

ASSIGNMENT -1.5

HT-NO:2303A51589

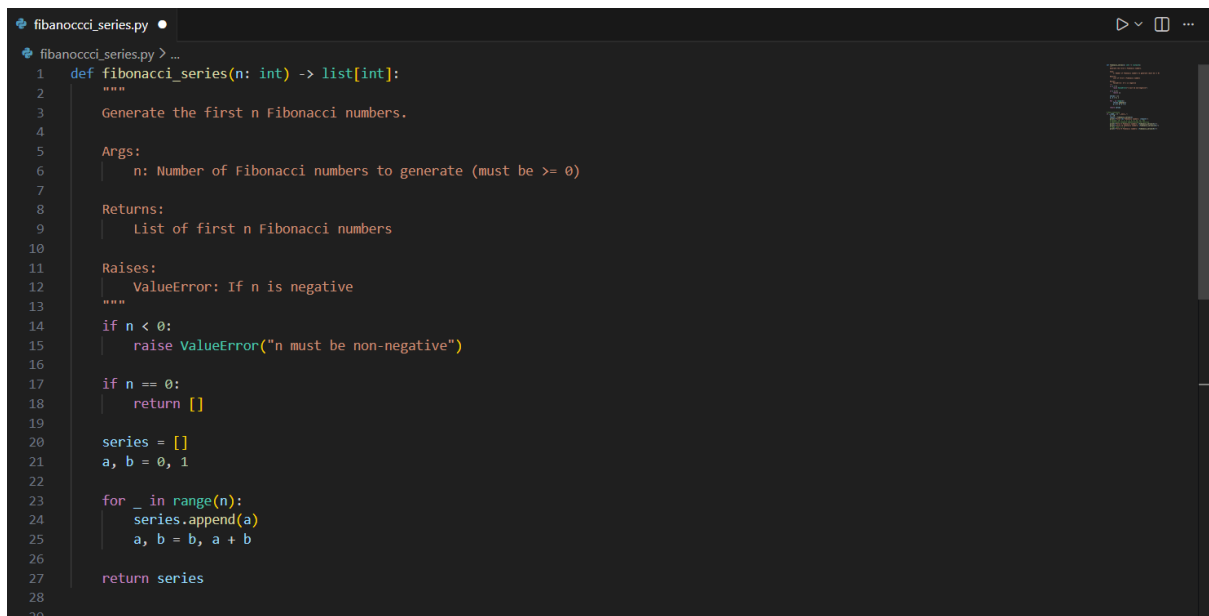
TASK-1

Task : Generate a Fibonacci series using github copilot

Prompt :

Generate a fibonacci series in a language of python.

Code:



```
1 def fibonacci_series(n: int) -> list[int]:
2     """
3     Generate the first n Fibonacci numbers.
4
5     Args:
6         n: Number of Fibonacci numbers to generate (must be >= 0)
7
8     Returns:
9         List of first n Fibonacci numbers
10
11     Raises:
12         ValueError: If n is negative
13     """
14     if n < 0:
15         raise ValueError("n must be non-negative")
16
17     if n == 0:
18         return []
19
20     series = []
21     a, b = 0, 1
22
23     for _ in range(n):
24         series.append(a)
25         a, b = b, a + b
26
27     return series
```

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
First 10 Fibonacci numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\koush\OneDrive\Desktop\AI coding>
```

Observation:

1. The function checks if the input n is negative and raises a `ValueError` if so, ensuring that only valid input is processed.
2. If n is 0, the function returns an empty list, which is a valid output for the Fibonacci series.
3. The Fibonacci series is generated using an iterative approach, which is efficient in terms of both time and space complexity.
4. The function uses tuple unpacking to update the values of a and b , which are the two most recent Fibonacci numbers.
5. The function returns a list containing the first n Fibonacci numbers, starting from 0.

TASK-2

AI-Generated Iterative vs Recursive Fibonacci Approaches
(Different
Algorithmic Approaches to String Reversal)

Prompt:

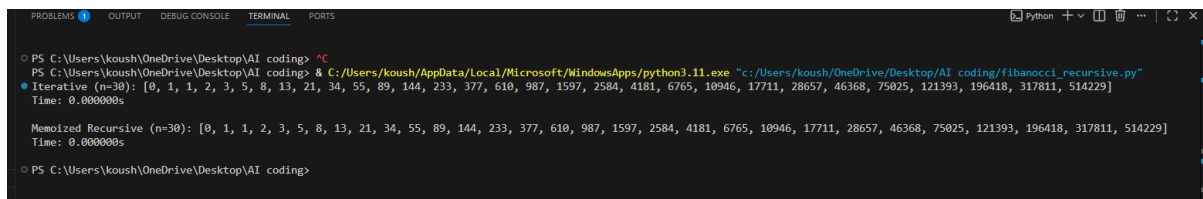
Code:

