

AI ASSISTED CODING ASSIGNMENT-1

RUDROJU RUPA SRI

2303A51918

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

❖ Scenario

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

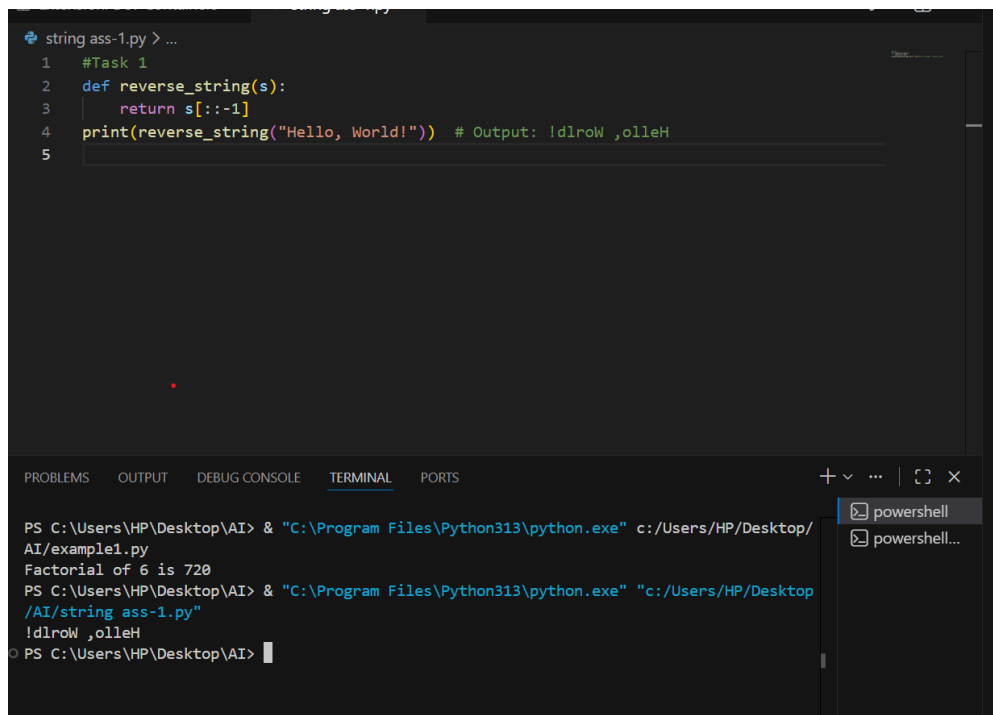
❖ Task Description

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



```
string ass-1.py > ...
1 #Task 1
2 def reverse_string(s):
3     return s[::-1]
4 print(reverse_string("Hello, World!")) # Output: !dlroW ,olleH
5
```

```
PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" c:/Users/HP/Desktop/AI/example1.py
Factorial of 6 is 720
PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" "c:/Users/HP/Desktop/AI/string ass-1.py"
!dlroW ,olleH
PS C:\Users\HP\Desktop\AI>
```

```
#Task 1
def reverse_string(s):
    return s[::-1]
print(reverse_string("Hello, World!")) # Output: !dlroW ,olleH
```

- This code defines a function named **reverse_string**.

- The function takes a string as input.
- It uses slicing `[::-1]` to reverse the string.
- The reversed string is returned by the function.
- Finally, the program prints the reversed output: **!dlroW ,olleH**.

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
6 def reverse_string(s):
7     """Reverse a string using slicing."""
8     return s[::-1]
9
10 print(reverse_string("Hello, World!")) # Output: !dlroW ,olleH
11
12 # Task 2
13
```

```
6 # Task 2
7 def reverse_string(s):
8     """Reverse a string using slicing."""
9     return s[::-1]
10
11 print(reverse_string("Hello, World!")) # Output: !dlroW ,olleH
12
13 # Task 2
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
!dlroW ,olleH
● PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" "c:/Users/HP/Desktop/AI/string ass-1.py"
!dlroW ,olleH
!dlroW ,olleH
● PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" "c:/Users/HP/Desktop/AI/string ass-1.py"
!dlroW ,olleH
!dlroW ,olleH
○ PS C:\Users\HP\Desktop\AI>
```

