity**: Both approaches have the same O(n) complexities.
    - **Performance**: Slicing is generally faster due to Python's internal optimizations.
    - **When to Use**: Loop-based for custom logic, slicing for cleaner, optimized production code.

Recommendation: Use slicing (`[::-1]`) for production code—it's faster and cleaner.