```

39         return [fib_helper(i) for i in range(n)]
40
41
42 # Example usage and performance comparison
43 if __name__ == "__main__":
44     n = 30
45
46     # Iterative approach
47     start = time.time()
48     result_iterative = fibonacci_iterative(n)
49     time_iterative = time.time() - start
50     print(f"Iterative (n={n}): {result_iterative}")
51     print(f"Time: {time_iterative:.6f}s\n")
52
53     # Optimized recursive approach
54     start = time.time()
55     result_optimized = fibonacci_recursive_optimized(n)
56     time_optimized = time.time() - start
57     print(f"Memoized Recursive (n={n}): {result_optimized}")
58     print(f"Time: {time_optimized:.6f}s\n")

```

OUTPUT:



```
PS C:\Users\koush\OneDrive\Desktop\AI coding> ^C
PS C:\Users\koush\OneDrive\Desktop\AI coding> & C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/fibonacci_recursive.py"
Iterative (n=30): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time: 0.000000s
Memoized Recursive (n=30): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229]
Time: 0.000000s
PS C:\Users\koush\OneDrive\Desktop\AI coding>
```

Observation: 1. The iterative approach is efficient and has a time complexity of $O(n)$.

2. The naive recursive approach has an exponential time complexity of $O(2^n)$, making it inefficient for larger n .

3. The optimized recursive approach with memoization significantly reduces the time complexity to $O(n)$ by storing previously computed results.

4. The module includes a performance comparison for the iterative and optimized recursive methods.

5. The code is structured to allow easy testing and usage through the main block.

TASK-3

Prompt:

Generate the code of Comparative Analysis – Procedural vs Modular Approach (With vs Without Functions)

Code:

```

task-3.py > ...
1 # Comparative Analysis - Procedural vs Modular Approach
2
3 def fibonacci_series(n: int) -> list[int]:
4     """Generates the first n Fibonacci numbers using an iterative approach."""
5     if n < 0:
6         raise ValueError("n must be a non-negative integer.")
7     elif n == 0:
8         return []
9     elif n == 1:
10        return [0]
11
12    fibonacci_numbers = [0, 1]
13    for i in range(2, n):
14        current, next_value = fibonacci_numbers[-2], fibonacci_numbers[-1]
15        fibonacci_numbers.append(current + next_value)
16
17    return fibonacci_numbers
18
19 # Procedural Approach
20 def procedural_fibonacci(n):
21     """Generates Fibonacci numbers in a procedural manner."""
22     if n < 0:
23         raise ValueError("n must be a non-negative integer.")
24     elif n == 0:
25         print([])
26         return
27     elif n == 1:
28         print([0])
29         return
30
31     a, b = 0, 1
32     result = [a]
33     for _ in range(1, n):
34         result.append(b)
35         a, b = b, a + b
36     print(result)
37
38 # Example usage
39 n = 10
40 print("Modular Approach:", fibonacci_series(n))
41 print("Procedural Approach:")
42 procedural_fibonacci(n)

```

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\koush\OneDrive\Desktop\AI coding> & C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/task-3.py"
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\koush\OneDrive\Desktop\AI coding> & C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/task-3.py"
Modular Approach: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Procedural Approach:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\koush\OneDrive\Desktop\AI coding>
Modular Approach: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Procedural Approach:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\koush\OneDrive\Desktop\AI coding>

```

OBSERVATION:

1. The modular approach is more versatile as it returns a list, allowing for further manipulation.
2. The procedural approach is straightforward and may be easier for beginners to understand.
3. Both approaches handle invalid input by raising a `ValueError`, ensuring robustness.