```
# Task 2
def reverse_string(s):
    """Reverse a string using slicing."""
    return s[::-1]

print(reverse_string("Hello, World!")) # Output: !dlroW ,olleH
```

1. The code defines a function named **reverse_string**.
2. It takes one parameter **s**, which is a string.
3. The line `s[::-1]` reverses the string using Python slicing.
4. The function returns the reversed string as output.
5. The print statement calls the function and displays **!dlroW ,olleH**.

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

❖ Scenario

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

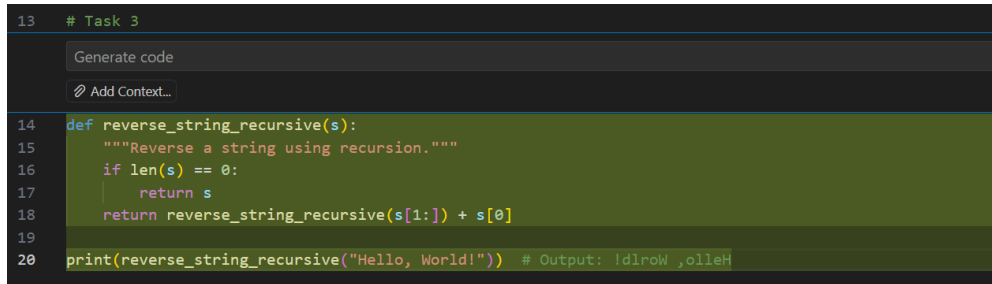
❖ Task Description

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

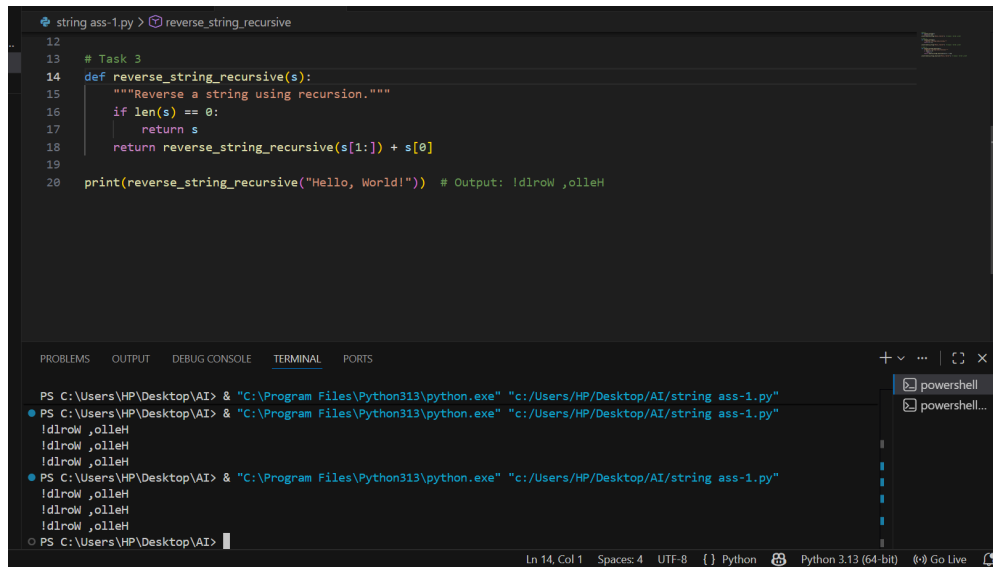
- Uses a user-defined function to reverse a string
- Returns the reversed string
- Includes meaningful comments (AI-assisted)

❖ Expected Output

- Correct function-based implementation
- Screenshots documenting Copilot's function generation
- Sample test cases and outputs



```
13 # Task 3
14 def reverse_string_recursive(s):
15     """Reverse a string using recursion."""
16     if len(s) == 0:
17         return s
18     return reverse_string_recursive(s[1:]) + s[0]
19
20 print(reverse_string_recursive("Hello, World!")) # Output: !dlroW ,olleH
```



```
# Task 3
def reverse_string_recursive(s):
    """Reverse a string using recursion."""
    if len(s) == 0:
        return s
    return reverse_string_recursive(s[1:]) + s[0]

print(reverse_string_recursive("Hello, World!")) # Output: !dlroW ,olleH
```

- This code defines a function called **reverse_string_recursive**.
- It uses **recursion** to reverse the string.
- If the string is empty, it returns the string (base case).
- The function calls itself with the remaining string and adds the first character at the end.
- The final output printed is **!dlroW ,olleH**.
- This code compares execution time of reversing a string **with and without a function**.
- First, the string is reversed directly using slicing inside a loop.
- Then, the same string is reversed using a **recursive function**.
- The `time` module measures how long each method takes.
- The results show that recursion is **slower than direct slicing**.

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

```
string ass-1.py X
string: C:\Users\JP\Desktop\AI\string ass-1.py - Pending changes from chat
38     "Not suitable - hard to manage",
39     "Similar - no overhead",
40     "Low - changes affect multiple places"
41 ],
42 "With Functions (Modular)": [
43     "Higher - clear, named logic blocks",
44     "Excellent - call function multiple times",
45     "Easier - isolated logic units",
46     "Highly suitable - scalable design",
47     "Similar - negligible function overhead",
48     "High - changes in one place"
49 ]
50 }
51
52 for i, criterion in enumerate(comparison["Criterion"]):
53     print(f"\n{criterion}:")
54     print(f"    Procedural: {comparison['Without Functions (Procedural)'][i]}")
55     print(f"    Modular:    {comparison['With Functions (Modular)'][i]}")
56
57 print("\n" + "="*70)
58 print("CONCLUSION: Modular approach (with functions) is superior for")
59 print("production code due to better maintainability and scalability.")
60 print("\n" + "="*70)
```

```
25 # Task 4: Comparative Analysis - Procedural vs Modular Approach
26 # Comparative Analysis Report
27 print("\n" + "="*70)
28 print("COMPARATIVE ANALYSIS: Procedural vs Modular Approach")
29 print("\n" + "="*70)
30
31 comparison = {
32     "Criterion": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability", "Performance", "Maintainability"],
33     "Without Functions (Procedural)": [
34         "Lower - inline code harder to follow",
35         "Poor - code duplication required",
36         "Difficult - logic scattered throughout",
37         "Not suitable - hard to manage",
38         "Similar - no overhead",
39         "Low - changes affect multiple places"
40     ],
41     "With Functions (Modular)": [
42         "Higher - clear, named logic blocks",
43         "Excellent - call function multiple times",
44         "Easier - isolated logic units",
45         "Highly suitable - scalable design",
46         "Similar - negligible function overhead",
47         "High - changes in one place"
48     ]
49 }
50
51 for i, criterion in enumerate(comparison["Criterion"]):
52     print(f"\n{criterion}:")
53     print(f"    Procedural: {comparison['Without Functions (Procedural)'][i]}")
54     print(f"    Modular:    {comparison['With Functions (Modular)'][i]}")
55
56 print("\n" + "="*70)
57 print("CONCLUSION: Modular approach (with functions) is superior for")
58 print("production code due to better maintainability and scalability.")
59 print("\n" + "="*70)
```

```
string ass-1.py 2...
25 # Task 4: Comparative Analysis - Procedural vs Modular Approach
26 # Comparative Analysis Report
27 print("\n" + "="*70)
28 print("COMPARATIVE ANALYSIS: Procedural vs Modular Approach")
29 print("\n" + "="*70)
30
31 comparison = {
32     "Criterion": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability", "Performance", "Maintainability"],
33     "Without Functions (Procedural)": [
34         "Lower - inline code harder to follow",
35         "Poor - code duplication required",
36         "Difficult - logic scattered throughout",
37         "Not suitable - hard to manage",
38         "Similar - no overhead",
39         "Low - changes affect multiple places"
40     ],
41     "With Functions (Modular)": [
42         "Higher - clear, named logic blocks",
43         "Excellent - call function multiple times",
44         "Easier - isolated logic units",
45         "Highly suitable - scalable design",
46         "Similar - negligible function overhead",
47         "High - changes in one place"
48     ]
49 }
50
51 for i, criterion in enumerate(comparison["Criterion"]):
52     print(f"\n{criterion}:")
53     print(f"    Procedural: {comparison['Without Functions (Procedural)'][i]}")
54     print(f"    Modular:    {comparison['With Functions (Modular)'][i]}")
55
56 print("\n" + "="*70)
57 print("CONCLUSION: Modular approach (with functions) is superior for")
58 print("production code due to better maintainability and scalability.")
59 print("\n" + "="*70)
60 print("\n" + "="*70)
```

PS C:\Users\JP\Desktop\AI> & "C:\Program Files\Python311\python.exe" "C:\Users\JP\Desktop\AI\string ass-1.py"

string ass-1.py

string ass-1.py

string ass-1.py

COMPARATIVE ANALYSIS: Procedural vs Modular Approach

Code Clarity:

Procedural: Lower - inline code harder to follow

Modular: Higher - clear, named logic blocks

Reusability:

Procedural: Poor - code duplication required

Modular: Excellent - call function multiple times

Debugging Ease:

Procedural: Difficult - logic scattered throughout

Modular: Easier - isolated logic units

Large-Scale Suitability:

Procedural: Not suitable - hard to manage

Modular: Highly suitable - scalable design

Performance:

Procedural: Similar - no overhead

Modular: Similar - negligible function overhead

Maintainability:

Procedural: Low - changes affect multiple places

Modular: High - changes in one place

CONCLUSION: Modular approach (with functions) is superior for production code due to better maintainability and scalability.

PS C:\Users\JP\Desktop\AI>