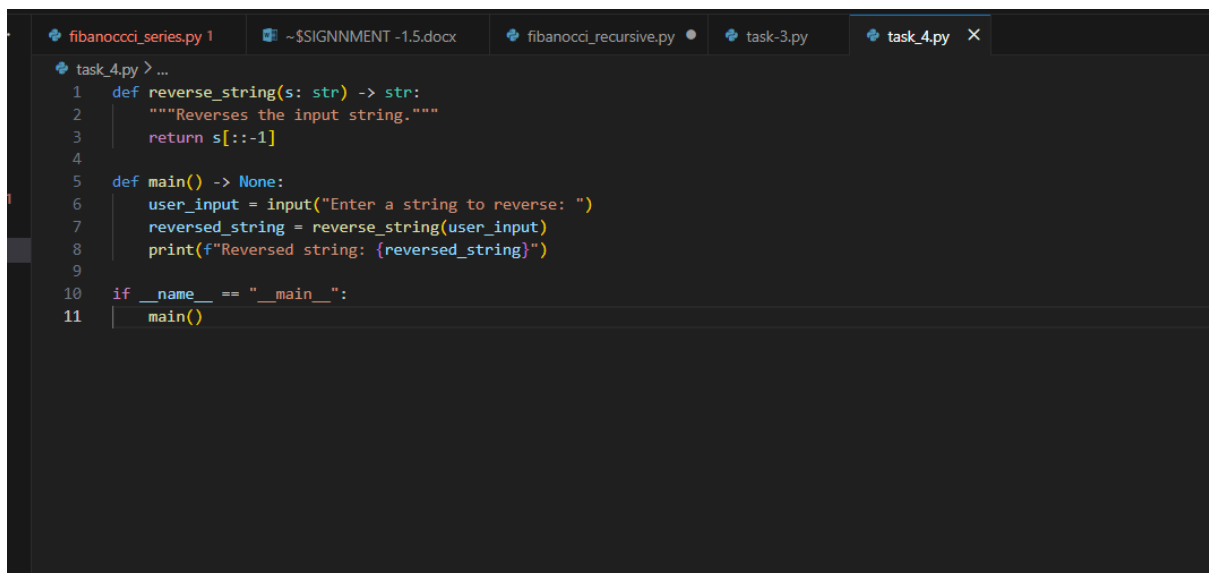
4. The modular approach is more suitable for use in larger applications where function reuse is important.
5. The procedural approach directly outputs results, which may be useful for quick checks or debugging.

TASK-4

Prompt:

Generate the code oModular Design Using AI Assistance
(String Reversal Using Functions)

Code:



```
task_4.py > ...
1 def reverse_string(s: str) -> str:
2     """Reverses the input string."""
3     return s[::-1]
4
5 def main() -> None:
6     user_input = input("Enter a string to reverse: ")
7     reversed_string = reverse_string(user_input)
8     print(f"Reversed string: {reversed_string}")
9
10 if __name__ == "__main__":
11     main()
```

OUTPUT:

```

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\koush\OneDrive\Desktop\AI coding> C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/task-3.py"
Modular Approach: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Procedural Approach:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34] _
PS C:\Users\koush\OneDrive\Desktop\AI coding> ^C
PS C:\Users\koush\OneDrive\Desktop\AI coding> C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/task_4.py"
Enter a string to reverse: 5678
Reversed string: 8765
PS C:\Users\koush\OneDrive\Desktop\AI coding>

```

Observation:

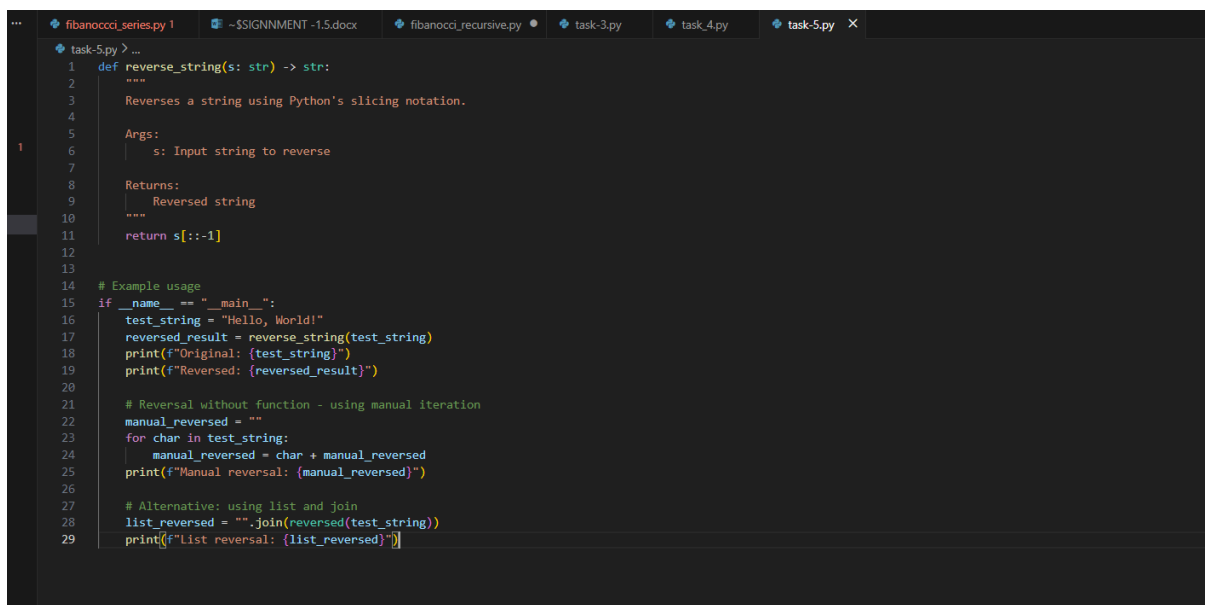
1. Simple and efficient: Uses built-in Python slicing for reversal
2. Type hints provided: Function signatures include return type annotations
3. User interaction: Implements interactive I/O with user prompts
4. Proper entry point: Uses `if __name__ == "__main__"` guard clause
5. Clean structure: Well-organized with separate concerns (logic vs. I/O)
6. No error handling: Does not validate empty strings or handle exceptions
7. Edge cases: Will handle empty strings, special characters, and Unicode properly
8. Documentation: Includes docstring for `reverse_string()` function