```

Maintainability:
  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

=====

Maintainability:
  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

Maintainability:
  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

Maintainability:

Maintainability:
  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

=====

Maintainability:
  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

=====

  Procedural: Low - changes affect multiple places
  Modular:    High - changes in one place

=====

  Modular:    High - changes in one place

=====

CONCLUSION: Modular approach (with functions) is superior for
CONCLUSION: Modular approach (with functions) is superior for
production code due to better maintainability and scalability.
=====

```

```

# Task 4: Comparative Analysis - Procedural vs Modular Approach
# Comparative Analysis Report
print("\n" + "="*70)
print("COMPARATIVE ANALYSIS: Procedural vs Modular Approach")
print("="*70)

comparison = {
    "Criterion": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability", "Performance", "Maintainability"],
    "Without Functions (Procedural)": [
        "Lower - inline code harder to follow",
        "Poor - code duplication required",
        "Difficult - logic scattered throughout",
        "Not suitable - hard to manage",
        "Similar - no overhead",
        "Low - changes affect multiple places"
    ],
    "With Functions (Modular)": [
        "Higher - clear, named logic blocks",
        "Excellent - call function multiple times",
        "Easier - isolated logic units",
        "Highly suitable - scalable design",
        "Similar - negligible function overhead",
        "High - changes in one place"
    ]
}

```

```

    ]
}

for i, criterion in enumerate(comparison["Criterion"]):
    print(f"\n{criterion}:")
    print(f"   Procedural: {comparison['Without Functions (Procedural)'][i]}")
    print(f"   Modular:      {comparison['With Functions (Modular)'][i]}")

print("\n" + "="*70)
print("CONCLUSION: Modular approach (with functions) is superior for")
print("production code due to better maintainability and scalability.")
print("="*70)

```

- This code compares **procedural (without functions)** and **modular (with functions)** programming.
- A dictionary stores comparison points like clarity, reusability, debugging, and performance.
- The program prints each criterion and shows both approaches side by side.
- It highlights that modular code is easier to manage, debug, and reuse.
- The conclusion states that **using functions is better for large and real-world projects**

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.

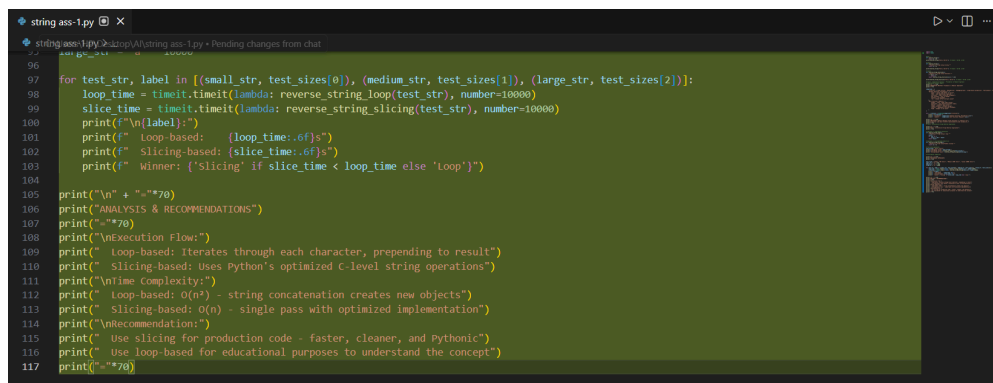
❖ Task Description

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