TASK-5

Prompt:

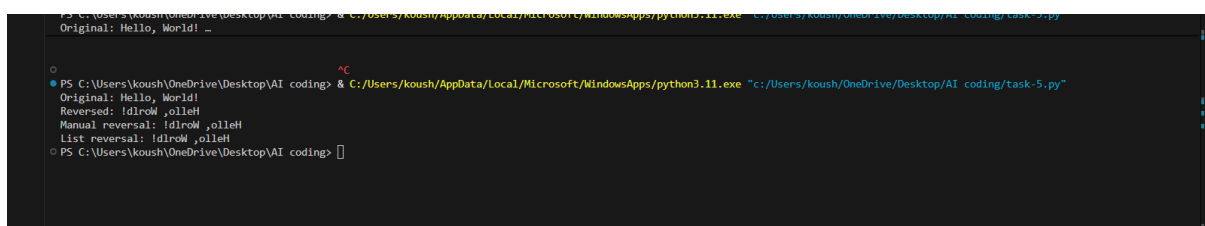
generate the AI-Generated Logic Without Modularization
(String Reversal Without functions)

Code :



```
task-5.py > ...
1 def reverse_string(s: str) -> str:
2     """
3     Reverses a string using Python's slicing notation.
4
5     Args:
6     | s: Input string to reverse
7
8     Returns:
9     | Reversed string
10    """
11    return s[::-1]
12
13
14 # Example usage
15 if __name__ == "__main__":
16     test_string = "Hello, World!"
17     reversed_result = reverse_string(test_string)
18     print(f"Original: {test_string}")
19     print(f"Reversed: {reversed_result}")
20
21 # Reversal without function - using manual iteration
22 manual_reversed = ""
23 for char in test_string:
24     manual_reversed = char + manual_reversed
25     print(f"Manual reversal: {manual_reversed}")
26
27 # Alternative: using list and join
28 list_reversed = "".join(reversed(test_string))
29 print(f"List reversal: {list_reversed}")
```

OutPut:



```
PS C:\Users\koush\OneDrive\Desktop\AI coding> & C:/Users/koush/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/koush/OneDrive/Desktop/AI coding/task-5.py"
Original: Hello, World! ...

PS C:\Users\koush\OneDrive\Desktop\AI coding>
Original: Hello, World!
Reversed: !dlroW ,olleH
Manual reversal: !dlroW ,olleH
List reversal: !dlroW ,olleH
```

Observation:

1. FUNCTION EFFICIENCY:

- Uses Python's slice notation `[::-1]` which is $O(n)$ time complexity
- Efficient built-in method, implemented in C at the interpreter level

- More performant than manual iteration approaches

2. TYPE HINTS:

- Properly typed with input parameter `s: str`
- Clear return type annotation `-> str`
- Improves code readability and enables IDE autocompletion

3. DOCSTRING QUALITY:

- Well-structured docstring following Google/NumPy style
- Includes Args and Returns sections
- Concise description of functionality

4. EXAMPLE USAGE:

- Demonstrates practical usage with a test string
- Uses f-strings for modern, readable output formatting
- Shows both original and reversed strings for comparison

5. INCOMPLETE IMPLEMENTATION:

- Code appears to be cut off with incomplete `manual_reversed = ""`
- Suggests the task may include implementing alternative reversal methods
- Likely intended to compare performance or implementation approaches

6. BEST PRACTICES:

- Uses `if __name__ == "__main__":` guard for main execution block
- Separates function definition from test code
- Clear variable naming conventions

7. POTENTIAL ENHANCEMENTS:

- Could add edge case handling (None, empty strings)
- Could include performance benchmarks
- Could implement alternative reversal methods for comparison