```

string-ass-1.py X
string-ass-1.py:2420/A/string-ass-1.py • Pending changes from chat
96
97 for test_str, label in [(small_str, test_sizes[0]), (medium_str, test_sizes[1]), (large_str, test_sizes[2])]:
98     loop_time = timeit.timeit(lambda: reverse_string_loop(test_str), number=10000)
99     slice_time = timeit.timeit(lambda: reverse_string_slicing(test_str), number=10000)
100     print(f"\n{label}:")
101     print(f"   Loop-based:      {loop_time:.6f}s")
102     print(f"   Slicing-based:    {slice_time:.6f}s")
103     print(f"   Winner: {'Slicing' if slice_time < loop_time else 'Loop'}")
104
105 print("\n" + "="*70)
106 print("ANALYSIS & RECOMMENDATIONS")
107 print("="*70)
108 print("\nExecution Flow:")
109 print("  Loop-based: Iterates through each character, prepending to result")
110 print("  Slicing-based: Uses Python's optimized C-level string operations")
111 print("\nTime Complexity:")
112 print("  Loop-based: O(n²) - string concatenation creates new objects")
113 print("  Slicing-based: O(n) - single pass with optimized implementation")
114 print("\nRecommendation:")
115 print("  Use slicing for production code - faster, cleaner, and Pythonic")
116 print("  Use loop-based for educational purposes to understand the concept")
117 print("\n" + "="*70)

```

```
string-ass-1.py X
# Task 5: Alternative String Reversal Approaches
"""
TASK 5: Alternative String Reversal Approaches
"""
print("\n ~~~~")
print("TASK 5: Alternative String Reversal Approaches")
print("\n ~~~~")

# Approach 1: Loop-based (Iterative)
def reverse_string_loop(s):
    """Reverse a string using a loop"""
    result = ""
    for char in s:
        result = char + result
    return result

# Approach 2: Slicing-based (Built-in)
def reverse_string_slicing(s):
    """Reverse a string using Python slicing"""
    return s[::-1]

# Test both approaches
test_string = "Hello, World!"
print(f"Original string: {test_string}")
print(f"Loop-based result: {reverse_string_loop(test_string)}")
print(f"Slicing-based result: {reverse_string_slicing(test_string)}")

# Performance comparison
print("\n ~~~~")
print("PERFORMANCE COMPARISON")
print("\n ~~~~")

test_cases = ["Small (100 chars)", "Medium (1000 chars)", "Large (10000 chars)"]
small_str = "a" * 100
medium_str = "a" * 1000
large_str = "a" * 10000

for test_str, label in [(small_str, test_cases[0]), (medium_str, test_cases[1]), (large_str, test_cases[2])]:
    loop_time = timeit.timeit(lambda: reverse_string_loop(test_str), number=1000)
    slice_time = timeit.timeit(lambda: reverse_string_slicing(test_str), number=1000)
    print(f"Label: {label}")
    print(f"Loop-based: {loop_time:.6f}s")
    print(f"Slicing-based: {slice_time:.6f}s")
    print(f"Winner: {'Slicing' if slice_time < loop_time else 'Loop'}")

print("\n ~~~~")
print("ANALYSIS & RECOMMENDATIONS")
print("\n ~~~~")

# Execution Flow
print("Execution Flow:")
print("Loop-based: Iterates through each character, prepending to result")
print("Slicing-based: Uses Python's optimized C-level string operations")

# Time Complexity
print("Time Complexity:")
print("Loop-based: O(n^2) - string concatenation creates new objects")
print("Slicing-based: O(n) - single pass with optimized implementation")

# Recommendation
print("Recommendation:")
print("Use slicing for production code - faster, cleaner, and Pythonic")
print("Use loop-based for educational purposes to understand the concept")
```

```
PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" "c:/Users/HP/Desktop/AI/string-ass-1.py"

=====
TASK 5: Alternative String Reversal Approaches
=====

Original string: Hello, World!
Loop-based result: !dlroW ,olleH
Slicing-based result: !dlroW ,olleH

=====
PERFORMANCE COMPARISON
=====

Small (100 chars):
Loop-based: 0.223679s
Slicing-based: 0.003623s
Winner: Slicing

Medium (1000 chars):
Loop-based: 2.409064s
Slicing-based: 0.047626s
Winner: Slicing

Large (10000 chars):
Loop-based: 36.823376s
Slicing-based: 0.254383s
Winner: Slicing

=====
ANALYSIS & RECOMMENDATIONS
=====

Execution Flow:
Loop-based: Iterates through each character, prepending to result
Slicing-based: Uses Python's optimized C-level string operations

Time Complexity:
Loop-based: O(n^2) - string concatenation creates new objects
Slicing-based: O(n) - single pass with optimized implementation

Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept

Time Complexity:
Loop-based: O(n^2) - string concatenation creates new objects
Slicing-based: O(n) - single pass with optimized implementation

Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
```



```

PS C:\Users\HP\Desktop\AI> & "C:\Program Files\Python313\python.exe" "c:/Users/HP/Desktop/AI/string ass-1.py"
Loop-based: 36.823376s
Slicing-based: 0.254383s
Winner: Slicing

=====
ANALYSIS & RECOMMENDATIONS
=====

Execution Flow:
Loop-based: Iterates through each character, prepending to result
Slicing-based: Uses Python's optimized C-level string operations

Time Complexity:
Loop-based: O(n²) - string concatenation creates new objects
Slicing-based: O(n) - single pass with optimized implementation

Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept

Time Complexity:
Loop-based: O(n²) - string concatenation creates new objects
Slicing-based: O(n) - single pass with optimized implementation

Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept
Slicing-based: O(n) - single pass with optimized implementation

Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept
Recommendation:
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept
Use slicing for production code - faster, cleaner, and Pythonic
Use loop-based for educational purposes to understand the concept
=====
PS C:\Users\HP\Desktop\AI> 

```

Task 5: Alternative String Reversal Approaches

```

print("\n" + "="*70)
print("TASK 5: Alternative String Reversal Approaches")
print("="*70)

# Approach 1: Loop-based (Iterative)
def reverse_string_loop(s):
    """Reverse a string using a loop."""
    result = ""
    for char in s:
        result = char + result
    return result

# Approach 2: Slicing-based (Built-in)
def reverse_string_slicing(s):
    """Reverse a string using Python slicing."""
    return s[::-1]

# Test both approaches
test_string = "Hello, World!"
print(f"\nOriginal string: {test_string}")
print(f"Loop-based result: {reverse_string_loop(test_string)}")
print(f"Slicing-based result: {reverse_string_slicing(test_string)}")

# Performance comparison

```

```

print("\n" + "="*70)
print("PERFORMANCE COMPARISON")
print("="*70)

test_sizes = ["Small (100 chars)", "Medium (1000 chars)", "Large (10000 chars)"]
small_str = "a" * 100
medium_str = "a" * 1000
large_str = "a" * 10000

for test_str, label in [(small_str, test_sizes[0]), (medium_str, test_sizes[1]), (large_str, test_sizes[2])]:
    loop_time = timeit.timeit(lambda: reverse_string_loop(test_str), number=10000)
    slice_time = timeit.timeit(lambda: reverse_string_slicing(test_str), number=10000)
    print(f"\n{label}:")
    print(f"  Loop-based:      {loop_time:.6f}s")
    print(f"  Slicing-based:    {slice_time:.6f}s")
    print(f"  Winner: {'Slicing' if slice_time < loop_time else 'Loop'}")

print("\n" + "="*70)
print("ANALYSIS & RECOMMENDATIONS")
print("="*70)
print("\nExecution Flow:")
print("  Loop-based: Iterates through each character, prepending to result")
print("  Slicing-based: Uses Python's optimized C-level string operations")
print("\nTime Complexity:")
print("  Loop-based: O(n²) - string concatenation creates new objects")
print("  Slicing-based: O(n) - single pass with optimized implementation")
print("\nRecommendation:")
print("  Use slicing for production code - faster, cleaner, and Pythonic")
print("  Use loop-based for educational purposes to understand the concept")
print("="*70)

```

Execution Complexity

- **Loop-based approach: $O(n^2)$** because strings are immutable and each concatenation creates a new string.
- **Slicing-based approach: $O(n)$** because Python reverses the string in one optimized operation.

◆ When to Use Each Approach

- **Loop-based reversal:** Use for **learning and understanding logic**, not for large data.
- **Slicing-based reversal:** Use in **real-world and production code** for speed and simplicity.

◆ Performance Comparison

- **Loop-based:** Slower, especially for long strings.
- **Slicing-based:** Much faster and more memory-efficient.

◆ Overall Comparison

Feature	Loop-Based	Slicing-Based
Time Complexity	$O(n^2)$	$O(n)$
Speed	Slow	Fast
Code Length	Longer	Very short
Best Use	Learning	Production